

Data Structures

Notes taken by Yeheli Fomberg <yp@campus.technion.ac.il>
Based on lectures by CS faculty

Winter 2025

These notes are not endorsed by the lecturers. I have revised them outside lectures to incorporate supplementary explanations, clarifications, and material for fun. While I have strived for accuracy, any errors or misinterpretations are most likely mine.

Contents

1	Introduction	4
1.1	Abstract data types	4
1.2	Time complexity	5
1.3	Exercises	9
1.4	Recursive equations	12
2	Arrays, matrices, sparse matrices and linked lists	17
2.1	Arrays	17
2.2	Initializing an array in $\mathcal{O}(1)$	18
2.3	Matrices	19
2.4	Linked lists	20
2.5	Sparse polynomials and matrices	21
2.6	Exercises	22
3	Trees and binary search trees	27
3.1	Trees	27
3.2	Traversals	31
3.3	Left-child right-sibling binary trees	34
3.4	Search trees	35
3.5	Exercises	39
4	AVL trees	43
4.1	AVL trees	43
4.2	Implementation of AVL trees	48
4.3	Exercises	50
5	B-trees, rank trees, and tries	52
5.1	2-3 trees	52
5.2	B-trees and B+ trees	59
5.3	Rank trees	60
5.4	Tries	64
5.5	Radix trees	66
5.6	Exercises	67
6	Skip lists and amortized complexity	70
6.1	Skip lists	70
6.2	Amortized complexity	75
6.3	Exercises	81
7	Hash tables	84
7.1	Hash tables	84
7.2	Hash functions	89
7.3	Universal hash functions	91
7.4	Exercises	95
8	Union-find	97
8.1	Implementations	97
8.2	Exercises	106

9	Heap sort and Quick sort	111
9.1	Heap sort	112
9.2	Quick sort	116
9.3	Finding the i th largest element	120
9.4	Exercises	123
10	Bucket sort and radix sort	127
10.1	Selection	127
10.2	Comparison-based sorting lower bound	129
10.3	Bucket sort	132
10.4	Radix sort	133
10.5	Exercises	135
11	Strings	139
11.1	Sorting using tries	139
11.2	Suffix trees	140
11.3	Finding substrings in a long text	143
11.4	Text compression	144
11.5	Finding longest common substring	146
11.6	Exercises	147
12	Graphs	152
12.1	Introduction	152
12.2	Topological sorting	155
12.3	Minimum spanning tree	159
12.4	Exercises	163
13	Bloom filters	167

1 Introduction

Data structures, and the motivation to study them, appear everywhere. They are found at the core of the art of problem solving in computer science, interviews, and even in the implementation of your favourite programming library.

Example 1.1. How can we make a data structure for 15,000 students that allows insertion and access to students by their id with expected time complexity of $\mathcal{O}(1)$?

1.1 Abstract data types

Definition 1.1 (Abstract data type). An abstract data type (ADT) is a mathematical model for a data type like a list, queue, or a stack, defined by its behaviour from the stand point of the user.

Example 1.2 (Stack). A stack has multiple basic functionalities.

<code>push(S,x)</code>	Add <code>x</code> to the top of the stack.
<code>top(S)</code>	Access the top of the stack. Returns <code>null</code> if the stack is empty.
<code>pop(S)</code>	Deletes the element on the top of the stack. Does not affect an empty stack.
<code>create()</code>	Returns an empty stack.
<code>is_empty(S)</code>	Returns <code>true</code> if the stack is empty, and returns <code>false</code> otherwise.

Remark 1.1. The stack manages data in such a way that the last element pushed is the first to get popped. This method of organizing the data is colloquially known as the “last in first out” (LIFO) method.

Question. We can implement a stack using either an array, or a list. Which is more efficient? Note that:

- (1) A clear disadvantage of an array is that the size of the stack is limited.
- (2) It may seem like a list has no memory limit, but even when we use a list, our physical memory is limited.

Answer. In the real world we use a list of arrays. We define an array of size N to implement the stack, and when we reach the maximum capacity of the array, we link a new array of size N .

Question. What is the abstract data type defined by the following operations?

<code>create(c)</code>	Returns an initial instance of the structure.
<code>read(c)</code>	Returns an integer.
<code>read(create(c))</code>	Returns 0.
<code>increment(c)</code>	Changes <code>c</code> such that <code>read(increment(c)) = read(c) + 1</code> .

Answer. The correct answer is: *a counter!* Here is one implementation:

<code>create(c)</code>	Allocate memory to an integer <code>c</code> .
<code>read(c)</code>	Returns <code>c</code> .
<code>increment(c)</code>	Increments <code>c</code> such that <code>c = c + 1</code> .

Note that there is a problem in our implementation since we could reach overflow! How can we solve this? There are many different possible solutions, try to think of one of your own.

Example 1.3 (Queue). A queue is defined by the following operations:

<code>enqueue(Q, x)</code>	Add <code>x</code> to the end of the queue.
<code>head(S)</code>	Access start of the queue. Return <code>null</code> in the queue is empty.
<code>dequeue(S)</code>	Deletes the first element of the queue, and return the first element of the modified queue. Does not affect an empty queue.
<code>create()</code>	Returns an empty queue.
<code>is_empty(S)</code>	Returns <code>true</code> if the queue is empty, and <code>false</code> otherwise.

Remark 1.2. The queue manages data in such a way that the first element enqueued is the first to get dequeued. This method of organizing the data is colloquially known as the “first in first out” (FIFO) method.

Remark 1.3. As with the stack, we can implement a queue using an array, or a linked list.

Remark 1.4 (Real life advice). In real life, the biggest advantage of abstract data types is that they allow the developer freedom of implementation, and freedom to change the implementation to a better one without the user noticing anything, or having to change the way he calls the methods.

In `c++`, if we implement an abstract data type in multiple ways, for example, if we implement the stack once using an array, and once using a list, we can change the implementation from the array based stack to the list based stack with the following the code:

```
//typedef ArrayBasedStack Stack;
typedef ListBasedStack Stack;
```

Remark 1.5. The quality of the implementation is determined by the time and space complexity of its functions, and the simplicity of the implementation.

1.2 Time complexity

Definition 1.2 (Run-time). The run-time of an algorithm A on input x is the number of machine commands that the algorithms runs for that input. We denote it by $\text{time}_A(x)$.

Remark 1.6. We do not care about the difference in run-time of different commands.

Definition 1.3 (Length of input). The length of input x denoted $|x|$ is the number of bits that represents the input.

Remark 1.7. Sometimes, in less formal contexts, we are more lax in the definition in input length. For example, we may say that if x is a char array of length n , then $|x| = n$ instead of $|x| = 8n$.

Definition 1.4 (Worst-case run-time). Since it’s impossible to consider the run-time of an algorithm A for all inputs, we define a new function, that gives us the worst-case run-time of A , for an input of a particular length:

$$t_A(n) := \max \{ \text{time}_A(x) : |x| = n \}.$$

Example 1.4 (For loop). Consider the following code:

```
1 sum = 0;
2 for (i = 0; i < n; i++)
3     sum = sum + a[i];
```

We have that

$$n + 1 \leq t_A(n) \leq c_1 \cdot n + c_2,$$

where c_1, c_2 are constants that depend on the the implementation of the specific programming language.

Example 1.5 (Check prime). Consider the following code:

```

1 bool checkPrime(int num) {
2     for (i = 2; i < num; i++) {
3         if (num % i == 0) return false;
4     }
5     return true;
6 }
```

We have that

$$\begin{aligned} \text{time}(9888) &= c_1 + c_2 \\ \text{time}(9973) &= 9971 \cdot c_1 + c_2 \\ \text{time}(4) &= 9971 \cdot c_1 + c_2 \\ \text{time}(n) &= 10^n \cdot c_1 + c_2 \end{aligned}$$

and in general, we have that $t(n) < 10^{|n|} \cdot c_1 + c_2$, where $|n|$ is the amount of digits of the numbers.

Remark 1.8. Notice that we try to bound $\text{time}(n)$ with the *size* of the input and not the *value* of the input. That is because bounding $\text{time}(n)$ with a function in $|n|$ gives a bound for $t(n)$.

Remark 1.9. In what follows we will begin to analyze the function $t(n)$ more rigorously. However, instead of writing $t(n)$ and using very specific notations, we will analyze the asymptotic behaviour of more abstract, general functions.

Definition 1.5 (Asymptotic upper bound). Let $f(n), g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) \in \mathcal{O}(g(n))$ if there exist $c > 0$ and $n_0 > 0$ such that for all $n > n_0$ we have $f(n) \leq c \cdot g(n)$.

Remark 1.10. If $f(n) \in \mathcal{O}(g(n))$ we usually denote $f(n) = \mathcal{O}(g(n))$ and say that $g(n)$ is an asymptotic bound of $f(n)$. We do this because usually $f(n)$ is a function with erratic behaviour (like $t(n)$), and by bounding it asymptotically by a simpler function $g(n)$ simplifies many of our calculations.

Proposition 1.1. Let k be a constant. Then

$$f(n) = \sum_{i=0}^k a_i n^i = \mathcal{O}(n^k).$$

Proof. Let $a_{\max} = \max\{1, a_0, \dots, a_k\}$. Then we have

$$f(n) \leq a_{\max}(1 + n + \dots + n^k) = a_{\max} \frac{n^{k+1} - 1}{n - 1} \leq a_{\max} \frac{n^{k+1}}{n/2} = 2a_{\max} n^k,$$

where the first equality is from the formula for a geometric series, and the last inequality holds for $n \geq 2$. Thus we can choose $n_0 = 2$ and $c = 2a_{\max}$ and then $f(n) \leq c \cdot n^k$ for all $n > n_0$. This implies that $f(n) = \mathcal{O}(n^k)$ and completes the proof. $\square$

Example 1.6. Here are some more examples of function and their asymptotic bounds:

$$\begin{aligned} f(n) &= 10006 = \mathcal{O}(1) \text{ (Constant)} \\ f(n) &= 4n + 27 = \mathcal{O}(n) \text{ (Linear)} \\ f(n) &= n \log_2(n) + 3n^2 = \mathcal{O}(n^2) \text{ (Polynomial)} \end{aligned}$$

The last example follows from the inequality $\log_2(n) < n$ for $n \geq 1$.

Proposition 1.2. For all $a, b > 1$ we have that

$$f(n) = \log_a(n) = \mathcal{O}(\log_b(n)).$$

Proof. Follows from the identity $\log_a n = \log_a b \cdot \log_b n$. □

Proposition 1.3. For every $a > 1$ and $\epsilon > 0$ we have $\log_a n = \mathcal{O}(n^\epsilon)$.

Proof. For all $\epsilon > 0$ and $a > 1$ we have

$$\log_a n = \frac{\log_2 n}{\log_2 a} = \frac{\epsilon \cdot \log_2 n}{\epsilon \cdot \log_2 a} = \frac{1}{\epsilon \cdot \log_2 a} \cdot \log_2 n^\epsilon \leq c \cdot n^\epsilon,$$

which completes the proof. □

Example 1.7. Here are some more examples of function and their asymptotic bounds:

$$\begin{aligned} f(n) &= a^n = \mathcal{O}(b^n) \text{ (Exponential)} \\ f(n) &= n^k + a^n = \mathcal{O}(a^n) \\ f(n) &= n \cdot \log_2 n = \mathcal{O}(n \log n) \end{aligned}$$

Remark 1.11. Sometimes we may want to show that $f(n) \neq \mathcal{O}(g(n))$.

Proposition 1.4. We have that

- (1) $f(n) = 3^n \neq \mathcal{O}(2^n)$.
- (2) $f(n) = e^n \neq \mathcal{O}(n^k)$ for all natural $k \geq 1$.
- (3) $f(n) = \log n \neq \mathcal{O}(\log \log n)$.

Proof. We will solve case by case.

- (1) Assume, by way of contradiction, that there exist $c > 0$ and $n_0 \geq 0$ such that for all $n > n_0$ we have $3^n \leq c \cdot 2^n$. This implies that $\left(\frac{3}{2}\right)^n \leq c$ for all $n > n_0$. But this is obviously a contradiction which completes the proof.
- (2) Let $k > 0$. We know from analysis (by L'Hôpital's theorem) that

$$\lim_{x \rightarrow \infty} \frac{x^k}{e^x} = 0.$$

This implies that $e^n \neq \mathcal{O}(n^k)$ for all $k > 0$ as wanted.

- (3) Similarly, we have that

$$\lim_{n \rightarrow \infty} \frac{\log \log n}{\log n} = 0$$

which completes the proof. □

Remark 1.12. Recall from Example 1.4 that if A is the algorithm for summing elements of an array then $t_A(n) \leq c_1 n + c_2$. This implies that $t_A(n) = \mathcal{O}(n)$. We say that “summing elements of an array of length n is $\mathcal{O}(n)$ ”.

Example 1.8 (Matrix multiplication). When multiplying matrices of order $m \times m$ the complexity of the algorithm is $\mathcal{O}(m^3)$. But m is not the size of the input, and we vowed to compute complexity by the size of the input. Since the input is of 2 matrices, its length is equal to $n := 2m^2$. Therefore,

$$t(n) = \mathcal{O}(m^3) = \mathcal{O}\left(\frac{n^{\frac{3}{2}}}{2^{\frac{3}{2}}}\right) = \mathcal{O}(n^{\frac{3}{2}}).$$

Example 1.9 (Binary search). Recall the algorithm of a binary search of an element x in a sorted array of length n . We will try to analyze the run-time of the algorithm. First, we notice that $t(n)$ satisfies the following recurrence relation:

$$t(1) = c_1 \quad \text{and} \quad t(n) = c + T(\lceil n/2 \rceil).$$

Thus, if $n = 2^m$ is a power of 2 we get

$$\begin{aligned} t(n) &= c + T(n/2) = c + c + T(n/4) = \cdots = \underbrace{c + c + \cdots + c}_{i \text{ times}} + t(n/2^i) \\ &= c + c + \cdots + c + t\left(\frac{n}{2^{\log_2 n}}\right) = c \log_2 n + t(1) = \mathcal{O}(\log n). \end{aligned}$$

Example 1.10 (Weird loops). Consider the following code:

```

1 S = 0;
2 for(i = 1; i < n; i++)
3     for(j = 0; j < n; j += i)
4         S++;

```

Roughly, we have that

$$t(n) \leq n(n-1) = \mathcal{O}(n^2).$$

However, we can be more precise. When $i = 1$, we run n commands. When $i = 2$, we run $\lfloor n/2 \rfloor$ commands. And similarly, when $i = k$ we run $\lfloor n/k \rfloor$ commands, and thus we get more precisely that

$$t(n) \leq n + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n} = n \left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right) = n \cdot H_n,$$

where H_n is the n th partial sum of the harmonic series $a_n = 1/n$. From basic analysis, we have the inequality

$$\ln(n+1) = \int_1^{n+1} \frac{dx}{x} \leq \underbrace{\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}}_{=H_n} \leq 1 + \int_1^n \frac{dx}{x} = 1 + \ln(n).$$

Thus, we have that $t(n) \leq n \cdot H_n \leq n(1 + \ln n)$, so

$$t(n) \leq n \cdot H_n \leq n(1 + \ln n) \leq n(2 \ln n) = 2n \ln n = \mathcal{O}(n \log n),$$

where the third inequality holds for all $n \geq e$. Now it is left to note that n is not the actual length of the input, since the length of an array of size n is $a \cdot n$, where a is some constant. However, even if we multiply all the equations by the constant a , we will still reach the rigorous result $t(n) = \mathcal{O}(n \log n)$.

Definition 1.6 (Asymptotic lower bound). Let $f(n), g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) \in \Omega(g(n))$ if there exist $c > 0$ and $n_0 > 0$ such that for all $n > n_0$ we have $f(n) \geq c \cdot g(n)$.

Remark 1.13. Similarly, if $f(n) \in \Omega(g(n))$ we denote $f(n) = \Omega(g(n))$.

Remark 1.14. We have the relation $g(n) = \mathcal{O}(f(n)) \iff f(n) = \Omega(g(n))$.

Definition 1.7 (Asymptotic bound). Let $f(n), g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) = \Theta(g(n))$ if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.

Definition 1.8 (Little-o notation). Let $f(n), g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) = o(g(n))$ if for all $c > 0$ there exists $n_0 \geq 0$ so that for all $n \geq n_0$ we have $f(n) \leq c \cdot g(n)$. Equivalently, we say that $f(n) = o(g(n))$ if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

Remark 1.15. Intuitively, we say that $f(n) = o(g(n))$ if $f(n)$ is “asymptotically negligible” with respect to $g(n)$.

Example 1.11. We have that $\log(n) = o(n)$ and that $n - 100 \neq o(n)$.

Definition 1.9 (ω notation). Let $f(n), g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) = \omega(g(n))$ if for all $c > 0$ there exists $n_0 \geq 0$ so that for all $n \geq n_0$ we have $f(n) \geq c \cdot g(n)$. Equivalently, we say that $f(n) = \omega(g(n))$ if

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0.$$

Remark 1.16. Note that asymptotic complexity isn’t always what we want to optimize.

Example 1.12. Suppose algorithm A has a running time of $t_A(n) = 100n$, and algorithm B has a running time of $t_B(n) = 5n \log_2 n$. Asymptotically, algorithm A is better because it is linear, but for algorithms with input length $n < 2^{20}$, algorithm B is more efficient, so we prefer it over algorithm A .

Example 1.13. If we have two algorithms such that $t_A(n) = n^2$ and $t_B(n) = 10^{160}$, we would always prefer algorithm A over B , even though B is $\mathcal{O}(1)$ and A is $\mathcal{O}(n^2)$.

Remark 1.17. All of that being said, in the absolute majority of cases, we will prefer the algorithm with better asymptotic complexity.

1.3 Exercises

Exercise 1.1. Let $P(n) = \sum_{i=0}^k a_i n^i \in \mathbb{R}[x]$ be a polynomial of degree k with nonnegative coefficients.

- (1) Show that $P(n) = \mathcal{O}(n^k)$.
- (2) Show that $P(n) = \Omega(n^k)$.
- (3) Conclude that $P(n) = \Theta(n^k)$.

Solution. Note that (3) follows from (1) and (2) by definition, so it suffices to prove those parts of the exercise.

- (1) We want to find constants $c, n_0 > 0$ such that for all $n \geq n_0$ we have:

$$P(n) \leq c \cdot n^k.$$

Since the coefficients of $P(n)$ are nonnegative we notice that for any $n \in \mathbb{N}$ and $0 \leq i \leq k$ we have $n^i \leq n^k$ and thus:

$$P(n) = \sum_{i=0}^k a_i n^i \leq \sum_{i=0}^k a_i n^k = \left(\sum_{i=0}^k a_i \right) n^k,$$

and therefore the inequality holds for $c := \sum_{i=0}^k a_i$ and $n_0 := 1$, which completes the proof.

- (2) We want to find constants $c, n_0 > 0$ such that for all $n \geq n_0$ we have:

$$P(n) \geq c \cdot n^k.$$

Since the coefficients of the polynomial are nonnegative we get that:

$$P(n) = \sum_{i=0}^k a_i n^i \geq a_k n^k,$$

and thus the inequality holds for $n_0 := 1$ and $c := a_k$, which completes the proof.

Exercise 1.2. Show that for any constant $k \geq 2$ we have:

$$\sum_{i=0}^n k^i = 1 + k + \cdots + k^n = \mathcal{O}(k^n).$$

Solution. We want to find constants $c, n_0 > 0$ such that for all $n \geq n_0$ we have:

$$\sum_{i=0}^n k^i \leq c \cdot n^k.$$

By the formula for the sum of a finite geometric series we get that:

$$\sum_{i=0}^n k^i = \frac{k^{n+1} - 1}{k - 1}.$$

Since $k \geq 2$ we have $k - 1 \geq \frac{1}{2}k$ and thus:

$$\sum_{i=0}^n k^i = \frac{k^{n+1} - 1}{k - 1} \leq \frac{k^{n+1} - 1}{\frac{1}{2}k} = 2 \cdot k^n,$$

so the inequality is satisfied for $n_0 = 1$ and $c = 2$ which completes the proof.

Exercise 1.3. Show that $\log n! = \Theta(n \log n)$.

Solution. We have that:

$$\log(n!) = \log(1 \cdot 2 \cdots n) = \log 1 + \log 2 + \cdots + \log n \leq n \log n,$$

hence $\log n! \in \mathcal{O}(n \log n)$. We also have that:

$$\begin{aligned} \log n! &= \log 1 + \log 2 + \cdots + \log n \geq \log \frac{n}{2} + \cdots + \log n \geq \frac{n}{2} \log \frac{n}{2} = \frac{n}{2} (\log n - \log 2) \\ &= \frac{n}{2} \log n - \frac{n}{2} = \frac{n}{4} \log n + \frac{n}{4} \log n + \frac{n}{4} \cdot 2 = \frac{n}{4} \log n + \frac{n}{4} (\log n - 2) \geq \frac{n}{4} \log n, \end{aligned}$$

where the last transition follows from the fact $\log n - 2$ is positive from $n \geq 4$. Hence $\log n! = \Omega(n \log n)$. This completes the exercise.

Exercise 1.4. Suppose that $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2(n) = \mathcal{O}(g_2(n))$. We know that $f_1(n) + f_2(n) = \mathcal{O}(g_1(n) + g_2(n)) = \mathcal{O}(\max(g_1(n), g_2(n)))$ and that $f_1(n) \cdot f_2(n) = \mathcal{O}(g_1(n) \cdot g_2(n))$ (it is also not too hard to verify this). Do we necessarily also have $f_1(n)^{f_2(n)} = \mathcal{O}(g_1(n)^{g_2(n)})$?

Solution. No. Suppose that $f_1(n) = g_1(n) = n$ and $f_2(n) = 2$ and $g_2(n) = 1$. Then clearly have that $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2(n) = \mathcal{O}(g_2(n))$, but n is not asymptotically bounded by n^2 . In other words n^2 is not $\mathcal{O}(n)$. We will show this in Exercise 1.5.

Exercise 1.5. Show that n^2 is not $\mathcal{O}(n)$.

Solution. Assume, by way of contradiction that exist $c > 0$ and $n_0 \geq 1$ so that for all $n \geq n_0$ we have $n^2 \leq cn$. We have that $f(n) = n^2 - cn$ converges to ∞ as $n \rightarrow \infty$, which is a contradiction.

Exercise 1.6. Compute the time complexity of the following program:

```

1  scanf ("%d",&n);
2  k = 2;
3
4  while (k < n) {
5      k = k * k * k;
6  }
```

Solution. The cost of the operations outside the loop is $\mathcal{O}(1)$. The cost of each iteration of the loop is $\mathcal{O}(1)$. Hence the function is $\mathcal{O}(\# \text{iteration count})$. We have that in the first iteration $2 \rightarrow 2^3$, in the second iteration $(2^3)^3 = 2^{3^2}$, and we can show easily that in the i th iteration the value would be 2^{3^i} . Hence the iteration count is $\log_3 \log_2 n$ and so the complexity of the program is $\mathcal{O}(\log \log n)$ (remember that in asymptotic complexity the base of the log is irrelevant).

Exercise 1.7. Show that for all $\epsilon > 0$ we have $\log n = o(n^\epsilon)$.

Solution. Sometimes the notations o and ω can be confusing. In order to remember them better you may want to read $\log n = o(n^\epsilon)$ as “ $\log n$ is asymptotically negligible to n^ϵ ” until you remember the notation well. Remember, negligible means that the limit of the quotient is zero and indeed:

$$\lim_{n \rightarrow \infty} \frac{\log n}{n^\epsilon} = 0.$$

This result is well known from calculus 1, and is just an application of L'Hôpital's rule.

Exercise 1.8. Let $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$. Define the notation $f(n) = S(g(n))$ whenever $f(n) = \mathcal{O}(\sum_{i=1}^n g(i))$. Prove or disprove the following:

- (1) $n^2 = S(n)$.
- (2) $n = S(\log n)$.
- (3) If $f(n) = S(g(n))$ and $g(n) = S(h(n))$ then $f(n) = S(h(n))$.

Solution.

- (1) The statement is **true**. We have that $\sum_{i=1}^n i = \frac{1}{2}(n^2 + n)$, and it is clear that $n^2 = \mathcal{O}(\frac{1}{2}(n^2 + n))$.
- (2) The statement is **true**. We have that $\sum_{i=1}^n \log i = \log n! = \Theta(n \log n)$, hence $n = \mathcal{O}(\log n!)$.
- (3) The statement is **false**. Denote $f(n) = n^2$, $g(n) = n$, and $h(n) = \log n$. By (1) and (2) the conditions of the statement are satisfied. However we have that $\sum_{i=1}^n \log i = \log n! = \Theta(n \log n)$, but $n^2 = \omega(n \log n)$ and in thus n^2 is not $\mathcal{O}(n \log n)$.

Exercise 1.9 (Winter 17-18, A, 1). Let $f: \mathbb{N} \rightarrow \mathbb{R}^+$, and suppose $f(x) = f(\lceil x \rceil)$ for $x \notin \mathbb{N}$. Prove or disprove the following:

- (1) If $n = \mathcal{O}(f(n))$ then for all $k \in \mathbb{N}$ we have $n = \mathcal{O}(f(n^k))$.
- (2) If $n = \mathcal{O}(f(n))$ then for all $k \in \mathbb{N}$ we have $n = \mathcal{O}(f^k(n))$.
- (3) If $f(n) = \omega(1)$ then for all $k \in \mathbb{N}$ we have $f(n^{k+1}) - f(n^k) = \Theta(f(n^{k+1}))$.

Solution.

- (1) The statement is **true**. By definition exist $c, n_0 > 0$ so that for all $n > n_0$ we have $n \leq c \cdot f(n)$, so since $n \leq n^k$ holds for any $n \in \mathbb{N}$ we also have $n \leq n^k \leq c \cdot f(n^k)$ so indeed $n = \mathcal{O}(f(n^k))$.
- (2) The statement is **true**. We can show this by (strong) induction on k . The base case $k = 1$ is trivial, so if $n = \mathcal{O}(f^k(n))$ we need to show $n = \mathcal{O}(f(f^k(n)))$. By induction hypothesis there exist $c_k, n_k > 0$ so that for $n > n_k$ we have $n \leq c_k \cdot f^k(n)$. By this and $n = \mathcal{O}(f(n))$ we get that for $n > n_1, n_k$:

$$n \leq c_k \cdot f^k(n) \leq c_k \cdot c_1 \cdot f(f^k(n)),$$

hence for $n_{k+1} = \max(n_1, n_k)$ and $c_{k+1} = c_k \cdot c_1$ we get that for $n > n_{k+1}$ we have $n \leq c_{k+1} \cdot f^{k+1}(n)$ so $n = \mathcal{O}(f(f^k(n)))$ which completes the proof.

- (3) The statement is **false**. Take $f(n) = \log \log n$ (when we see exponents in input it is worth checking logarithmic functions). We see that $f(n) = \omega(1)$ since $\log \log n \rightarrow \infty$ as $n \rightarrow \infty$. For $k = 1$ the statement is false since by logarithm properties:

$$\begin{aligned} f(n^{k+1}) - f(n^k) &= \log((k+1) \log n) - \log(k \log n) \\ &= (\log(k+1) + \log n) - (\log k + \log n) = \log \frac{k+1}{k} = \log 2, \end{aligned}$$

and $\log 2$ is not $\Theta(2 \log \log n)$.

1.4 Recursive equations

Definition 1.10 (Recursive function). A recursive function is a function that is defined using its values on smaller values.

Example 1.14. We can define:

$$T(n) := \begin{cases} \Theta(1) & n = 1, \\ 2 \cdot T(n/2) + \Theta(n) & n > 1. \end{cases}$$

This is not a function in the usual sense, since $\Theta(1)$ is a family of functions, but we simply interpret these as any function of that family. With this interpretation, $T(n)$ can be thought of as a recursive function. One example of a known algorithm with running-time described by this $T(n)$ is *merge sort*.

Example 1.15. Consider the following algorithm:

```

1 Max(h, t)
2   if (h == t) return A[t];
3   else {
4       m = (h + t) / 2;
5       M1 = Max(h, m - 1);
6       M2 = Max(m, t);
7       return (M1 > M2) ? M1 : M2;
8   }
```

We want to find a recursive equation describing its running-time, and it is not too hard to see that it is:

$$T(n) := \begin{cases} c_1 & n = 1, \\ 2 \cdot T(n/2) + c_2 & n > 1. \end{cases}$$

for some constants $c_1, c_2 \geq 1$, where $n = t - h + 1$. In what follows we will introduce methods to help us finding the complexity of recursive algorithm by analyzing recursive functions like $T(n)$.

Definition 1.11 (The inductive method). The inductive method is simply induction. Once we have a guess for $f(n)$ so that $T(n) = \mathcal{O}(f(n))$, we can use induction to prove this.

Exercise 1.10. Define:

$$T(n) := \begin{cases} 1 & n = 1, \\ 2 \cdot T(\lfloor n/2 \rfloor) + n & n > 1. \end{cases}$$

Use the induction method to show that $T(n) = \mathcal{O}(n \log n)$.

Solution. In the claim is true for $\lfloor n/2 \rfloor$ we can see that:

$$T(n) = 2 \cdot T(\lfloor n/2 \rfloor) + n \leq 2 \left(c \cdot \left\lfloor \frac{n}{2} \right\rfloor \log(\lfloor n/2 \rfloor) \right) + n \leq 2 \left(c \cdot \frac{n}{2} \log \frac{n}{2} \right) + n = \mathcal{O}(n \log n),$$

but does this complete the proof? We must show the base case and for $n = 1$ the claim fails. However we only need to show the claim holds starting from some point, for example from $n = 2$, and indeed it holds for $n = 2$. But does this complete the proof? It turns out that not.

Question. Why does the solution for Exercise 1.10 does not work?

Answer. In this case, the induction only works when we show the claim holds for all numbers larger than 2. In our case, we have shown that it works for $n = 2$, and by the induction step, it also works for all numbers in the sequence: 2, 4, 5, 8, 9, ... (because we have shown that the claim holds for n only if it holds for $\lfloor n/2 \rfloor$). In order to cover all cases, we can notice that if we apply $\lfloor n/2 \rfloor$ on any $n \geq 2$ sufficiently many times it would eventually be equal to either 2 or 3. Hence if in our base case we show that the claim holds for $n = 2$ and $n = 3$ the proof will be complete. Indeed the claim holds for $n = 2$ and $n = 3$ so the proof is complete.

Definition 1.12 (Iterations method). The iterations method is again, not very much of a method. We simply iterate the recursion until we reach the base case.

Exercise 1.11. Find the time complexity of $T(n) = n + 3T(\lfloor n/4 \rfloor)$ with $\mathcal{O}(1)$ base cases.

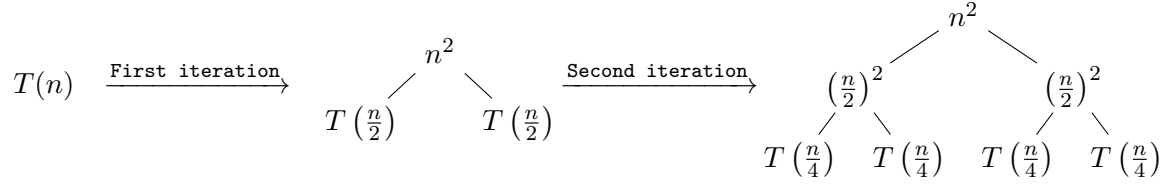
Solution. We notice that we have to iterate $\log_4 n$ times until we reach the base case. We have that:

$$\begin{aligned} T(n) &= n + 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) = n + 3 \left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\left\lfloor \frac{n}{16} \right\rfloor\right) \right) = n + 3 \left\lfloor \frac{n}{4} \right\rfloor + 9 \left\lfloor \frac{n}{16} \right\rfloor + 27T\left(\left\lfloor \frac{n}{64} \right\rfloor\right) \\ &\leq n + \frac{3}{4}n + \frac{9}{16}n + \cdot 3^i \cdot T\left(\left\lfloor \frac{n}{4^i} \right\rfloor\right) \leq n \underbrace{\sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i}_{=4n} + 3^{\log_4 n} \cdot \Theta(1) \\ &= 4n + 3^{\log_4 n} \cdot \Theta(1) = 4n + n^{\log_4 3} \cdot \Theta(1) = \mathcal{O}(n), \end{aligned}$$

where the second to last transition follows from log properties, and the last transition follows from $\log_4 3 < 1$.

Exercise 1.12 (Recursion trees one). Find an asymptotic bound for $T(n) = 2T(n/2) + n^2$ with $T(1) = 1$.

Solution. One nice visualization of the iterations method is recursive trees. Consider the following:



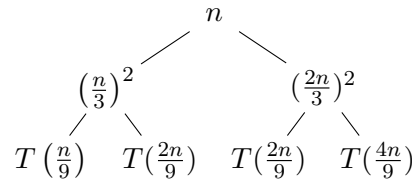
The sum of all nodes of each of these tree is $T(n)$. Hence:

$$T(n) \leq n^2 + 2 \left(\frac{n}{2}\right)^2 + 4 \left(\frac{n}{4}\right)^2 + \cdots = n^2 \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) = 2n^2,$$

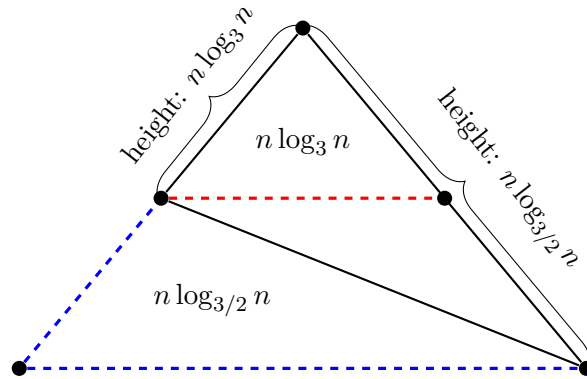
hence $T(n) = \mathcal{O}(n^2)$.

Exercise 1.13. Find a tight bound for a recursive algorithm given by $T(n) = n + T(n/3) + T(2n/3)$.

Solution. In this case the recursion tree we be unbalanced as follows:



The sum of nodes in this tree (and trees describing the next iterations) is again $T(n)$. This tree is not a complete binary tree (unbalanced), so the computations could seem harder. However, note that each full level in this tree sums to n , and that it is bounded between the two red and blue full trees as follows:



Hence we have that:

$$n \log_3 n \leq T(n) \leq n \log_{3/2} n,$$

and so $T(n) = \Theta(n \log n)$.

Definition 1.13 (Master theorem). The master theorem is a strong theorem in analysis of algorithms which goes as follows: Let $a \geq 1$ and $b > 1$ be real constants, let $f(n)$ be a function, and let $T(n) = a \cdot T(n/b) + f(n)$. Then:

- (1) If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$.

- (2) If $f(n) = \mathcal{O}(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.
- (3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some $c < 1$ eventually then $T(n) = \Theta(f(n))$.

Remark 1.18. Although the master theorem is very useful in practice, we will not prove it and you do not have to remember it since it is always available on the formula sheet.

Definition 1.14 (Variable change). Unlike what you have seen in calculus, this method is a lot less complex. We simply change variables in order to do the computations, then change them again to get the result in the original variables.

Exercise 1.14. Find a tight asymptotic bound for the following recursive equation:

$$\begin{cases} 1 & n = 1, \\ 2 \cdot T(\sqrt{n}) + (\log n - 1) & n > 1. \end{cases}$$

Solution. In order to get rid of the log denote $m = \log n$ so that the formula becomes:

$$T(2^m) = 2 \cdot T(2^{m/2}) + (m - 1),$$

and so if we define $S(m) = T(2^m)$ we see that the formula becomes:

$$S(m) = 2 \cdot S\left(\frac{m}{2}\right) + m - 1.$$

In Exercise 1.10 we saw that $S(m) = \Theta(m \log m)$ hence:

$$T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log n \cdot \log \log n).$$

Exercise 1.15. Find a tight asymptotic bound for the following recursive equation:

$$\begin{cases} 1 & n = 1, \\ 2\sqrt{n} \cdot T(\sqrt{n}) + n(\log n - 1) & n > 1. \end{cases}$$

Solution. These coefficients are terrible. Let us first divide the equation by n so that:

$$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + \log n - 1.$$

Next, notice that if we define $U(n) = \frac{T(n)}{n}$ then the equation becomes:

$$U(n) = 2 \cdot U(\sqrt{n}) + \log n - 1.$$

In Exercise 1.14 we saw that $U(n) = \Theta(\log n \cdot \log \log n)$ and hence:

$$T(n) = n \cdot U(n) = n \cdot \Theta(\log n \cdot \log \log n) = \Theta(n \cdot \log n \cdot \log \log n).$$

Exercise 1.16 (Spring 14, B, 1). Let $f_1(n), f_2(n), g_1(n), g_2(n)$ be positive and monotonically increasing functions. Prove or disprove the following claims:

- (1) If $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2(n) = \mathcal{O}(g_2(n))$ then $f_1(f_2(n)) = \mathcal{O}(g_1(g_2(n)))$.
- (2) Find a tight asymptotic bound for the recursive equation $T(n) = 3T(\lfloor \sqrt[4]{n} \rfloor) + \log^2 n$ with the initial condition $T(1) = 1$.

Solution.

- (1) Let us try $f_1(n) = g_1(n) = 2^n$. In this case we need $f_2(n) = \mathcal{O}(g_2(n))$ so that $2^{f_2(n)}$ is not $\mathcal{O}(2^{g_2(n)})$. If we choose $f_2(n) = \log(n^2)$ and $g_2(n) = \log(n)$ then since $\log(n^2) = 2 \log n$ clearly $f_2(n) = \mathcal{O}(g_2(n))$, but we have $2^{f_1(n)} = n^2$ which is not $\mathcal{O}(2^{g_1(n)}) = \mathcal{O}(n)$.
- (2) Clearly we need to do a change of variables so denote $n = 2^m$ (so that $m = \log n$) and see that:

$$T(2^m) = 3 \cdot T\left(\left\lfloor \sqrt[4]{2^m} \right\rfloor\right) + (\log(2^m))^2 = 3 \cdot T\left(\left\lfloor 2^{m/4} \right\rfloor\right) + m^2,$$

so for $S(m) = T(2^m)$ we get:

$$S(m) = 3 \cdot S(m/4) + m^2,$$

and this is a standard recursive equation with $a = 3$, $b = 4$, $f(m) = m^2$. We can see that the conditions for the master theorem are satisfied and hence $S(m) = \Theta(m^2)$ so $T(2^m) = \Theta(m^2)$ and thus $T(n) = \Theta(\log^2 n)$.

2 Arrays, matrices, sparse matrices and linked lists

2.1 Arrays

Example 2.1 (Array). An array is defined by the following operations:

`create(type, I)` Returns an array of elements of type `type` with the index set `I`.
`store(A, i, e)` Stores `e` of type `type` with index `i` in the array. In code, this usually looks like `A[i] = e`.
`get(A, i)` Returns the element at index `i` in the array. This usually looks like `A[i]`.

Remark 2.1. All the values in the array must be of the same type. Notice that we didn't define what happens if we try to store an element of a different type, or what to return when we use `get` on an uninitialized value. In what follows, we will give a few trivial implementations of the array.

Example 2.2 (Continuous memory block). This is the most trivial implementation of an array. In this implementation, we have

- The functions `store` and `get` in $\mathcal{O}(1)$
- The time it takes to initialize the array depends on the time it takes for the machine to allocate the necessary memory, and also on whether we want to initialize it with a certain default value.

Example 2.3 (Multiple continuous memory block of constant size). In order to implement an array of size m , we allocate $k = \lceil m/n \rceil$ blocks of memory. In this way, we only need to keep a pointer array of size $k = \lceil m/n \rceil$. It will hold a pointer to the start of each of the k blocks. In order to access an element in this kind of array, we can use basic arithmetics. For example, if $n = 6$ and $i = 16$ the address of $A[16]$ is

$$\text{base} \left(\left\lfloor \frac{16}{6} \right\rfloor \right) + 16 \% 6 = \text{base}(2) + 4,$$

where `base` is the pointer array. The address computation time is still $\mathcal{O}(1)$.

Example 2.4 (2d-array). In a 2d-array $A[i][j]$ of size $m \times n$ each row $A[i]$ is a 1d-array of size n . In order to find the element $A[i][j]$ we compute

$$\text{base} + i \cdot n + j,$$

where `base` is pointer to $A[0][0]$.

Example 2.5 (n -dimensional array). An n -dimensional array $A[n_d] \dots [n_2][n_1]$ is very similar to a 2-dimensional array. In order to find the address of $A[i_d] \dots [i_1]$ we compute

$$\text{base} + i_d \cdot n_{d-1} \cdots n_1 + i_{d-1} n_{d-2} \cdots n_1 + \cdots + i_1.$$

The number of operations in this calculations is approximately $1 + 2 + \cdots + n = \mathcal{O}(n^2)$.

Question. Is there a way to optimize this calculation?

Answer. Yes! using Horner's method we can do the same computation with a linear amount of operations. As you may expect, it is not a very interesting method, it is simply a formula for getting the same number using a different amount of operations. However, it is very efficient, so even if you don't remember its implementation you should remember that it exists. It also allows computations of polynomials using a linear amount of operations.

Remark 2.2 (Locality). In general, when scanning the elements of an n -dimensional array it is better to scan them by rows and not by columns because of *locality*. It is “easier” for the computer to access memory that is close to the previously accessed memory. Reading memory is more expensive than calculating an operation in the GPU, so this is very important. However, it is not a part of this course’s syllabus, so we won’t get into it.

2.2 Initializing an array in $\mathcal{O}(1)$

With the trivial implementations presented in the previous subsection, initializing an array of size n takes $\mathcal{O}(n)$ operations. We want to be able to initialize an array (give all entries a default value) in $\mathcal{O}(1)$. For this speed, we will have to use more memory, and also a slightly more tricky implementation.

Example 2.6 (Problematic implementation). In addition to our main array, we will also allocate an auxiliary array of size n . Each time we store a value in index i , we add index i to the top of the auxiliary array. In this way, before trying to access to array, we check if index i is in the auxiliary array, and if not, return 0 (if we wanted to initialize the array to 0).

This implementation is problematic for multiple reasons, for example, when accessing the element in index i , we first need to scan the auxiliary array which takes $\mathcal{O}(n)$ operations.

Example 2.7 (True implementation). This time, we will hold three arrays. An array for values, a shortcuts array, and an array for indices with “defined” value which we denote A, B, and C respectively. There isn’t too much intuition behind this apart from what previously stated. Here is a simple implementation of this method in pseudocode resembling c:

```
1 int is_assigned(int i) {
2     return (B[i] < top && B[i] >= 0 && C[B[i]] == i);
3 }
```

```
1 void init(int V[], int d) {
2     top = 0;
3     constant = d;
4 }
```

```
1 int get(int V[], int i) {
2     if (is_assigned(i))
3         return A[i];
4     else
5         return constant;
6 }
```

```
1 void store(int V[], int i, int e) {
2     if (!is_assigned(i)) {
3         C[top] = i;
4         B[i] = top;
5         top = top + 1;
6     }
7     A[i] = e;
8 }
```

Remark 2.3. After $\text{top} \geq n$ we don’t have to use arrays B or C anymore. The advantages of this method are:

- Asymptotically this is a great solution.
- We optimized the initialization process dramatically, especially for large arrays.

The disadvantages are:

- This method takes three times more space than the trivial method.
- The fast initialization time is later backstabbed by the new implementations of `get` and `store` (we will learn more about this when we learn about amortized analysis).

2.3 Matrices

Example 2.8 (Symmetric matrices). Suppose we have a symmetric matrix:

$$\begin{pmatrix} 1 & 2 & 4 & 7 & 12 \\ 2 & 3 & 5 & 8 & 13 \\ 4 & 5 & 6 & 9 & 14 \\ 7 & 8 & 9 & 10 & 15 \\ 12 & 13 & 14 & 15 & 16 \end{pmatrix}.$$

Instead of allocating memory for each element in the matrix, we can just allocate a single continuous memory block containing all the elements under (and including) its main diagonal as follows:

$$[1, 2, 3, 4, \dots, 16].$$

In order to calculate the address of the element $A[i, j]$ we can use the formula

$$\sum_{i=1}^N i = \frac{N(N+1)}{2},$$

and see that

$$\text{Addr}(i, j) = \begin{cases} \text{base} + \left\lfloor \frac{i(i+1)}{2} \right\rfloor + j & i \geq j, \\ \text{Addr}(i, j) & \text{otherwise.} \end{cases}$$

Example 2.9 (Sparse matrices). A sparse matrix is a matrix where “most elements” are 0 (or some other constant). For example, the identity matrices I_n are considered sparse matrices.

Definition 2.1 (Sparse matrix). We do not need a formal definition for a sparse matrix in this course, but usually in the context of a single sparse matrix, we say that M of order $n \times n$ is sparse if the amount of nonzero elements is approximately n .

Definition 2.2 (Sequence of sparse matrices). A sequence of matrices $\{M_n\}_{n=1}^{\infty}$ is said to be a sequence of sparse matrices if $v(n) \in o(m(n))$, where $v(n)$ is the amount of nonzero elements in M_n , and $m(n)$ is the total amount of elements in M_n , and $m(n) \xrightarrow{n \rightarrow \infty} \infty$.

Example 2.10. Any increasing sequence of diagonal matrices is a sparse sequence matrices. For example the sequence of identity matrices $M_n = I_n$ satisfies $m(n) = n^2$ and $v(n) = n$ so since $n \in o(n^2)$ we conclude that indeed $\{I_n\}_{n=1}^{\infty}$ is a sparse sequence.

Remark 2.4. In sparse matrices of order $n \times n$ the number of elements different than zero (or the chosen constant) is $o(n^2)$. From now on we will always assume the constant is zero.

One way to represent these matrices is by a lexicographical list of triples $(i, j, A_{i,j})$ that represent the nonzero values in the matrix. On one hand, we lose the ability to call `get` in constant time. On the other hand, in this way allocation is way more efficient (requires a lot less memory). Addition and multiplication of sparse matrices is also way more efficient.

Example 2.11 (Addition of sparse matrices). Addition in usual matrices of order $n \times n$ takes $\Theta(n^2)$ time (note this time n is not the size of the input, we are not always strict with our conventions). When summing list based sparse matrices, the sum takes the same time complexity as merging the two lists. Suppose that the lists are of sizes m and k , then the time complexity of addition is now only $\mathcal{O}(m + k)$, and since the matrices are sparse it is $o(n^2)$.

Exercise 2.1. Show that matrix multiplication with the list based implementation takes $o(n^3)$ time instead of $\mathcal{O}(n^3)$ (yes, we broke the vow to compute complexity with the real length of input yet again).

2.4 Linked lists

Recall that a linked list is a data type that allows dynamic allocation, which means that there is no theoretical limit for the amount of elements in a list. However, we lose the access to elements in constant time, because we need to scan them one by one. Here is a simple implementation of a list in c:

```
1 void init(NODE *head){
2     *head = NULL;
3 }
```

```
1 NODE* find(DATA_TYPE x, NODE *head) {
2     NODE* t;
3     t = head;
4     while (t != NULL && t -> info != x)
5         t = t -> next;
6     return t;
7 }
```

```
1 int insert(NODE *t, DATA_TYPE x) {
2     NODE *p;
3     if (t == NULL) return 0;
4     p = (NODE *) malloc (sizeof (NODE));
5     p -> info = x;
6     p -> next = t -> next;
7     t -> next = p;
8     return 1;
9 }
```

```
1 void delete(NODE *t) {
2     NODE *temp;
3     temp = t -> next;
4     t -> next = temp -> next;
5     free(temp);
6 }
```

```
1 int delete_first(NODE **p_head){
2     NODE *temp;
3     if(*p_head == NULL) return 0;
4     temp = *p_head;
5     *p_head = temp -> next;
6     free(temp);
7 }
```

```

7 |   return 1;           /* success code */
8 | }

```

Remark 2.5. There are different types of linked lists. We will now proceed to introduce the more important ones.

Example 2.12 (Dummy node linked list). A dummy node linked list is just like a standard linked list, with an additional node at the beginning without any meaningful value. This additional dummy node allows us to get rid of the `delete_first` function and not deal with pathological cases like empty lists anymore, without impairing the standard functionalities of the linked list.

Example 2.13 (Circular linked list). A Circular linked list is a linked list where the last element is pointing to the head node. The advantages of a circular linked list are:

- We can reach any element from any other element.
- We can concatenate lists in constant time, because we have constant time access to the pointer of the last node.

Example 2.14 (Doubly linked list). In a doubly linked list, each node has not only a pointer to the next node, but also another pointer to the previous node. The advantages of a doubly linked list are:

- We can reach any element from any element.
- Given access to a node t we can delete it in constant time, because we have constant time access to the node before it.

Question. Given a pointer t to a node in a linked list, can we still remove it in constant time even if the list is not doubly linked (for example, a standard linked list)?

Answer. Yes! yes we can! Refer to the node following N by M . We just copy the value of M to N and then proceed to delete M . This method indeed works and essentially removes N in constant time. Unfortunately, there are problems with this method:

- We can't delete the last node.
- There might be more pointers to M and we may not be allowed to modify them.
- There might be more pointers to N and we may not be allowed to modify them.
- The nodes could potentially hold a lot of information, and copying M to N may take a long time.

2.5 Sparse polynomials and matrices

Definition 2.3 (Sparse polynomial). A polynomial

$$p(x) = a_n x^n + \cdots + a_1 x + a_0 \in F[x]$$

is called sparse if most of its coefficients are 0 (or some other constant). This definition is not very formal, but formally, it should be very similar to the definition of a sparse matrix.

Remark 2.6. Instead of representing sparse polynomials in an array of size of the degree of the polynomial, we can represent it efficiently by only storing the nonzero coefficients in a linked list. We can also use linked lists to represent multivariate polynomials, for example polynomials in x and y , which we consider as polynomials in x with coefficients which are polynomials in y . We do this by letting each node store both the power of x associated with the nonzero coefficient, and also store a pointer to another linked list which represents the polynomial in y that is the coefficient that the original node represents.

Example 2.15 (Sparse matrix node). When we represent sparse matrices our nodes will only need to store the values of the nonzero entry, its row and column, and pointers to the next nodes on the row and column it is located on. In pseudocode, the structure of the node looks something like this:

```

1 typedef struct node {
2     float value;
3     struct node* next, *down;
4     int row, column;
5 } NODE;
```

Example 2.16 (Sparse matrix multiplication). Here is an efficient implementation for multiplication of sparse matrices in pseudocode:

```

1 #define N_row 20
2 #define N_col 20
3 NODE *row_header[N_row],
4     *col_header[N_col];
5 /* Compute the value of c[i][j] */
6 float mult_row_col(int i, int j) {
7     float c = 0;
8     NODE *r = row_header[i],
9         *k = col_header[j];
10    while (r != NULL && k != NULL){
11        if (r -> column < k -> row) r = r -> next;
12        else if (r -> column > k -> row) k = k -> down;
13        else /* r->column = k->row */
14            c += r -> value * k -> value;
15            r = r -> next; k = k -> down;
16    }
17    return c;
18 }
```

2.6 Exercises

Exercise 2.2 (Matrix of diagonals). A matrix of diagonals is a square matrix $A \in \mathbb{R}^{n \times n}$ where all diagonals have the same value. For example, the following matrix is a matrix of diagonals:

$$A = \begin{pmatrix} 1 & 5 & 8 & 4 \\ 7 & 1 & 5 & 8 \\ 5 & 7 & 1 & 5 \\ 6 & 5 & 7 & 1 \end{pmatrix}.$$

Describe a data structure which supports the following operations:

- get(i,j)** Returns element in coordinates (i,j).
put(i,j,x) Updates the element in coordinates (i,j) and all elements in its diagonal to the value x.

Solution. The trivial solution is to simply use the standard matrix implementation. In this case **get** would be $\mathcal{O}(1)$, **put** would be $\mathcal{O}(n)$, and the space complexity would be $\mathcal{O}(n^2)$.

However, notice that we only need one element of each diagonal to maintain the described array, so instead of storing the entire matrix we will only store the first row and column:

$$\begin{pmatrix} 1 & 5 & 8 & 4 \\ 7 & 1 & 5 & 8 \\ 5 & 7 & 1 & 5 \\ 6 & 5 & 7 & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 5 & 8 & 4 \\ 7 & & & \\ 5 & & & \\ 6 & & & \end{pmatrix} \longrightarrow \begin{array}{|c|c|c|c|c|c|c|} \hline 6 & 5 & 7 & 1 & 5 & 8 & 4 \\ \hline \end{array}$$

In this way we can clearly implement **get** and **put** in $\mathcal{O}(1)$ and the space complexity drops to $\mathcal{O}(1)$.

Exercise 2.3 (Tridiagonal matrix). A tridiagonal matrix is a square matrix $A \in \mathbb{R}^{n \times n}$ where all entries store a constant value c except the three longest diagonals that can store any value. For example the following is a tridiagonal matrix:

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 0 \\ 5 & 4 & 1 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 7 & 9 \end{pmatrix}$$

Describe a data structure which supports the following operations:

- get(i,j)** Returns element in coordinates (i,j).
put(i,j,x) Updates the element in coordinates (i,j) the value x unless it is not on the three longest diagonals, in which case do nothing.

Solution. The trivial solution is to simply use the standard matrix implementation. In this case **get** would be $\mathcal{O}(1)$, **put** would be $\mathcal{O}(1)$, and the space complexity would be $\mathcal{O}(n^2)$.

However, note that we could implement this using an array that stores the three longest diagonals and the constant stored in the rest of the matrix, and thus improve the implementation to $\mathcal{O}(n)$ space complexity.

Exercise 2.4 (Sparse matrix). A sparse matrix is a rectangular matrix $A \in \mathbb{R}^{n \times m}$ so that only $r \ll n \cdot m$ are nonzero.

- get(i,j)** Returns element in coordinates (i,j).
put(i,j,x) Updates the element in coordinates (i,j) the value x unless it is not on the three longest diagonals, in which case do nothing.
transpose() Transposes the matrix.

Solution. The trivial solution is to simply use the standard matrix implementation. In this case **get** would be $\mathcal{O}(1)$, **put** would be $\mathcal{O}(1)$, the **transpose** complexity would be $\mathcal{O}(nm)$ and the space complexity would be $\mathcal{O}(nm)$.

However, notice that we could store only the nonzero elements with keys (i,j,x) which contain their position and values, ordered lexicographically by (i,j). We can then implement

get using binary search, **put** by insertion into a sorted array (can be done in linear time ¹), and implement **transpose** by holding a boolean value **isTransposed**, so that we access (i, j) when it is 0 and (j, i) if it is 1. In this implementation **get** would be $\mathcal{O}(\log r)$, **put** would be $\mathcal{O}(r)$, the **transpose** complexity would be $\mathcal{O}(1)$ and the space complexity would be $\mathcal{O}(r)$.

Exercise 2.5 (Magic matrix). A $n \times n$ matrix is called magic if the following conditions are satisfied:

- (1) The matrix only contains values from 1 to n^2 , and each value appears exactly once.
- (2) For any $1 \leq i, j \leq n$ the sum of the i th and j th rows is equal.
- (3) For any $1 \leq i, j \leq n$ the sum of the i th and j th columns is equal.

Implement a data structure that supports the following operations:

init(n) Initializes a matrix of order $n \times n$.
insert(i, j, x) In the first n^2 valid insertions, try to insert x to position (i, j) . In case that position already has a value do nothing and return **try again** (this is not a valid insertion). After n^2 valid insertions output **magic!** if the matrix is magic, and **not magic!** otherwise.

The required time complexity for all operations is $\mathcal{O}(1)$, and the space complexity is $\mathcal{O}(n^2)$.

Solution. In order to implement this structure we store an $n \times n$ matrix A , two array R and C of length n which store the current sum of the i th row/column in their i th entry respectively. Additionally note that since we know the values in the matrix must be between 1 and n^2 (otherwise we treat it is invalid input), we get that the sum of elements in the matrix is $\frac{n^2(n^2+1)}{2}$, hence the sum in each row and column must be $\frac{n^2(n^2+1)}{2n}$. Whenever we update the current sum in R and C we also check if the sum is equal to the correct value. If it is we add 1 to another value called **correctSums** (initialized to 0, expected to be $2n$), and if it were the correct value and the new update changes that, we decrement **correctSums** by 1. In order to make sure each number is inserted exactly once, we also maintain an $n \times n$ one-dimensional boolean array, initialized to 0 (remember this can be done in constant time), and once we insert a number $1 \leq k \leq n^2$ we change the $k - 1$ th entry of the array to 1. We modify the **insert** operation so that it first checks if we already inserted that value before. After n^2 valid insertions we return **magic!** if and only if the number of correct sums is $2n$.

Exercise 2.6 (Doctor appointments). In this exercise we help a doctor to organize his appointments. Implement a data structure that supports the following operations:

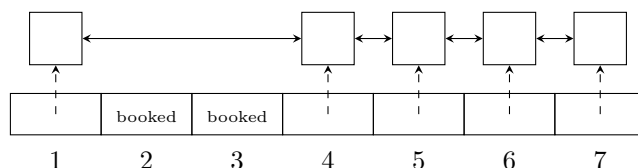
init(m) Initilize a notebook with m different possible dates for appointments, denoted by $1, \dots, m$. After initialization, all dates are available.
 Time complexity: $\mathcal{O}(m)$.
addAppointment(i) Set an appointment on date i . If i is not available, set the appointment for some other available date and return the other date.
 Time complexity: $\mathcal{O}(1)$.
cancelAppointment(i) Cancel the appointment on date i .
 Time complexity: $\mathcal{O}(1)$.

The required space complexity for the structure is $\mathcal{O}(m)$.

¹No need to worry about the size of the array since we will have better solutions after we learn about trees.

Solution. The trick in this exercise is the following observation: We can initialize an array of size n , and a doubly-linked list of size n with pointers from each entry in the array to the respective node.

The solution is not obvious from the previous observation, but it is immediate. Intuitively, the array represents the notebook, where all values are initialized to 0 meaning the date is available, and are set to 1 once we set an appointment. Additionally, the linked list represents only the available dates, and each time an appointment is booked, we delete the respective node from the list and set the entry in the array to 1. If an appointment is cancelled, we set the 1 back to 0, and insert a node for that available date at the beginning of the list (if the list is cyclic we can insert it anywhere we want but we still have to make sure it is a place to which we have constant time access).



Exercise 2.7 (Winter 18-19, A, 4, part 1). A student in the optical engineering department wants to conduct an experiment that includes a row of n incandescent light bulbs with identifies $1, 2, \dots, n$. A light bulb can either be active or inactive. Implement a data structure that supports the following operations:

- Init(x)** Initialize the structure with n light bulbs which all start inactive.
Time complexity: $\mathcal{O}(1)$.
- Activate(i)** Activate light bulb number i .
Time complexity: $\mathcal{O}(1)$.
- LongestActiveSequence()** Return the longest continuous active sequence of light bulbs.
Time complexity: $\mathcal{O}(1)$.

Solution. We would like to store the length of all active sequences somewhere and maintain the value k of the length of the longest active sequences. The precise details of the implementation are as follows:

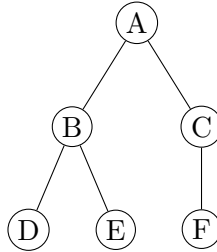
- Init(x)** Initialize an array with n elements in all set to zero in $\mathcal{O}(1)$ time as we saw in class. Additionally initialize a value $k = 0$ for the current length of the longest active sequence.
Time complexity: $\mathcal{O}(1)$.
- Activate(i)** If the i th light bulb is active do nothing. If the $i - 1$ th and $i + 1$ th light bulbs are inactive mark $i-i$ on the i th bulb to represent the limits of the active sequence it belongs to. We call these values “limit values”, the bulbs they represent “limit values”, and denote them by L_i-R_i . If i has one or more active neighbours, use the limit values marked on them in order to write the correct limit values for i , and for the limit bulbs of i ’s neighbours. After this process, the boundary bulbs of i ’s sequence have the correct boundary values, which guarantees the correctness of the next **Activate** operation. Since after this process the i belongs to has length $R_i - L_1 + 1$, set k to be the maximum of that value and its current value. Now by construction k is the length of the longest active sequence of light bulbs.
Time complexity: $\mathcal{O}(1)$.
- LongestActiveSequence()** Return k .
Time complexity: $\mathcal{O}(1)$.

We can verify rigorously the correctness of the algorithms, but since this is a course focusing on data structures many times we simply omit the proof of correctness when it is sufficiently easy to see why the algorithm works.

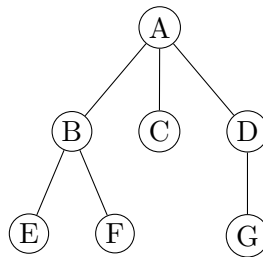
3 Trees and binary search trees

3.1 Trees

Example 3.1 (Binary Tree). Here is an example of a binary tree:



Example 3.2 (Ternary tree). Here is an example of a ternary tree:



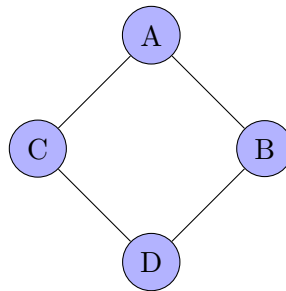
Definition 3.1 (Undirected graph). An undirected graph is an ordered pair (V, E) such that

- (1) V is a set of vertices (or nodes).
- (2) $E \subseteq V \times V$ is a set of ordered pairs of vertices called directed edges (or arcs).

The undirected edge $\{u, v\} \in E$ implies a connection between u and v , without respect to order or direction.

Remark 3.1. In the context of undirected graphs, we sometimes denote $\{u, v\} = (u, v) = (v, u)$.

Example 3.3. Here is a simple example of an undirected graph:

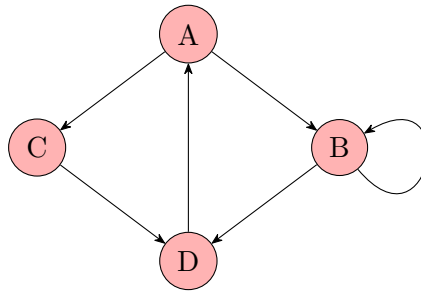


Definition 3.2 (Directed graph). A directed graph (or digraph) is an ordered pair (V, E) such that:

- (1) V is a set of vertices (or nodes).
- (2) $E \subseteq V \times V$ is a set of ordered pairs of vertices called directed edges (or arcs).

The directed edge $(u, v) \in E$ implies a connection *from* vertex u *to* vertex v , but not necessarily from v to u .

Example 3.4. Here is a simple example of a directed graph:



Remark 3.2. The following definitions (Definition 3.3 to Definition 3.10) are mostly technical and formal and you probably won't use them in this course. It is important to understand how all of them are distinct, but you don't have to learn the names by heart.

Definition 3.3 (Walk). A walk is a finite or infinite sequence of edges which joins a sequence of vertices.

Definition 3.4 (Trail). A trail is a walk in which all edges are distinct.

Definition 3.5 (Path). A path is a trail in which all vertices (and therefore also all edges) are distinct.

Definition 3.6 (Circuit). A circuit is a nonempty trail in which the first and last vertices are equal (a closed trail).

Definition 3.7 (Cycle). A cycle (or simple circuit) is a circuit in which only the first and last vertices are equal.

Remark 3.3. In math, a continuous path that does not intersect itself is called simple. Compare this with the terminology "simple circuit" for a cycle.

Definition 3.8 (Underlying graph). Given a directed graph $G = (V, E)$, its induced underlying graph is the undirected graph obtained by replacing every directed edge with an undirected edge.

Definition 3.9 (Source). A source is a vertex that has no incoming edges.

Definition 3.10 (Sink). A sink is a vertex that has no outgoing edges.

Definition 3.11 (Directed tree). A directed tree is a directed acyclic graph (DAG) whose underlying undirected graph is a tree. That is, a directed tree is a DAG such that:

- (1) There is a unique vertex called the *root* with no incoming edges.
- (2) Every other vertex has exactly one incoming edge.

Remark 3.4. The notion of a directed acyclic graph (DAG) is very important in algorithmics. Note that a directed tree cannot have undirected cycles, but a DAG can have undirected cycles.

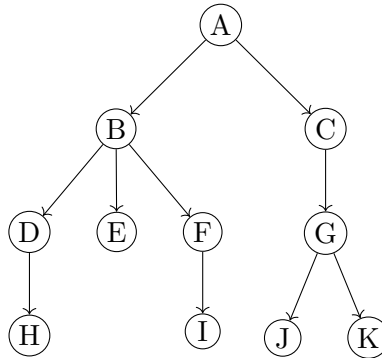
Remark 3.5. We say that v is a *child* of u if there exists an edge from u to v , and that u is a *parent* of v if there exists an edge from v to u . We also say that a vertex w is an *ancestor* of v if there is a path from w to v , and w is closer to the root, and we say that a vertex u is a *descendant* of v if v is an ancestor of u .

Definition 3.12 (Degree). The degree of v denoted $\deg(v)$ is the number of children of v .

Definition 3.13 (Leaf node). We say that v is a leaf node (or terminal vertex) if $\deg(v) = 0$.

Definition 3.14 (Internal node). A node that is not a leaf node is called an internal node.

Example 3.5. In the directed tree below, F is a child of B , and B is a parent of E . G is a descendant of A , and B is an ancestor of H . H is a leaf node, $\deg(A) = 2$, and A is an internal node.



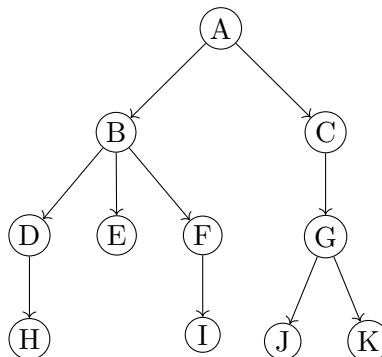
Definition 3.15 (Height of a vertex). The height of a vertex v , denoted $h(v)$, is the length of the longest path from v to a leaf in the tree. If v is a leaf, then its height is defined to be 0.

Definition 3.16 (Depth of a vertex). The depth of a vertex v , denoted $d(v)$, is the length of the unique path from the root to v . The depth of the root is defined to be 0.

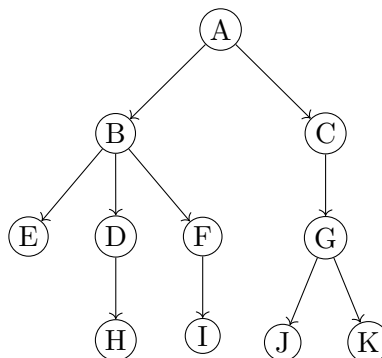
Definition 3.17 (Height of a tree). The height of a tree, denoted $h(T)$, is the height of its root, that is, the length of the longest path from the root to any leaf in the tree.

Definition 3.18 (Ordered tree). An ordered tree is a directed tree such that the order of the children matters.

Example 3.6. The tree



is different from the tree

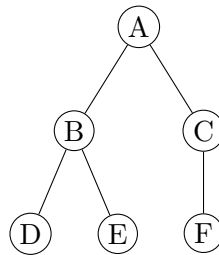


because the order of the children of B has changed.

Remark 3.6. From now on we refer to directed trees simply as trees.

Definition 3.19 (Binary Tree). A binary tree is a tree where each node has at most two children.

Example 3.7. Here is a simple example of a binary tree:



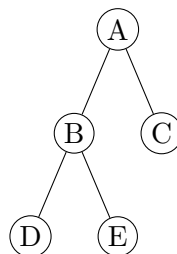
We call the children of a node in a binary tree the left child and the right child.

Definition 3.20 (Binary tree, recursive definition). An equivalent definition to a binary tree, is the following recursive definition. A binary tree is a data structures which is one of the following:

- (1) An empty tree (has no nodes).
- (2) A data structure with three parts, a node called the *root*, a left binary tree called the left subtree, and a right binary tree called the right subtree.

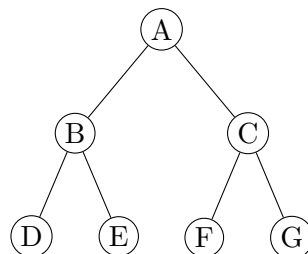
Definition 3.21 (Full Binary Tree). A full binary tree is a binary tree in which every node has precisely 0 or 2 children. In other words, a full binary tree is a binary tree where every non-leaf node has exactly 2 children.

Example 3.8. Here is an example of a full binary tree:



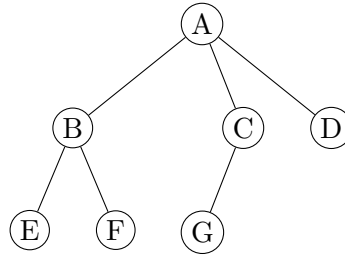
Definition 3.22 (Complete binary tree). A complete binary tree is a full binary tree in which all the leaves are of the same depth.

Example 3.9. Here is an example of a complete binary tree:



Definition 3.23 (Almost complete binary tree). An almost complete binary tree is a binary tree where all levels, except possibly the last, are fully filled, and all nodes are as far left as possible (as if we deleted leaves from the right side of the last level of a complete binary tree).

Example 3.10. Here is an example of an almost complete binary tree:

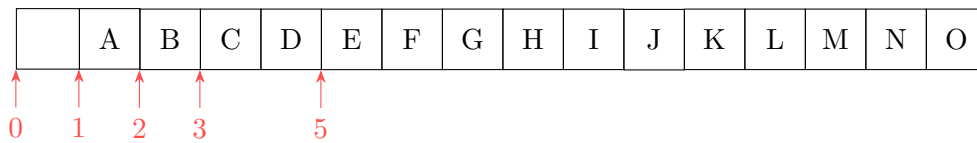


Remark 3.7. A complete binary tree T with n nodes, L leaves, and height $h(T) = h$ has the following properties:

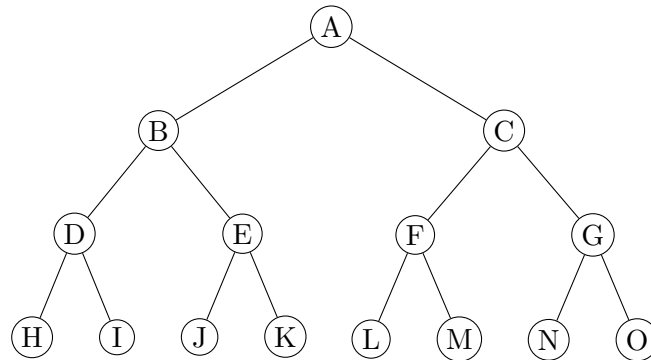
- (1) The number of nodes at level i is 2^i , where i is the index of the level, starting from 0.
- (2) The number of leaves in the tree is $L = n_h = 2^h$.
- (3) The number of nodes in the tree is $n = \sum_{i=0}^h n_i = \sum_{i=0}^h 2^i = 2^{h+1} - 1$.
- (4) The height of a complete binary tree is $h = \log_2(n + 1) - 1$.
- (5) The number of internal nodes is $n - L = 2^h - 1 = L - 1$.

Remark 3.8. In order to store complete binary trees in an array we can store the left child of the node at index i at index $2i$ and its right child at index $2i + 1$. Using this method, the father of the i th node is located at index $\lfloor i/2 \rfloor$.

Example 3.11. The following array

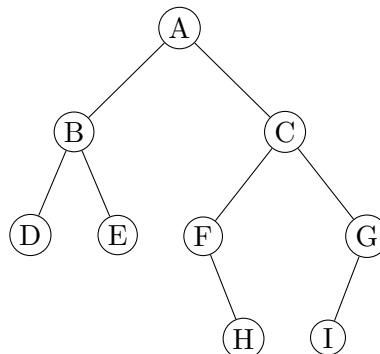


represents the following complete binary tree:



3.2 Traversals

Given a binary tree T



we can traverse across its nodes in many different ways.

Definition 3.24 (Preorder traversal). In a preorder traversal:

- (1) Visit the root node.
- (2) Visit the left subtree.
- (3) Visit the right subtree.

In this order the output of the traversal over T is A B D E C F H G I.

Definition 3.25 (Inorder traversal). In an inorder traversal:

- (1) Visit the left subtree.
- (2) Visit the root node.
- (3) Visit the right subtree.

In this order the output of the traversal over T is D B E A F H C I G.

Definition 3.26 (Postorder traversal). In a postorder traversal:

- (1) Visit the left subtree.
- (2) Visit the right subtree.
- (3) Visit the root node.

In this order the output of the traversal over T is D E B H F I G C A.

Remark 3.9. When computing the algorithm manually, it could be useful to remember that the order of the steps (1 – 3) works just like when programming recursion; we follow step (1), then follow the first step we haven't done yet in the current node (note that step (1) may change our current node or it may not), and so on and so forth.

Example 3.12 (Postorder implementation). Here is a simple implementation of a postorder traversal in c.

```

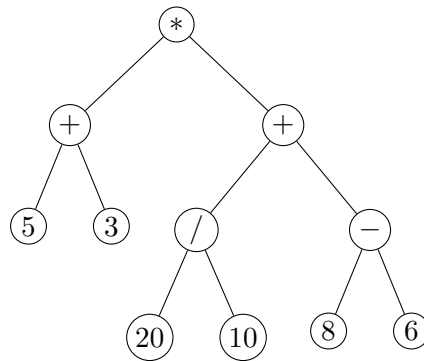
1  typedef struct node {
2      int value;
3      struct node *left, *right;
4  } NODE;
5
6  void postorder (NODE *T){
7      if (T == NULL) return;
8      postorder(T -> left);    /*1*/
9      postorder(T -> right);   /*2*/
10     'visit'(T);              /*3*/
11 }
12

```

Remark 3.10. By switching the order of (2) and (3) in Example 3.12 we would get an implementation for an inorder traversal.

Remark 3.11. Since we use recursion the space complexity is extra $\mathcal{O}(h)$, but there exists an algorithm called Morris traversal which allows all traversals (but particularly inorder) in extra $\mathcal{O}(1)$ space (without affecting time complexity).

Example 3.13 (Arithmetic expression). Suppose we have the following tree,



An inorder traversal would give us the following expression

$$((5 + 3) * ((20/10) + (8 - 6))) = 32$$

which we call an *infix expression* because the arithmetic operator comes between its operands. A postorder traversal would give us the following expression:

$$5\ 3\ +\ 20\ 10\ /\ 8\ 6\ -\ +\ * = 32$$

which we call a *postfix expression* because the arithmetic operator comes after its operands.

Remark 3.12. In order to calculate a postfix expression we can follow the following algorithm using a stack:

- (1) Start with an empty stack.
- (2) Scan the leftmost element in the expression (and remove it from the expression) and then:
 - (a) If the element is an operand (number), store it in the stack.
 - (b) If the element is an operator (operation), apply it on the two elements on the top of the stack, and store the result in the stack.
- (3) If the expression isn't empty return to step (2). If it is empty, there is a single element in the stack which is the result of the expression.

Exercise 3.1. Write a program that computes an arithmetic expression.

Exercise 3.2. Write the program from Exercise 3.1 without using recursion (use a stack).

Example 3.14 (Postfix arithmetics). Here is how to calculate an expression using recursion in postfix order:

```

1 typedef struct node {
2     int value;
3     struct node *left, *right;
4 } NODE;
5
6 void postorder (NODE *T){
7     if ("T is a leaf") return(T -> value);
8     val1 = postorder(T -> left);    /*1*/
9     val2 = postorder(T -> right);   /*2*/
10    return ("val1 operation val2"); /*3*/
11 }

```

Example 3.15 (Height function implementation). Here is an example of a program that computes the height of a tree:

```

1  int height (NODE *T) {
2      int L,R;
3      if ("T is a leaf") return 0;
4      else {
5          L = height(T -> left);
6          R = height(T -> right);
7          return 1 + max(L,R);
8      }
9  }
```

3.3 Left-child right-sibling binary trees

Definition 3.27 (Arborescence). An arborescence is a directed graph where there exists a unique node r called the *root*, so that there exists a unique directed path from v to r for each vertex v .

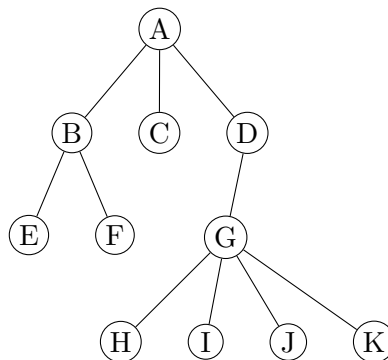
Remark 3.13. Every arborescence is a DAG, but not every DAG is an arborescence.

Remark 3.14. Collins dictionary defines arborescence as “a tree-like branching pattern”.

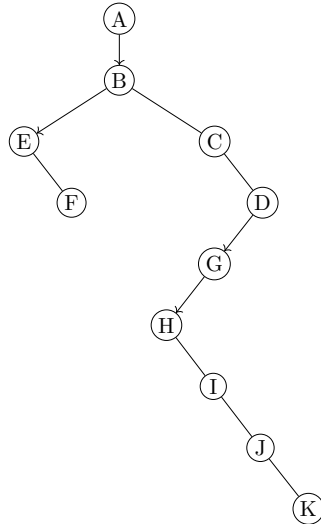
Definition 3.28 (k -ary tree). A k -ary tree is an arborescence $T = (V, E)$ such that for any $v \in V$ we have $d(v) < k$. In other words, each node has at most k children.

Example 3.16. A 2-ary tree is a binary tree. Similarly, a 3-ary tree is called a ternary tree.

Example 3.17 (LC-RS tree). Let $T = (V, E)$ be an ordered k -ary tree (so the children of each node are in a specific order), for example:



with $k = 4$. We want and we can convert this T into a binary tree (a 2-ary tree) by using a process sometimes called the Knuth transform. The generation of the tree starts by taking the root of the original tree to be the root of the binary tree. Then, starting with the root, each node's leftmost child in T is made its left child in the binary tree, and its nearest sibling to the right in the original tree is made its right child in the binary tree. Note that siblings form linked lists. Note that if a node is the rightmost child, then its right child in the binary tree is a null pointer. Similarly, if it is a leaf, then its left child in the binary tree is a null pointer. The resulting tree is called a LC-RS (left-child right-sibling) binary tree, or a doubly chained tree. If T is the 4-tree above, then its associated LC-RS tree is the tree:



Note that the $\rightarrow$ represents a parent-child relation, and the $-$ represents a siblings relation.

Remark 3.15. By converting the tree to a binary tree, we are now able to apply all algorithms related to binary trees, which is a great benefit, but it is also important to note that we had to compensate with the height of the tree. Although the tree is now binary (which is extremely useful), the height of the tree increases by a factor of k (by a factor of 4 in Example 3.17). In other words, $h_{\text{new}} \leq d \cdot h_{\text{old}}$. This makes the algorithms slower.

3.4 Search trees

Definition 3.29 (Dictionary). A dictionary is a data structure that stores pairs called key-value pairs, composed from a key and a value. We denote the collection of theoretically acceptable keys by U . A dictionary must support multiple basic operations such as:

<code>create()</code>	Returns an empty dictionary.
<code>insert(D,x,info)</code>	Adds (x, info) to the collection of pairs in D .
<code>delete(D,x)</code>	Deletes the key-value pair with key x from the dictionary.
<code>find(D,x)</code>	Access the key-value pair (x, info) . Returns <code>null</code> if no such pair exists in the dictionary.

Note that we require x to be in U and usually we also require that each x appears at most once in the dictionary.

Example 3.18. We can create a dictionary with $U = \mathbb{N}$ in order to implement a database of people with their IDs.

Definition 3.30 (Set). A set is a specific type of dictionary, and which simply stores a collection of keys from U . Its basic operations are:

<code>create()</code>	Returns an empty set.
<code>insert(S,x)</code>	Adds x to the collection S .
<code>delete(S,x)</code>	Deletes x from the set S .
<code>find(S,x)</code>	Returns <code>true</code> if x is in S and <code>false</code> otherwise.

We also require x to be in U and we also require that each x appears at most once in the set. A set could be defined as a dictionary where we “forget” about the value part of the key-value pairs.

Example 3.19. Some applications of the dictionary data structure are found in the implementations of:

- A language dictionary, where the key is a word and the information (value) is the definition of the word.
- A social network, where the key is a user and the information is the data of the user.
- A video site, where the key is the name of a video and the information is the video itself.

Remark 3.16. Since the convention is to keep a small key for a lot of information, we usually keep the key with a pointer to the information, and not the information itself.

Remark 3.17. Some more possible functionalities of a dictionary could be

`min(D)` Returns the minimal key in D with respect to some order on U .
`next(D, x)` Returns the minimal key with a value larger than x with respect to some order on U .

However, it is clear that these functionalities can only make sense if we define a linear order on U (and the dictionary always stores a finite amount of entries, which is always the case).

Remark 3.18 (Current goal). Our goal now is to be able to use all of the basic functionalities of a dictionary in (worst case) $\mathcal{O}(\log n)$ where n is the number of keys in the dictionary at the time of calling the function.

Definition 3.31 (Search tree). Search trees are a family of data structures that implement a dictionary when a linear order is defined on the collection of possible keys U .

Remark 3.19. The main reason we have studied the theory of graphs and trees, is because they allow us to implement dictionaries. Intuitively, a tree may not seem like dictionary, but as we mentioned at the beginning of the course, it does not matter how things seem or actually are behind the scenes, it only matters that we implement the abstract data types (in this case of a dictionary) correctly.

It may be useful to remember that a binary tree is not an abstract data type (since it does not have built-in functionalities), but a dictionary is (a binary tree by itself is a mathematical construction). A binary search tree is neither of those. For our purposes, a binary search tree is simply an implementation of a dictionary.

Definition 3.32 (Binary search tree). A binary search tree is an ordered binary tree such that in every node we store a key-value pair from a dictionary, such that for every node N with value x , all the keys on the left subtree of N are smaller than x and all the keys on the right subtree of N are bigger than x . It should also implement the functions of a dictionary.

Example 3.20 (Implementation of `find(T, x)`). Here is an implementation of `find(T, x)`

- (1) If T is empty return that x is not in the tree
- (2) Let y be the value of the root.
- (3) If $x = y$, return the pointer to the node that holds x .
- (4) If $x < y$, continue the search on the left subtree of T .
- (5) Otherwise ($x > y$), continue the search on the right subtree of T .

Example 3.21 (Implementation of `insert(T, x, info)`). Here is an implementation of the function `insert(T, x, info)`

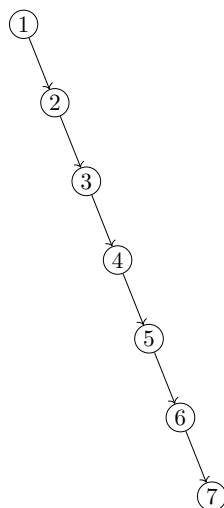
- (1) Search x in T .
- (2) If x is in T return.
- (3) Let (y, z) be the last key-value pair on the search path of x .
- (4) If $x < y$, insert a node N with the key-value pair (x, info) as a left child of the node of (y, z) .
- (5) Otherwise ($x > y$), insert a node N with the key-value pair (x, info) as a right child of the node of (y, z) .

Remark 3.20. In an ordered binary search tree, an inorder traversal would return the elements in an order corresponding to the linear order of the search tree.

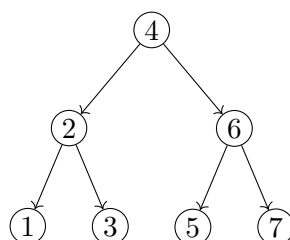
Example 3.22 (Implementation of `delete(T, x)`). Here is an implementation of the function `delete(T, x)`. Let N be the node we want to delete.

- (1) If N is a leaf node, delete it.
- (2) If N has one child, let the father of N point to that child.
- (3) Let W be the node immediately following N (we can get this using an inorder traversal). Note that W has at most one child.
- (4) Switch between nodes N and W .
- (5) Now N has at most one child, continue using step (1) or (2) as needed.

Remark 3.21. When inserting certain values to binary tree, the order matters. For example if we insert the numbers 1–7 in order, we would get the following tree:



These type of trees (with exactly one node at each level) are called degenerate binary trees. And if we insert the elements in the order 4 2 6 1 3 5 7 we would get



and it is clear that the second order of inserting elements is better (or for now at least, more neat) than the other.

Remark 3.22. Let T be a binary search tree. Using the implementations we have seen so far, we know that lookups, insertions and deletions can be done in linear time with respect to the height of T . In case the binary tree is almost complete we have $n \geq 2^h$, so $\log_2 n \geq h$, and thus lookups, insertions and deletions can be done in $\mathcal{O}(\log n)$.

However, if we have a degenerate (or pathological) tree, we get the relation $h = n - 1$, and then lookups, insertions and deletions can only be done in $\mathcal{O}(\log n)$.

Remark 3.23. A natural question to ask is what is the expectation of the height? That is, in the random life of a software engineer, what is the actual average height of a binary tree?

We won't go into details here, but given a discrete random variable X , its expectation is defined as

$$\mathbf{E}[X] := \sum_x x \cdot \mathbf{P}(X = x).$$

It is the weighted average of all possible values X can take.

Although it's very difficult to consider the expectation of the height of the tree when allowing many insertions and deletions, we can quite easily consider the expectation of the height if we only allow insertions. This simplification makes the result irrelevant in most real life scenarios, but this is the only thing we can compute without complicating things too much.

The number of possible orderings of insertion of n elements to a tree is $n!$ because there are $n!$ permutations on n elements. For each such permutation $p \in S_n$ we denote the height of the tree after insertion in that order $h(p)$. We get that

$$\mathbf{E}[h] = \frac{1}{n!} \cdot \sum_{p \in S_n} h(p).$$

It can be shown that $\mathbf{E}[h] = \mathcal{O}(\log n)$. This means that the expectation of the height of the tree after n insertions is $\mathcal{O}(\log n)$. The proof for this is omitted, however, we can prove a similar proposition.

Proposition 3.1. *The expected build time of a binary search tree with n elements is $\mathcal{O}(n \log n)$.*

Proof. First we denote the amount of comparisons we need to make by $T(n)$, the order of the elements by $b_1, \dots, b_n$ and the order of insertion of the elements by $p_1, \dots, p_n$. According to the law of total probability,

$$\mathbf{E}[T] = \sum_{j=1}^n \mathbf{P}(b_j \text{ chosen first}) \cdot \mathbf{E}(T \mid b_j \text{ chosen first}).$$

The probability that each b_j is chosen first is equal to $\frac{1}{n}$. After choosing the head we need to make $n - 1$ comparisons to divide the rest of the elements to the left and right subtrees of b_j , and build two more trees with $j - 1$ and $n - j$ elements. This means that

$$T(n) = \frac{1}{n} \cdot \sum_{j=1}^n [(n - 1) + T(j - 1) + T(n - j)], \quad (\text{and } T(0) = 0).$$

We have that

$$T(n) = \frac{1}{n} \cdot \sum_{j=1}^n [(n - 1) + T(j - 1) + T(n - j)] = n - 1 + \frac{2}{n} \sum_{j=1}^{n-1} T(j).$$

Multiplying both sides by n we get:

$$nT(n) = n(n-1) + 2 \sum_{j=1}^{n-1} T(j) \implies (n-1)T(n-1) = (n-1)(n-2) + 2 \sum_{j=1}^{n-2} T(j).$$

We can now subtract the second equation from the first and get that:

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1),$$

which implies the relation

$$nT(n) = 2n - 2 + (n+1)T(n-1).$$

Now divide both sides by $n(n+1)$ and get

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{2}{n+1} - \frac{2}{n(n+1)} + \frac{T(n-1)}{n} < \frac{2}{n+1} + \frac{T(n-1)}{n} \\ &< \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1} < \dots < 2 \sum_{j=1}^n \frac{1}{j+1} + \frac{T(0)}{1} = 2(H_{n+1} - 1) + 0 \\ &\leq 2H_{n+1} = \mathcal{O}(\log(n+1)) = \mathcal{O}(\log n), \end{aligned}$$

where the second to last equality follows from a computation we did in Example 1.10. Thus we conclude that,

$$\frac{T(n)}{n+1} = \mathcal{O}(\log n) \implies T(n) = \mathcal{O}(n \log n),$$

which completes the proof. $\square$

Remark 3.24 (The Euler–Mascheroni constant). The n th harmonic number H_n satisfies the equality

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \ln(n) + \gamma + \Theta\left(\frac{1}{n}\right),$$

where $\gamma \approx 0.57721$ is called the Euler–Mascheroni constant. This relation implies that

$$H_n - \ln(n) - \gamma = \Theta\left(\frac{1}{n}\right),$$

and thus we get that $H_n = \mathcal{O}(\ln n)$.

3.5 Exercises

Exercise 3.3. In an apocalyptic future where binary trees randomly disappear every once in a while, a team of researchers tries to understand if keeping the output of traversals on a tree is sufficient information in order (pun intended) to regenerate the exact same tree. They figured it out easily but now it's your turn. Decide from which of the following traversals we can regenerate the exact tree they came from:

- (1) Preorder.
- (2) Postorder.
- (3) Inorder.
- (4) Preorder and inorder.

Solution.

- (1) Preorder is not sufficient information. The following trees:



both have the same output under preorder: 1 2.

- (2) Postorder is not sufficient information. The trees from (1) both have the same output under postorder: 2 1.

- (3) Inorder is not sufficient information. The following trees:



both have the same output under inorder: 1 2.

- (4) Preorder and inorder is sufficient information. By properties of the preorder traversal we know that the first key in the preorder scan is the root. Denote that key by x . By properties of the inorder traversal we know that all keys to the left of x in the inorder scan are elements of the left subtree of x , and the elements to its right are the elements of its right subtree (note that the tree is not necessarily a binary search tree, so it is not possible to regenerate the tree without this scan). We denote the left and right elements of x by L and R respectively. Next, by properties of preorder, the first element in the preorder scan and L is the root of the left subtree, and similarly we can find the root of the right subtree. It is not hard to see that by using this process recursively, we can regenerate the tree.

Exercise 3.4. Let T_1 and T_2 be binary search trees, each of size n . Describe the most efficient algorithm you can think of that merges them to a binary search tree of minimal height.

Solution. The trivial solution is to simply insert all elements of T_2 into T_1 (using any traversal we want), but since each insertion costs worst-case $\mathcal{O}(n)$ time, the time complexity of this algorithm is $\mathcal{O}(n^2)$, and it leaves us with a tree of height $\mathcal{O}(n)$.

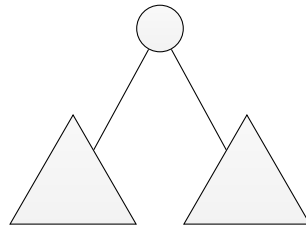
The key to solving this exercise efficiently, is remembering that an inorder traversal of a binary search tree returns its elements in a sorted order. Also, we can achieve efficient height by using a more complicated trick which we will explain soon.

The efficient solution is as follows:

- (1) Use inorder to store elements of T_1 and T_2 in a sorted order in sorted arrays A and B of size n .
- (2) Merge the two sorted arrays into a sorted array C of size $2n$.
- (3★) Generate an empty almost complete binary tree T of size $2n$.
- (4) Perform an inorder traversal on T and insert the elements of C in their sorted order.

All steps of this algorithm are $\mathcal{O}(n)$, and hence the result is an algorithm of $\mathcal{O}(n)$ time and the merged tree is of height $\mathcal{O}(\log n)$. It is left to discuss step (3★), which we will do in the following remark.

Remark 3.25 (Constructing an empty almost complete binary tree). Note that we can construct an empty complete binary tree by the following recursive relation:



The time complexity of this algorithm is $\mathcal{O}(n)$ where n is the number of nodes. The main problem is that Exercise 3.4 requires us to create a tree of size $2n$, and all complete binary search trees have an odd amount of nodes. In order to create the required almost complete binary tree (say with size N), we follow the following steps:

- (1) Create the smallest complete binary tree with more than N nodes. Denote its size by M .
- (2) Use a modified inorder traversal (which visits right subtrees before left subtrees) on the tree. By properties of inorder traversal we will visit the leaves in right-to-left order so deleting the first $M - N$ leaves we are left with an empty almost complete binary tree of size N .

Since both of these steps require $\mathcal{O}(N)$ time, we have completed our goal, and this remark is complete.

Exercise 3.5. Let T be a binary search tree. A random bit flip changed the value of exactly one node in T to a random value (assume T is no longer a BST). Describe an algorithm which fixes T back into a valid binary search tree in worst-case $\mathcal{O}(n)$ time.

Solution. The solution is as follows:

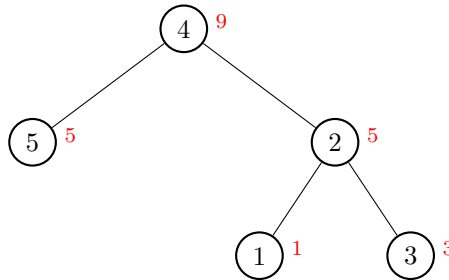
- (1) Perform an inorder traversal on T in order (pun intended) to find the bad node. By properties of inorder it will be the node closest to the root of the two/three nodes which are not of increasing order in the scan. (it will be two nodes if 0 2 3 4 changed to 0 2 1 4 and three nodes if 0 2 3 4 changed to 0 2 7 4).
- (2) Use the algorithm from Exercise 3.4 in order to merge the two subtrees of the bad node.
- (3) Insert the bad node into the newly merged tree.
- (4) Replace the subtree rooted at v with the new tree we created.

All steps take $\mathcal{O}(n)$ time, hence the algorithm is $\mathcal{O}(n)$ time.

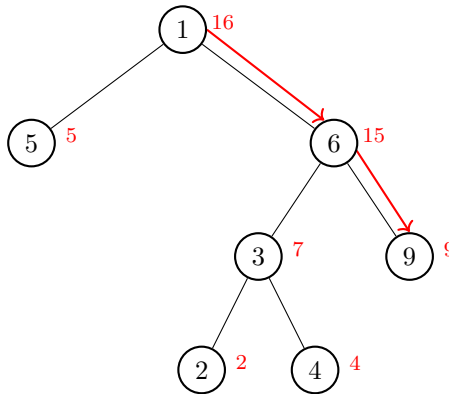
Exercise 3.6 (Extra). Let T be a full binary tree with n nodes (not necessarily a binary search tree). Describe an algorithm that computes the sum of the heaviest path from the root to any of its leaves (the path where the sum of the node values is maximal) in time complexity $\mathcal{O}(n)$. Moreover, the algorithm must print this path. You may add additional fields to the nodes for your solution.

Solution. Note that if the only requirement was to compute the sum of the heaviest path, we could have solved the problem using a postorder traversal (the type of traversal that visits the children of nodes before the nodes), that updates a field `weight` that is given by the value of that node for leaves, and otherwise by `v->value + max(v->left.weight, v->right.weight)`. Intuitively, this value represents the heaviest path from v to any of the leaves in the subtree rooted at v . After this process we can return `root.weight`.

In order to print the nodes in the heaviest path one may think to retrace the heaviest path from the root by subtracting the value of the root from `root.weight` and going in the direction of the child with the weight that is equal to the result. However, it is possible that the roots child have the same value weight value as in the following example:



where the red values represent the weight of the nodes. Hence this method does not work (we will not be able to tell where to go). In order to print the heaviest path we must additionally add pointers that tell us which child has the weight value we got when computing `max(v->left.weight, v->right.weight)`. Here is an example of what it would look like:



so we can easily retrace the pointers from the root in order to print the heaviest path. This solution is correct and it particular it solves the previous counterexample (think what happens).

4 AVL trees

4.1 AVL trees

As we saw earlier, we would much rather work with more balanced trees like complete or almost complete trees, than with pathological trees. The problem is that it's very hard to keep a tree complete (or almost complete) in each insertion and deletion. We are going to solve this problem using AVL trees. After we complete the construction and implementation of an AVL, we would get an implementation for a dictionary with all functions done in $\mathcal{O}(\log n)$!

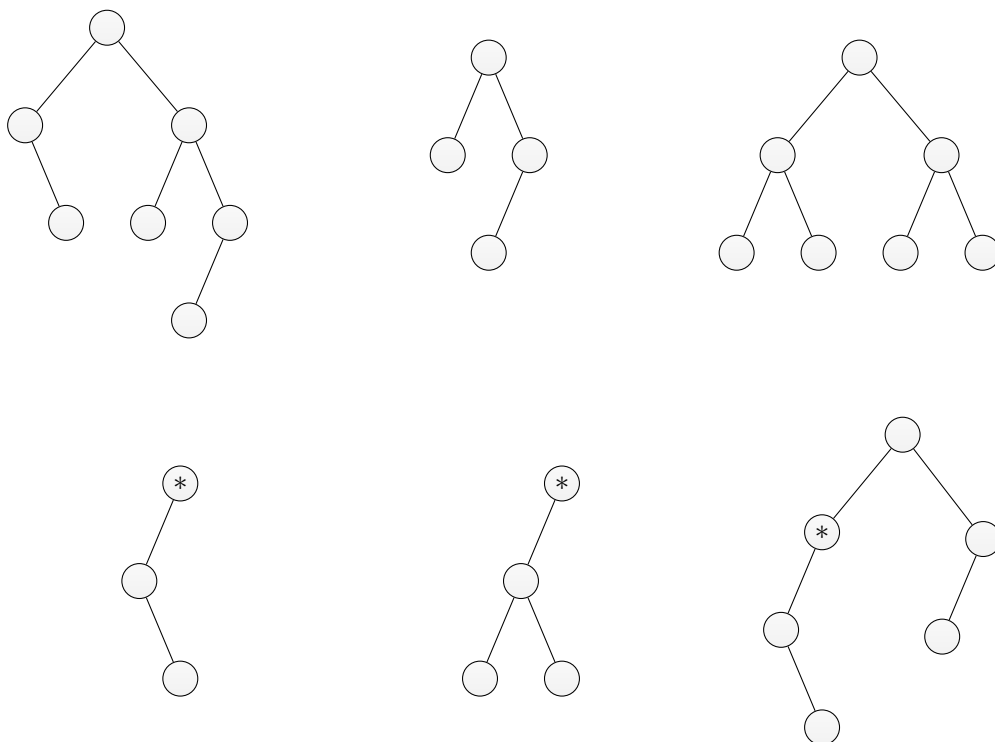
Definition 4.1 (Balanced tree). Let $\mathcal{T}$ be a family of trees. Then $\mathcal{T}$ is said to be balanced if there exists a function $f(n) = \mathcal{O}(\log n)$ so that $h(T) < f(n)$ for all $T \in \mathcal{T}$.

Definition 4.2 (AVL tree). An AVL tree (named after its inventors **A**delson-**V**elsky and **L**andis), is a binary search tree such that for any node n we have

$$|h(v \rightarrow \text{left}) - h(v \rightarrow \text{right})| \leq 1,$$

where $h(v \rightarrow \text{left})$ is the left subtree of v , and $h(v \rightarrow \text{right})$ is the right subtree of v .

Example 4.1. Below are some examples, and non examples of AVL trees. The unbalanced nodes (those that violate the balance condition) are marked with a *.



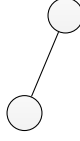
Remark 4.1. A leaf node is a subtree of height 0 and an empty tree is of height -1 .

Remark 4.2. We see that the definition of an AVL tree is balancing the tree in some way. Our goal is to show that in an AVL tree of height h , the number of nodes is at least $n > (1.6)^h$ and thus $h < \log_{1.6} n = \mathcal{O}(\log n)$.

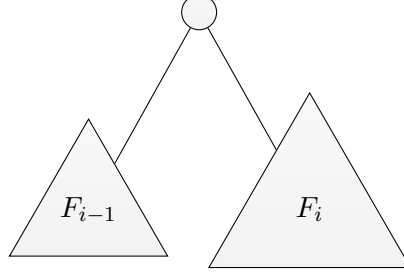
Definition 4.3 (Fibonacci trees). The Fibonacci trees are defined as a recursive sequence of trees. First, we define F_0 by



and F_1 by



Then we define F_{i+1} by the recurrence relation:



Remark 4.3. The Fibonacci trees are the “least balanced” AVL trees. We also have that F_i are the smallest (by node amount) of all AVL trees of height $h(F_i)$. Thus, if we will show that $h(F_i) = i$, and that $|F_i| \geq a^i$ for some $a > 1$ (where $|F_i|$ is the amount of node in F_i), we will achieve our goal described in Remark 4.2.

Proposition 4.1. *For every n , the height of $h(F_n) = n$.*

Proof. We can see that $h(F_0) = 0$ and $h(F_1) = 1$ and by the construction of F_n we have

$$h(F_n) = h(F_{n-1}) + 1 = n$$

which completes the proof (by induction). □

Remark 4.4. That that by Theorem 4.1 we know that not only does $h(F_n) = n$, but also that in the family of AVL trees of height n , there is exactly one unique Fibonacci tree (intuitively, this is the “least balanced tree”).

Proposition 4.2. *Denote the number of nodes of a tree T by $|T|$. Then*

$$|F_n| = 1 + |F_{n-1}| + |F_{n-2}| \geq |F_{n-1}|.$$

Proof. This is obvious from the construction of F_n . □

Proposition 4.3. *Let T be an AVL tree with height n . Then $|T| \geq |F_n|$.*

Proof. We can prove this by induction on n . We can verify that the proposition holds for $n = 0$ and $n = 1$. Denote L the left subtree of T and R the right subtree of T . We know that the heights of L and R are less than h . Since T is an AVL tree without loss of generality assume we have $h(R) = n - 1$ and then by the balancing property of AVL trees $h(L)$ is either $n - 1$ or $n - 2$. Since $n - 1 < n$ using the induction hypothesis we have that $|R| \geq |F_{n-1}|$ and from Theorem 4.2 (and the induction hypothesis) we have that

$$|L| \geq \min \{|F_{n-1}|, |F_{n-2}|\} = |F_{n-2}|.$$

Thus,

$$|T| = 1 + |R| + |L| \geq 1 + |F_{n-1}| + |F_{n-2}| = |F_n|$$

which completes the proof. □

Remark 4.5. We have now shown that $|T| \geq |F_n|$ for any AVL tree T of height n . It is now suffices to show that there exists a number $\phi > 1$, so that $|T_n| > \phi^n$ in order to achieve our goal from Remark 4.2 (we will actually show a slightly different inequality, but we will see that it also suffices).

Proposition 4.4. Let $\{n_i\}_{i=0}^{\infty}$ be the sequence of Fibonacci numbers. Then

$$n_i = \frac{\phi^i - \bar{\phi}^i}{\sqrt{5}}$$

where

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \quad \text{and} \quad \bar{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618.$$

Remark 4.6. We call $\phi \approx 1.618$ the golden ratio.

Remark 4.7. Since $|\bar{\phi}| < 1 < |\phi|$ it follows that

$$n_i \approx \frac{\phi^i}{\sqrt{5}}$$

for a sufficiently large i .

Remark 4.8 (The beauty behind the golden ration). The golden ration is one of the most recurring numbers in nature. A 15th century mathematician Luca Pacioli wrote an entire book called “Divina proportione” on the golden ration. The golden ration is an important and useful constant. Some other parts in math where ϕ appears include the positive solution of the equation

$$x^2 = x + 1,$$

or even the result of the following continuous fraction:

$$\phi = \sqrt{1 + \sqrt{1 + \sqrt{1 + \cdots}}} \quad \text{or} \quad \phi = 1 + \frac{1}{1 + \frac{1}{1 + \cdots}}.$$

Proof of Theorem 4.4. We are given the equations:

$$n_0 = 0 \quad n_1 = 1 \quad n_{i+1} = n_i + n_{i-1}.$$

Ignore the constraints $n_0 = 0$ and $n_1 = 1$ and assume that $n_i = x^i$ are solutions (to the variables $\{n_i\}_{i=0}^{\infty}$). Then we get the relation:

$$x^{i+1} = x^i + x^{i-1}.$$

By dividing both sides by x^{i-1} we get the relation $x^2 = x + 1$, and the solutions of this equation are exactly

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \quad \text{and} \quad \bar{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618.$$

This implies that $n_i = \phi^i$ and $n_i = \bar{\phi}^i$ are both solutions of $n_{i+1} = n_i + n_{i-1}$, and similarly, any linear combination $n_i = a\phi^i + b\bar{\phi}^i$ is also a solution (although, not of the form $n_i = x^i$). We can now consider the constraints $n_0 = 0$ and $n_1 = 1$ again in order to obtain

$$\begin{aligned} n_0 = 0 &\implies a\phi^0 + b\bar{\phi}^0 = 0 && \implies b = -a \\ n_1 = 1 &\implies a\left(\frac{1 + \sqrt{5}}{2}\right) + (-a)\left(\frac{1 - \sqrt{5}}{2}\right) = 1 && \implies a = \frac{1}{\sqrt{5}} \end{aligned}$$

and therefore the solution to the equation is $n_i = \frac{\phi^i - \bar{\phi}^i}{\sqrt{5}}$. □

Remark 4.9. We can also prove Theorem 4.4 by using induction — or even by using linear algebra, as you may have seen in your first semester!

Proposition 4.5. *The i th Fibonacci tree F_i has $|F_i| = n_{i+3} - 1$ nodes, where n_i is the i th Fibonacci number.*

Proof. We have the following equations:

$$|F_0| = 1 \quad |F_1| = 2 \quad |F_{i+1}| = |F_i| + |F_{i-1}| + 1.$$

Note that if we denote $t_i = |F_i| + 1$, then $|F_i| = t_i - 1$, and thus the equations become

$$t_0 = 2 \quad t_1 = 3 \quad t_{i+1} - 1 = (t_i - 1) + (t_{i-1} - 1) + 1 \iff t_{i+1} - 1 = t_i + t_{i-1}.$$

Thus the set $\{t_i\}_{i=0}^\infty$ must be equal to the Fibonacci sequence, except the indices are shifted by 3 places. This implies that $|F_i| = n_{i+3} - 1$ which completes the proof. $\square$

Proposition 4.6. *Let T be an AVL tree with n nodes and height $h(T) = h$. Then $h = \mathcal{O}(\log n)$.*

Proof. By Theorem 4.3 we obtain

$$n = |T| \geq |F_h|.$$

By Theorem 4.4 and Theorem 4.5 we get that

$$n \geq |F_h| = n_{h+3} - 1 = \frac{\phi^{h+3} - \bar{\phi}^{h+3}}{\sqrt{5}} - 1 \geq \frac{\phi^{h+3}}{\sqrt{5}} - 2,$$

where the last transition follows from the fact $\bar{\phi}^{h+3} < 1$. This inequality implies that

$$\sqrt{5}(n+2) \geq \phi^{h+3}.$$

By taking the logarithm base ϕ on both sides of the equation we get

$$\log_\phi \left(\sqrt{5}(n+2) \right) \geq h+3,$$

and by applying logarithm identities and subtracting 3 we get

$$\log_\phi(n+2) + \log_\phi \sqrt{5} - 3 \geq h,$$

so in conclusion $h = \mathcal{O}(\log n)$. $\square$

Remark 4.10. If we denote the family of AVL trees by $\mathcal{T}$ and let $f(n) = \log_\phi(n+2) + \log_\phi \sqrt{5} - 3 = \mathcal{O}(\log n)$, then for any $T \in \mathcal{T}$ with n nodes we have $h(T) \leq h(F_n) \leq f(n)$, so clearly $\mathcal{T}$ is a balanced family of trees (by definition the inequality should be strict, but we can easily fix that by considering $g(n) = f(n) + 1$ for example).

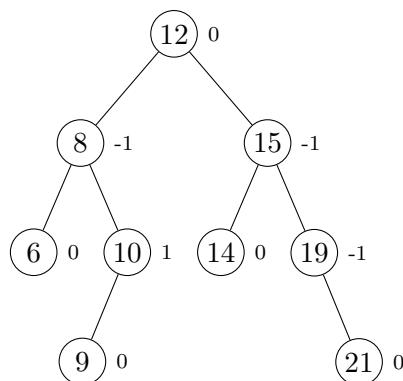
Remark 4.11. By Theorem 4.6 we get that all binary search tree functions on AVL trees can be done in $\mathcal{O}(\log n)$. Our goal now is to make sure that after an insertion/deletion to an AVL tree, the tree remains an AVL tree, so that we could keep using the functions in $\mathcal{O}(\log n)$ indefinitely.

Definition 4.4 (Balance factor). Let T be a binary tree, and let v be a node in T . Then the balance factor of v is defined by

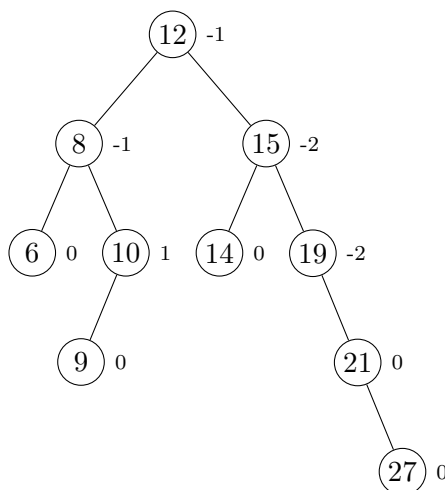
$$BF(v) = h_L(v) - h_R(v),$$

where L and R are the left and right subtrees of v respectively.

Example 4.2. Here is a binary tree with the balance factor of each node written near the nodes:



This is an AVL tree, since all the balance factors are in the range $[-1, 1]$. Note that if we insert 27 into the tree we would get



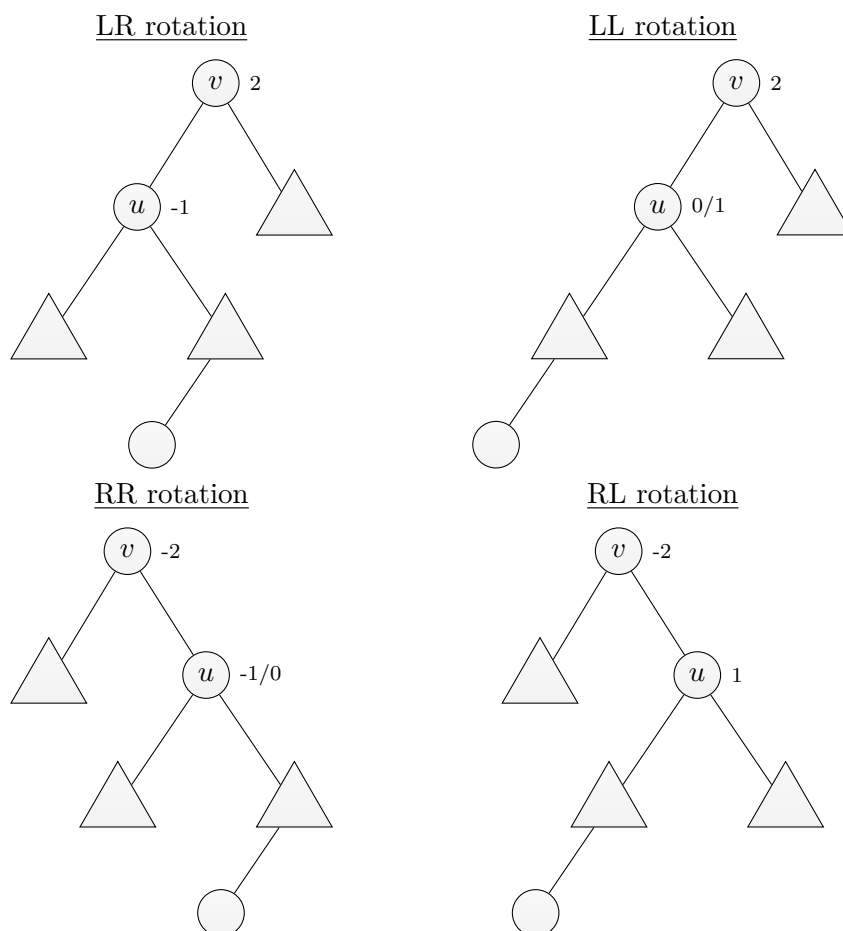
and now the tree is no longer an AVL tree. In order to solve this problem we will use rotations. Before learning about what kind of rotations there are, we need to notice the following facts about AVL trees:

- (1) The only nodes in which the balance factor may violate the AVL balance condition, are those along the insertion/deletion path.
- (2) If v is a node in the insertion/deletion path and the height of the subtree rooted at v has the same height after the insertion/deletion, then the balance factor of the nodes “above it” in the path has not changed.
- (3) If the balance factor condition was violated, then the balance factors of the tree must remain in the range $[-2, 2]$ (for example, they cannot become 3 after one insertion).
- (4) Each time the balance factor condition is violated, we can make the tree an AVL tree again by using rotations (explained below).

After each insertion/deletion when the balance factor condition is violated, we can notice that:

- (1) By fixing the balance factor of the “lowest” node in the insertion/deletion path, we would fix the balance factor condition in every node, and thus complete our goal. Also that “lowest” node always exists since trees are finite in this course.

- (2) If we denote the “lowest” problematic node by v , then we can always classify the subtree rooted at v into 4 cases:



Convince yourselves that indeed, these four cases exhaust all possibilities, and that these are the possible balance factors (note that we could also interpret these as six cases, and that balance factors 0 are only achievable with deletions).

Remark 4.12. The name LL rotation comes from the fact that the lowest unbalanced node v is “left-heavy”, then its child at the direction of imbalance u is also “left-heavy”, and similarly for the other types of rotations. This explains why an LL rotation actually requires a rotation to the right and not to the left for example.

Remark 4.13 (Implementation of rotation). The logic behind the rotations themselves is not very integral to the theory, and one may try to derive the algorithm of each rotation by themselves or see them online.

4.2 Implementation of AVL trees

Our last remaining task is to make sure we can keep the balance factor field at each node and update it according to the actions on the tree without changing the complexity of our functions (insertions, deletions, and lookups).

Example 4.3 (Balance factor). We want to keep the balance factor at every node, but in order to do that we will instead keep something else. At every node v we will keep the value $h(v)$, which is the height of the subtree rooted at v . If we have these fields we can easily compute the balance factor of every node in $\mathcal{O}(1)$. Note that since $h(v)$ is determined by $h(v \rightarrow \text{right})$ and $h(v \rightarrow \text{left})$, we can always easily update the field even after insertions and deletions.

Example 4.4 (Insertions in AVL trees). The insertion algorithm in an AVL tree is very similar to the one in a binary search tree, with slight modifications. The algorithm is as follows:

- (1) Insert x to T like a binary search tree, and let v be its associated leaf node.
- (2) Put 0 in $h(v)$.
- (3) While $v \neq \text{root}$
 - (a) Let p be the node $\text{parent}(v)$.
 - (b) If $h(p) \geq h(v) + 1$ return (end the function).
 - (c) Set $h(p)$ to be $h(v) + 1$.
 - (d) If p violates the balance factor condition, execute the relevant rotation and return. Otherwise, set v to be p .

Question. How can we compute $\text{parent}(v)$?

Answer. We can implement insertion recursively, then by the process of recursion we would have access to the parent. Alternatively, we can keep the insertion path in a stack while descending in the tree, then we would have access to the parent through the stack.

Note that the complexity of the algorithm is $\mathcal{O}(\log n)$ since insertion in a binary search tree is $\mathcal{O}(\log n)$, allocating and adding the node is $\mathcal{O}(1)$, finding the lowest node that violates the balance factor condition is $\mathcal{O}(\log n)$, and fixing it using a rotation is $\mathcal{O}(1)$, so in total we still get that this algorithm is $\mathcal{O}(\log n)$.

Example 4.5 (Deletions in AVL trees). The deletion algorithm in an AVL tree is very similar to the one in a binary search tree, with slight modifications. In contrast to insertion, however, in the deletion algorithm one rotation may not be enough to balance the tree, so we use multiple rotations when necessary. The algorithm is as follows:

- (1) Delete v just like in a binary search tree.
- (2) Fix the created imbalance in the following way: for every v in the deletion path, starting from the bottom up to the root:
 - (a) Update $h(v)$.
 - (a) If $|BF(v)| = 2$, perform the relevant rotation.
 - (a) If the height of the subtree rooted at v has not changed, return, and otherwise the height has changed (and the balance factor condition is satisfied), so continue with the next node in the deletion path.

Remark 4.14. As mentioned earlier, in deletion, we may have to perform more than one rotation. Also note that this algorithm is a generalization of the insertion algorithm.

Note that similarly to insertion, the modified deletion algorithm is still $\mathcal{O}(\log n)$. That is because deletion in a binary search tree is $\mathcal{O}(\log n)$, finding the first node of imbalance is $\mathcal{O}(\log n)$, each rotation necessary is $\mathcal{O}(1)$, which we perform at most $\mathcal{O}(\log n)$ times so balancing the tree also takes $\mathcal{O}(\log n)$ time, and in total the deletion algorithm is $\mathcal{O}(\log n)$.

Example 4.6 (Deletion in Fibonacci trees). Let F_n be the n th Fibonacci tree. If we delete its first (leftmost) leaf node, we would necessarily have a rotation on every node in the path from the leaf to the root.

Exercise 4.1. Compute the sequence of rotations (for example $a_1 = \text{LL}$, $a_2 = \text{RL}$ and so on) that occur when we delete the first (leftmost) leaf node of the n th Fibonacci tree F_n .

Remark 4.15 (Conclusion). We achieved our goal and found a data structure (AVL trees) that allows lookups, insertions, and deletions in $\mathcal{O}(\log n)$ time. We had to modify the commands to make sure they preserve the tree balanced, but we managed to do it such that all commands still take $\mathcal{O}(\log n)$ time.

Remark 4.16 (A taste for more). Some more interesting families of trees include:

Red-Black trees: Another famous balanced family of trees, commonly taught in universities, and also used in the industry.

Splay Trees: A family of trees used when recently accessed elements are likely to be accessed again soon.

B-Trees: Another balanced family of trees, it includes 2-3 trees and mostly used for filesystems and databases.

TRIE: Another family of trees mainly useful when working with strings.

4.3 Exercises

Exercise 4.2. Let $\{T_i\}_{i=1}^k$ be a sequence of nonempty AVL trees with a total of n nodes. Describe an algorithm that outputs all n numbers in increasing order in $\mathcal{O}(n \log k)$ time.

Solution. Note that the trivial solution is to print all elements to an array, and sort the array then print it. This solution takes $\mathcal{O}(n \log n)$ time. Another solution is to perform an inorder traversal on all trees, then merge the k scans into an array of size nk . This solution takes $\mathcal{O}(nk)$ time (we merge k arrays simultaneously).

The efficient solution is using an AVL tree based merge. The algorithm is as follows:

- (1) Perform an inorder traversal on $\{T_i\}_{i=1}^k$ and get the respective sorted arrays $\{A_i\}_{i=1}^k$.
- (2) Initialize pointers $\{P_i\}_{i=1}^k$ so that P_i points to the beginning of A_i .
- (3) Insert $\{P_i\}_{i=1}^k$ into an AVL tree where the values they are sorted by is the values they point to.
- (4) Start filling an array A of size n in increasing order as follows:
 - (a) Delete the minimal element P_i from the AVL tree.
 - (b) Insert the element P_i points to.
 - (c) Increment P_i to point to the next element in A_i and insert it to A again.

Step (1) takes $\mathcal{O}(\sum_{i=1}^k |T_i|) = \mathcal{O}(n)$ times, step (2) takes $\mathcal{O}(k)$ time, step (3) takes $\mathcal{O}(k \log k)$ times, and step (4) takes $\mathcal{O}(n \log k)$ times since we perform the subroutine in $\mathcal{O}(k)$ time for a total of n elements.

Exercise 4.3. We say that two binary trees (not necessarily AVL trees) are equivalent under AVL rotations, if there exists a sequence of AVL rotations which turns one tree into the other. Notice that this is indeed an equivalence relation. Let

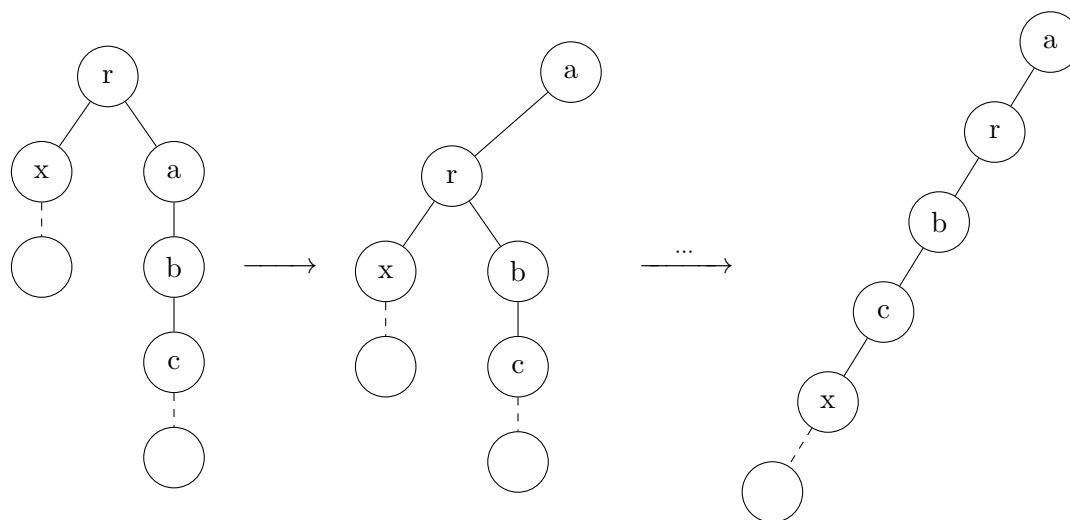
$$T_N = \{T \mid T \text{ is a binary tree with } n \text{ elements}\}$$

. Find $|T_n| / \sim$ where $\sim$ is the relation described above.

Solution. We know that AVL rotations preserve the order of the inorder scan, hence we can find the number of equivalence classes of binary trees with n elements inserted in a specific order and multiply by $n!$ to get the result.

We claim that every $T \in T_n$ is equivalent to a degenerate tree (without loss of generality assume all children nodes are left children) $L_n \in T_n$ with the same inorder ordering of the elements. After proving this claim we would get that the total number of equivalence classes for trees with a specific ordering of the elements is 1, hence $|T_n / \sim| = n!$. We will show the claim by induction on n .

For $n = 1$ the claim is clearly true without any rotations. Assume that the claim is true in T_n . Let $T \in T_{n+1}$. If T has no right subtree, use the induction step to turn its left subtree into L_n , and if T has both a right and a left subtree use the induction step on both subtrees, then perform a left rotation on r until we get L_{n+1} . This completes the proof of the induction. Here is how the last step would look like:



Note that at each level the degenerate right subtree decreases in height by one so eventually we indeed get L_{n+1} . Thus, indeed the result is $|T_n / \sim| = n!$.

5 B-trees, rank trees, and tries

Recall that

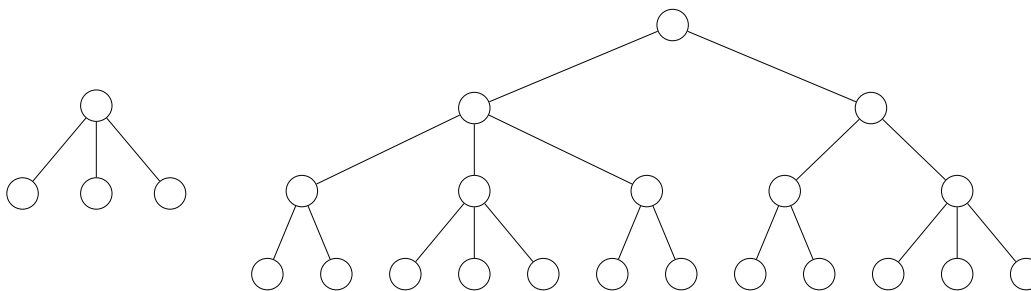
- (1) Our goal is to store **(key,data)** pairs efficiently.
- (2) Usually, **data** stores a point to another place where the actual data is stored (for efficiency when dealing with the nodes).
- (3) We have previously seen binary search trees, in which we can insert, delete, and lookup keys in $\mathcal{O}(h)$ time where h is the height of the tree.
- (4) In a balanced tree we are promised that $h = \mathcal{O}(\log n)$, and we have seen that AVL trees are a family of balanced trees.

In this section, we will introduce 2-3 new types of trees (and more generally B-trees), which are used in the heart of storage systems, databases, and more.

5.1 2-3 trees

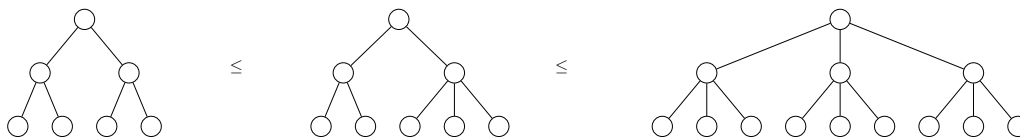
Definition 5.1 (2-3 tree). A 2-3 tree is an ordered tree in which all leaf nodes are on the same level (depth), and every inner node has either 2 or 3 children.

Example 5.1. Here are some examples of 2-3 trees:



Proposition 5.1. The number of leaves in a 2-3 tree satisfies $2^h \leq L \leq 3^h$ where h is the height of the tree.

Proof. In order to achieve the lower bound, we can remove every third child, and thus obtain a complete binary tree, in which the number of leaves is 2^h . In order to achieve the upper bound, we can add a third child to every node, and complete the tree to a complete ternary tree, in which the number of leaves is 3^h . For example:



in the left tree there are 2^h leaves, and in the right tree 2^3 leaves. □

Corollary 5.2. We have that $\log_3 L \leq h \leq \log_2 L$, and in particular $h = \Theta(\log L)$.

Remark 5.1. Similarly we can show that if n is the number of nodes of a 2-3 tree then $2^{h+1} - 1 \leq n \leq (3^{h+1} - 1)/2$ so $h = \Theta(\log n)$.

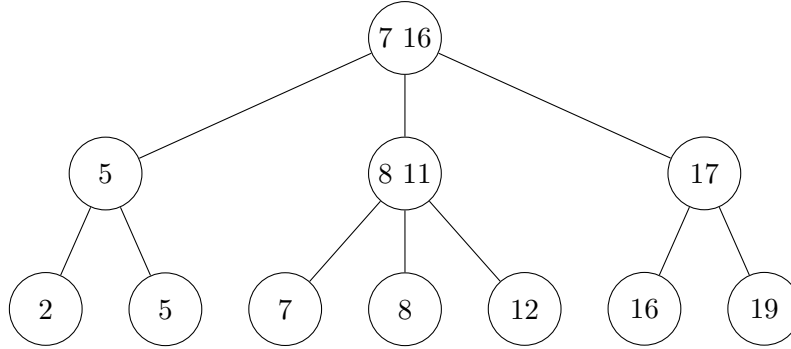
Remark 5.2 (Node structure). Recall that so far the node structure was simply **(key,data)**, and the key was unique for every node. In a 2-3 tree the situation is different. The leaves have a unique key (we will assume keys are numbers), and inner nodes with two children also have a unique key, however, inner nodes with three children, have two distinct keys. In order to make search (lookups) efficient, we need to impose a few rules on the keys. Recall that in a binary search tree the rules were:

- (1) All keys are distinct.
- (2) The key of a right child must be strictly greater than the key of his parent, and the key of the left child must be strictly lesser than the key of his parent.

In a 2-3 tree, we do not require the keys to be unique, and the rules are different. They are:

- (1) Two different nodes may have the same key, but all leaves must have distinct keys. Intuitively, all of our data is stored in the leaves (if an inner node contains k but no leaf contains k we want `search(T,k)` to return `false`).
- (2) In the case of a node with two children v_2 , denote its key by k and its left and right children by v_2^L and v_2^R respectively. Then we must have $\text{key}(v_2^L) < k \leq \text{key}(v_2^R)$. In the case of a node with three children v_3 , denote its two keys by k_1 and k_2 , and denote its left, middle, and right children by v_3^L , v_3^M and v_3^R respectively. Then we must have $\text{key}(v_3^L) < k_1 \leq \text{key}(v_3^M) < k_2 \leq \text{key}(v_3^R)$.

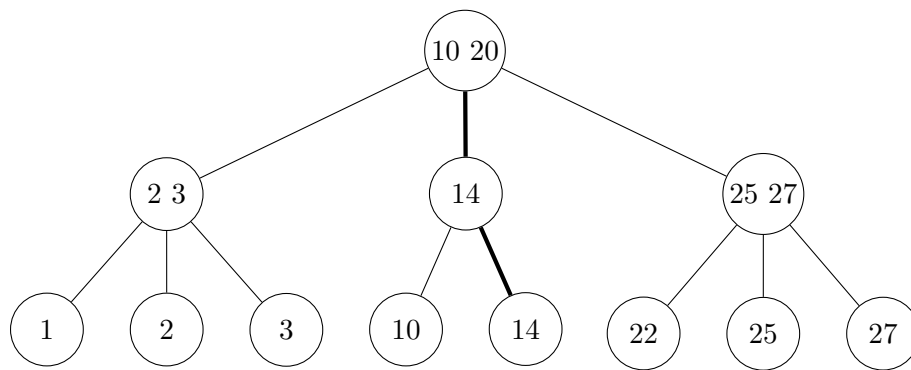
Example 5.2. Here is an example of a 2-3 tree with its keys written on the nodes, and its data omitted:



Definition 5.2 (Search in 2-3 trees). In the following algorithm, v_L , v_M , and v_R are the left, middle, and right children of v accordingly (v_M does not exist if v has two children, and none exist if v is a leaf). The keys of v are denoted by k_1 and k_2 such that $k_1 < k_2$ (and k_2 does not exist if v has two children or is a leaf node). We also denote the key we are searching for by k . The search algorithm in a 2-3 tree is as follows:

- (1) Initialize v to be the root of the tree.
- (2) If v is a leaf node, return `true` if $k_1 = k$ and `false` otherwise.
- (3) If $k < k_1$ put $v \leftarrow v_L$ and go back to step (2). Else if v has two children, put $v \leftarrow v_R$ and go back to step (2). Else if v has three children and $k < k_2$, put $v \leftarrow v_M$ and go back to step (2). Else (in this case v has three children and $k_2 \leq k$) put $v \leftarrow v_R$ and go back to step (2).

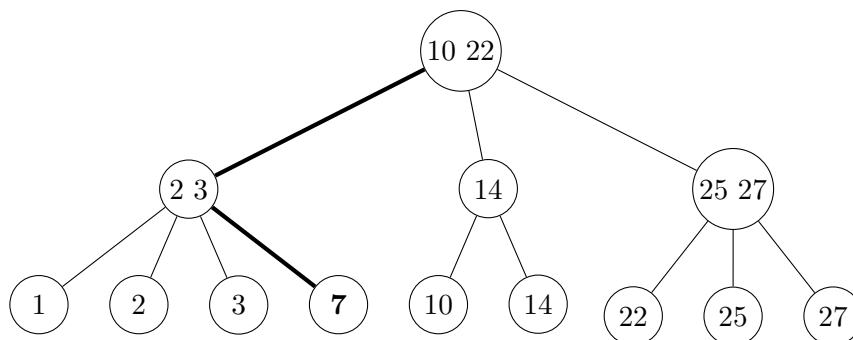
Example 5.3. If we search for the key 14 in the following tree:



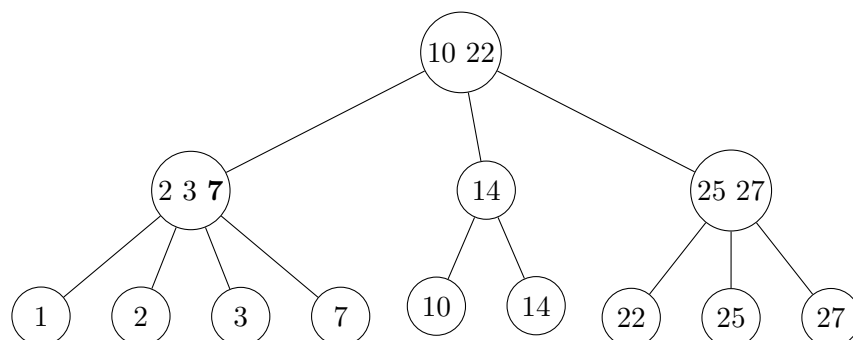
then the path to the node is the bolded path.

Remark 5.3. When inserting a node into a 2-3 tree we must search for its right place, then insert it, and then if the 2-3 tree now has a node with 4 children we need to fix this. The fixing is done at each level, along the insertion path (in reverse), and we must perform $\mathcal{O}(1)$ operations at each layer in order to keep the rest of the operations at time complexity $\mathcal{O}(\log n)$.

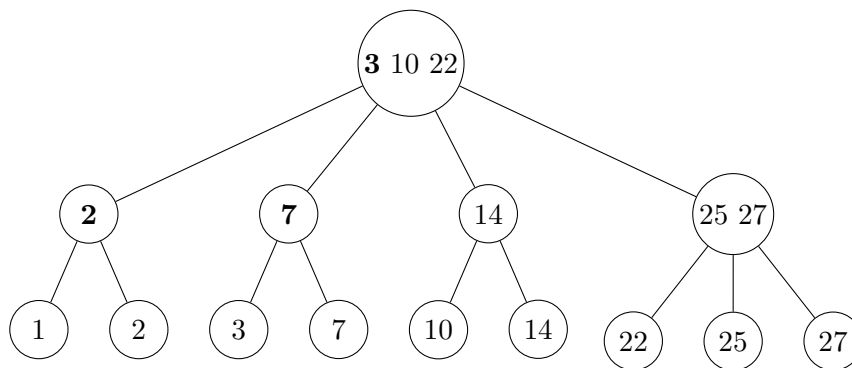
Example 5.4. Here is an example of how insertion works: First suppose we inserted 7 to our tree:



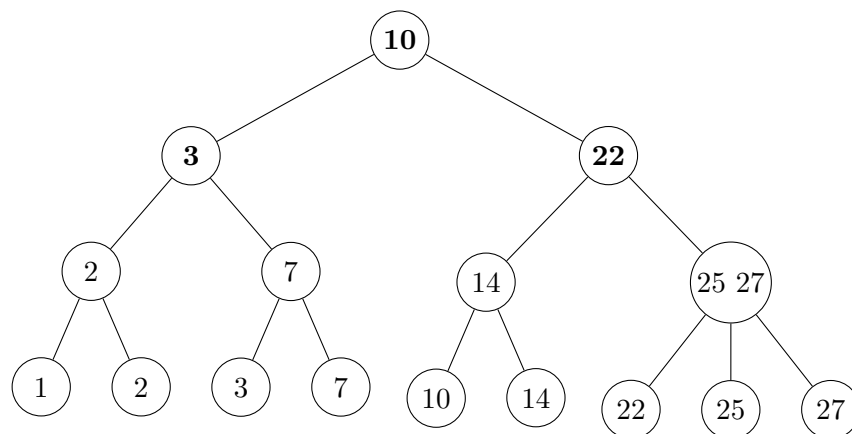
Then in order to fix the node v with 4 children, we first add 7 to the list of keys of v , which must be its third key (but it isn't always the biggest key).



Then we split v from a node with 3 keys and 4 children to two nodes with 1 key each, and 2 children each, and we send the third key (always the middle key with respect to order, since $2 < 3 < 7$ we send 3) to the parent of v .



Now our problem is that the former parent of v has 4 children, so we need to apply the same algorithm from earlier. We send the middle of the three children (with respect to order, since $3 < 10 < 22$ we send 10) to the problematic root's parent (in this case the empty node), split the remaining node with two keys into two nodes with one key each, both nodes are the children of the previously empty node, and then the four children become the children of the two nodes (two for each).



Definition 5.3 (Insertion in 2-3 trees). The insertion algorithm is as follows:

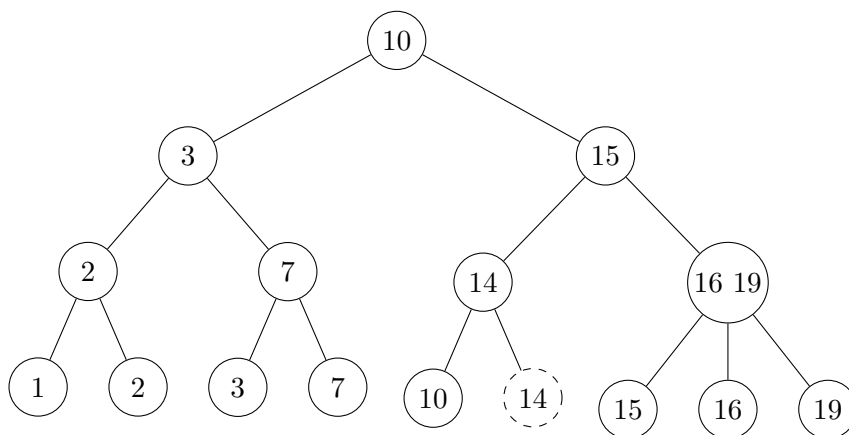
- (1) Search for k in T (remember, we only search the roots). If k is already in the tree, then do nothing and return.
- (2) If k is not in T , then we can denote the last node in the search path of k by f (excluding the leaf node, that is, one node before the last node in the search path).
- (3) Create a new node with key k , and set it as a child of f while keeping the order between the children of f (remember, a 2-3 tree is ordered). It may be the case f now has 4 children.
- (4) Add a new key to f according to the 2-3 tree conditions, for example if k is the biggest child of f , you may add k as f 's additional key. It may be the case f now has 3 keys.
- (5) Let v be another pointer of f .
- (6) If v has 3 children (it also follows from this that it has exactly two keys) then do nothing and return.
- (7) If v has 4 children, split v into two nodes v_1, v_2 with one key each, and send the middle key (with respect to order) to v 's father node (if v is the root then we create a new root, and return since the algorithm finished successfully), to which we assign v_1 and v_2 as children (when assigning we respect the order).

(8) If the algorithm didn't terminate at step (7), return to step (5).

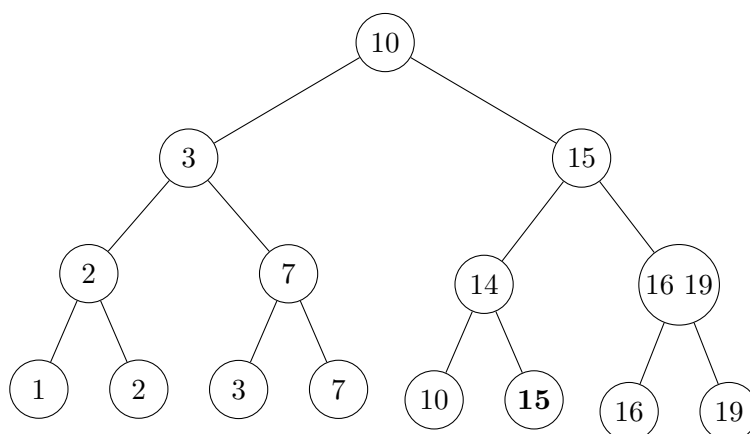
Remark 5.4 (On insertion in 2-3 trees). We perform splits only on the along search path, and every split is done in $\mathcal{O}(1)$. When splitting v into v_1 and v_2 , the depth of v_1 and v_2 is exactly the depth of v , however if we split the root then the depth of *all* nodes increases by one. We also note that the splitting is done in a way that preserves the properties of 2-3 trees, so after insertions the tree is still balanced. All of this implies that we can perform lookups and insertions in a 2-3 tree in $\mathcal{O}(\log n)$ indefinitely.

Remark 5.5. When deleting a node into a 2-3 tree we must search for its right place, then delete it, and then we may have to “fix” the tree so that it remains a 2-3 tree after deletion. The fixing is done at each level, along the deletion path, and we perform $\mathcal{O}(1)$ commands at each node in order to keep all the algorithms at time complexity $\mathcal{O}(\log n)$.

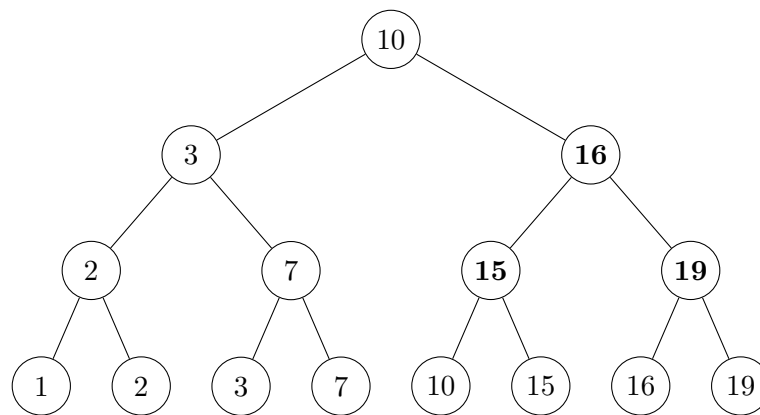
Example 5.5 (Deletion by borrowing). Suppose we wanted to delete the node 14 from the following tree:



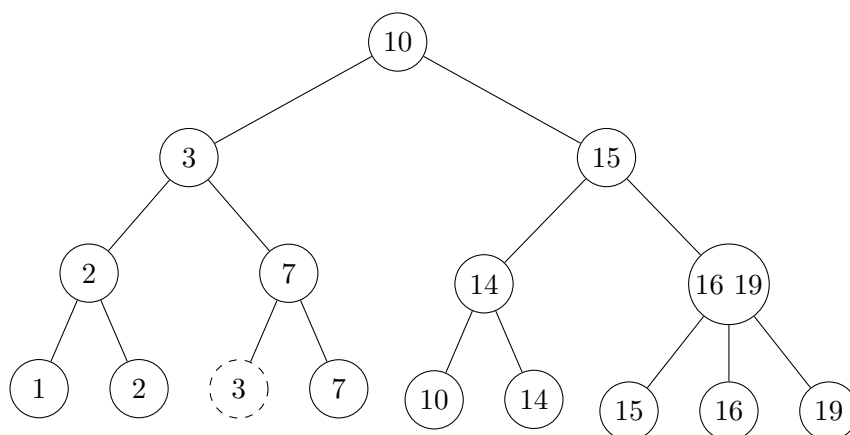
Since the level of the leaves is ordered increasingly from left to right, we can borrow 14's immediate successor leaf from the right (if we wanted to delete 10 we would borrow its predecessor leaf from the left).



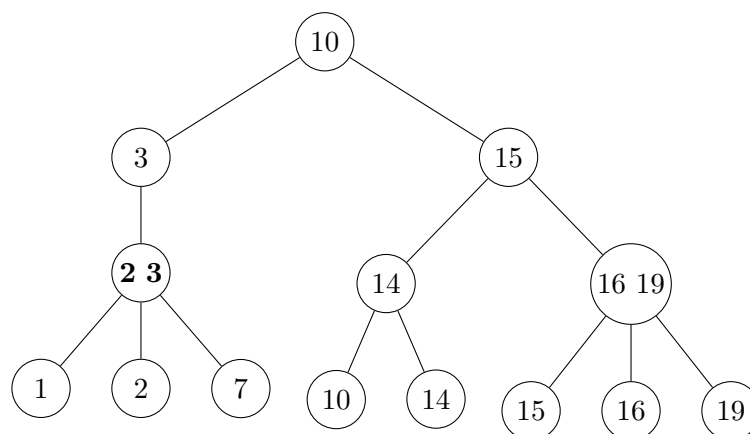
Then we need to fix the keys of the parents of the leaves. This can be done as follows:



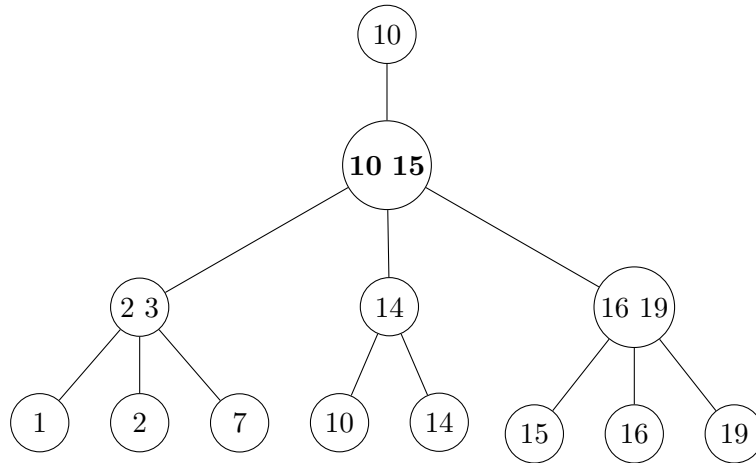
Example 5.6 (Deletions by merging). A different example is as follows. Suppose we wanted to delete the node 3 from the following tree:



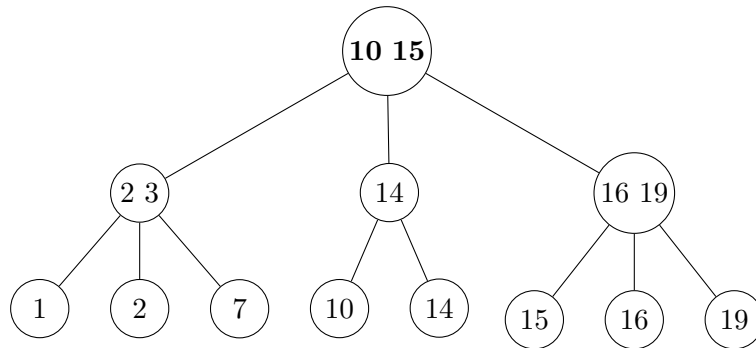
In this case, borrowing won't help, so we need to merge the parent nodes. Also, it would be better to merge 2 with 3 instead of 7 so we will do that instead.



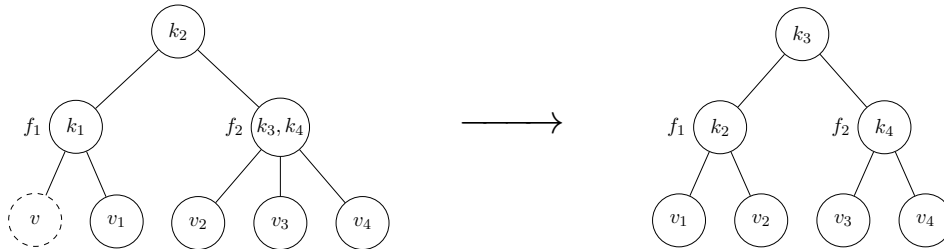
Similarly, we need to merge the parents again, in a way that preserves the properties of 2-3 trees.



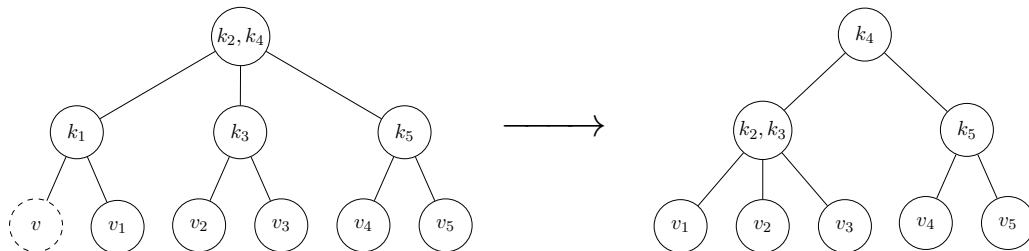
Now we simply need to erase the superfluous root we get:



Remark 5.6. In general, the borrowing process looks like this:



and the merging process looks like:



Definition 5.4 (Deletion in 2-3 trees). The algorithm for deletion of a node with key k in 2-3 trees is as follows:

- (1) Search for k in the tree and if its not found return.
- (2) If k is found let t be the leaf with value k and let f be its parent.
- (3) Delete t from the tree.
- (4) Set $v \leftarrow f$.

- (5) If v is a root with a single child, delete v and return.
- (6) If v has two children, return. Otherwise continue to step (7).
- (7) Check the siblings of v .
 - (a) If v has a single sibling with 3 children, borrow from the sibling and return.
 - (b) If any immediate sibling of v has two children, merge v with an immediate sibling of v and update the keys in the created node accordingly. Also let f be the parent of v .
- (8) Return to step (4).

Remark 5.7 (Correctness of deletion). Notice that

- (1) The depth of all nodes does not change in the deletion process, unless we delete the extra root and then the depth of all nodes decreases by 1.
- (2) If the number of children of a node goes under 2, then the algorithm fixes this either by borrowing or merging.

Thus, we conclude that after a deletion all leaves still have the same depth, and every inner node still has 2-3 children ordered correctly.

Remark 5.8 (Complexity of deletion). Deletion requires a search ($\log(h)$ operations) and $\mathcal{O}(1)$ operations on every level of the tree which is $\mathcal{O}(h)$ operations in total. Since 2-3 trees are balanced, we have that $h = \mathcal{O}(\log n)$, so the modified deletion function requires $\mathcal{O}(\log n)$ operations.

Remark 5.9. Insertions, deletions, and lookups all have a time complexity of $\mathcal{O}(\log n)$ as wanted.

Remark 5.10. We could also put values in the inner nodes, and just in the leaves. In the real world, we usually generalize this construction to bigger nodes, while keeping all the data only in the leaves. The generalization is called B-trees.

5.2 B-trees and B+ trees

In the previous subsection, we have introduced 2-3 trees, in which each inner node has either 2 or 3 children. We want to generalize this notion to trees with between min and max inner nodes. These type of trees are called B-trees (if we include data in inner nodes) or B+ trees (if we only include data in leaves).

Remark 5.11. A necessary condition for splitting nodes is $\lfloor (\max+1)/2 \rfloor \geq \min$, and for merging we must have $2\min - 1 \leq \max$. Thus, in order to fulfill these conditions and also have easy lives, we always consider trees with $\min = \lceil \max/2 \rceil$. For example (2,3)-trees (these are 2-3 trees), or (2,4)-trees, or (20,40)-trees, but not (30,55)-trees or (5,7)-trees.

Definition 5.5 (B+ tree). An ordered tree is called a B+ tree if it satisfies all of the following conditions:

- (1) All of the data is kept in the leaves, and all of the leaves have the same depth.
- (2) Every inner node, except possibly the root, has c children where $\lceil m/2 \rceil \leq c \leq m$.
- (3) The root has exactly c children where $2 \leq c \leq m$, unless it is a leaf.
- (4) Every inner node with c children has exactly $c - 1$ keys ordered by size $k_1, \dots, k_{c-1}$.

- (5) All the key k in the i th child of any node satisfies $k \in [k_{i-1}, k_i)$ for all $2 \leq i \leq c - 1$.

Remark 5.12. Lookups, insertions, and deletions in B+ trees are almost identical to lookups, insertions, and deletions in 2-3 trees, and as such, they are also $\mathcal{O}(\log n)$.

Remark 5.13 (Memory and B+ trees). The memory in a computer is managed by two main parts. The first part is called the RAM (random-access memory), which can be accessed very quickly, but cannot store a large amount of information. The second part is called the SSD (solid-state drive), which has slow accessibility, but can store large amounts of information.

When reading/writing to the disc (another term for SSD), we always read/write blocks of memory and not just a single bit every time.

Filesystems and databases utilize the nature of B+ trees, and commonly use them in their implementations, where the nodes are usually very big (the size SSDs read at a time), and low heights.

Since SSDs read chunks of memory at a time, this is a way more natural and efficient way to access information than binary trees.

Remark 5.14 (B+ trees and AVL trees). Both data structure support the dictionary functions in $\mathcal{O}(\log n)$ time complexity, so in theory the difference between them is mostly negligible. However, since B+ allow bigger nodes in the tree, and consequently low heights, they are way more efficient when we implement filesystems and databases.

Remark 5.15 (Data structure invariants). In data structures, an invariant of a structure is a property that is preserved throughout its lifespan, which help us prove other properties on the structure. Here are some examples:

- (1) **Binary search tree invariant:** For any given node, its key is greater than all keys in its left subtree and less than all keys in its right subtree. It allows us to prove lookups, insertions, and deletions can be implemented in $\mathcal{O}(h)$ time complexity.
- (2) **AVL tree invariant:** Every node has a balance factor in the range $[-1, 1]$. It allows us to prove that $h = \mathcal{O}(\log n)$. If we additionally “add” the binary search tree invariant, we would get that lookups, insertions, and deletions can be implemented in $\mathcal{O}(\log n)$ time complexity.
- (3) **2-3 tree invariant:** All leaves are on the same level, and every inner node has 2-3 children. It allows us to prove that $h = \mathcal{O}(\log n)$. If we additionally “add” the binary search tree invariant, we would get that lookups, insertions, and deletions can be implemented in $\mathcal{O}(\log n)$ time complexity.

5.3 Rank trees

Our next immediate goal, is to be able to add functionality to our trees without increasing the complexity of our current operations.

Question. We want to find the median value in a dictionary. Can we do that with our current data structures?

Answer. We can copy all of the data of the dictionary from the tree to an array, sort the array and find the median value. However this is a very expensive solution. Is there a way of doing this in $\mathcal{O}(\log n)$ time complexity without changing the complexity of the standard operations? It turns out, that there is.

Remark 5.16. The most efficient way to implement new operations like “finding the median” in a data structure, is to maintain relevant information in every node.

Definition 5.6 (Rank). Let S be an ordered set and let $x \in S$. Then the rank of x in S is its index in the ordered sequence S .

Example 5.7. The order of 5 in $\{8, 3, 7, 5\}$ is 3 (yes we start counting from 1 not 0).

Remark 5.17. We could now theoretically implement two new functions. The first new function is `rank(T,x)`, which returns the rank of x in T . The second function is `select(T,k)`, which returns the element of rank k in T .

Question. Can we implement a search tree that implements insertions, deletions, lookups, and the new operations rank and select in $\mathcal{O}(h)$, where h is the height of the tree? Can we additionally implement a `sum(T,N)` function that returns the sum of all elements in T smaller than N , in $\mathcal{O}(h)$ time?

Answer. The answer to these questions is yes, however, as mentioned in Remark 5.16, we need to track new information in the nodes.

Definition 5.7 (Rank tree). A rank tree is a tree in which each node v additionally stores the number of nodes in the subtree rooted at v .

Remark 5.18. As you should expect, this new information will allow us to compute the rank of elements in the tree efficiently.

Example 5.8 (Implementation of rank and select). Let T be a rank tree, and denote the number of elements in the subtree rooted at a node u by $n(u)$. Then we can notice that the rank of x in the tree is:

$$\text{rank}(T, x) = \sum_{\substack{\text{nodes on search path} \\ \text{where we go right}}} (1 + n(v \rightarrow \text{left})),$$

and we can compute this in $\mathcal{O}(h)$. In order to implement `select(T,k)` we could apply the following algorithm:

- (1) Let v be the root of T .
- (2) If $n(v \rightarrow \text{left}) = k - 1$ return the root.
- (3) If $n(v \rightarrow \text{left}) > k - 1$, search recursively for the element of rank k in the left subtree of v .
- (4) If $n(v \rightarrow \text{left}) < k - 1$ return the root, search recursively for the element of rank $k - n(v \rightarrow \text{left}) - 1$ in the right subtree of v .

As long as $k < n$ where n is the number of nodes of T , we can easily verify the correctness of the algorithm, and we can also see that we can compute it in $\mathcal{O}(h)$.

Example 5.9 (Implementation of sum). Let T be a rank tree where each node u stores the sum of elements in the subtree rooted at u , denoted by $S(u)$. Then we can notice that if P is the search path to a value x , then the value `sum(T,x)` is simply:

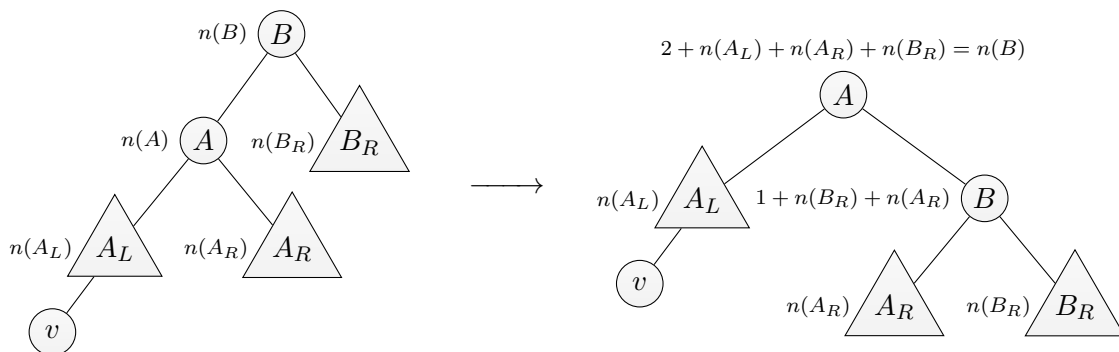
$$\text{sum}(T, x) = \sum_{\substack{\text{nodes on search path} \\ \text{where we go right}}} (v + S(v \rightarrow \text{left})),$$

and we can compute this in $\mathcal{O}(h)$.

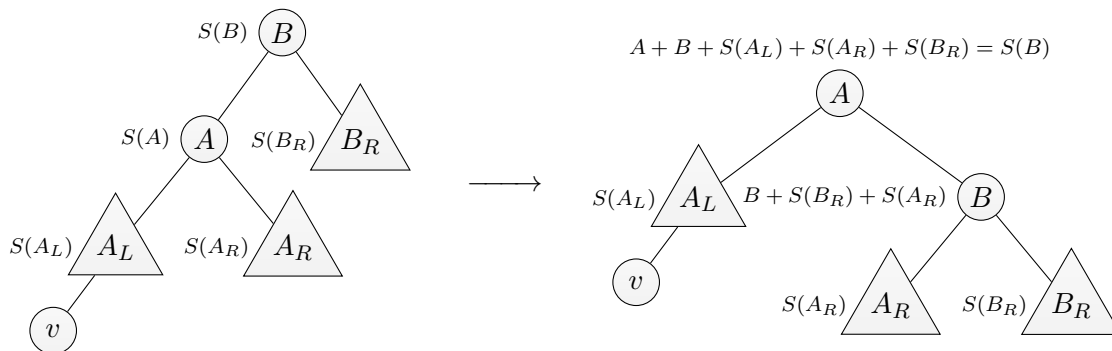
Remark 5.19 (Return the balance to the trees). We now have multiple operations in $\mathcal{O}(h)$, but how can we make sure that they run in $\mathcal{O}(\log n)$ time? The answer is simple, we will use balanced rank trees. For example, we will use an AVL tree where each node additionally stores the number of element in its associated subtree, denoted $n(v)$. Now we only need to make sure we can maintain this new field of information efficiently.

Remark 5.20 (Maintaining fields). In general, when solving data structure problems, we need to add new fields to existing structures. The ways we maintain different new fields is, while computationally different, not inherently different. The value of the field of a node is almost always determined completely by the values of the fields of its children. This important property allows us to maintain the new field efficiently (in $\mathcal{O}(\log n)$ time). For example, consider the field $n(v)$ introduced in Example 5.8. If we want to use it in the final, we would have to justify that we can maintain it in an AVL tree for example. The justification is as follows:

- (1) In a successful insertion (no rotations required), we increase the value of $n(v)$ by 1 in each node on the insertion path.
- (2) In a successful deletion (no rotations required), we decrease the value of $n(v)$ by 1 in each node on the deletion path.
- (3) Since the value of $n(v)$ is determined completely by the values of $n(v \rightarrow \text{left})$ and $n(v \rightarrow \text{right})$, we can maintain $n(v)$ efficiently even in the case of a rotation. For example, in the case of an LL rotation we update the field as follows:



If we want to maintain the field $S(v)$ introduced in Example 5.9, we could similarly maintain it easily when the insertion/deletion does not require rotations, and in the case of rotations, for example, and LL rotation, we update it as follows:



Remark 5.21. In Remark 5.20, there is a great similarity between the implementation of the functions $n(v)$ and $S(v)$ in a rank tree. First we notice that the functions n and S are similar themselves, since both are functions that take a key as input, and return a value. As you may expect, these type of functions are extremely useful, and you will definitely come across exercises that ask you to implement such functions. Luckily, there is a general working scheme for implementing such functions, which we will cover in the following remark.

Remark 5.22 (Adding utilities). Suppose we have a rank tree T , and denote the set of keys already inserted by $K \subset U$ (U is the set of possible keys). Suppose we wanted to implement a new function $\text{ComputeSomething}(x)$ where $x \in U$ (for example, the amount of prime numbers in the tree, lesser than x). In order to implement such a function efficiently we follow the general scheme:

- (1) Find an auxiliary function $f: 2^U \rightarrow \mathbb{Z}$ (we don't have to choose $\mathbb{Z}$ as the codomain, we could choose any set), such that if we knew the value of $f(\{y \mid y < x\})$, we could determine the value of $\text{ComputeSomething}(x)$. In the prime number example, $f(S)$ could be the number of prime numbers in S and T for $S \in 2^U$. By abuse of notation denote:

$$f(x) := f\left(\{v \in T \mid v \text{ is in the subtree rooted at } x\}\right).$$

- (2) In each node we keep and maintain the value of $f(x)$. This can be done so long as there exists an auxiliary function $g: 2^U \rightarrow \mathbb{Z}$, such that for any $S \in 2^U$ such that $S = U_1 \sqcup U_2 \sqcup U_3$, we have that:

$$f(S) = g(f(S_1), f(S_2), f(S_3)).$$

Note that we only care for unions of three sets and not more because of the properties of rotations in AVL trees. In the prime number example

$$f(S_1 \sqcup S_2 \sqcup S_3) = g(f(S_1), f(S_2), f(S_3)) = f(S_1) + f(S_2) + f(S_3).$$

This is very theoretical, so in practice simply mention how we maintain the field $f(x)$ after insertions/deletions/rotations at every node x .

- (3) Now in order to compute $\text{ComputeSomething}(x)$, we can use the properties from steps (1) and (2), such that the computation is done along the search path of x in $\mathcal{O}(\log n)$ time by using g .

In the prime number example, we have that $\text{ComputeSomething}(x) = f(\{y \mid y < x\})$, and also that

$$f(\{y \mid y < x\}) = \sum_{\substack{\text{nodes on search path} \\ \text{where we go right}}} f(v \rightarrow \text{left}) + f(\{v\}),$$

which allows computation of $\text{ComputeSomething}(x)$ in logarithmic time.

Note that $f(x)$ should be a function that we can always compute on the leaves, and it is usually a function that computes some value associated with the subtree rooted at x .

Exercise 5.1. In a planet very distant, not so different from our own, giants and dwarfs live peacefully. However, while the dwarfs are very talkative, the giants don't really understand the dwarfs... The planet is isomorphic to the real line $\mathbb{R}$, and every giant and dwarf has a unique real coordinate of his location. You are tasked with creating a data structure that support the following operations:

<code>Init()</code>	Creates an empty planet as described above. Required time complexity: $\mathcal{O}(1)$.
<code>InsertDwarf(location)</code>	Inserts a dwarf at location <code>location</code> . The location of a dwarf is its identifier and cannot change over time. Required time complexity: $\mathcal{O}(\log n)$.
<code>InsertGiant(location)</code>	Inserts a giant at location <code>location</code> . The location of a giant is its identifier and cannot change over time. Required time complexity: $\mathcal{O}(\log n)$.
<code>IsTalking(L1,L2)</code>	Returns whether two dwarfs at locations <code>L1</code> and <code>L2</code> can talk with each other (two dwarfs can talk if each other if and only if there isn't a giant standing between them). Required time complexity: $\mathcal{O}(\log n)$.
<code>Remove(location)</code>	Removes the creature at location <code>location</code> . Required time complexity: $\mathcal{O}(\log n)$.

Solution. Let T be an AVL based rank tree, which also keeps a field at each node, indicating whether the node is a dwarf or a giant. This structure supports all of the required operations by default except `IsTalking(L1,L2)`.

Note that two dwarfs are talking if and only if there is the same amount of giants to their left. Thus the function we want to compute is `ComputeLeftGiants( $x$ )`. If we have this function, we can implement `IsTalking(L1,L2)` simply by verifying both $L1$, $L2$ are dwarfs, and that `ComputeLeftGiants( $L1$ ) = ComputeLeftGiants( $L2$ )`.

In order to implement this function, we do a similar to what we did before. We let $f: 2^U \rightarrow \mathbb{Z}$ be such that $f(S)$ is the number of giants in S and T , and let $g(S_1 \sqcup S_2 \sqcup S_3) = f(S_1) + f(S_2) + f(S_3)$. Then in order to implement `ComputeLeftGiants( $x$ )`, we simply go along the search path of x and compute the giants to its left by computing $f(\{y \mid y < x\})$ as usual.

5.4 Tries

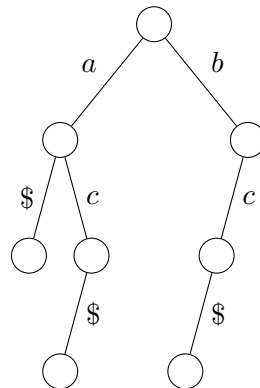
A trie is a data structure that allow insertions, deletions, lookups, and finding the (lexicographical) minimum of strings in an efficient way.

Remark 5.23. The name *trie* comes from the word **re**trieval.

Definition 5.8 (Trie). Let Σ be a set of characters we call the alphabet, and let $\Sigma' := \Sigma \cup \{\$$ be our set of characters. A trie is an ordered tree where every inner node has at most $|\Sigma'|$ children, and every edge has an associated character. Moreover, a node is a leaf if and only if its connected edge holds the character $\$$ which represents the end of a string.

Remark 5.24. The alphabet is almost always the English alphabet $\Sigma = \{a, b, \dots, z\}$ with the usual lexicographical order, and $\$$ is considered the smallest character lexicographically ($\$ < a$).

Example 5.10. A trie which stores the words $\{ac, a, bc\}$ is represented by:



Remark 5.25. Each nodes stores an array of $|\Sigma'|$ pointers, each is an edge representing one character, and in the case of the standard English alphabet it is implmented like this:

```

1  #define SIGMA 26;
2  typedef struct node {
3      struct node[SIGMA + 1] *children;
4  } NODE;
5  NODE *root;

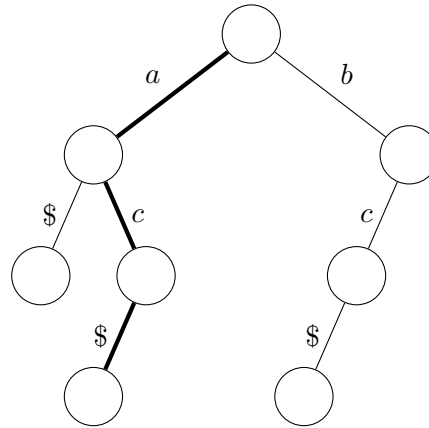
```

Remark 5.26. Any path from the root to a leaf is called a string, and it represents a word. Since all of the classic operations (lookups/insertions/deletions) are done along a path on the tree their time complexity is $\mathcal{O}(|s|)$, where $|s|$ is the length of the path (almost always a string). The space complexity of a trie T is linear in the total number of characters across all stored strings, that is, it is $\mathcal{O}(\sum_{s \in T} |s|)$.

Definition 5.9 (Lookups in a trie). Our goal is to determine whether a string $s = s_1s_2 \dots s_k$ is in the trie. The algorithm (as you should expect) is as follows:

- (1) Initialize $i \leftarrow 1$ and $v \leftarrow \text{root}$.
- (2) Follow the s_i th pointer by $v = v \rightarrow \text{sons}[s_i]$ and increment i by 1. If v is NULL, the string is not present, so return **false**, otherwise continue to step (3).
- (3) If $i \leq k$ return to step (2). If not, then we have that $i = k + 1$, so if the last edge was \$ return **true**, and otherwise return **false** (in this case the string s is a substring of an existing string, but is not in the trie itself).

Example 5.11. A search path for $s = ac$ in the trie $T = \{a, ac, bc\}$ looks like:



Remark 5.27. Notice that a trie is also similar to a B+ tree, since all the data (in this case strings) is “stored” at the leaves. We can think of each leaf as if it stores a word, and then the words stored in the trie correspond exactly to the leaves.

Definition 5.10 (Insertions in a trie). Our goal is to insert a string $s = s_1s_2 \dots s_k$ to the trie. The algorithm (as you should expect) is as follows:

- (1) Initialize $i \leftarrow 1$ and $v \leftarrow \text{root}$.
- (2) Follow the s_i th pointer by $v = v \rightarrow \text{sons}[s_i]$. If v is NULL, create a new node u and set $v \rightarrow \text{sons}[s_i] = u$, then increment i by 1.
- (3) If $i \leq k$ return to step (2). If not, then we have that $i = k + 1$. If $v \rightarrow \text{sons}[0] \neq \text{NULL}$ then the string is already in the trie so we could return. Otherwise, create a new node u , and set $v \rightarrow \text{sons}[0] = u$ (the pointer $v \rightarrow \text{sons}[0]$ is associated with \$), and return.

Definition 5.11 (Deletions in a trie). Our goal is to delete a string $s = s_1s_2 \dots s_k$ from the trie. The algorithm (as you should expect) is as follows:

- (1) Search from the string in the tree while also keeping the last node in the trie with more than one child denoted by v . If not found, there is nothing to delete, so return. If found, continue to step (2).
- (2) Delete the relevant subtree of v .

5.5 Radix trees

Definition 5.12 (Radix). Given a value representation system such as base-10, base-2, or the English alphabet, the radix of the system is the amount of different symbols used to represent values in the system.

Example 5.12. The radix of base-2 is 2, and the radix of the English alphabet is 26.

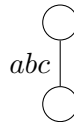
Definition 5.13 (Radix tree). A radix tree (also called a compressed trie), is a data structure similar to a trie, except we merge any node v with a single child with its parent, adding the lost edge's information to the new edge connecting the parent and the child.

Remark 5.28. The node of a radix tree may be implemented as follows:

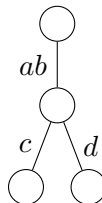
```

1 #define SIGMA 26;
2 typedef struct node {
3     struct node[SIGMA + 1] *children;
4     char[SIGMA + 1] *string;
5 } NODE;
6 NODE *root;
```

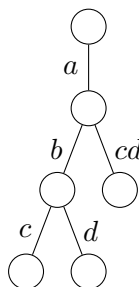
Note that if we insert to an empty tree the words *abc*, *abd*, *ace*, in that order, the radix tree would first compress *abc* into a single edge as follows:



then after inserting *abd* it would have to *split* as follows:



and then after inserting *acd* it would have to split again:



Remark 5.29. Note that in radix trees (actually in tries too) we have some room of freedom in how we choose where to end words. In the implementation used above we didn't have \$ edges. We could either add \$ like we did in tries, or we could add to each node a variable that says whether the string terminates here or not.

Remark 5.30. Let M be the number of nodes in a radix tree, and let ℓ be the number of leaves (which is also the number of strings). Then after the compression each of the $(M - \ell)$ inner nodes has at least 2 children.

Proposition 5.3. *Let $T = (V, E)$ be a tree with M nodes and ℓ leaves, such that every inner node has at least 2 children. Then $M \leq 2\ell - 1$.*

Proof. Since any inner node has at least two children we have

$$2(M - \ell) \leq \sum_{v \in V} d_{\text{out}}(v),$$

where $d_{\text{out}}(v)$ is the amount of edges going “out” (or in the case of trees “down”) of v . Since T is a tree it satisfies $|E| = |V| - 1 = M - 1$, and thus

$$2(M - \ell) \leq \sum_{v \in V} d_{\text{out}}(v) = |E| = M - 1.$$

The inequality $2(M - \ell) \leq M - 1$ implies that $M \leq 2\ell - 1$, which completes the proof. $\square$

Corollary 5.4. *In a radix tree, the number of nodes is linear in the number of leaves (which is the number of strings). In other words, the space complexity of a radix tree is $\mathcal{O}(n)$, where n is the number of nodes in the tree.*

5.6 Exercises

Exercise 5.2. Implement a data structure that supports the following operations:

- Insert(x)** Insert x into the structure.
Time complexity: $\mathcal{O}(\log n)$.
- Delete(x)** Delete x from the structure. Time complexity: $\mathcal{O}(\log n)$.
- Member(x)** Check if x is in the structure. Time complexity: $\mathcal{O}(\log n)$.
- OddSum()** Find sum of all elements with an odd rank.
Time complexity: $\mathcal{O}(1)$.

Solution. We will show a solution using a rank tree based on a 2-3 tree (although we might as well use a rank tree based on an AVL tree). We will also not worry too much about maintaining the values through deletions that require merge and other complicated operations like that, if you want to show that (for example in a test), it is better to use AVL trees since we have shown a general way to do this when working with them.

The **Member(x)** operation works in the standard way for 2-3 trees. In our data structure, each node would maintain three extra values:

- **oddSum** which represents the sum of elements with odd rank in the subtree rooted at that node (the rank is computed with respect to that subtree).
- **evenSum** which represents the sum of elements with even rank in the subtree rooted at that node (the rank is computed with respect to that subtree).
- **n** which represents the size of the tree rooted at that node.

When using **insert(x)** we initialize its values to be **n** $\leftarrow$ 1, **oddSum** $\leftarrow$ x , and **evenSum** $\leftarrow$ 0. Note that when inserting x the only nodes which change their values are on its search path which contains $\mathcal{O}(\log n)$ nodes. We can update the values of each such node by using the following formulas:

```
n <- n(left) + n(middle) + n(right)
oddSum <- A + B + C
```

for $A = \text{odd}(\text{left})$ and:

$$B = \begin{cases} \text{oddSum}(\text{middle}) & n(\text{left}) \text{ is even} \\ \text{evenSum}(\text{middle}) & \text{otherwise} \end{cases}$$

$$C = \begin{cases} \text{oddSum}(\text{right}) & n(\text{left}) + n(\text{middle}) \text{ is even} \\ \text{evenSum}(\text{right}) & \text{otherwise} \end{cases}$$

and `evenSum` is updated similarly. The `delete(x)` operation would work similarly to the standard implementation except we update the nodes on the search path exactly in the same way we updated them for `insert(x)`.

Since the search path is always of length $\mathcal{O}(\log n)$ and we maintain the values in $\mathcal{O}(1)$ time, the time complexity of the insert and delete operations is $\mathcal{O}(\log n)$ as desired, and since we can access `oddSum(root)` in $\mathcal{O}(1)$ time, this completes the exercise.

Exercise 5.3 (Winter 20-21, A, 2, part 1). In response to the outbreak of COVID-19 a group of biologist were given the task to monitor and analyze the amount of vaccinated people in cities indexed by $1, \dots, n$. Since they are not familiar with data structure they ask you politely to help them implement a data structure that supports the following operations:

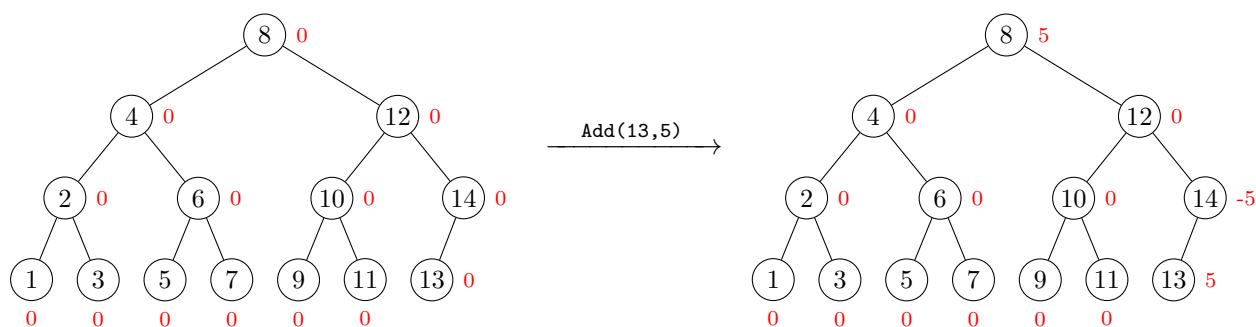
- Init(Cities)** `Cities` is an array of size n that stores the amount of vacciated people. This operations uses that array to initialize the data structure.
Time complexity: $\mathcal{O}(n)$.
- Add(i,j,x)** Add to all cities with an index in the range $[i, j]$ exactly x more vaccinated people.
Time complexity: $\mathcal{O}(\log n)$.
- Find(i)** Return the amount of vaccinated people at the i th city.
Time complexity: $\mathcal{O}(\log n)$.

Solution. We will use an almost complete binary tree which we know we can initialize in $\mathcal{O}(n)$. We can implement `Add(i,j,x)` easily by using `Add(j,x)` then `Add(i-1,-x)` hwere `Add(i,x)` is a command that adds x vaccinated people to all cities in the range $[1, i]$, so it suffices to implement `Add(i,x)` instead. The idea is as follows: In each node in the tree (which represents a city) we will maintain a field called `extra`, and we will compute the amount of vaccinated people in city i by summing the `extra` values on its search path. The implementation details are as follows:

- Init(Cities)** Construct an almost complete binary tree for n cities in $\mathcal{O}(n)$, then perform an inorder traversal over the cities and add a field called `extra` initialized to be `Cities[i]`.
- Add(i,x)** Search for i starting from the root. In each sequence of right turns in the path we add x to the `extra` field of the first node in that sequence. If such a sequence is terminated by a left turn from a node j , we decrease the `extra` value of j by x (since it is bigger than i). When we reach the node we searched for, if we didn't reach it after a sequence of right turns we increase its `extra` value by x , and in any case we decrease the `extra` value of its right child (if it exists).
- Find(i)** We search for i and sum the `extra` values of the nodes on the search path.

Here is an example of how `Add(i,x)` works with the assumption all cities start with 0 vaccinated

people and the **extra** values marked in red:



Remark 5.31. Exercise 5.3 appeared in a past test, and the part we solved in these notes is the easier of the two. A brave reader can try to solve the second part which is to additionally implement the following operation:

$\text{NumOfVaccinated}(i, j)$ Returns the total number of vaccinated people in cities in the range $[i, j]$.
Time complexity: $\mathcal{O}(\log n)$.

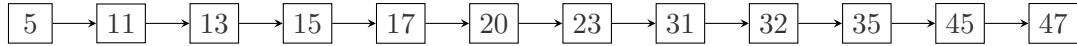
Note that this operation can be implemented easily if we implement $\text{NumOfVaccinated}(i)$ which returns the total number of vaccinated people in cities in the range $[1, i]$. This part of the question is a better representation of the actual difficulty level of questions in this course. Unfortunately, we do not solve many questions of such difficulty level in these notes, because its goal is to teach the basic concepts.

6 Skip lists and amortized complexity

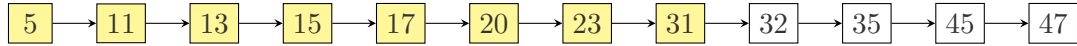
6.1 Skip lists

We have previously discussed all types of lists and their advantages over arrays. However, one clear disadvantage of a list compared to an array is that lookups take $\mathcal{O}(n)$ time, even when the list is sorted. In order to improve that, we have to add additional structure to the list, that allows *skipping* over large parts of it. This is the only way to overcome the $\mathcal{O}(n)$ lookup complexity barrier. There is no “going one by one in a smart way”. Lists that implement this type of “skipping” are called skip lists.

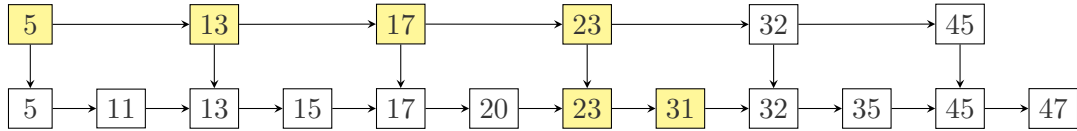
Example 6.1 (Layers). As mentioned, in a standard list, for example:



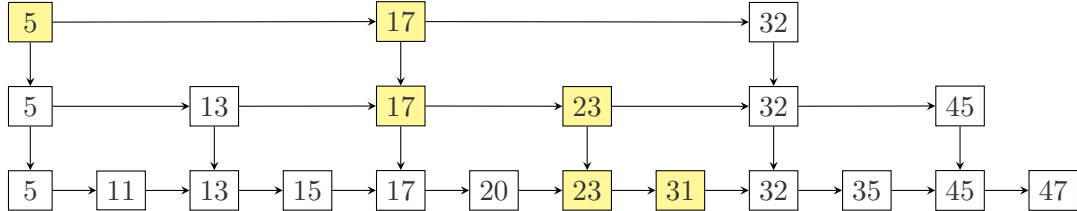
insertions, deletions, and lookups all take $\mathcal{O}(n)$ time. For example if we look for 31, we would have to go through the nodes one by one as follows:



However, we can improve on this tremendously, if we add a sort of “express lane”, that is, if we add another layer, that is a copy of our linked list, except it only holds every second node. In this case, the search path of 31 would be:



and as we see, this is a much shorter search path. Similarly, we can get better results with each layer we add, for example if we add a third layer, with every second node of the previous layer, we could get to 31 even faster:

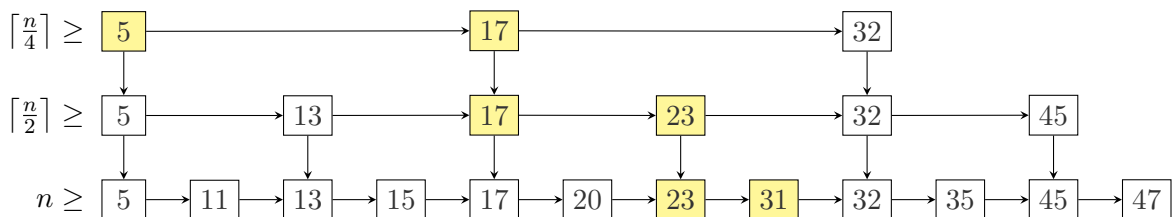


and we could keep going like this ad infinitum, except that layers after the $\lceil \log n \rceil$ layer would only hold the first element of the list, so we could stop there, or maybe even stop one layer before it.

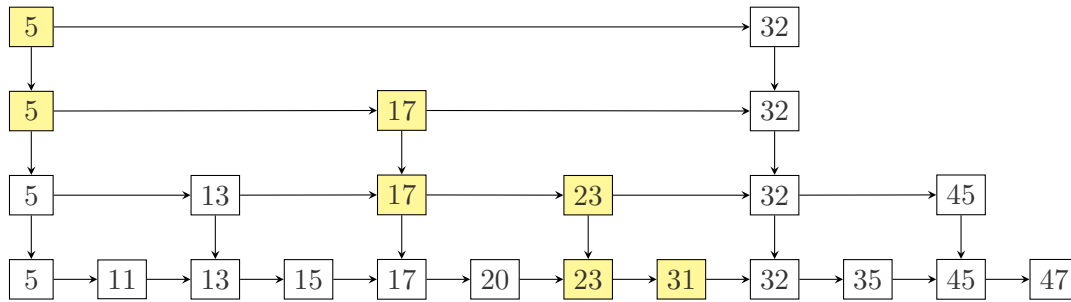
Remark 6.1 (Space complexity of skip lists). In a skip list as described before, with $\lceil \log n \rceil$ layers, the number of nodes we need is

$$n + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{4} \right\rceil + \cdots + 1 \leq n + \left(\frac{n}{2} + 1 \right) + \left(\frac{n}{4} + 1 \right) + \cdots + 1 \leq 2n + \log n = \mathcal{O}(n),$$

and therefore the space complexity of such a skip list is $\mathcal{O}(n)$. Here is an auxiliary diagram:



Remark 6.2 (Time complexity of skip lists). Consider the previous search path for 31 in the previous skip list with four layers:



First, it might be worth to mentioend that the searching algorithm for k can be stated as “go down unless the number to your right smaller than k in which case go right”. We could also determine the search path backwards by (starting from k) “go up unless there is not way up in which case go left”.

Since the length of the path top to bottom is the same length of the path bottom to top, we can count the nodes traveled bottom to top according the the previously mentiond “reverse algorithm”, and find out that any search path has at most $2 \lceil \log n \rceil = \mathcal{O}(\log n)$ steps.

Remark 6.3 (Oh oh). There is a critical problem in our current implementation of a skip list. When we try to insert or delete nodes, it is inevitable that we would have to restructure all of the layers from scratch and that is clearly not efficient complexity wise. Luckily, we can solve this problem by using randomized algorithms.

Definition 6.1 (Randomized algorithm). A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. In our course, we could equivalently define a randomized algorithm, as an algorithm that has at least one random variable as a parameter.

Definition 6.2 (Las Vegas algorithm). A randomized algorithm where the output is *always* correct, but the running time depends on the random variables are called Las Vegas algorithms.

Example 6.2. One example of a Las Vegas algorithm is the following:

Algorithm 1: Las Vegas exact minimum

Input: Array $A[1 \dots n]$

Output: $\min(A)$

```

1 Generate a random permutation  $\pi$  of  $\{1, \dots, n\}$ 
2  $m \leftarrow A[\pi(1)]$ 
3 for  $j \leftarrow 2$  to  $n$  do
4   if  $A[\pi(j)] < m$  then
5      $m \leftarrow A[\pi(j)]$ 
6   end
7 end
8 return  $m$ 
```

Here the random variable is $X: S_n \rightarrow \mathbb{R}$, which we assume to be uniform. Clearly, the output will always be correct, however, certain permutations would certainly make the algorithm run faster than other permutations.

Definition 6.3 (Monte Carlo algorithm). A randomized algorithm where the output may sometimes be incorrect, but we can bound the probability of an incorrect output (usually at the cost of running time) is called a Monte Carlo algorithm.

Example 6.3. One example of a Monte Carlo algorithm is the following:

Algorithm 2: Monte Carlo approximate minimum

Input: Array $A[1 \dots n]$, sample size k

Output: Approximation of $\min(A)$

- 1 Select k indices $i_1, i_2, \dots, i_k$ uniformly at random from $\{1, \dots, n\}$
 - 2 $m \leftarrow \min\{A[i_1], A[i_2], \dots, A[i_k]\}$
 - 3 **return** m
-

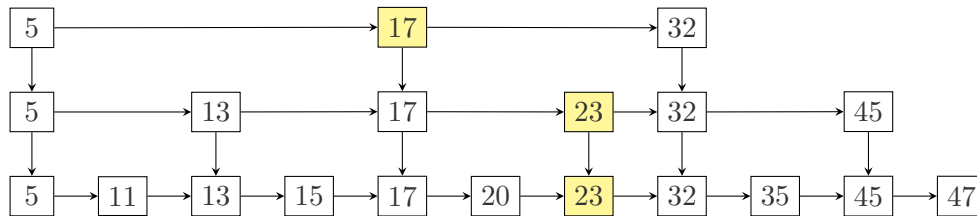
In this algorithm, variable is given by $X: A \rightarrow \mathbb{R}$, where $A \subset 2^{\{1, \dots, n\}}$ is the set of subsets of $\{1, \dots, n\}$ of size k . It is also given that X is distributes uniformly. Clearly, the output may be incorrect, but we can bound the probability of an incorrect output by increasing k .

Definition 6.4 (Insertion in a skip list). The insertion algorithm for a key k in the randomized (and from now on standard) implementation of a skip list is as follows:

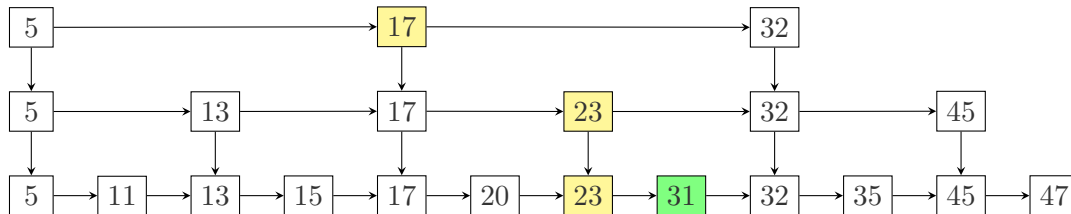
- (1) Search for k in the list, while also keeping pointers to the rightmost node at every level of the search path. If k is found return, and otherwise continue to step (2).
- (2) Add a new node at the lowest layer of the list with the inserted value k .
- (3) Flip a random coin (this can be implemented using the various **random** functions programming languages offer).
- (4) If the result of the coin flip is tails, then return. If the result is heads, create a new node with the value k , and insert it one level above the lowest layer. Note that we can only do this by using the pointers we kept in step (1). Keep flipping the coin and repeat this process (insert k on the second, third additional layers and so on) until flipping tails.

Note that if we flipped heads at the top layer, we need to create a new layer (this isn't a problem).

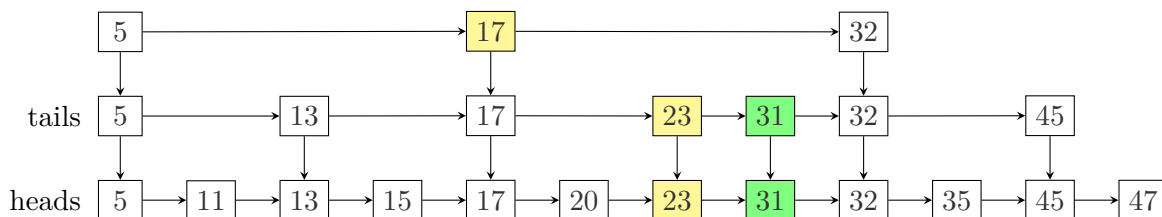
Example 6.4. Suppose we wanted to insert 31 into the following skip list:



where the highlighted nodes are the pointers we kept in order to possibly add 31 to their layer later. Then we add 31 to the bottom list:



and now we start flipping coins and so long as we get heads we keep adding 31 to the layers above:



where the results of the coin flips at each layer are written on the left side. Note that the result at the coin flip at the i th layer is relevant only to the $i + 1$ th layer, and that we always insert k (in this case 31) to the lowest layer.

Remark 6.4 (Generating randomness). In different programming languages there are many different methods that return a “random” number. For example the functions `Math.random`, and `java.util.Random` in Java, or `rand()` from `stdlib.h` in C.

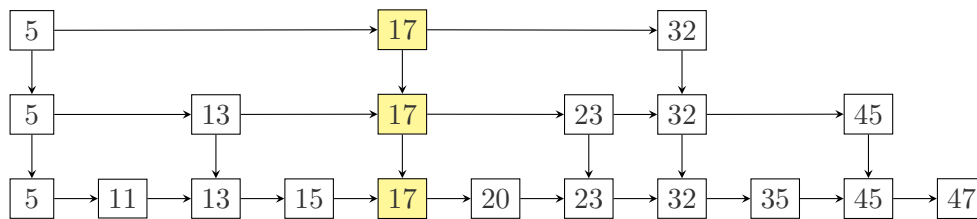
The implementation of these functions usually depends on a value that is “practically random”, like the least significant bit in the binary value of the time today (the computer’s inner clock also counts milliseconds, so this is very “random”), or something similar.

When we need “absolute randomness” we usually need to obtain special physical parts for the computer that allow us to get random values in a uniform and independent way.

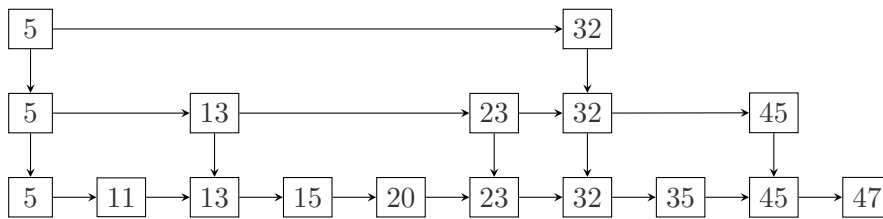
Definition 6.5 (Deletion in skip lists). In a standard skip list, the algorithm for deleting k is quite simple:

- (1) Search for k in the list. If not found, return. If found, continue to step (2).
- (2) Delete k from all the levels in which it appears.

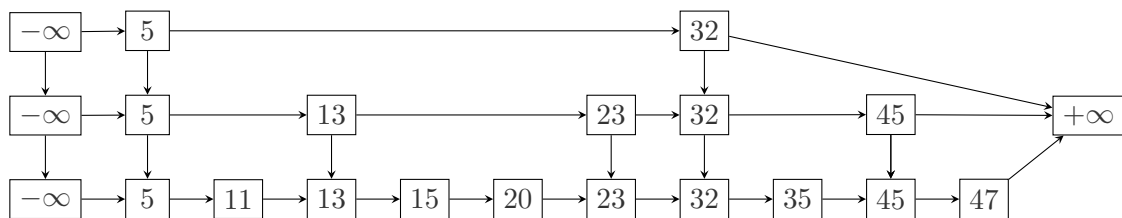
Example 6.5. Suppose we wanted to delete 17 from the following list:



where the highlighted nodes are the the part of the search path for 17 which with the value 17 (by the construction of the skip list these must be all nodes with the value 17 in the entire skip list). We now proceed to delete these nodes from bottom to top and remain with the following skip list:



Remark 6.5. When implementing skip lists it is also very common to add an additional node at the beginning of every layer with the value $-\infty$ and one node that connects to the end of each layer as follows:



We can also implement insertions, deletions, and lookups recursively (or not recursively), and some of these implementations are available in the course site’s material, however we *skip* discussing the particular implementation details of these functions.

Proposition 6.1. *The expected space complexity of a standard skip list storing n elements is $\mathcal{O}(n)$.*

Proof. First, we need to prove a lemma.

Lemma 6.2. *Suppose we flip a standard fair coin repeatedly until the first time we flip tails, and then we stop. Let Y be the number of times we flipped the coin. Then $\mathbf{E}[Y] = 2$.*

Proof. Since the coin flips are independent Bernoulli trials with probability $p = 0.5$ of success, it is clear that Y is a geometric variable with parameter $p = 0.5$, so we can write $Y \sim \text{Geo}(0.5)$. It is well known that $\mathbf{E}[Y] = \frac{1}{0.5} = 2$, but we can also compute manually that

$$\mathbf{E}[Y] = \sum_{i=1}^{\infty} i \cdot \mathbf{P}(Y = i) = \sum_{i=1}^{\infty} i \cdot p^{i-1}(1-p) = (1-p) \sum_{i=1}^{\infty} i \cdot p^{i-1} = \frac{1-p}{(1-p)^2},$$

where the last transition follows from the derivation of the series expansion of the function $\frac{1}{1-x}$. In our case

$$\mathbf{E}[Y] = \frac{1-p}{(1-p)^2} = \frac{1}{(1-p)} = \frac{1}{(1-\frac{1}{2})} = 2,$$

which completes the proof of the lemma. $\square$

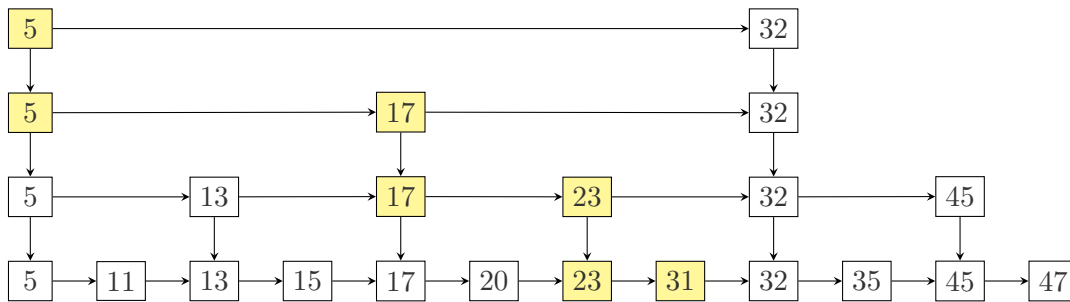
Let S denote the amount of total nodes in the tree, let L_i denote the amount of total nodes in the i th tower for all $1 \leq i \leq n$. We can write $S = \sum_{i=1}^n L_i + 1$, where 1 is for the ∞ node. Then by Theorem 6.2 and linearity of expectation we get

$$\mathbf{E}[S] = \mathbf{E}\left[\sum_{i=1}^n L_i + 1\right] = \sum_{i=1}^n \mathbf{E}[L_i] + 1 = 2n + 1 = \mathcal{O}(n),$$

which completes the proof. $\square$

Proposition 6.3. *Let L be the expected length of a search path in a skip list with n elements. Then $L \leq 2 \log_2(n) + 2$.*

Proof. Consider the path in reverse:



We always go up when possible, and left otherwise. Let $c(k)$ denote the expected length of a path which includes exactly k points where we go up (note that this is equivalent to a path from layer $M+1$ to layer 1 since every time we go up we go up by exactly one layer). Then, since at each insertion an element has a 50% chance of creating a copy of itself in the second layer, and a 50% chance of not doing that, we can conclude that for a path ending at some node v , there is a 50% chance going left and a 50% chance going right. This implies the following inequality:

$$c(k) \leq \frac{1}{2}(1 + c(k)) + \frac{1}{2}(1 + c(k-1)),$$

where the $+1$ come from the step we took either left or up. This implies that $c(k) \leq 2 + c(k-1)$, and thus

$$c(k) \leq 2 + c(k-1) \leq 4 + c(k-2) \leq \dots \leq 2k.$$

This implies that the expected length of a path from layer M to layer 1 is at most $2(M-1)$. We know that M distributes like the maximum over n geometric variables with parameter 0.5. It can be shown that $\mathbf{E}[M] \leq \log_2(n) + 2$, and therefore

$$L \leq c(\log_2(n) + 2) \leq 2(\log_2(n) + 2 - 1) = 2\log_2(n) + 2,$$

which completes the proof. $\square$

Remark 6.6. In the real world we bound the amount of possible layers by an amount close to $\log_{1/p} n$. Also, in the real world we usually let the probability to go up a layer be less than $p = 0.5$, since p controls the amount of layers quite strictly, and sometimes we would rather have less layers (save space), instead of decreasing the running time.

Question. What is the worst possible sequence of coin flipping for a skip list?

Answer. Flipping heads 100000 times in a row would be pretty bad. Never flipping heads would result in a standard linked list which is also bad.

Corollary 6.4. *The expected time complexity of the search, insertion, and deletion operations in a standard skip list with n elements is $\mathcal{O}(\log n)$.*

Proof. In Theorem 6.3 we have seen that length of search paths are $\mathcal{O}(\log n)$. Since in every one of the standard operations we run $\mathcal{O}(1)$ commands at every node in the path, we get that their time complexity is $\mathcal{O}(\log n)$. $\square$

Remark 6.7. In we let the probability of flipping heads be some arbitrary $0 < p < 1$ instead of $p = 0.5$, then we could get similarly conclude that $L \leq 2\log_{1/p}(n) + 1/p$, and thus that the operations can be done in $\mathcal{O}(\log_{1/p} n) = \mathcal{O}(\log n)$. In order to save space it is also common to choose $p = 0.25$.

Remark 6.8. Skip lists are a popular alternative for self balancing trees because of their ease of computation

Example 6.6 (Deterministic skip lists). So far, we have only worked with probabilistic skip lists. But we can also construct deterministic skip lists which are based on the principle that there cannot be three consecutive node on the i th without nodes right above them in the $i+1$ th layer. These type of trees are not very complex, and correspond to the Knuth transform (which we discussed in Example 3.17) of 2-3 trees exactly. That is, each Knuth transform of a 2-3 tree can be easily made into a deterministic skip list. These type of lists are not very relevant to our course so we will *skip* them as well.

6.2 Amortized complexity

We have seen multiple types of complexity so far:

Worst-case complexity: What is the worst time that the program could take for an input of size n ? Formally defined by:

$$T_{\text{worst}}(n) := \max_{|x|=n} T(x).$$

We usually care for its upper bound, and only analyze average-case complexity if it has better results.

Average-case complexity: What is the average time that the program would take for an input of size n ? This is the most common type of complexity we have seen so far, however this

is a slightly less meaningful notion of complexity, since it assumes that the input is uniformly distributed, while in real life it may be controlled by an “adversary” that always gives us bad input. It is formally defined by:

$$T_{\text{avg}}(n) := \mathbf{E}_{x \sim D_n}[T(x)] = \sum_{|x|=n} T(x) \mathbf{P}(x),$$

where D_n is the distribution of inputs of size n .

Expected complexity (of a randomized algorithm): We first encountered this type of complexity in the last subsection when we talked about skip lists. For every input x of size n , the algorithm “flips coins” with a certain distribution, and we ask what is the expected running time over the algorithm internal randomness. This notion of complexity is stronger than average-case complexity, since even if an adversary saw our algorithm it could not give us bad input because he cannot know the future outcome of the coin flippings. Formally, we define:

$$T_{\text{exp}}(x) := \mathbf{E}_r[T(x, r)],$$

where r is the randomness factor of the algorithm. As mentioned, randomized algorithms can be thought of as algorithms that also take random variables as parameters, so r can be thought of as one or more random variables with different distributions.

Amortized complexity: We have not encountered this type of complexity yet. This type of complexity becomes useful when a sequence of n operations is executed in succession, and each of them has a different running time. For example if we execute $n - 1$ operations in $\mathcal{O}(1)$ each, then 1 operation in $\mathcal{O}(n^2)$, we would like to know what is the expected running time of a random (uniformly chosen) operation out of the sequence.

Definition 6.6 (Amortized complexity). Let $m = (m_1, \dots, m_k)$ be a sequence of k operations, and let $T(m) = \sum_{i=1}^k T(m_i)$ be the total time it takes to run them in succession. Then the amortized complexity of m is defined by:

$$T_{\text{amort}}(m) := \frac{T(m)}{k} = \frac{\sum_{i=1}^k T(m_i)}{k}.$$

When the sequence is obvious (for example we run the same command k times in a row) we can also denote

$$T_{\text{amort}}(k) := \frac{\sum_{i=1}^k T(m_i)}{k}.$$

Definition 6.7 (Amortized complexity, sets). Let F be a set of operations. We say that F runs in time complexity $\mathcal{O}(g(n))$ if for (almost) any sequence of operations $m = (m_1, \dots, m_k) \in F^k$ we have that $T_{\text{amort}}(m) = \mathcal{O}(g(n))$ or equivalently $\sum_{i=1}^k T(m_i) = \mathcal{O}(k \cdot g(n))$.

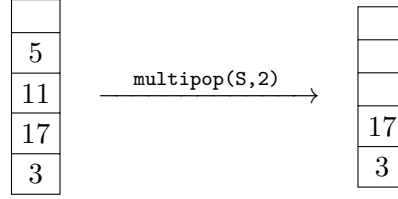
Remark 6.9. In Definition 6.7 we had the equation $\sum_{i=1}^k T(m_i) = \mathcal{O}(k \cdot g(n))$ which may look weird since the left hand side has no seeming dependency on n , while the right hand side does. Indeed, we made a slight simplification and were not rigorous. The value of n is defined to be the maximum size of the structure (or input) throughout m . We will mostly use Definition 6.6 so leaving Definition 6.7 slightly not very well defined is fine.

Remark 6.10. We are mostly interested in finding an upper bound for $\frac{1}{n} \sum_{i=1}^n T(m_i)$. The nature of the proofs is not going to be inherently different from the proofs for upper bounds of normal worst-case running times, however, there are three new methods which we will learn that will be very helpful to us when dealing with sums of the form $\frac{1}{n} \sum_{i=1}^n T(m_i)$ specifically.

Definition 6.8 (Multipop stack). We define a new data type which is similar to the stack introduced in Example 1.2, with one additional operation:

multipop(S,k) - this operation performs the operation **pop(S)** k times in a row, or until the stack is empty.

We also assume that each of the standard operations on the stack takes $\mathcal{O}(1)$ time, so that the new operation **multipop(S,k)** takes $\mathcal{O}(\min(k, s))$ time, where s is the amount elements currently in the stack. Here is an example of multistack being used:



Example 6.7 (Worst-case complexity). Let S be the multipop stack defined in Definition 6.8, and suppose that S is currently empty. If m is sequence of k operations, then since the most heavy operation takes about k time (**multipop(S,k)** after inserting $k - 1$ elements), and since there are k operations in the sequence, we could bound $\sum_{i=1}^k T(m_k)$ by $k \cdot k = k^2$. Thus $T(m) = \mathcal{O}(k \cdot n)$ or equivalently $T_{\text{amort}}(m) = \mathcal{O}(n)$. This method is somewhat useful, but can do much better with the following three amortized complexity methods.

Definition 6.9 (Aggregate analysis). The aggregate method is a very straightforward method. We compute the bound of $\frac{1}{k} \sum_{i=1}^k T(m_k)$ directly by some function $g(n)$, and then we get $T_{\text{amort}}(m) = \mathcal{O}(g(n))$.

Example 6.8 (Aggregate analysis). Let S be the multipop stack defined in Definition 6.8, and suppose that S is currently empty. We will find the amortized complexity of a sequence of k operations on S . In every sequence of k operations there are exactly k_1 calls for push, and k_2 calls of pop (we consider **multipop(S,t)** as however many **pop(S)** operations is calls). Note that $k_1 + k_2$ is not necessarily equal to k or less than k or anything of the sort. However, since the sequence with the most amount of push has exactly k calls for push, and the sequence with the most amount of pop has exactly $k - 1$ calls for pop, we have

$$T_{\text{amort}}(m) = \frac{\sum_{i=1}^k T(m_k)}{k} = \frac{k_1 + k_2}{k} \leq \frac{(k) + (k - 1)}{k} \leq 2,$$

and thus $T_{\text{amort}}(m) = \mathcal{O}(1)$.

Definition 6.10 (The accounting method). The accounting (bank) method is a method in which we treat time as currency. That is, for a given sequence of operations $m = (m_1, \dots, m_k)$, we give ahead of time each operation m_i an amortized cost a_i , and denote the time it takes by $T(m_i) := t_i$, which we also treat as currency. The setting is that at every operation we pay a_i , and note to ourselves that the cost was t_i , so we add to our credit $a_i - t_i$ (intuitively, if this is positive we paid more than required at the i th step). Then after we finish executing all of the operations, we check the total balance of our credit. If the balance is positive, we paid more than enough, which intuitively means that the amortized costs a_i subsidized the actual costs of the operations t_i . Formally, we get that $\sum_{i=1}^k (a_i - t_i) \geq 0$, and therefore,

$$\sum_{i=1}^k T(m_i) = \sum_{i=1}^k t_i \leq \sum_{i=1}^k a_i,$$

so we found a bound for the amortized complexity $T_{\text{amort}}(m) = \mathcal{O}(\sum_{i=1}^k a_i/k)$ (here the a_i are usually a function in n , and usually most are constant functions so they are not even dependent on n).

Remark 6.11. This is a good time to make sure you understand the concept of amortization in finance. It will help you understand amortized complexity.

Example 6.9 (The accounting method). Let S be the multipop stack defined in Definition 6.8, and suppose that S is currently empty. We will find the amortized complexity of a sequence of k operations on S . When solving a problem with the accounting method, we first start by computing the actual cost of operations t_i . It is clear that $t_{\text{pop}()} = 1$ (or 0 when S is empty), $t_{\text{push}(x)} = 1$, and $t_{\text{multipop}(t)} = \min(t, |S|)$ where $|S|$ is the current size of the stack. We define the amortized cost of $\text{push}(x)$ to be $a_{\text{push}(x)} = 2$, and set the amortized costs of all the other operations to 0. We do this, because for each $\text{pop}()$ operation that costs 1, there had to be exactly one corresponding $\text{push}(x)$ operation that had cost exactly 1. Intuitively, the amortized cost of the $\text{push}(x)$ method subsidizes the cost of a future $\text{pop}()$ method. Thus, our balance will always be positive so

$$\sum_{i=1}^k t_i \leq \sum_{i=1}^k a_i \leq 2k,$$

and therefore $T_{\text{amort}}(m) = \mathcal{O}(2k/k) = \mathcal{O}(1)$.

Definition 6.11 (The potential method). In the potential method, we treat the data structure as if it has “potential” energy (yes, like in physics), and we do that by defining a potential function Φ that assigns each state of the data structure $\text{state}(D)$ the potential $\Phi(\text{state}(D)) \in \mathbb{R}$. Intuitively, the higher the potential of a state of a data structure is, the more it is vulnerable to heavy “fines” by costly operations. For example, in the multipop stack defined in Definition 6.8, we want to think of S at state D as having a potential roughly (or even exactly) equal to the number of elements in it. That is because the more elements it stores, the higher the “fine” we could potentially get by $\text{multipop}(S, k)$.

In order to apply this notion of potential to our problem (finding amortized complexity), we first denote the actual costs of operations in t_i , and set the amortized cost to be $a_i := t_i + \Phi(D_i) - \Phi(D_{i-1})$, where D_i is the state of the data structure after i operations. Intuitively, for each operation, we pay the cost of the operation, then additionally we pay for the potential of the structure caused by the i th operation (this is a pay “in case of” fine in the $(i+1)$ th step), and we add to our balance the potential of the structure after the previous step (this is necessary since this is what prevents us from paying twice for the same potential).

Finally, notice that we get that if $m = (m_1, \dots, m_k)$ is a sequence of m operations we get

$$\sum_{i=1}^k a_i = \sum_{i=1}^k (t_i + \Phi(D_i) - \Phi(D_{i-1})) = \left(\sum_{i=1}^k t_i \right) + \Phi(D_m) - \Phi(D_0).$$

Since the sequence of operations m is almost surely assumed to start right after the initialization of the structure, $\Phi(D_0)$ is the potential of the structure after initialization, so it is natural to define Φ such that $\Phi(D_0) = 0$. Thus, if we make sure that $\Phi(D_m) \geq 0$ for any possible state of the data structure, we get that

$$\sum_{i=1}^k t_i \leq \left(\sum_{i=1}^k t_i \right) + \Phi(D_m) \leq \sum_{i=1}^k a_i,$$

and thus $T_{\text{amort}}(m) = \mathcal{O}(\sum_{i=1}^k a_i/k)$.

Example 6.10 (The potential method). Let S be the multipop stack defined in Definition 6.8, and suppose that S is currently empty. We will find the amortized complexity of a sequence of k operations on S . The potential function $\Phi(\text{state}(S))$ is defined to be the current number of elements in the stack. It is clear that $\Phi(D_0) = 0$, and that $\Phi(\text{state}(S)) \geq 0$ for any state of S . We now remember that the real costs of the operations are $t_{\text{pop}()} = 1$ (or 0 when S is

empty), $t_{\text{push}(x)} = 1$, and $t_{\text{multipop}(t)} = \min(t, |S|)$ where $|S|$ is the current size of the stack, and compute the amortized costs by the potential function:

$$\begin{aligned} a_{\text{push}(x)} &= t_{\text{push}(x)} + (\Phi(D_i) - \Phi(D_{i-1})) = 1 + (1) = 2 \\ a_{\text{pop}()} &= t_{\text{pop}()} + (\Phi(D_i) - \Phi(D_{i-1})) = \min(1, 0) - \min(1, 0) = 0 \\ a_{\text{multipop}(t)} &= t_{\text{multipop}(t)} + (\Phi(D_i) - \Phi(D_{i-1})) = \min(t, |S|) - (\min(t, |S|)) = 0, \end{aligned}$$

where for example the $\min(1, 0)$ operation is not actually taking the minimum between 1 and 0, rather it returns 0 in both cases if the stack is empty, and 1 in both cases otherwise (remember we define $t_{\text{pop}()} = 0$ when S is empty). We now conclude that

$$\sum_{i=1}^k t_i \leq \sum_{i=1}^k t_i + (\Phi(D_m) - \Phi(D_0)) = \sum_{i=1}^k a_i \leq 2k,$$

and thus $T_{\text{amort}}(m) = \mathcal{O}(2k/k) = \mathcal{O}(1)$.

Remark 6.12 (Coincidence? I think not). Note that both in Example 6.9 and Example 6.10 we got the same amortized costs. Is this a coincidence? Well, no. While it is true that we could have chosen a different potential function, then get different amortized costs and still arrive at the same solution, there actually is a connection between both methods. The connection reveals itself when we consider the potential as “current bank balance”, or equivalently consider the “current bank balance” as potential. Note that while both methods are equivalent, sometimes it is more convenient to use one than the other, so it is important to know both.

Exercise 6.1 (Binary counter problem). Let A be a binary counter with k bits. A represents numbers by keeping the value of the bits in an array $A = a_{k-1}a_{k-2} \dots a_0$, so that the value of A is the value of $(a_{k-1}a_{k-2} \dots a_0)_2$, for example when $k = 3$ we get $(100)_2 = 8$ and so on. The counter is initialized with the value 0. The main operation of A is the **Increment(A)** operation with the following implementation:

```

1 void Increment(A) {
2     int i = 0;
3     while (i < length(A) and a_i == 1) {
4         a_i = 0;
5         i = i + 1;
6     }
7     if (i < length(A)) {
8         a_i = 1;
9     }
10 }
```

where $\text{length}(A) = k$ in our original notation. For example if $A = 00100111$ then after **Increment(A)** we get $A = 00101000$. We define that **Increment(A)** costs the amount of bits that it changed. Find the amortized complexity of a sequence $m = (m_1, \dots, m_r)$ of r calls for **Increment(A)** using all three methods (aggregate analysis, the accounting method, and the potential method). You may assume that A has just been initialized.

Solution. First, we notice that each operation can flip at most k bits, and therefore the amortized complexity of m is clearly $T_{\text{amort}}(m) = \mathcal{O}(k)$. However, this bound is not strict, and we can show this using three methods.

- (1) First we will use aggregate analysis. Note that not all bits flip during each call to **Increment(A)**. In fact:

- The bit a_0 flips every single call to **Increment(A)**.

- The bit a_1 flips every two calls to **Increment**(A).
- The bit a_2 flips every four calls to **Increment**(A).
- ...
- The bit a_i flips every 2^i calls to **Increment**(A).

This shows that a_i flips exactly $\left\lfloor \frac{r}{2^i} \right\rfloor$ times. Thus, we have that

$$\sum_{i=1}^r T(m_i) = \sum_{i=0}^{k-1} \left\lfloor \frac{r}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{r}{2^i} \leq \sum_{i=0}^{\infty} \frac{r}{2^i} = 2r,$$

and therefore $T_{\text{amort}}(m) = \mathcal{O}(2r/r) = \mathcal{O}(1)$.

- (2) We will now use the accounting method. Note that every bit flipped from 1 to 0, must have previously been flipped from 0 to 1. Similarly, nodes flipped from 1 to 0 exactly s times throughout m must have been flipped at least s times from 0 to 1 throughout m . We know that the cost of each m_i is

$$t_i := T(m_i) = \sum_{i=0}^{k-1} \mathbb{1}_{i\text{th bit flipped}}.$$

We set the amortized cost of m_i to be

$$a_i := 2 \sum_{i=0}^{k-1} \mathbb{1}_{i\text{th bit flipped from 0 to 1}}.$$

It follows from the previous explanation that the amortized cost necessarily suffices to subsidize the actual cost, so we have $\sum_{i=1}^r t_i \leq \sum_{i=1}^r a_i$. Note that at every execution of **Increment**(A), at most one bit flips from 0 to 1, and therefore

$$\sum_{i=1}^r a_i = 2 \sum_{i=1}^r \sum_{j=0}^{k-1} \mathbb{1}_{i\text{th bit flipped from 0 to 1}} \leq 2 \sum_{i=1}^r 1 = 2r,$$

Therefore, we obtain that $T_{\text{amort}}(m) = \mathcal{O}(2r/r) = \mathcal{O}(1)$.

- (3) We will now use the potential method. Define the potential function $\Phi(\text{state}(A))$ be the current number of 1s in the counter A at $\text{state}(A)$. In this way both $\Phi(D_0) = 0$ and $\Phi(D_m) \geq 0$ for all m . The intuition behind the definition of Φ , is that **Increment**(A) can only flip as many bits as there are 1s in A (maybe +1 bit that can be flipped to 0), so the more 1s there are, the more A is prone to “fines”, or equivalently, it has more potential. We already saw in (2) what the value of t_i is, so our next step is to compute a_i for all $1 \leq i \leq r$. Denote by z_i the number of bits that flip from 1 to 0, so that $t_i \leq z_i + 1$. This inequality implies that $\Phi(D_i) \leq \Phi(D_{i-1}) - z_i + 1$. Therefore, by using these two inequalities we can obtain that

$$a_i = t_i + (\Phi(D_i) - \Phi(D_{i-1})) \leq (z_i + 1) + (1 - z_i) = 2.$$

This, together with our previous conclusions that $\Phi(D_0) = 0$ and $\Phi(D_m) \geq 0$ for all m , suffices to show that $\sum_{i=1}^r t_i \leq \sum_{i=1}^r a_i = 2r$, and thus $T_{\text{amort}}(m) = \mathcal{O}(2r/r) = \mathcal{O}(1)$.

6.3 Exercises

Exercise 6.2. Implement a stack that supports the following functions:

- Init(*k*)** Initializes the stack with parameter *k*.
Time complexity: $\mathcal{O}(1)$.
- Push(*x*)** Inserts *x* into the top of the stack for $x \leq k$.
Time complexity: $\mathcal{O}(1)$ amortized together with **Pop()**.
- Pop()** Deletes the element on top of the stack. Time complexity: $\mathcal{O}(1)$ amortized together with **Push(*x*)**.
- Delete(*x*)** Deletes all elements in the stack with value *x*. After a call for **Delete(*x*)** it is not possible to insert elements with value *x* to the stack.
Time complexity: $\mathcal{O}(1)$.

Solution. In our implementation we use a stack *S* and a boolean array *A* of size *k* initialized to all **false** indicating where an element has been deleted or not. The implementation details are as follows:

- Init(*k*)** Initializes an empty stack *S* and a boolean array *A* full of zeros.
- Push(*x*)** Standard **Push(*x*)** as long as $A[x] == \text{false}$.
- Pop()** Performs a standard **Push()** for each element on top of the stack that has been deleted, then performs one other **Pop()** after that.
- Delete(*x*)** Sets $A[x] == \text{true}$.

While **Pop()** takes $\mathcal{O}(n)$ worst-case, we will show that the amortized complexity of **Pop()** together with **Push(*x*)** is $\mathcal{O}(1)$. Recall that in amortized complexity analysis we always assume the sequence of *m* operations starts from an empty structure. We will show that the complexity conditions hold using the three methods we have learned:

Aggregate Analysis: Suppose that in the sequence of *m* operations $\{S_i\}_{i=1}^m$ we have *m*₁ **Push** operations and *m*₂ **Pop** operations, and suppose that the *i*th call to **Pop** popped exactly *k*_{*i*} elements from the stack. Since we cannot pop more elements to the stack than elements we inserted we have that:

$$\sum_{i=1}^{m_2} k_i \leq m_1,$$

hence:

$$\sum_{i=1}^m \mathcal{O}(S_i) = m_1 \cdot \mathcal{O}(1) + \sum_{i=1}^{m_2} \mathcal{O}(1 + k_i) \leq \mathcal{O}(m_1) + \mathcal{O}(m_2 + m_1) = \mathcal{O}(m),$$

so the amortized complexity of **Push** and **Pop** is $\mathcal{O}(1)$.

The Accounting Method: In the accounting method we define amortized cost of the **Push(*x*)** to be 2 and **Pop()** to be 1 (intuitively **Push(*x*)** is 2 since we assume each thing we insert we may delete).

Note that $\sum_{i=1}^m (a_i - t_i)$ (where *a*_{*i*} is the amortized cost and *t*_{*i*} is the actual cost) is nonnegative since each **Delete** operations is subsidized by the amortized cost of **Push**.

Hence the running time of a sequence of *m* operations is bounded by 2*m* (if all operations are **Push**), hence the amortized complexity of **Push** and **Pop** is $\mathcal{O}(1)$.

The Potential Method: As usual, we define the potential of the structure Φ_i to be the amount of elements in the stack after *i* operations. Note that $\Phi_0 = 0$ and $\Phi_m \geq 0$ for $m \geq 1$. In

the potential method the amortized cost of each operation is defined to be the sum of its real cost and the difference in potential it made. Hence:

$$a_{\text{Push}} = t_{\text{Push}} + (\Phi_i - \Phi_{i-1}) = 1 + 1 = 2,$$

and

$$a_{\text{Pop}} = t_{\text{Pop}} + (\Phi_i - \Phi_{i-1}) = 1 + k_i - k_i = 1,$$

where the 1 is from access to the array of deleted elements. We now get that:

$$\sum_{i=1}^m t_i \leq \sum_{i=1}^m a_i + \Phi_0 - \Phi_m \leq \sum_{i=1}^m a_i \leq 2m,$$

hence the amortized complexity of **Push** and **Pop** is $\mathcal{O}(1)$.

Exercise 6.3. A dynamic array is a data structure that is initialized to be an array of size K , but is dynamic in the sense that whenever it is filled completely we initialize an array of size $2K$ and copy the data into the bigger array. Moreover, if the size of the current size of the array is $K/4$ (this happens for example when the array implements a stack), we initialize an array of size $2K$ and copy the data into the smaller array. In this way, arrays can theoretically store as many elements as needed. Find the amortized complexity of **Push** and **Pop** for dynamic arrays.

Solution. Given a sequence $\{S_i\}_{i=1}^m$ of m **Push** and **Pop** operations, we may partition it into L contiguous subarrays which end on an operation which changes the size of the array. Suppose the size of the i th such partition is m_i , then since starting from a newly initialized array until a new initialization we have to make $\mathcal{O}(m_i)$ operations, the total complexity of $\{S_i\}_{i=1}^m$ is $\mathcal{O}(m)$ hence the amortized complexity is $\mathcal{O}(1)$.

Exercise 6.4 (Extra). Solve Exercise 6.3 using the potential method.

Solution. Let n_i be the number of elements in the array after i operations, and let K_i be the size of the array after i operations. Define the potential function as follows:

$$\Phi_i = \left| n_i - \frac{K_i}{2} \right|.$$

The intuition for this potential method is that there is more potential in the structure the further we are from K_i since we are more prone to operations that cause the array to change its size. In order to make later computations easier we redefine the potential function to be:

$$\Phi_i = \left| n_i - \frac{K_i}{2} \right| \cdot c$$

for some constant $c > 0$ which we will express explicitly later. When using the potential method we would like to show that the amortized costs of the operations (the a_i) are smaller or equal to some constant value. Once we do that we will have that a sequence of m such operations has complexity m times that constant, and hence this will show that the amortized complexity of the operations is $\mathcal{O}(1)$. In this case, we will show that $a_i \leq 1 + c$ (the intuition here is vague but we know that $a_i \geq 1$ and $c > 0$ so it makes sense to try this). We see that one push/pop operation that does not change the size of the array changes the potential by exactly c hence:

$$a_{\text{Push}} = a_{\text{Pop}} = 1 \pm c \leq 1 + c,$$

for **Push** and **Pop** that do not change the size of the array.

Suppose that the i th **Push** did cause a size change for the array. In that case we have $n_{i-1} = K_{i-1} - 1$ and $n_i = K_i/2$. We have that $t_i = K_{i-1}$ since we need to copy all elements, and we also have that:

$$\Phi_{i-1} = \left| n_{i-1} - \frac{K_{i-1}}{2} \right| \cdot c = \left| K_{i-1} - 1 - \frac{K_i}{2} \right| \cdot c = c \cdot \left(\frac{K_{i-1}}{2} - 1 \right),$$

and $\Phi_i = \left| n_i - \frac{K_i}{2} \right| \cdot c = |n_i - n_i| \cdot c = 0$. Hence for a size changing **Push** we have:

$$a_{\text{Push}} = t_i + \Phi_i - \Phi_{i-1} = K_{i-1} - c \cdot \left(\frac{K_{i-1}}{2} - 1 \right),$$

so for $c \geq 2$ we get that for a size changing **Push** we still have $a_{\text{Push}} \leq 1 + c$.

In case the i th operation is a size changing **Pop** we have that $n_{i-1} = \frac{K_{i-1}}{4} + 1$ and $n_i = K_i/2$. Hence $t_i \leq K_{i-1}$ since we copy only some part of the old array (the part that has values) to the new array. We then similarly get that $\Phi_i = \left| n_i - \frac{K_i}{2} \right| \cdot c = |n_i - n_i| \cdot c = 0$ and by calculation for a size changing **Pop** we have:

$$\Phi_{i-1} = \left| n_{i-1} - \frac{K_{i-1}}{2} \right| \cdot c = \left| \left(\frac{K_{i-1}}{4} + 1 \right) - \frac{K_i}{2} \right| \cdot c = c \cdot \left(\frac{K_{i-1}}{4} + 1 \right),$$

and so:

$$a_{\text{Pop}} = t_i + \Phi_i - \Phi_{i-1} \leq K_{i-1} - c \cdot \left(\frac{K_i}{4} + 1 \right),$$

which is smaller than $1 + c$ for $c \geq 4$.

For conclusion, if $c \geq 4$, then for any operation a_i we have $a_i \leq 1 + c = 5$. We also know that Φ_0 is bounded since the size of the array at initialization is bounded. We conclude that for a sequence of m operations we have:

$$(1 + c) \cdot m \geq \sum_{i=1}^m a_i = \sum_{i=1}^m t_i + \Phi_m - \Phi_0 \geq \sum_{i=1}^m t_i - \Phi_0,$$

and Φ_0 is bounded as mentioned. It follows that:

$$\sum_{i=1}^m t_i \leq (1 + c) \cdot m + \Phi_0 = \mathcal{O}(m),$$

hence the amortized time of the operations is $\mathcal{O}(1)$.

7 Hash tables

Suppose we have $n < 1000$ people, such that each person is identified with his own ID number. And suppose that we want a data structure that implements all standard operations of a dictionary:

Memeber(x) Returns **true** if the element is stored in the data structure, and **false** otherwise.

Insert(x) Inserts **x** to the data structure.

Delete(x) Deletes **x** from the data structure.

What is the most efficient way of implementing such a structure? We already know of multiple structures that supports these operations in $\Theta(\log n)$ time (for example AVL trees). Can we think of a structure that allows implementing all of these operations in $\mathcal{O}(1)$ time?

One possible implementation is by using a really, really long array. First, we initialize the array to 0 (this can be done in $\mathcal{O}(1)$ as we saw in Section 2.2) When we want to insert a person into the structure, we simply change the value of $A[ID]$ to 1, we delete by setting $A[ID]$ to 0, and we implement **memeber** by checking the value of $A[ID]$. The problem with this naïve implementation, is that unfortunately, there is a really large amount of IDs, and we would have to allocate memory for every single one of them. This is not very practical, especially considering the fact $n < 1000$, and we would like to allocate about n “space units” (for example an array with 1000 indices would theoretically work, however, now we have a problem since there are more than a 1000 possible values of IDs).

Remark 7.1. Implementing a dictionary using an array as described is sometimes called *direct addressing*. In direct addressing, the keys themselves are the indices of the array.

It seems we have reached an insurmountable wall, an absolute limit on the complexity of the operations ($\Theta(\log n)$), and that we cannot go faster. Indeed, we have to somewhat compromise. The compromise that we have to make is that we require the operations to be $\mathcal{O}(1)$ *on average over all possible inputs*.

Remark 7.2. Note that we require the operations to be on average over all possible inputs. We do not talk about amortized complexity, nor about expected complexity of a randomized algorithm.

7.1 Hash tables

Remark 7.3 (The general setting). Suppose U is a large set of values, and let $K \subset U$ such that $|K| \ll |U|$. Our goal is to define a data structure that supports the standard operations of a dictionary, in time complexity $\mathcal{O}(1)$ on average over all possible inputs, and space complexity $\mathcal{O}(n)$.

Example 7.1. In the beginning of the subsection, the set of possible IDs is U , the set of IDs of the people is K , and indeed $|K| \ll |U|$.

Definition 7.1 (Hash function). Let the setup be as in Remark 7.3, and let $m = \Theta(n)$. Then a hash function is a function $h: U \rightarrow \{0, 1, \dots, m-1\}$, such that:

- (1) The computation time of h is $\mathcal{O}(1)$.
- (2) h is surjective.
- (3) h distributes elements of U uniformly to m different “buckets”, i.e. $h^{-1}(k) \approx \frac{|U|}{m}$ for any value $k \in \{0, 1, \dots, m-1\}$.

Remark 7.4. The conditions in the definition of the hash function (Definition 7.1), are all necessary for reasons that should be very clear. Condition (1) is necessary since we need to compute h at every operation (search/insert/delete), and thus h should be $\mathcal{O}(1)$. Condition (2) is not strictly necessary, but we are going to store each $x \in U$ in $T[h(x)]$, where T is the hash table, and $h(x)$ is the index. Therefore if $h(x)$ is not surjective, there are indices in the array T which will never be used, which is a waste. Condition (3) is the most important condition, and it is necessary for reasons that we will understand more thoroughly later, but for example if h is a constant function $h \equiv c$, then we would have to somehow store all elements in the same space in memory $T[c]$, which is impossible, and even when we will show that how to deal with collisions, we will see that if h is constant the complexity requirements aren't met.

Remark 7.5 (Index). As vaguely mentioned before, for $x \in U$, the value $h(x)$ is called the *index* (or hash code) of x .

Example 7.2. Let U be the set of all possible IDs, and let $K \subset U$ be a subset of 100 different IDs. We can define a hash map $h: U \rightarrow \{0, 1, \dots, 99\}$, defined by $h(x) = x \pmod{100}$. There is no need to show that modulo maps are hash maps, and in fact, in this course there is no need of constructing any hash function at all. Except for universal hash maps (which we will discuss later), all hash maps in this course are going to be modulo maps, and more specifically the modulo of m , where m is the size of the hash table.

Definition 7.2 (Hash table). Let the setup be as in Remark 7.3, and let h be a hash function. A hash table, the ordered pair (T, h) , where T is an array of size $m = \Theta(n)$, which we call the *hash table*. Additionally, the hash table has to be able to deal with collisions. Consequentially, there are different types of hash tables, each with its own way of dealing with collision.

Remark 7.6. The array T is sometimes implemented by a dynamic array.

Remark 7.7 (Collisions). A hash map $h: U \rightarrow \{0, 1, \dots, m-1\}$ is required to be surjective, but it is not necessarily injective (in fact, if h is injective it is called a *perfect hash map*). This means that there exist distinct $x, y \in U$ (that is, $x \neq y$) such that $h(x) = h(y) = k \in \{0, 1, \dots, m-1\}$. In this case, since we are unable to store two elements in the same “bucket”, i.e. in $T[k]$, and we say that we have a collision.

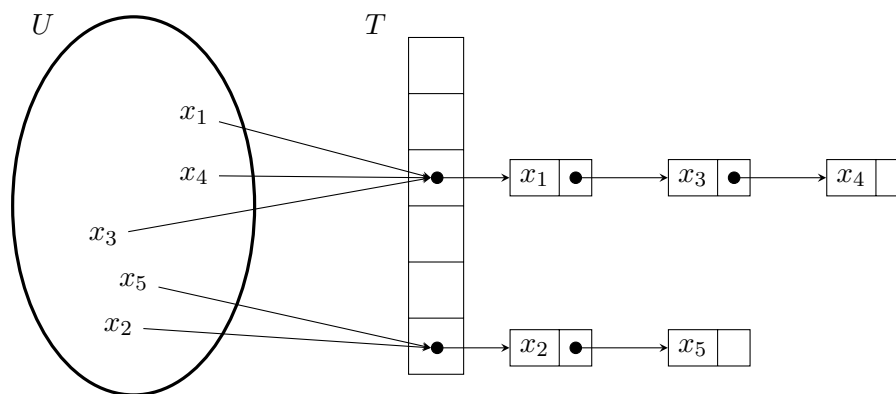
Remark 7.8. In this course, we will only discuss two types of hash tables. The first is a hash table implementing chain hashing, and the second is a hash table implementing double hashing (which is a type of open addressing).

Definition 7.3 (Chain hashing). A chain hashing based hash table, is a table which deals with collisions by initializing an empty linked list at each cell of the array T . The exact implementation of the operations is as follows:

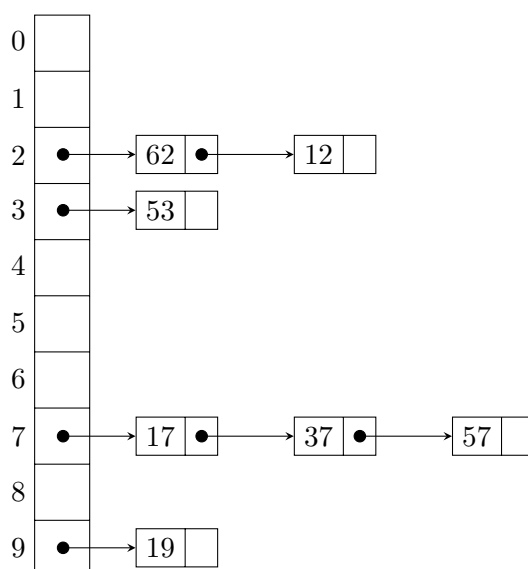
- Search(x)** Compute $h(x)$, and search for x at the list at $T[h(x)]$. Return **true** if the element was found, and **false** otherwise.
- Insert(x)** Searches for x and if not found, it *chains* (inserts) x at the front of the linked list that sits at $T[h(x)]$.
- Delete(x)** Deletes x from the linked list that sits at $T[h(x)]$.

We will not prove this, but all of these operations have an average-case time complexity of $\mathcal{O}(1)$ and a worst-case complexity of $\Theta(n)$.

Example 7.3 (Chain hashing). Below we have an example for chain hashing, where the elements have been inserted at the order of their indices x_1 x_2 x_3 x_4 x_5 :



Example 7.4 (Chain hashing). Here is a more concrete example of chain hashing with the hash function $h(k) = k \pmod{m}$ for $m = 10$, and the input 53 62 17 19 37 12 57



Remark 7.9. In Example 7.4 we used $m = 10$ for convenience. We will see later that $m = 11$ is a better choice.

Remark 7.10 (SUHA). It is clear that constructing the perfect hash function is one of the most important steps towards crafting the the perfect hash table. We would like a perfect hash function to be:

- Easy to compute. That is, it is computable in $\mathcal{O}(1)$.
- Uniform. That is, we want a completely random function from U to $\{0, 1, \dots, m-1\}$. which means that $h(x) = k_1$ and $h(y) = k_2$ are independent events, and moreover $\mathbf{P}(h(x) = k_1) = 1/m$, for all $x, y \in U$ and $k_1, k_2 \in \{0, 1, \dots, m-1\}$.

A function that satisfies both of these conditions is called a uniform hash function, and we say that it satisfies the simple uniform hash assumption (SUHA). We also introduce a stronger and purely theoretical assumption (which never happens in practice), called the uniform hashing assumption (UHA). The UHA only applies to open addressing hashing schemes. Whereas in SUHA we assume that every key is equally likely to hash into any of the m slots, independent of where any other keys goes, in UHA we assume that each key's probe sequence is equally likely. That is, for any probe sequence $(p_1, \dots, p_m)$, the probability of a key $k \in U$ to have that probe sequence is $\frac{1}{m!}$.

Remark 7.11. Under the SUHA, given a hash function h , and a hash table of size m , the probability of a hash collision is

$$\mathbf{P}(h(a) = h(b)) = \frac{1}{m}.$$

Definition 7.4 (Load factor). A load factor α is defined as follows:

$$\alpha := \frac{n}{m},$$

where n is the number of keys stored, and m is the size of the hash table.

Remark 7.12. Under the SUHA, for a simple uniform h , the expected value of the length of a chain is α , since the keys are distributed uniformly.

Proposition 7.1 (Unsuccessful lookup). *Under the assumption of simple uniform hashing (SUHA), in a hash table using chaining, the expected time of an unsuccessful lookup is $\alpha + 1$.*

Proof. As mentioned in Remark 7.12, the expected chain length is α . An unsuccessful lookup scans an entire list. Thus, the expected of an unsuccessful lookup is $\alpha + 1$ (we also compute the hash functions). $\square$

Proposition 7.2 (Successful lookup). *Under the assumption of simple uniform hashing, in a hash table using chaining, the expected time of a successful lookup is at most $\alpha/2 + 1$.*

Proof. Let $k_1, \dots, k_n$ be the keys inserted to the hash table, indexed according to their insertion order. We want to find the expected lookup time of k_i . The number of keys to its left is thus expected to be $\frac{i-1}{m}$, and so the expected time of lookup is $\frac{i-1}{m} + 1$. Thus, the expected time of a successful lookup for a random k_i is

$$\begin{aligned} t &= \frac{1}{n} \sum_{i=1}^n \left(\frac{i-1}{m} + 1 \right) = \frac{1}{n} \sum_{i=1}^n \left(\frac{n-i}{m} + 1 \right) = 1 + \frac{1}{n \cdot m} \sum_{i=1}^n (n-i) = 1 + \frac{1}{n \cdot m} \sum_{i=0}^{n-1} i \\ &= 1 + \frac{1}{n \cdot m} \frac{(n-1)n}{2} = 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{1}{2m} < 1 + \frac{\alpha}{2} \end{aligned}$$

$\square$

Corollary 7.3. *Under the assumption of simple uniform hashing (SUHA), in a hash table using chaining and $m = \mathcal{O}(n)$, all operations have an average-case time complexity of $\mathcal{O}(1)$.*

Proof. We have seen in Theorem 7.1 and Theorem 7.2 that a search operation takes at most $\alpha + 1$ operations, so since $m = \mathcal{O}(n)$, we obtain $\alpha = \frac{n}{m} = \mathcal{O}(1)$, and the result follows. $\square$

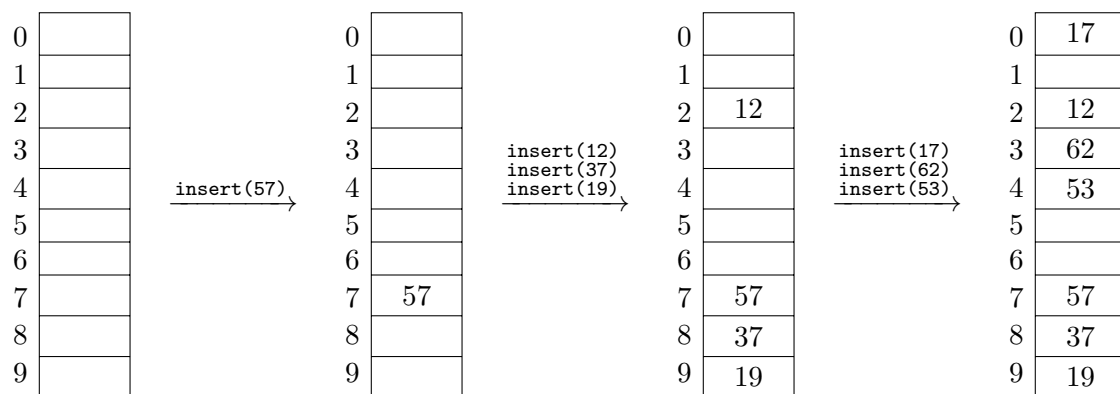
Example 7.5. Suppose that $|U| = 10^6$ and $K \subset U$ with $|K| = 2100$. We could initialize a hash table of size 700 with chaining, and the length of an average chain would be 3, so all operations will have an average-case time complexity of $\mathcal{O}(1)$.

Definition 7.5 (Open addressing). Open addressing (or closed hashing), is a method of collision resolution in hash tables, where instead of chaining, we keep *probing* the table until we find an empty cell.

Definition 7.6 (Linear probing). In a hash table using linear probing, we assume that $n \leq m$, and the basic operations are implemented as follows:

- Insert(x)** Tries to insert x to the data structure at $T[h(x)]$. If another key (or **tomb**) is already there, we resolve the collision by linearly scanning the table, trying to insert it first at $T[h(x) + 1]$, and if that doesn't work then at $T[h(x) + 2]$ and so on. We could implement the table as a circular linked list in order to prevent overflow.
- Delete(x)** Searches for x in the table by linearly scanning it from $T[h(x)]$, if not found, return **false**, and if found delete it from the table and insert a **tomb** (could be any value outside U) there.
- Search(x)** Start by computing $h(x)$, and linearly scan for the element starting from $T[h(x)]$. If a **tomb** is found, keep searching, and if an empty cell is found return **false**.

Example 7.6 (Linear probing). Here is an example of a hash table using linear probing with a hash function $h(k) = k \pmod{m}$ and $m = 10$. The input is inserted in the following order: 57 12 37 19 17 62 53.



Remark 7.13. The main advantage of linear probing is in its simplicity, however, linear probing has a tendency to fill continuous chunks of memory, which doesn't work well with the SUHA, and thus hampers lookups, insertions, and deletions. If we want our structure to implement deletions, we better use the chaining method, since linear probing makes each lookup and insertion slower because of **tombs**, which are there out of necessity, not merit.

Example 7.7. Hash tables using linear probing could still be helpful in data structure where deletion isn't necessary. For example:

(ID databases) – In ID databases, there is not need to delete IDs...

(Symbol table) – A symbol table is a table used by a programming language processor in order to store names of variables, constants, functions, and similar data in a program.

Remark 7.14. We will now proceed to introduce more open addressing methodologies, which utilize the SUHA more efficiently. However, keep in mind that when we need to implement deletions, a hash table using chaining is preferable.

Definition 7.7 (Rehashing). Let $\{h_i\}_{i=0}^{\infty}$ be a sequence of hash functions. Suppose that $T[h_0(x)] = 0$ causes a hash collision, then we resolve that by trying $T[h_1(x)] = 0$, and so on, until the collision is resolved. Sometimes rehashing refers to the process of copying the content of the hash table into a new hash table to get rid of the tombs. This process is expensive, but if we perform it once every $\Omega(n)$ operations, it would be amortized average-case complexity $\mathcal{O}(1)$.

Remark 7.15 (Linear probing). It turns out that linear probing is a special case of rehashing, by defining $h_i(x) := h(x) + i \pmod{m}$.

Remark 7.16 (Average-case complexity). Under the SUHA the probability that we have a collision is $\alpha < 1$ (remember, in open addressing $n < m$). Assuming that the hash functions are mutually independent the probability that $\{h_i\}_{i=0}^{k-1}$ result in a collision is α^k , so the probability that the search path is of length k is exactly $\alpha^{k-1} \cdot (1 - \alpha)$. It follows that the expected search path length is

$$l = \sum_{i=1}^{\infty} i \cdot \alpha^{i-1} \cdot (1 - \alpha) = \frac{1}{1 - \alpha}.$$

For example if $\alpha = 2/3$ then the expected search length is at most 3.

Remark 7.17. In rehashing, when implementing `delete(x)` we still have to use a `tomb`.

Definition 7.8 (Double hashing). Double hashing is also a special case of rehashing. It is implemented by using two independent functions. A hash function $h: U \rightarrow \{0, 1, \dots, m-1\}$, and a step function $d: U \rightarrow \{0, 1, \dots, m-1\}$ (which is also a hash function in itself). We then define $h_i(x) := h(x) + i \cdot d(x) \pmod{m}$ and proceed with the rehashing method.

Remark 7.18. Ideally, we would want that for any $x \in U$, we would have

$$\{h_0(x), \dots, h_{m-1}(x)\} = \{0, \dots, m-1\},$$

so that collisions would always be resolved in at most $m-1$ steps. However, this only happens when $d(x)$ is coprime with m .

Question. How can we make $d(x)$ and m are coprime for any $x \in U$?

Answer. If we set m to be a prime number, then since for any $x \in U$ we have $d(x) \in \{0, 1, \dots, m-1\}$, so we necessarily have $\gcd(d(x), m) = 1$, so by definition $d(x)$ and m are coprime. This also explains Remark 7.9.

Remark 7.19. When implementing hash tables, we always assume that $\alpha = 1$, which follows from the assumption that $m = \mathcal{O}(n)$. However we can only choose $m = \mathcal{O}(n)$ if we know what the value of n (the expected number of total keys) is, and this isn't always the case. The solution is that when the hash table gets "too full", we initialize a new hash table, double the size of the current table, and copy all values one by one. Although this copying operation is very expensive, since it is very rare, the expected complexity for an operation is still $\mathcal{O}(1)$.

7.2 Hash functions

Remark 7.20. Recall that when implementing hash functions our primary goal is making sure that they are simple (to compute) and uniform (on all possible inputs). In this subsection we will show multiple examples of hash functions and impose a few conditions on them so they will be approximate the SUHA.

Example 7.8 (Modulo function). The modulo m function is the function $h(x) = x \pmod{m}$. For such a function, we would prefer that:

- m is not a power of 2 or 10. Note that if m is a power of 2 or 10 then $h(x)$ only depends on the $\log_2(m)$ or $\log_{10}(m)$ least significant bits/digits of x respectively. It is clearly better when $h(x)$ utilizes all of the information in x .
- m is not close to a power of 2. Since in many cases keys represented in binary are close to a certain power of 2, if m is also close to a power of 2, we wouldn't get a uniform distribution.

- m is prime. We already discussed why a prime m is better in Section 7.1.

Example 7.9 (Multiplicative hash). Let $0 < \beta < 1$ be a constant. Then its associated multiplicative hash function is

$$h(k) = \lfloor m * \text{frac}(\beta \cdot k) \rfloor,$$

where $\text{frac}(x) := x - \lfloor x \rfloor$. The size of the table m can be anything. A good value of β for the SUHA is $\beta = (\sqrt{5} - 1)/2 \approx 0.61803 \dots$. For example if $m = 10000$ and $k = 123456$ then

$$\begin{aligned} h(k) &= \lfloor 10000 \cdot \text{frac}(0.61803 \cdot 123456) \rfloor \\ &= \lfloor 10000 \cdot \text{frac}(76300.0041151) \rfloor \\ &= \lfloor 10000 \cdot 0.0041151 \rfloor \\ &= \lfloor 41.151 \rfloor \\ &= 41. \end{aligned}$$

Example 7.10 (XOR hash). In order to discuss hash functions of strings, we first convert strings to their ASCII code, for example:

$$\begin{aligned} \text{"a"} &= 97 = (0110\ 0001)_2 \\ \text{"b"} &= 98 = (0110\ 0010)_2 \end{aligned}$$

Then the first XOR hash we may try is to simply take the XOR of all characters in the string, that is, for a string $s = s_1 \dots s_k$ with $\text{ASCII}(s_i) = a_i$ we compute

$$h(s) = a_1 \oplus a_2 \oplus \dots \oplus a_k.$$

For example:

$$h(\text{"ab"}) = (0110\ 0001)_2 \oplus (0110\ 0010)_2 = (0000\ 0011)_2 = 3$$

However, this method isn't very efficient since

$$\begin{aligned} h(\text{"aa"}) &= (0110\ 0001)_2 \oplus (0110\ 0001)_2 = 0 \\ h(\text{"bb"}) &= (0110\ 0010)_2 \oplus (0110\ 0010)_2 = 0, \end{aligned}$$

and similarly $h(s) = 0$ for any string where each character appears an even amount of times. This means that even though the distribution of this implementation of XOR hash looks really good, we see that it does not really approximate SUHA well for a natural family of strings. Another problem is that its range is limited since $h(x) \leq 255$.

However, we can improve on the simple XOR hash function by choosing an arbitrary permutation $\sigma = (\sigma_0, \dots, \sigma_{255}) \in S_{256}$, and store it in an array $A[256]$. Then we define the hash function recursively by

$$h(s_1) = s_1 \oplus A[0] \quad h(s_1 s_2 \dots s_n) = s_n \oplus A[h(s_1 \dots s_{n-1})]$$

For example, if we have $\sigma_0 = 23$ and $\sigma_{118} = 38$, then we get that

$$h(\text{"a"}) = (0110\ 0001)_2 \oplus T[0] = (0110\ 0001)_2 \oplus \underbrace{(0001\ 0111)_2}_{=23} = (0111\ 0110)_2 = 118,$$

and since $\sigma_{118} = 38$ we get that

$$h(\text{"aa"}) = a \oplus T[h(a)] = (0110\ 0001)_2 \oplus \underbrace{(0010\ 0110)_2}_{=38} = 71.$$

Using permutations we see that we don't have to worry about strings where each character appears an even amount of times.

In order to solve the limited range problem, we notice that if we have multiple independent hash functions $h_1, h_2, \dots, h_m$ with limited ranges we can simply concatenate them

$$h(k) = h_1(k) || h_2(k) || \dots || h_m(k).$$

For example, in the example with strings we could simply choose two hash functions h_1, h_2 based on different permutations $\sigma_1, \sigma_2 \in S_{256}$ and then define:

$$h(k) = h_1(k) || h_2(k) = h_1(k) \cdot 256 + h_2(k),$$

in order to get a hash function with range $256^2 - 1$. We could keep concatenating hash function in order to get to the desired range (remember, the desired range is the size of the table which should be $m = \mathcal{O}(n)$).

Note that we could also define h_1 to be based on σ_1 , and then set $h_2(s) = h_1(\bar{s}_1 s_2 \dots s_k)$, where $\bar{s}_1$ is the character following s_1 in ASCII. For example:

$$h(\text{"acb"}) = h_1(\text{"acb"}) \cdot 256 + h_2(\text{"acb"}) = h_1(\text{"acb"}) \cdot 256 + h_1(\text{"bcb"}).$$

7.3 Universal hash functions

For any hash function $h: U \rightarrow \{0, 1, \dots, m-1\}$, there exists $K \subset U$ such that there would be a very large amount of collisions. For example if U is the set of IDs and $h(x) = x \pmod{11}$ then choose $K = \{k \in U \mid k \pmod{11} = 4\}$ and then unfortunately we would have a collision in every insertion.

So far we have worked under the assumption that keys are chosen uniformly and independently, however, in real life this is usually not the case. There may be an “adversary” that would always give us a set $K \subset U$ so that we would have many collisions. It may also be the case that there is a programmer who simply likes to use similar names for his variables, which would also cause many collisions in the hash function of the symbol table.

In this subsection, we will construct a hash function which always works efficiently, regardless if there is an adversary, or if the programmer on the other side is terrible. Such a hash function is called a *universal hash function*.

Definition 7.9 (Universal family). Let $H = \{h_i: U \rightarrow \{0, 1, \dots, m-1\}\}_{i \in I}$ be a family of hash functions. Then H is said to be a universal family if for any $x, y \in U$ such that $x \neq y$ we have:

$$\left| \{h \in H \mid h(x) = h(y)\} \right| \leq \frac{|H|}{m}.$$

Moreover, H is said to be strongly universal if it is pairwise independent. Equivalently, we may define that H is strongly universal if

$$\left| \{h \in H \mid h(x) = h(y)\} \right| = \frac{|H|}{m}.$$

Remark 7.21. Note that in this course we always assume a universal family to be strongly universal. Effectively this means that you may forget about the notion of a “weak” universal family and take the latter as a definition for a universal family from now on.

Remark 7.22. If H is a universal family of hash functions then for a $h \in H$ chosen uniformly at random we have

$$\mathbf{P}[h(x) = h(y)] = \left(\frac{|H|}{m} \right) / |H| = \frac{1}{m}.$$

Proposition 7.4. Let H be a universal family of hash functions mapping into a hash table T that is using chaining of size m . If $h \in H$ is chosen uniformly at random, and we use it to insert n keys into T , then the expected number of hash collisions in the n th insertion is $\frac{n-1}{m}$.

Proof. In Remark 7.22 we saw that the probability for a hash collision of distinct $x, y \in U$ is $p = 1/m$. Thus, the expected value of some $x \in U$ with any of the other keys $y \in T$ is

$$\bar{L} = \sum_{y \in T \setminus \{x\}} \mathbf{P}[h(x) = h(y)] = \sum_{y \in T \setminus \{x\}} \frac{1}{m} = \frac{n-1}{m} = \alpha - \frac{1}{m} < \alpha,$$

where $\alpha = n/m$ is the load factor. This completes the proof. $\square$

Corollary 7.5. *The expected number of collisions is less than α , and in particular it is expected $\mathcal{O}(1)$.*

Remark 7.23. Our next goal is constructing a universal family of hash functions.

Example 7.11 (Universal family). Let m be a prime number. Suppose that any $x \in U$ can be partitioned into r segments $\mathbf{x} = [x_0, \dots, x_{r-1}]$ of constant size c such that $x_i < m$ for all $0 \leq i \leq r-1$. For example, if $U = \{0, 1, \dots, 255\}^2$, we can partition any $x \in U$ into two parts of length 8 by representing each coordinate in base 2. For any $\mathbf{a} = (a_0, \dots, a_{r-1}) \in A := \{0, 1, \dots, m-1\}^r$, we define $h_{\mathbf{a}}(\mathbf{x})$ by

$$h_{\mathbf{a}}(\mathbf{x}) = \langle \mathbf{a}, \mathbf{x} \rangle \pmod{m},$$

where $\langle \cdot, \cdot \rangle$ is the standard inner product in $\mathbb{R}^r$. Then we define $H := \bigcup_{\mathbf{a} \in A} \{h_{\mathbf{a}}\}$. In this case $|H| = m^r$ and H is a universal family of functions. Note that all conditions mentioned (for example that m is prime) are necessary for H to be a universal family.

Example 7.12 (Universal family, concrete example). Let $m = 503$ (yes, 503 is prime), let $h_{\mathbf{a}}$ with $\mathbf{a} = (352, 223, 101) \in A$ be the hash function chosen uniformly at random, and suppose that $U = \{0, 1, \dots, 2^{24} - 1\}$. Note that any $k \in U$ can be partitioned into three 8 bit segments which form its binary representation. For example for $x = 3077 \in U$ we have

$$x = 3077 = \underbrace{(00000000 \ 00001100 \ 00000101)}_{0 \ 12 \ 5}_2,$$

and thus $\mathbf{x} = (0, 12, 5)$. It follows that

$$h_{\mathbf{a}}(\mathbf{x}) = \langle \mathbf{a}, \mathbf{x} \rangle \pmod{m} = (352, 223, 101) \cdot (0, 12, 5) \pmod{503} = 163.$$

Remark 7.24 (How to use universal families). Whenever the user defines a hash table T of size m , a vector $\mathbf{a} \in A$ is chosen uniformly at random, and then $h_{\mathbf{a}}$ is used for all of the hashing in T from that point.

Proposition 7.6. *The set H from Example 7.11 is a (strongly) universal family.*

Proof. Let $x, y \in U$ be such that $x \neq y$, and let $\mathbf{x} = (x_0, \dots, x_{r-1})$ and $\mathbf{y} = (y_0, \dots, y_{r-1})$ be their corresponding partitions. Without loss of generality assume that $x_0 \neq y_0$. By Definition 7.9 it follows that it suffices to show that

$$\left| \{h \in H \mid h(x) = h(y)\} \right| = \frac{|H|}{m} = \frac{m^r}{m} = m^{r-1}.$$

Thus, it suffices to show that for any $a_1, \dots, a_{r-1}$, there exists a unique $a_0 \in \{0, 1, \dots, m-1\}$ such that $\mathbf{a} = (a_0, \dots, a_{r-1}) \in A$ satisfies $h_{\mathbf{a}}(x) = h_{\mathbf{a}}(y)$ (this is very much equivalent to writing $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$). Note that

$$h_{\mathbf{a}}(x) \equiv_m h_{\mathbf{a}}(y) \iff \langle \mathbf{a}, \mathbf{x} - \mathbf{y} \rangle \equiv_m 0 \iff a_0(x_0 - y_0) \equiv_m - \sum_{i=1}^{r-1} a_i(x_i - y_i),$$

and $x_0 \neq y_0$ so $(x_0 - y_0) \neq 0$. Since m is prime $(x_0 - y_0)$ is invertible, and thus

$$a_0 \equiv_m - \sum_{i=1}^{r-1} a_i (x_i - y_i) (x_0 - y_0)^{-1},$$

which completes the proof. $\square$

Remark 7.25 (Conclusion). We have seen the definition of a universal family of hash functions (Definition 7.9). We have seen that with universal hashing the expected number of collisions is less than the balance factor $\alpha = n/m$ (Theorem 7.4). We have constructed a universal family of hash functions (Example 7.11).

Remark 7.26 (Trees and hash tables comparison). While we have seen that hash tables can be very useful, it is important to remember in which cases using a tree or a skip list is better. For example:

- When we are required to work with worst-case complexity bounds, balanced trees and skip lists are $\mathcal{O}(\log n)$, while hash tables are $\mathcal{O}(n)$.
- A *range query* is a query regarding a range of elements within our data structure. For example, given an array $A[n]$, a range minimum query (also known as RMQ) asks what is the minimal number in A in a certain range $[m, M] \subset \mathbb{R}$. In case we are tasked with a range query, using a tree or a skip list is better than hash tables since they store keys by their order.

Example 7.13 (RMQ). In order to solve RMQ (range minimum query), we can use hash tables, but it would take us $\mathcal{O}(n)$ time since we would have to go over all the elements. However, if we use an AVL tree, and modify it so that each node stores the size of its subtree, we could solve the problem in $\mathcal{O}(\log n)$.

Remark 7.27. Sometimes when we talk about a range query on $A[n]$ we mean a query on a subarray $A[1, \dots, r]$ of $A[n]$, not a query on $A[n]$ that involves a certain range $[m, M] \subset \mathbb{R}$. In these cases using a hash tree is rather meaningless since we lose the order of the elements, but we can still use an AVL tree if we use the indices of the elements as keys, and keep the value of the elements in a separate field of the same node. If the range query is to compute the sum of elements in the subarray $A[1, \dots, r]$, we would still need to further modify the tree and store at each node the sum of values of elements of its subtree (this is not the same as the sum of the keys in its subtree since the keys are now indices and the values are in a separate field). Now we could solve the query in $\mathcal{O}(\log n)$.

Remark 7.28. In practice, universal hash tables aren't really used for the following reasons:

- (1) Universal hashing usually requires arithmetic operations in $(\text{mod } p)$ for big primes as we saw, and in general calculating it is a lot slower than using a modern hash function that does not involve modulo operations.
- (2) If we want to be safe against adversarial input, we have other, more modern solutions that are safer than universal hashing.

Remark 7.29. It turns out that when using simple hash functions (for example the standard implementations in many programming languages), a malicious attacker can generate an input sequence that causes millions of collisions, which usually leads to a 100% CPU usage for multiple hours. Such behavior is impossible with modern hashing algorithms or with universal hashing, whose predictability would practically require foreseeing the result of a dice roll.

Remark 7.30. We have that

	Tree	Balanced tree	Skip list	Hash	Universal hashing	Array
Worst-case complexity	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Uniform coin tosses			$\mathcal{O}(\log n)$		$\mathcal{O}(\alpha)$	
Good world (uniform input and tosses)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\alpha)$	$\mathcal{O}(\alpha)$	$\mathcal{O}(1)$

It may seem that a hash table is better than a skip list in every scenario, but remember Remark 7.26.

Exercise 7.1 (Element uniqueness). Given an array of n natural numbers $0 \leq x_1, x_2, \dots, x_n \leq T$, find if there exist $i \neq j$ such that $x_i = x_j$.

Solution. We can solve this problem in multiple ways.

- (1) First, we could check all pairs (x_i, x_j) , which solves the problem in $\mathcal{O}(n^2)$.
- (2) We can sort the array, then check if there are equal neighbouring elements in the sorted array. This method solves the problem in $\mathcal{O}(n \log n)$ (sorting an array efficiently takes $\mathcal{O}(n \log n)$).
- (3) If we impose the condition that T is sufficiently small so that we could create an array of size T , we could use a histogram and solve the problem in $\mathcal{O}(n)$ (so long as we are allowed to initialize the array in $\mathcal{O}(1)$, otherwise the solution is in $\mathcal{O}(T + n)$). Note that this solution is conditional, since it only works for sufficiently small values of T .
- (4) We could simply initialize a hash table of size n , then check for equality in each collision. This solves the problem in $\mathcal{O}(n)$ on average on all possible inputs.

Exercise 7.2. Given an array $A[n]$ of n natural numbers $0 \leq x_1, x_2, \dots, x_n \leq T$, find if there exist two different numbers which sum up to 0. Then do the same with 37 instead of 0.

Solution. First we will find if there exist two different number which sum up to 0. We can do this by simply inserting the elements x_i one by one into a hash table of size $\mathcal{O}(T)$, then after every insertion search for $y_i = -x_i$. If at any iteration y_i is found, then there is such a pair, otherwise, there isn't. This solves the problem in average-case time complexity $\mathcal{O}(n)$ since at every iteration we perform $\mathcal{O}(1)$ operations on average and there are n iterations in total.

In order to solve the problem with 37 instead of 0, after each insertion of x_i we simply search for $y_i = 37 - x_i$ instead of $y_i = -x_i$. Similarly, this solution has average-case time complexity $\mathcal{O}(n)$.

Proposition 7.7. Under the uniform hash assumption, in a hash table using open addressing we have:

- (1) Expected time of an unsuccessful lookup is less than $\frac{1}{1-\alpha}$.
- (2) Expected time of a successful lookup is less than $\frac{1}{\alpha} \ln \frac{1}{\alpha} + \frac{1}{\alpha}$.

Remark 7.31. Theorem 7.7 is largely theoretical, as achieving anything close to the uniform hashing assumption is practically infeasible. There is almost never sufficient space to store $m^{|U|}$ functions, so we cannot sample a random hash function.

Proposition 7.8. Under the simple uniform hash assumption, in a hash table using open addressing with linear probing we have:

- (1) Expected time of an unsuccessful lookup is less than $\frac{1+1/(1-\alpha)}{2}$.
- (2) Expected time of a successful lookup is less than $\frac{1+1/(1-\alpha)^2}{2}$.

Remark 7.32. Theorem 7.8 shows that under the uniform hash assumption, we can use linear probing with complexity $\mathcal{O}(1)$ for any of the basic dictionary operations.

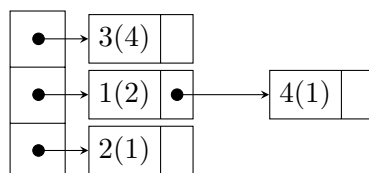
Remark 7.33 (Conclusion). In this section, we have studied hash tables, which implements dictionaries with expected complexity $\mathcal{O}(\alpha = n/m)$, and usually $\mathcal{O}(\alpha) = \mathcal{O}(1)$. We have seen multiple ways to resolve collisions, for example by using chaining, or the different open addressing methods (linear probing/rehashing/double hashing). We then continued to talk about different hashing functions, and we also discussed universal families of functions.

7.4 Exercises

Exercise 7.3. Describe an algorithm which finds the mode of an array A of size n in $\mathcal{O}(n)$ expected time complexity.

Solution. A mode of an array is simply the value that appears most in the array. Expected time complexity means that we cannot use a particular hash function for our hash table (clearly this is an exercise in hash tables), so we state that we use a universal hash function. In that way, as we know even against input of an omniscient adversary the operations would run in $\mathcal{O}(1)$ almost surely.

The solution is as follows: We maintain a hash table with universal hashing and linearly scan the elements of A and insert them into the table. Moreover, when we insert an element into the table we also keep a count of how many times we saw it in A so far. We can do this by keeping a simple counter, which we initialize to 1 with the first appearance of an element, and increment by 1 each time we see it in A again. For example if the array is $[1, 3, 2, 3, 3, 4, 1, 3]$ with hash function $x \mapsto x \pmod{3}$ and hash table of size 3 then the hash table after this process would be:



In order to return the mode we also keep track of the element with the highest counter as we scan the array. We could also simply scan the array after this process and check which element has the highest counter.

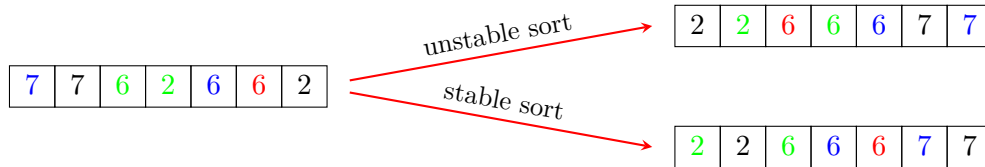
Exercise 7.4. Describe an algorithm that takes an array of natural numbers of size n and a natural number k , and returns whether there exists some natural i so that the total number of appearances of i and $i + 1$ in the array is k in average-case time complexity $\mathcal{O}(n)$.

Solution. Construct the same hash table as in Exercise 7.3 (this time we do not have to use universal hashing since the complexity requirement is average-case not expected).

After the construction of the table we scan the original array again, and for any x in the array we query the amount of appearances of x and $x + 1$ in the hash table, and if it sums up to k we return **true**. If the scan of the array is complete then we return **false**.

Exercise 7.5. A sorting algorithm is called stable if whenever $A[i] == A[j]$ and $i < j$ then $A[i]$ comes before $A[j]$ in the output. Show that any sorting algorithm A that takes an array of size n and sorts it in worst-case $T_A(n)$ time can be transformed (modified) into a stable sort B with an average-case running-time $T_A(n) + \mathcal{O}(n)$, and worst-case running-time $T_A(n) + \mathcal{O}(n \log n)$.

Solution. Below is a colorful example of a stable and a nonstable sort for reference:



Note that the exercise involves average-case and worst-case so it is most likely a exercise in hash table, and since we see the exercise mentions $\mathcal{O}(n \log n)$ it also has to do with trees, hence the solution is probably using chain hashing with AVL trees.

Indeed we can turn get a stable sorting algorithm by executing the following steps:

- (1) Denote the array by `arr`, the algorithm by A , initialize an output array `out`, apply A on `arr` to get a sorted array `s`, and initialize a hash table based on chain hashing with AVL trees T so that each node in the AVL trees is a linked list.
- (2) Scan elements of `arr` from left to right and insert them to to T . If we try to insert an element into a tree and it is already there, add it at the end of the list in that node.
- (3) Scan the elements of `s` and when you encounter a new value search for it in T and print the linked list of the node found.

8 Union-find

8.1 Implementations

In some cases we may want a data structure that supports storing sets of elements, and supports unions between these sets. This may sound a slightly unnatural data structure that is only helpful in very specific situations, but it turns out that it very useful. This type of data structure is called a union-find data structure.

Definition 8.1 (Union-find). Let U be a universe of elements with $|U| = n$, and assume for simplicity that $U = \{1, \dots, n\}$. An abstract data type for union-find includes:

- Makeset**(i) Creates and returns the singleton $\{i\}$. Note that i cannot be an element of any currently stored set.
- Find**(i) Returns the identifier of the set that contains i .
- Union**(p, q) Removes the sets p, q , from the structure, then creates and returns the set $p \cup q$.

Example 8.1. Suppose that our union-find data structure currently contains the sets:

$$\{1, 3\} \quad \{5\} \quad \{2, 4, 6, 7\}$$

and then we perform $p = \text{Find}(6)$, $q = \text{Find}(5)$, $r = \text{Union}(p, q)$, and $s = \text{Find}(6)$. We get that $r = \{5, 2, 4, 6, 7\}$, and also that p and q are currently null pointers, and that s and r are both pointers to the same set.

Exercise 8.1. Let $\{1, \dots, n\}$ be the identifiers of n cities, and suppose all cities start unconnected (there are no roads connecting them). Implement a data structure that supports the following functions:

- AddRoad**(x, y) Adds a road (x, y) that connects between two cities x and y .
- CheckConnectivity**(x, y) Returns **true** if two cities are connected (note that connectivity is transitive), and returns **false** otherwise.

Solution. We use a union-find data structure and initialize the structure by **Makeset**(i) for all $1 \leq i \leq n$. Then we implement the functions as follows:

- AddRoad**(x, y) Perform **Union**(**Find**(x), **Find**(y)).
- CheckConnectivity**(x, y) Return the boolean value **Find**(x) == **Find**(y).

Remark 8.1. In what follows we will show multiple implementations for the union-find data structure. We will start by trying some trivial implementations, and gradually move on to more advanced implementations and we reach the standard implementation.

Example 8.2 (First implementation). The first implementation is by using an array $A[n]$ and a counter initialized to zero **counter** = 1. Then intuitively the counter is representing the identifier of the next created set, and each time we create a set we increment the counter by one in order to make the identifier of each set unique. The array stores the identifier of the set containing i at $A[i]$. More formally, the functions of the union-find data structure are implemented as follows:

- Makeset**(i) Perform $A[i] = \text{counter}++$, and return $A[i]$.
- Find**(i) Return $A[i]$.
- Union**(p, q) Perform a linear scan on $A[1, \dots, n]$ and replace any occurrence of p or q by **counter**, then increment the counter with **counter**++.

For example if we suppose that **counter** is 13 we represent

$$\{1, 3\} \quad \{5\} \quad \{2, 4, 6, 7\}$$

in the following way:

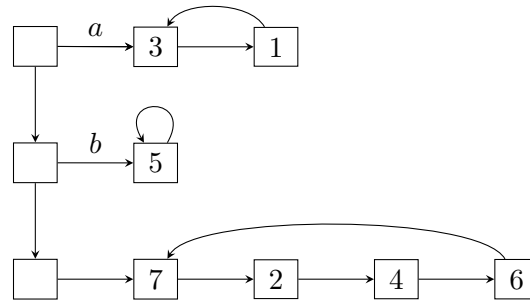
1	4		1	13
2	12		2	12
3	4		3	13
4	12	$\xrightarrow{\text{Union}(4,5)}$	4	12
5	5		5	13
6	12		6	12
7	12		7	12

Note that the complexity of **Makeset** and **Find** is $\mathcal{O}(1)$, but **Union** is $\mathcal{O}(n)$. We can do better.

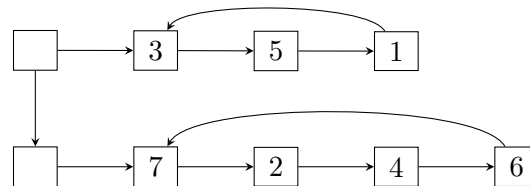
Example 8.3 (Second implementation). In this implementation each set is represented by a circular linked list, and we additionally maintain another linked list which contains pointers to all current sets. For example we represent

$$\{1, 3\} \quad \{5\} \quad \{2, 4, 6, 7\}$$

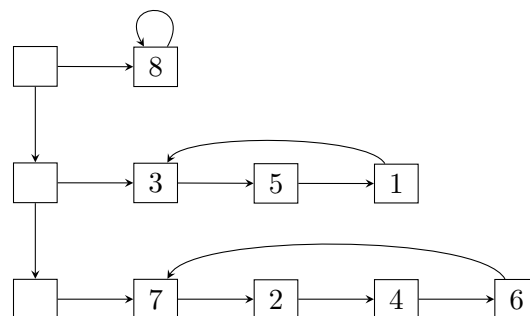
in the following way:



In this method, we can implement **Union(p,q)** by merging the cyclic lists with identifying pointers q and p , which we can do in $\mathcal{O}(1)$. For example, after **Union(a,b)** we get:



We implement **Find(i)** by scanning all lists until we find i , which takes $\mathcal{O}(n)$. The last operation **Makeset(i)** is implemented by initializing a list with a single element and linking it to the structure. For example if we run **Makeset(8)** we would get:



Remark 8.2 (A tip for implementation). Note that sometimes the identifier of a set could also be an element in that set, or a different set, and it could be very difficult and annoying to implement union-find while always having to keep this in mind. In order to prevent confusion, it is recommended to use the `typedef` command available in `c++` as follows:

```
typedef int VALUE;
typedef int SET;
typedef NODE* SET;
SET Makeset(VALUE value);
SET Find(VALUE value);
SET Union(SET p,q);
```

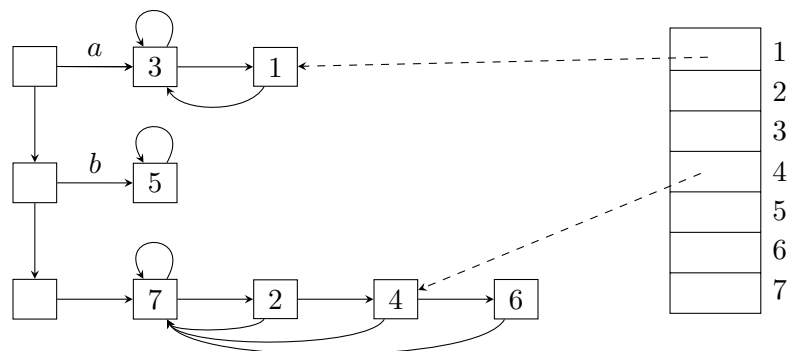
Remark 8.3 (Set identifiers). Note that so far we have assumed that set identifiers are numbers, but sometimes it is more useful to use strings to represent sets, for example when each set contains people of a certain rank, it would be better if we identified each set with the name of the rank instead of an arbitrary number. In order to use strings as identifiers, we need a data structure that easily converts between strings and numbers, for example a hash table using open addressing (which we may choose to be dynamic if the number of sets we need isn't known). In this way, each string would be assigned an index in the hash table, and since we are using open addressing, each index identifies a unique string, which allows using string the string as an identifier by accessing the set with $h(\text{string})$.

Remark 8.4. In general when implementing a union-find data structure we have assumed that the number of elements in U is known and thus we were able to bound the number of possible sets by $|U|$ and use a static array of size $|U|$. If this isn't the case, we could simply implement a dynamic array instead of a static array.

Example 8.4 (Third implementation). This time each set is implemented as a special list, where every node points to the head of the list as well as to the next node. Additionally, we maintain an array which stores a pointer to the element i at $A[i]$. For example we represent

$$\{1, 3\} \quad \{5\} \quad \{2, 4, 6, 7\}$$

in the following way:



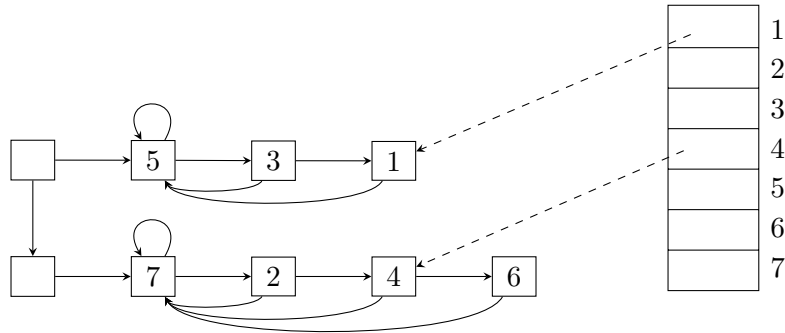
We then implement the functions as follows:

Makeset(i) We create a new list with one node (other than the pointer node) and create pointers to it from itself and $A[i]$. This takes $\mathcal{O}(1)$ time.

Find(i) We first follow $A[i]$, then the pointer to the head of the list, then return the pointer pointing at the header from the dummy node. This takes $\mathcal{O}(1)$ time.

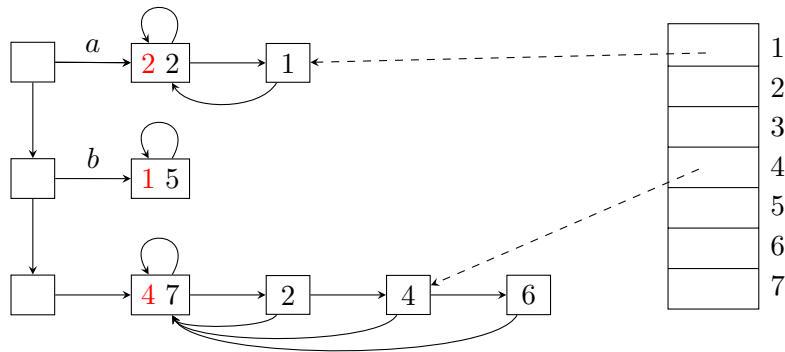
Union(p, q) We merge the lists with pointers p and q , the update the pointers to point at the new head. We then return the pointer to the new head. This takes $\mathcal{O}(n)$ time.

For example if we run **Union(p, q)** we get the new structure:

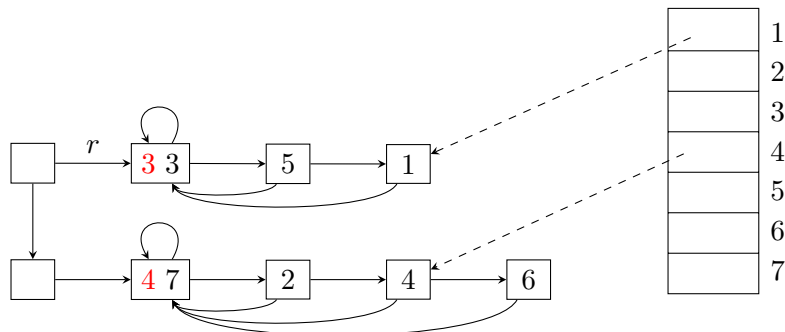


Note that when we merge sets (lists) we must update the pointers of all nodes at one list from the old head to the new head, and it is clearly better if we update the pointers of the shorter list.

Example 8.5 (Third implementation, modified). This implementation is identical to the third implementation, except we modify the list so that each list head would also contain the list's length. With this new field we can now modify the union function so that it updates the pointers of the shorter list in order to save time. For example, if we our union-find structure was at the following state:



where the left numbers at the head represent the length of the list, then when we run the command `r = Union(a,b)` we would get:



Note that the smaller set $\{5\}$ (equivalently, the shorter list) was merged into $\{1, 3\}$ and not the other way around.

Lemma 8.1. *If, during each union operation, the smaller set is always merged into the larger set, then any element x will have its parent set changed no more than $\log_2 n$ times, where n is the total number of elements in the data structure.*

Proof. Whenever an element x changes from set A to a new set $C = A \cup B$, since $|A| \leq |B|$ we get $|A| \leq \frac{1}{2}|C|$. Therefore after $\log_2 n$ unions in which x changed its parent set, we would get $|A| \leq 2^{-\log_2 n}|C|$, where $A = \{x\}$ is the first set of x , and C is its group after the $\log_2 n$ merges. In this case we get $n \leq |C|$, and thus x cannot change its set anymore, which completes the proof. $\square$

Proposition 8.2. *In an empty union-find data structure, if, during each union operation, the smaller set is always merged into the larger set, then any sequence of m operations from the set $\{\text{Makeset}, \text{Union}, \text{Find}\}$ takes at most $\mathcal{O}(m \log n)$ time where n is the number of elements in the structure by the end of the sequence.*

Proof. We know that both **Makeset** and **Union** cost $\mathcal{O}(1)$ time. We also know that each **Union** operation takes $\mathcal{O}(n_c)$ time, where n_c is the number of elements that had their parents changed by the union. By Theorem 8.1 any element in the structure will have its parents changed no more than $\log n$ times, and therefore all **Union** operations in the sequence together take at most $\mathcal{O}(n \log n)$ time. Therefore the time complexity of the total sequence of operations is $\mathcal{O}(m + n \log n)$. However, since $m > n$ (each element in the data structure requires exactly one **Makeset**) we conclude that the time complexity is simply $\mathcal{O}(m \log n)$. $\square$

Remark 8.5. Note that a statement of the form $T(n, m) \in \mathcal{O}(f(n, m))$ means that there exists constants $C > 0$ and $N > 0$ such that for all $n, m > N$ we have $T(n, m) \leq C|f(n, m)|$.

Proposition 8.3. *In any state of a union-find data structure, if, during each union operation, the smaller set is always merged into the larger set, then any sequence of m operations from the set $\{\text{Makeset}, \text{Union}, \text{Find}\}$ takes at most $\mathcal{O}(m \log n)$ time where n is the number of elements in the structure by the end of the sequence.*

Proof. We know that **Makeset** and **Find** operations take $\mathcal{O}(1)$, so it suffices to prove the proposition for a sequence of m **Union** operations. We know that each **Union** takes $\mathcal{O}(1)$ time for some simple calculations, and then $\mathcal{O}(n_c)$ time for the merging, where n_c is the number of elements that had their parents changed by the union. Therefore by Theorem 8.1, the total time of the sequence is bounded by $\mathcal{O}(m + t \log n)$, where t is the number of elements that had their parents changed by the union. Note that while each union changes the parents of multiple nodes, there is only a single unique node which has its parent node for the first time which is the head node. There may be other nodes which had their parents changed, but they are subsidized by previous **Union** operations. Therefore we have m operations and at every **Union** exactly one node (the head of the smaller set) which we subsidize by $\mathcal{O}(\log n)$, and thus we will get that the total time of the sequence (not including time subsidized by previous operations) is $\mathcal{O}(m + m \log n) = \mathcal{O}(m \log n)$. $\square$

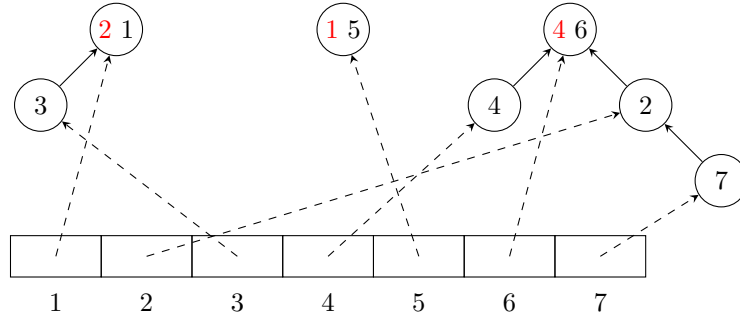
Remark 8.6. The last argument in Theorem 8.3 may be a little confusing but it is a very subtle and important point about amortized complexity. When computing the amortized complexity of a sequence of m operations that start from a certain state of the data structure, we only count the time that we need to subsidize. There might be some actions (such as changing a pointer, storing a value in an array etc.) which are subsidized by previous operations (operations executed before our sequence), and in that case, counting them in the total execution time of our sequence would be counting them twice, so it is important to not count them.

Corollary 8.4. *The amortized complexity of any sequence of m operations of **Makeset**, **Union**, and **Find** is $\mathcal{O}(\log n)$. In particular, the amortized complexity of m **Union** operations is $\mathcal{O}(\log n)$.*

Example 8.6 (Fourth implementation). In the fourth (and final) implementation of union-find, each set is represented by a data structure called an up-tree (this is technically a new data structure, but it is simply a tree where children nodes point to their parents), where each node represents an element in the set, the root of the tree would also contain the size of the tree, and we also let the root of the tree be the identifier of the set. We additionally maintain an array with access to the elements just like the one we maintained in the third implementation. For example we represent

$$\{1, 3\} \quad \{5\} \quad \{2, 4, 6, 7\}$$

in the following way:



We can now implement the union-find operations in the following way:

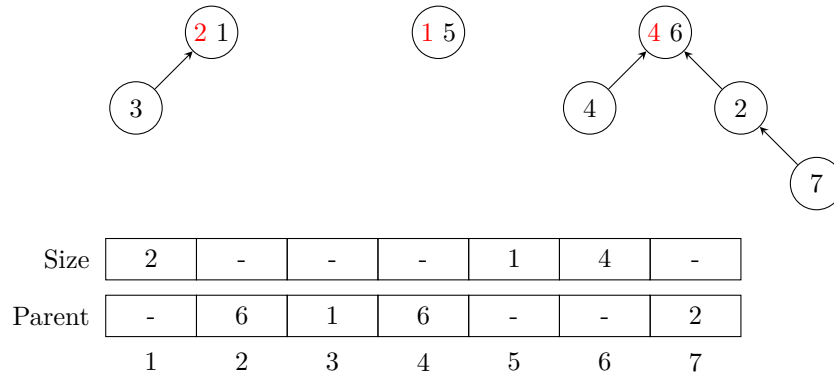
Makeset(i) Create a new tree with a single node i . Time complexity: $\mathcal{O}(1)$.

Find(i) Follow $A[i]$ then climb the up-tree until we reach the node, and return the node. Time complexity: $\mathcal{O}(h)$ where h is the height of the tree.

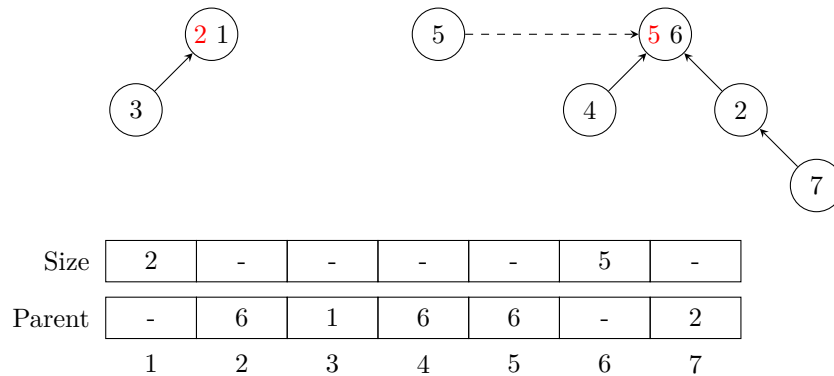
Union(p, q) Merge the trees with roots p and q , such that the root of the smaller tree becomes the child of the root of the bigger tree. In this implementation the merging condition is called *union by rank*. Time complexity: $\mathcal{O}(1)$.

Note that **Union**(p, q) is done in $\mathcal{O}(1)$, since if we assume that p and q are roots, then **Find**(p) and **Find**(q) are necessarily $\mathcal{O}(1)$. We can also assume p and q are pointers to the root node and then it is also obvious why the complexity is $\mathcal{O}(1)$.

Remark 8.7 (Implementing up-trees). Note that if the number of pointers is known beforehand, we can implement the up-trees from Example 8.6 by maintaining two arrays as follows:



Then when a **Union**(5,6) is executed we update the arrays in the following way:



Proposition 8.5. In a forest of up-trees with total n nodes and max height h that is constructed by merging the smaller trees into the bigger trees we have $h \leq \log_2 n$.

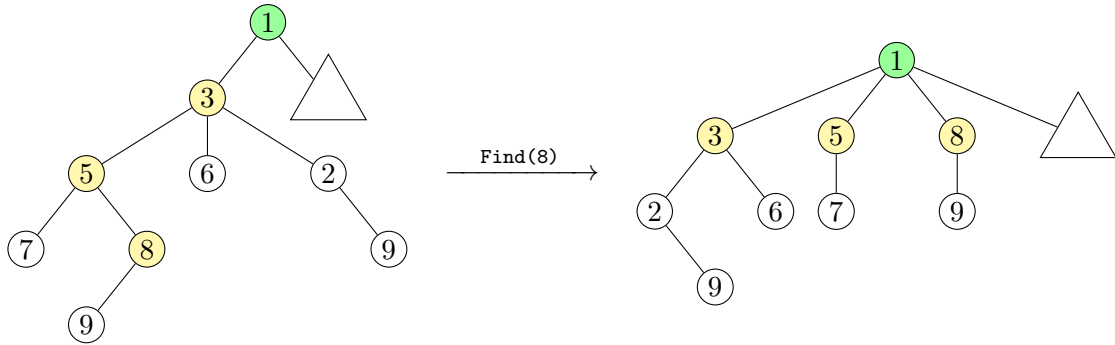
Proof. Note that if we merge the smaller tree into the bigger tree then any key x in the forest cannot have its parent changed more than $\log_2 n$ times.

Since every **Union** of trees implies that all the nodes of the smaller tree have their depths increased by exactly 1, and every node starts at depth 0 (all keys start as up trees with a single node), we get that the maximal depth of a key is $\log_2 n$, which implies that $h \leq \log_2 n$ and completes the proof. $\square$

Corollary 8.6. *The worst-case time complexity of the **Find** operation is $\mathcal{O}(\log n)$, where n is the total number of nodes in the data structure.*

Remark 8.8 (Path compression). We can also make a final modification to the final implementation of the union-find data structure introduced in Example 8.6 known as *path compression*. With path compression we implement **Find**(i) a bit differently. We start by following $A[i]$ then going up the tree in order to find the root, but then once we have the root we follow $A[i]$ again and go up the tree a second time, but now we also update the parent of each node in our path to be the root. Since **Find**(i) takes two traversals to the root instead of one its complexity remains $\mathcal{O}(h) = \mathcal{O}(\log n)$.

Example 8.7 (Path compression). Here is an example of path compression:



Definition 8.2 (Iterated logarithm). The iterated logarithm function $\log^* n$ (read “log star of n ”), is defined as the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1. Formally, we can define it recursively as follows:

$$\log^* n := \begin{cases} 0 & \text{if } n \leq 1, \\ 1 + \log^*(\log n) & \text{if } n > 1. \end{cases}$$

Remark 8.9. When we talk about $\log^* n$ we always mean the iterative binary logarithm, that is, the iterative logarithm with respect to base 2.

Remark 8.10. Although the function $\log^* n$ is monotonically increasing, for any practical value of n we have $\log^* n \leq 5$. In fact, the binary iterated logarithm is equivalent to the inverse of the infamous tetration operation (up to rounding). For example:

$$\begin{aligned} \log^*(1) &= 0 & \log^*(2) &= 1 & \log^*(2^2) &= 2 \\ \log^*(2^{2^2}) &= 3 & \log^*(2^{2^{2^2}}) &= 4 & \log^*(2^{2^{2^{2^2}}}) &= \log^*(2^{65536}) = 5 \end{aligned}$$

Proposition 8.7. *In the fourth implementation of the union-find data structure with union by rank and path compression, the time complexity of m **Union/Find** operations is bounded by $\mathcal{O}(m \log^* n)$. In particular, the amortized time of each operation is bounded by $\mathcal{O}(\log^* n)$.*

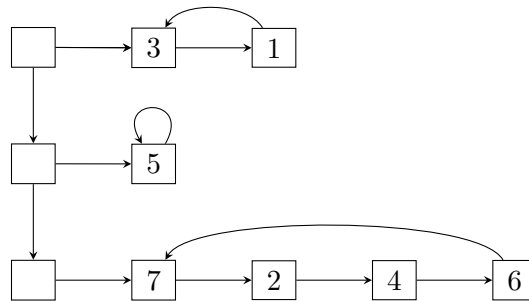
Proof. The proof is omitted. Nonetheless, it is important to appreciate how powerful path compressions are. For practical values of n we can *informally* say $\mathcal{O}(\log^* n) \approx \mathcal{O}(1)$. $\square$

Remark 8.11. A tighter bound than the one in Theorem 8.7 can be obtained using Ackermann functions.

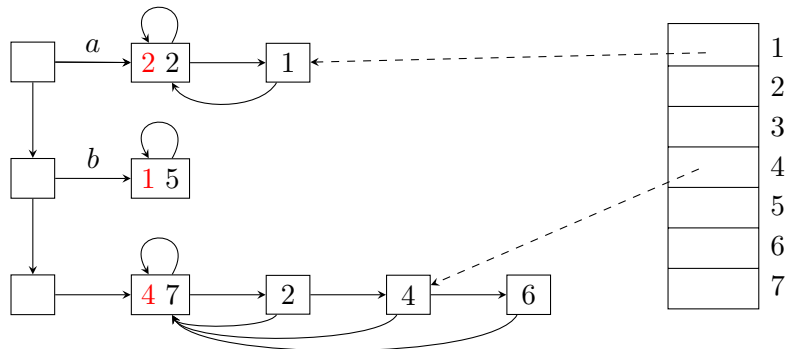
Remark 8.12 (Conclusion). In this section, we have introduced four different implementations for the union-find data structure (not including the non-modified versions). In the first implementation (Example 8.2):

1	4
2	12
3	4
4	12
5	5
6	12
7	12

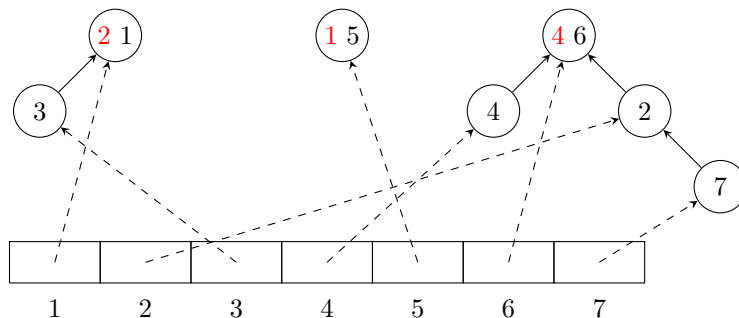
we achieved **Makeset** in $\mathcal{O}(1)$, **Find** in $\mathcal{O}(1)$, and **Union** in $\mathcal{O}(n)$. In the second implementation (Example 8.3):



we achieved **Makeset** in $\mathcal{O}(1)$, **Find** in $\mathcal{O}(n)$, and **Union** in $\mathcal{O}(1)$. In the modified third implementation (Example 8.5):



we achieved **Makeset** in $\mathcal{O}(1)$, **Find** in $\mathcal{O}(1)$, and **Union** in (amortized) $\mathcal{O}(\log n)$. In the modified fourth implementation (Example 8.6):



we achieved **Makeset** in $\mathcal{O}(1)$, **Find** in $\mathcal{O}(\log n)$, and **Union** in $\mathcal{O}(1)$, however, the amortized complexity of the **Union/Find** operations is $\mathcal{O}(\log^* n)$.

Exercise 8.2. There are n balls numbered $1, 2, \dots, n$ in a dark room. Every ball is painted in one of two possible colors, but we are not able to tell the color of each ball by ourselves since the room is too dark. Implement a data structure that supports the following operations:

Init (n)	Initialize a dark room with n balls of unknown colors. Required complexity: $\mathcal{O}(n)$.
Same (i, j)	This command returns nothing, but it announces that balls i and j have the same color.
Different (i, j)	This command returns nothing, but it announces that balls i and j have different colors.
AreSameColor (i, j)	Returns TRUE if i and j are of the same color, and FALSE if they have different colors. If it is currently impossible to know whether i and j have the same color or not, return UNKNOWN .

The required amortized complexity of **Same**, **Different**, and **AreSameColor** is $\mathcal{O}(\log^* n)$.

Solution. It is clear that we need to use the union-find data structure. We are going to initialize the structure with n singletons, and maintain them such that each node also has a pointer (which starts as a null pointer) that points to a set with all the nodes that are painted in a color different than its own. We will also maintain the up-trees such that each tree (set) will only contain nodes that are known to be painted in the same color. Finally, since in each tree all balls are of the same color, the extra pointers at each node in a certain tree should all point to the same set, so instead of maintaining the extra pointer and set at every node, we would only maintain the extra pointer at the root, and leave the extra pointers of the other nodes as null pointers. The exact implementation of the operations is as follows:

Init(n) – Initialize a union-find data structure by **Makeset**(i) for all $1 \leq i \leq n$, and let $i.\text{diff} = \text{null}$ (the pointer to the set of balls painted in the other color is initialized to be the null pointer).

AreSameColor(i, j) – Let G_i and G_j be the roots (representing the sets) of i and j respectively. If $G_i == G_j$ return **TRUE**, if $G_i == G_j.\text{diff}$ return **FALSE**, and otherwise return **UNKNOWN**.

Same(i, j) – Let G_i and G_j be the roots (representing the sets) of i and j respectively. Then perform **Union**(G_i, G_j) and **Union**($G_i.\text{diff}, G_j.\text{diff}$), and then update the pointers of the merged sets respectively.

Different(i, j) – Let G_i and G_j be the roots (representing the sets) of i and j respectively. Then perform **Union**($G_i, G_j.\text{diff}$) and **Union**($G_i.\text{diff}, G_j$), and then update the pointers of the merged sets respectively. Note that resulting two sets are the **diff** of one another.

Now we proceed to analyze the complexity of the operations. At **Init**(n) we use **Makeset**(i) exactly n times, so its complexity is $\mathcal{O}(n)$ as wanted. Similarly, in every sequence of the **Same**, **Different**, and **AreSameColor** of length m we perform a linear (in m) amount of **Union** and **Find** operations (and some $\mathcal{O}(1)$ operations for managing the pointers), and therefore their amortized complexity is $\mathcal{O}(\log^* n)$ as wanted.

It is left to show correctness of the data structure. We can use structural induction in order to show that the result of **AreSameColor**(i, j) is always true (whether it be **TRUE**, **FALSE**, or **UNKNOWN**).

8.2 Exercises

Exercise 8.3. Suggest a modification to the standard union-find implementation, that allows finding the maximal element in each set in $\mathcal{O}(\log^* n)$ time complexity (amortized with the union and find operations).

Solution. We use the same structure of a union-find with up trees and path compressions, and we additionally maintain a **max** field at the root which stores the maximal value of the elements in its up tree. The implementation details are as follows:

Find(i) As in the standard implementation.

Max(i) Perform **Find(i)** and return the **max** value at the root.

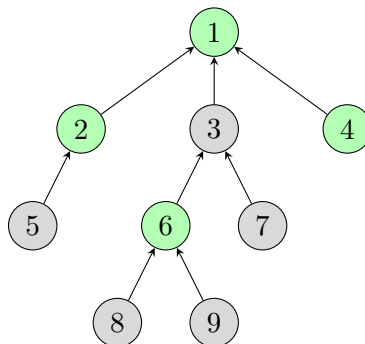
Union(Q) Perform standard **Union(p,q)** and update **max** to be the maximal value out of the **max** values of the roots the two sets.

Exercise 8.4. Suppose we have a family tree of numbers given as a directed up tree. At first assume that all members of the tree are still *alive*. Implement a data structure that support the following operations:

Close(x) Declares **x** as *dead*.

Master(x) Returns the closest living relative of **x**. If **x** is alive then **Master(x) = x**.

Here is an example of a possible family tree (green nodes are alive and gray nodes are dead):



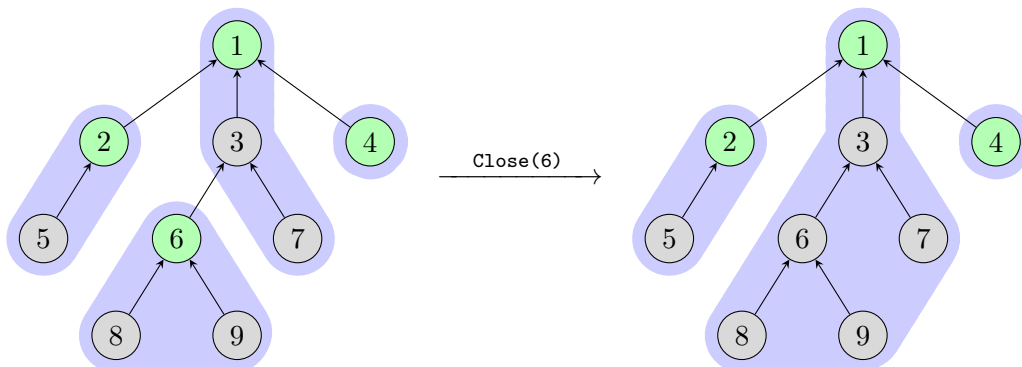
In this case **Master(7) = 1** and **Master(5) = 2**.

Solution. Our solution is based on the union-find data structure. In initialization we make a set for each node in the (family) tree, so that the root of each tree is the master of all nodes in the (union-find) tree (this is true in the beginning since **Master(x) = x** at initialization). The implementation details are as follows:

Master(x) Return the root of the union-find tree of **x**.

Close(x) Let **y** be the parent node of **x**, then we perform **Union(Find(x), Find(x))**.

For example we have that:



Since all operations use a constant amount of union-find operations, their amortized complexity is $\mathcal{O}(\log^* n)$ as needed.

Exercise 8.5. There are n given boxes with different sides of lengths $\{(l_i, w_i, h_i)\}_{i=1}^{10}$ for $(l_i, w_i, h_i) \in (0, \infty)^3$. The goal is to allow management of towers we build out of these boxes. The rules for building a tower are as follows:

- (1) Each box has exactly one face marked with “This Face Up”, meaning that when building a tower this fact must be facing up.
- (2) A box can be put above another if and only if its down bottom face is strictly contained in the top face of the box below it.

We need a data structure which supports the following operations:

<code>Init(Boxes, n)</code>	Initializes the n given boxes. Each box is given an identifier $1, \dots, n$, and each box counts as a tower with one box.
<code>PutOnTop(B1, B2)</code>	Puts the tower B1 belongs above the tower that contains B2 if the rules for building towers are satisfied.
<code>TowerHeight(B)</code>	Returns the height of the tower that contains B.
<code>HeighFromGround(B)</code>	Returns the height of the bottom face of B from the floor.

The required time complexity for `Init(Boxes, n)` is $\mathcal{O}(n)$, and the rest of the operations must be $\mathcal{O}(\log^* n)$ amortized complexity.

Solution. Note that we can solve the problem easily if we ignore the `HeighFromGround` operation. We use a union-find structure where each set also stores the size of the top and bottom faces of the tower, and also the height of the tower. In this way we can implement the operations as follows:

<code>Init(Boxes, n)</code>	Perform <code>Makeset</code> for each box. Initialize the additional tower information according to the sizes of the box.
<code>PutOnTop(B1, B2)</code>	First perform <code>T1 <- Find(B1)</code> and <code>T2 <- Find(B2)</code> . We can now check in $\mathcal{O}(1)$ if the building rules are satisfied, and if the are we perform <code>Union(T1, T2)</code> , and update the tower information as needed.
<code>TowerHeight(B)</code>	Perform <code>Find(B)</code> and return the height of the tower which we always maintain.

Since the operations use a constant amount of union-find operations, the complexity conditions are satisfied. Implementing `HeighFromGround` is a more difficult problem.

First of all, we can always maintain the height of the boxes that are roots of up trees in the union-find structure. That is because we know which box is going to be the root after each union, and we can update its height from the ground either by not changing it (if it is the bottom tower in the union), or by adding to it the height of the second tower (if it is the upper tower in the union). We can now answer `HeighFromGround` for boxes that are roots of up trees.

In order to determine the height of each box from the ground, we make a couple of observations:

- (1) In an up tree, the relative distance of a box from the box it points to remains the same (up to path compressions!).
- (2) When perform a `Union`, the absolute heights of both towers is known, hence we can compute the relative distance of the root of the small tree from the bigger tree.

- (3) If each *edge* (not node) would store the relative height for the source from the sink, then the sum of the values on the edges the form the path from each node to the root would equal to relative height of the node from the root. Note the this value can be computed as explained in observation (2) up to path compressions which we will address soon.
- (4) Since by previous discussion we can compute the height of roots of up trees, together with observation (3) we could compute **HeighFromGround** of every box up to path compressions.
- (5) It is finally time to openly address path compressions (pun intended). Recall that when doing a path compression we traverse the path from the given node to the root twice. We modify the path compression algorithm so that in the first traversal we compute the relative distance of the node from the root, and in the second traversal when we attach the first node to the root we put the sum we got on the edge from that attachment, then for the second node we can subtract the relative distance of the first node from the second node from the current sum, and since by construction that is the relative distance of the second node from the root, we can put that value on the edge we use to attach the second node to the root (remember we are in a compression). We keep subtracting the values from each edge on the path from the sum one by one and update the values of edges we create in the compression process until we reach the root, and by our construction these values would represent the relative distance of these nodes from the root, which completes the proof of correctness of our algorithm.

There is not reason to try to understand these observations with numerical examples, the intuition comes from understanding the construction. These five observations are also a detailed implementation of **HeighFromGround** in $\mathcal{O}(\log^* n)$ time. This completes the exercise.

Exercise 8.6 (Spring 14, A, 3). A train consists of k carriages and a total of n seating sections ($n \geq k$). Each carriage contains exactly n_i seating sections. Each section contains exactly 6 seats. There are k entrances to the train, with one entrance located at the front of the first seating section of each carriage.

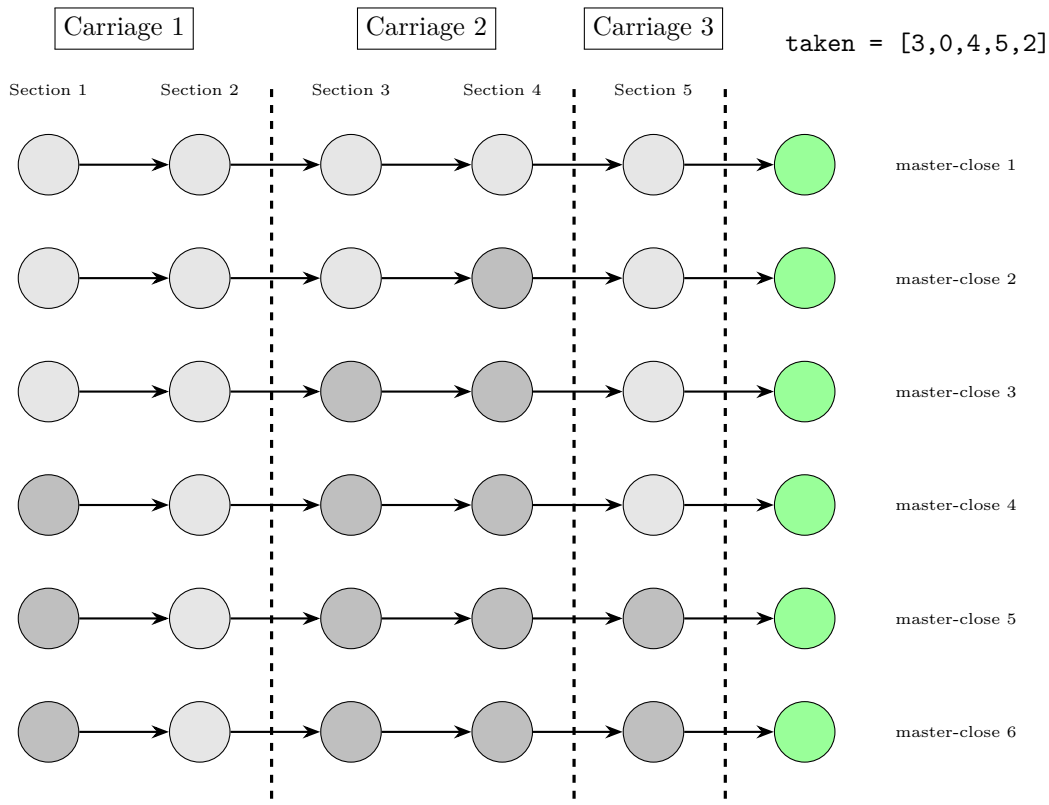
Groups of x passengers ($1 \leq x \leq 6$) board the train one-by-one (shocking) seeking a single seating section with enough vacant seats to accommodate the entire group. They may choose to board through any of the k entrances. From their entry point, they move toward the right (toward the k th carriage) until they either find a section with sufficient space or reach the end of the train.

Implement a data structure that supports the following operations:

- Init(k,sizes)** Initializes the structure with an empty train with k carriages. **sizes** is an array of size k so that **sizes[i] = n_i**.
Time complexity: $\mathcal{O}(n)$.
- Board(x,i)** A group of $1 \leq x \leq 6$ passengers boards the train through the entry at the i th carriage. This operation updates the train so that the group seats at the first seating section with at least x vacant seats.
Time complexity: Amortized $\mathcal{O}(\log^* n)$.

Solution. In this case we have to notice that there are 6 hierarchical structure to the seating sections. Each structure $1 \leq i \leq 6$ contains all seating sections with exactly i available seats. When each group of x passengers boards the train, they need access to the leftmost seating section in the x th hierarchical structure. In order to implement the 6 hierarchical structures, we can use 6 master-close structures, and some more arrays in order to maintain all necessary data for the train.

The structure would contain the following:



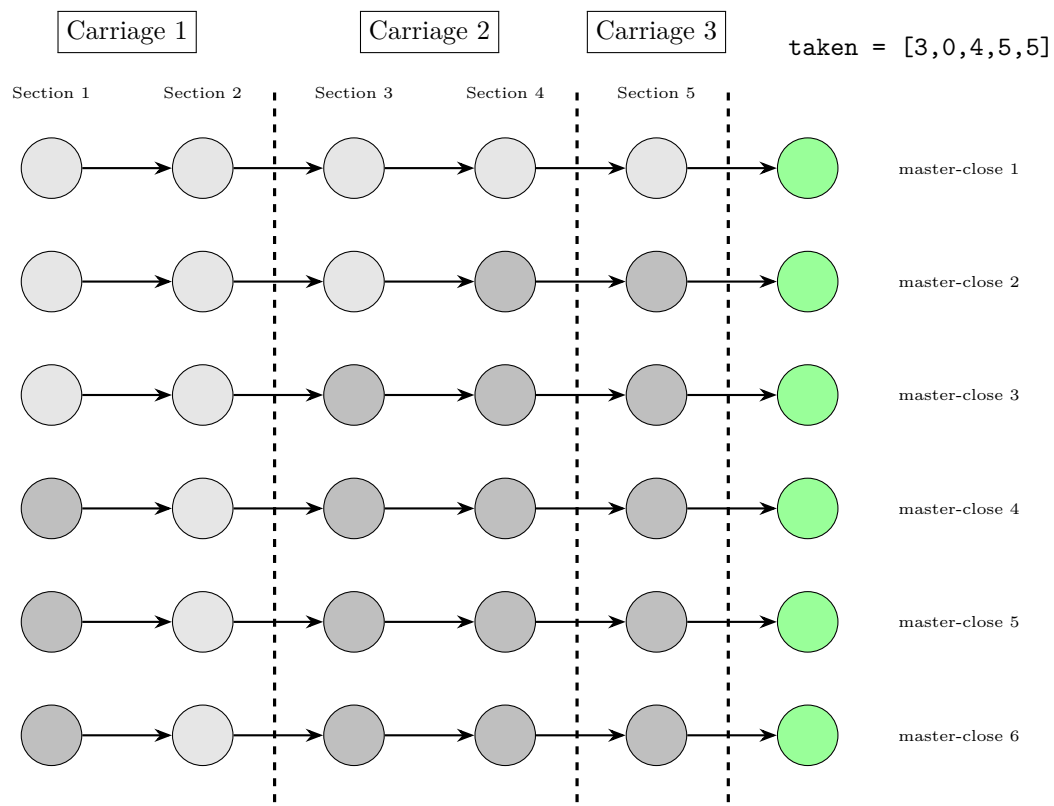
In the diagram we see an example of a train with 3 carriages, The first carriage has 2 seating sections, the second has 2 and the third has 1, so in total we have $n = 5$ seating sections. The seating sections have **taken** = $[3, 0, 4, 5, 2]$ seats taken respectively. We will also compute this array in practice for reasons you will see in the implementation. The i th master-close structure is a chain of $n = 5$ nodes, so that we mark a node in light gray if it has i vacant seats, dark gray if it does not have i vacant seats, and finally since we mentioned a master-close structure must always have an active root, we add a dummy root and the root of each structure in green. Note that the boxed “Carriage 1” etc. at the top of the diagram each represent an array of 6 pointers to the 6 nodes in all the master-close structures corresponding to the first seating section in that carriage. All of these arrays can be also put into an array of arrays called **Carriage**.

The implementation details are as follows:

- Init(k,sizes)** We initialize all the master-close structures we described, the auxiliary arrays etc.
- Board(x,i)** We use the **Carriage[i]** array of pointers to access the node of the first seating section of carriage i in the x th master-close structure. We call this node v . We then perform **Master(v)** and reach some node u which is by construction the first seating section with sufficiently vacant seats. Suppose u corresponds to seating section k , we update **taken[k]**, and it is left to update the master-close structure. Since we know how many seats are vacant in k (say we have $t = 6 - \text{taken}[k]$ vacant seats), we simply execute **Close(w)** for all nodes w corresponding to seating section k in the master-close structures with index $6, 5, \dots, t - 1$.

Let us consider an example to make sure we understand why everything works. In case the train is in the state described by the previous diagram and we perform **Board(3,2)**, we use **Carriage[2]** to access the node corresponding to section 3 in the third master-close and apply **Master** on it and get the node corresponding to section 5 in the third master-close structure. We now update **taken[5]** from 2 to 5 since 3 people joined this seating section, and there is now

$6 - \text{taken}[k] = 1$ empty seats in section 5, so we must perform **Close** on the node corresponding to section 5 in master-close structures 6, 5, 4, 3, 2. After this operation the structure would look like:



9 Heap sort and Quick sort

We already know about naïve sorting algorithms in $\mathcal{O}(n^2)$ such as bubble sort:

```

1 void BubbleSort(int* A, int n) {
2     for (int i = 0; i < n-1; i++) {
3         for (int j = n-2; j >= i; j--) {
4             if (a[j] > a[j+1])
5                 swap(&a[j], &a[j+1]);
6         }
7     }
8 }
```

Remark 9.1. In practice, we usually don't sort an array of numbers, rather we want to sort objects according to a certain field, so when we sort numbers you may assume in the back of your head that each number has a pointer to the rest of the data.

Definition 9.1 (Priority queue). A priority queue is an abstract data type that stores objects from the object of lowest priority the object of highest priority. It supports the following operations:

- Create(A)** Create an empty priority queue, or take n parameters and create a priority queue that stores them.
- Insert(Q,x)** Insert x into the queue according to its priority.
- Max(Q)** Return the element of highest priority.
- DelMax(Q)** Delete the element with highest priority.

Example 9.1 (Jobs). A priority queue is a good way to store computer *jobs*. A computer job is a certain task that the computer has to complete, and ideally we would like to have some task prioritized over other tasks. If we store all jobs in a priority queue according to their priority we could tell the computer to always complete the job with the highest priority first.

We could implement a priority queue using a balanced search tree. In this case any of the **Insert/Max/DelMax** operations can be completed in $\mathcal{O}(\log n)$ time. We could also implement a priority queue using a skip list. However, the focus of the following subsection is on the implementation of a priority queue using a heap.

Definition 9.2 (Heap). A heap is a tree based data structure that supports the following operations:

- MakeHeap(A)** Create an empty heap, or take n parameters and create a heap that stores them.
- Insert(Q,x)** Insert x into the queue according to its key.
- Max(Q)** Return the element with the highest key value.
- DelMax(Q)** Delete the element with the highest key value.

Remark 9.2. Sometimes we also add the command **DecKey(p,x)** which decreases the key of a certain element to x (and does nothing if the value at p is greater than x). This can be done in $\mathcal{O}(\log n)$ by changing the value of the node, then sifting-up once from the node p represents. Sifting up means that while the value at p is smaller than the value of its parent you swap it with its parent (this process continues indefinitely, possibly ending at the root if x is smaller than the root).

Remark 9.3. As mentioned, we could use a heap to implement a priority queue, however, when it was first constructed in a paper in 1964, it was constructed in order to write a new sorting algorithm, which we will learn in the following subsection.

9.1 Heap sort

The heap sort is a sorting algorithm that uses a heap, however, it is clear that the complexity of this algorithm depends on the specific implementation of the heap.

Definition 9.3. Given an implementation for a heap, the heap sort algorithm is:

Algorithm 3: Heap Sort

```

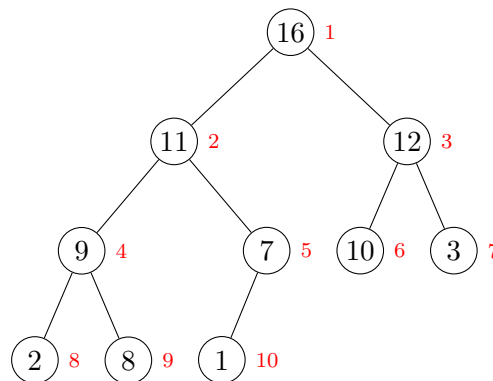
1 Q = MakeHeap()
2 foreach  $x$  in input do
3   | Insert(Q,  $x$ )
4 end
5 while  $Q$  is not empty do
6   | print(Max(Q))
7   | DelMax(Q)
8 end
```

Remark 9.4. With the naïve implementation using a balanced search tree each insertion/deletion/lookup takes $\mathcal{O}(\log n)$ time, so the time complexity of heap sort with this implementation is $\mathcal{O}(n \log n)$, and in fact, it is $\Theta(n \log n)$.

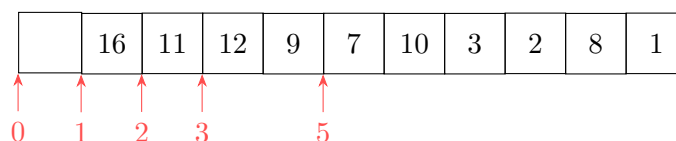
Remark 9.5. We will soon see that in the standard implementation of the heap, we can implement $\text{Max}(Q)$ in $\mathcal{O}(1)$ time and $\text{MakeHeap}(A[1, \dots, n])$ in $\mathcal{O}(n)$ time (with the binary search tree implementation Max takes $\mathcal{O}(\log n)$ and MakeHeap takes $\mathcal{O}(n \log n)$). Note that the command $\text{MakeHeap}(A[1, \dots, n])$ replaces lines 1-4 in Algorithm 3.

Definition 9.4 (Standard heap implementation, part 1). A heap is implemented as an almost complete binary tree that satisfies a *heap property*. The heap property in a *max-heap* is that each parent node has a higher key value than the key values of its children.

Example 9.2. The following almost complete binary tree satisfies the (max) heap property and is thus a heap:



Remember that we saw in Example 3.11 how to represent an almost complete binary tree as an array:



the parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$, the left child of $A[i]$ is $A[2i]$, and the right child of $A[i]$ is $A[2i+1]$. In a standard programming language if i is of type `int` we have that the parent of $A[i]$ is simply $A[i/2]$ since the flooring is automatic.

Remark 9.6. A clear advantage of a heap over a balanced search tree is that we can access the maximum element in $\mathcal{O}(1)$ time. It is also easier to balance a heap, and we will soon see that we can also initialize a heap quickly.

Definition 9.5 (Sift-down). The sift-down operation is an operation that takes the root of an almost complete binary tree where all nodes except the root v satisfy the heap property, and turns it into an almost complete binary tree where all nodes satisfy the heap property (a heap).

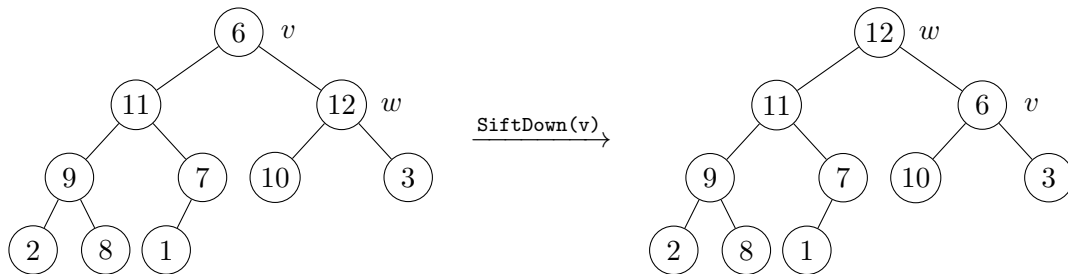
Algorithm 4: Sift-down

```

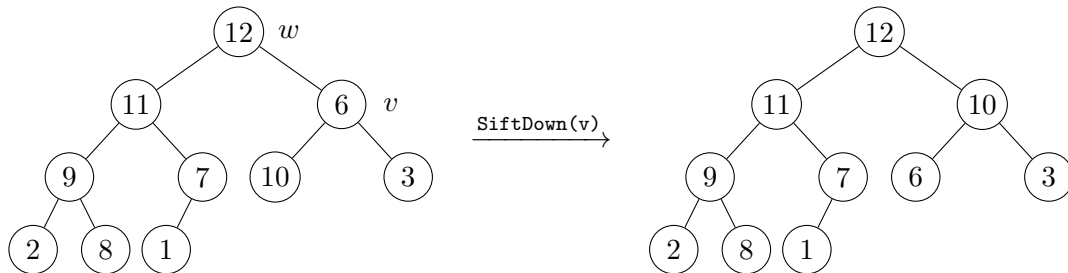
1 if  $v$  is a leaf or  $v$  satisfies the heap condition then
2   | return
3 end
4 Exchange  $v$  with its child of maximum value
5 call Sift-down on that child

```

Example 9.3 (Sift-down example). In the following almost complete tree, note that only the root v violates the heap property. Therefore when sifting down v we first exchange it with its child of maximum value w .

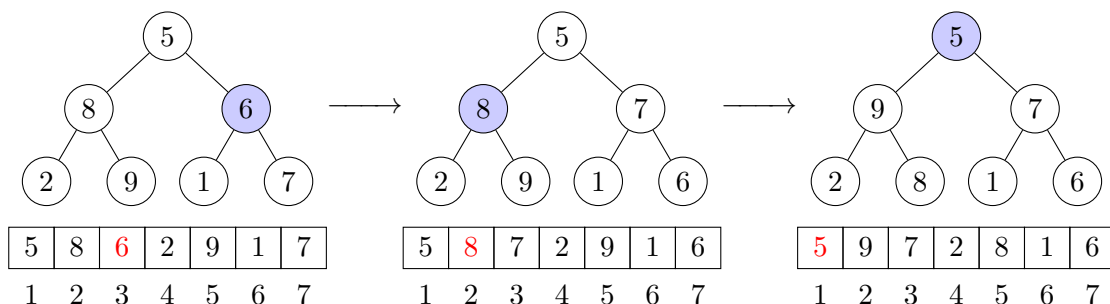


Then we get a new almost complete tree but since 6 still violates the heap property we sift it down again and change it with its child of maximum value 10 as follows:

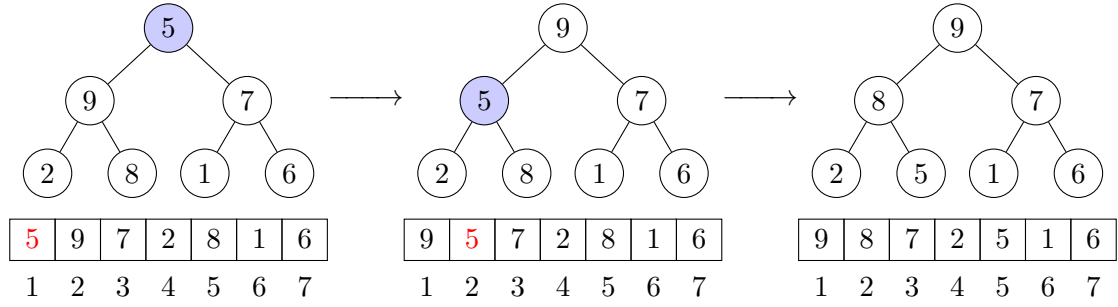


Now since 6 is a leaf the algorithm returns, and indeed the result is a heap.

Definition 9.6 (MakeHeap implementation). We implement **MakeHeap** by first initializing an array to represent an almost complete binary tree, and then we turn that almost complete binary tree into a heap by sifting down all internal nodes, processed from the last internal node up to the root (in reverse level order). For example:



where the blue nodes are the ones we sift down in each iteration. The algorithm continues:



Remark 9.7 (Correctness). The correctness of **MakeHeap** is guaranteed since in Definition 9.6 we required sifting down all the nodes in an order so that before sifting down a node, we first sift down its children, so by correctness of Algorithm 4 the algorithm for **MakeHeap** is also correct.

Remark 9.8 (Complexity analysis). When we first initialize the array for an almost complete binary tree it costs us $\mathcal{O}(n)$. Then each sift-down operation takes $\mathcal{O}(h(v))$ where $h(v)$ is the height of the node we sift down v . Note that in an almost complete binary tree with $n = 2^{h+1} - 1$ nodes (we assume it has the most amount of nodes since we are trying to analyze worst-case complexity), there are 2^i nodes at height $h - i$. Therefore the complexity of all sift-down operations combined is:

$$S = \sum_{i=0}^h 2^i(h-i) = \sum_{i=0}^{h-1} 2^i(h-i) = 1 \cdot h + 2 \cdot (h-1) + \dots + 2^{h-1} \cdot 1,$$

note that

$$2S = 2 \cdot h + 4 \cdot (h-1) + \dots + 2^h \cdot 1.$$

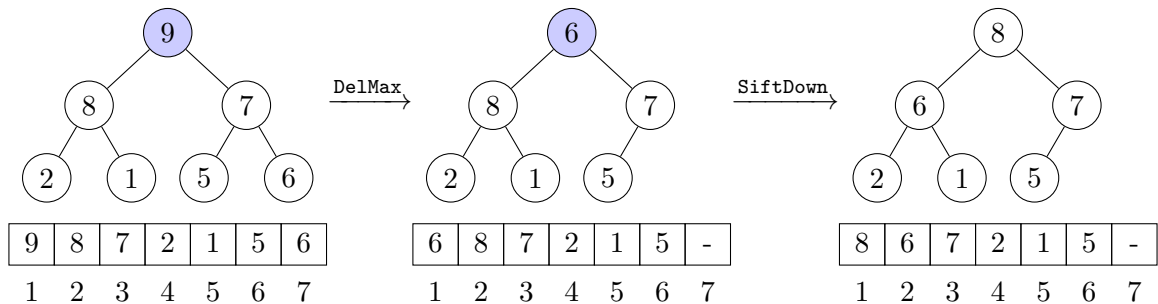
Therefore, by subtracting the first equation from the second we get

$$\begin{aligned} S &= \left(2 \cdot h + 4 \cdot (h-1) + \dots + 2^h \cdot 1\right) - \left(1 \cdot h + 2 \cdot (h-1) + \dots + 2^{h-1} \cdot 1\right) \\ &= -h + \left(2 \cdot h + 4 \cdot (h-1) + \dots + 2^h \cdot 1\right) - \left(2 \cdot (h-1) + \dots + 2^{h-1} \cdot 1\right) \\ &= -h + (-1 + 1) + 2 + 4 + \dots + 2^h \\ &= -h - 1 + \underbrace{2^{h+1} - 1}_{=n} < n, \end{aligned}$$

so the complexity of **MakeHeap** is $\mathcal{O}(n)$ and in fact it is $\Theta(n)$.

Definition 9.7 (**Max(Q)** implementation). Return the root node. This is done in $\mathcal{O}(1)$ complexity.

Definition 9.8 (**DelMax(Q)** implementation). Delete the root and replace it with the rightmost leaf in the tree. Then perform sift-down on the new root in order to satisfy the heap property. For example:



Remark 9.9 (HeapSort complexity). Recall that the HeapSort algorithm is as follows:

Algorithm 5: Heap Sort

```

1 Q = MakeHeap()
2 foreach x in input do
3   Insert(Q,x)
4 end
5 while Q is not empty do
6   print(Max(Q))
7   DelMax(Q)
8 end

```

Since initializing the heap takes $\mathcal{O}(n)$ and executing DelMax n times takes $\mathcal{O}(n \log n)$ time, the total time complexity is $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$.

The space complexity is $\mathcal{O}(n)$ since we only use an array of length n and $\mathcal{O}(1)$ temporary pointers.

Remark 9.10 (HeapSort implementation). We can implement HeapSort more explicitly as follows:

```

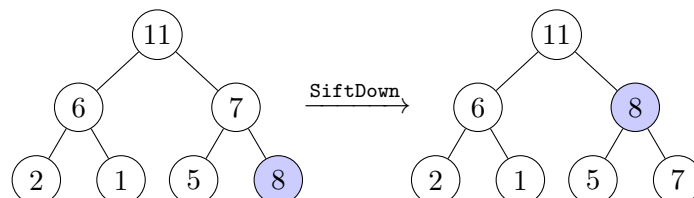
1 void HeapSort(int n) {
2   for (int i = n/2; i > 0; i--) {
3     sift_down(i,n);      /* initialize the heap */
4   }
5   for (int i = n; i > 0; i--) {
6     swap(1,i);          /* the sorting process */
7     sift_down(1,i-1);
8   }
9 }

```

where `sift_down(i,n)` takes the index of the problematic node i and size of heap n . The reason the first for-loop is over the first $\lfloor n/2 \rfloor$ elements, is because these are the internal nodes, and the rest are leaves. The second for-loop move the root (maximal element) to the end of the array, the “pretends” it works with a new heap represented by $A[1, \dots, n-1]$ and repeats the process until the array is sorted.

Remark 9.11 (Why use HeapSort). The HeapSort algorithm is helpful because its worst-case complexity is $\mathcal{O}(n \log n)$, and the space complexity is as minimal as possible: $\mathcal{O}(n)$.

Definition 9.9 (Insertion). When inserting to a heap we create a new leaf in the first available place on the lowest level with the inserted value. Then we perform the sift-up operation in order to satisfy the heap property. For example:



We will elaborate more on the sift-up operation in the recitations.

Definition 9.10 (Min-Heap). A min-heap is a tree based data structure that supports the

following operations:

MakeHeap(A) Create an empty heap, or take n parameters and create a heap that stores them.
Insert(Q,x) Insert x into the queue according to its key.
Min(Q) Return the element with the lowest key value.
DelMin(Q) Delete the element with the lowest key value.

Remark 9.12. We implement a min-heap exactly how we would implement a max-heap (what are used to simply call a heap), except that the heap property is now different. The heap property in a *min-heap* is that each parent node has a lower key value than the key values of its children.

Remark 9.13 (HeapSort with a min-heap). We can implement the HeapSort algorithm with a min-heap as follows:

Algorithm 6: Heap Sort with a min-heap

```

1 Q = MakeHeap()
2 foreach  $x$  in input do
3   Insert(Q,x)
4 end
5 while  $Q$  is not empty do
6   print(Max(Q))
7   DelMax(Q)
8 end
```

9.2 Quick sort

The quick sort algorithm uses the divide and conquer methodology. In each step, we sort an array $A[1, \dots, r]$, then pick a pivot element $A[p]$ for some $l \leq p \leq r$, either by random, or simply by always choosing $p = 1$. We then sort $A[1, \dots, r]$ such that all elements smaller than the pivot element $A[p]$ are in $A[1, \dots, i-1]$, and all elements bigger than the pivot element $A[p]$ are in $A[i, \dots, r]$, such that the pivot element will be in $A[i]$. We then proceed to use quick sort on $A[1, \dots, i-1]$ and $A[i, \dots, r]$ recursively.

Algorithm 7: Quick sort

Input: Array $A[1 \dots n]$
Output: Sorted array A

```

1 Function Quicksort( $A, l, r$ ):
2   if  $l \geq r$  then
3     return;
4   else
5      $p \leftarrow \text{ChoosePivot}(A, l, r)$ ;
6      $i \leftarrow \text{Partition}(A, l, r, p)$ ;
7     Quicksort( $A, l, i - 1$ );
8     Quicksort( $A, i + 1, r$ );
```

Remark 9.14. The function **Partition** returns the index i of the pivot element after sorting the elements according to the pivot element.

Remark 9.15 (QuickSort correctness). Note that after **Partition**(A, l, r, p) the pivot element $A[p]$ is sorted into its correct position i . We then recursively apply the algorithm so that all elements are eventually pivot elements, and therefore after the algorithm the output would be a sorted array.

Remark 9.16 (QuickSort complexity). We cannot tell the complexity of QuickSort without knowing the specific implementations of Partition and ChoosePivot, but we will soon see that:

- (1) In the worst case scenario the algorithm is $\Theta(n^2)$.
- (2) If we always pick the pivot to be approximately the middle element, the algorithm is $\mathcal{O}(n \log n)$.
- (3) If we choose the pivot element uniformly at random, we would get an expected complexity $\mathcal{O}(n \log n)$ (complexity of a randomized algorithm).

Remark 9.17 (Implementation of Partition). Here is the standard implementation of the Partition operation in $\Theta(n)$ time:

```

1  int partition(KEY *A, int left, int right, int pivot) {
2      swap (A+pivot, A+right); /* move pivot to end of array */
3      int i = left;
4      for (int j = left; j < right; j++) { /* sort according to
5          pivot */
6          if (A[j] < A[right]) {
7              swap(A+i, A+j);
8              i++;
9          }
10     }
11     swap(A+right, A+i); /* put pivot in correct place */
12     return i; /* return correct index of pivot */
13 }
```

Example 9.4 (Partition algorithm example). Here is an example of the partition algorithm in action, where the pivot is chosen to be the leftmost element, in this case 7, and the array is:

7	11	12	6	9	1	3	2	8	10
---	----	----	---	---	---	---	---	---	----

First we move the pivot element to the right:

10	11	12	6	9	1	3	2	8	7
----	----	----	---	---	---	---	---	---	---

After moving it to the right we scan the array from the left until finding an element smaller than the pivot, in this case 6, and then switch 6 with the element in the first cell of the array:

6	11	12	10	9	1	3	2	8	7
---	----	----	----	---	---	---	---	---	---

Then we increment i , and continue the scan, until we find the next element in the array smaller than the pivot, in this case 1, and switch it with the second element in the array:

6	1	12	10	9	11	3	2	8	7
---	---	----	----	---	----	---	---	---	---

We then continue the process until the scan is complete:

6	1	3	10	9	11	12	2	8	7
---	---	---	----	---	----	----	---	---	---

6	1	3	2	9	11	12	10	8	7
---	---	---	---	---	----	----	----	---	---

At this point the scan is complete since none of the following keys are smaller than 7, so it is left to put the pivot in its correct position:

6	1	3	2	7	11	12	10	8	9
---	---	---	---	---	----	----	----	---	---

Remark 9.18 (QuickSort worst-case analysis). Suppose that we are given an array $A[n]$, and that we choose the pivot element to be the leftmost element in the array. Then according to Algorithm 7 in every recursive call we would have to sort the subarrays $A[1, l-1]$ and $A[l+1, n]$. Since sorting $A[1, l-1]$ takes $\mathcal{O}(1)$ (since $l > l-1$), we can ignore it when computing the complexity, and infer that if $T(n)$ is the time complexity of QuickSort for an array of size n then $T(n) = \Theta(n) + T(n-1)$, and thus,

$$T(n) = cn + T(n-1) = cn + c(n-1) + T(n-2) = \dots = c \sum_{i=1}^n i = \Theta(n^2).$$

Remark 9.19 (QuickSort analysis, median pivot). Suppose that we are given an array $A[n]$, and that we choose the pivot element to be the median element in the array. In this case we get the relation $T(n) = 2T(n/2) + \Theta(n)$, and thus $T(n) = \Theta(n \log n)$. Similarly, if the pivot is some element such that 10% of the array is smaller than the pivot, and 90% of the array is larger than the pivot then we get that $T(n) = T(n/10) + T(9n/10) + \Theta(n)$, and similarly by induction we get that $T(n) = \Theta(n \log n)$.

Remark 9.20 (QuickSort analysis, random pivot). Suppose that we are given an array $A[n]$, and that we choose the pivot to be a random element chosen uniformly. We get that the *expected* time is $T(n) = \Theta(n) + \frac{1}{n} (\sum_{i=1}^n T(i-1) + T(n-i))$. Then, similarly to the process we have seen in Theorem 3.1, we can conclude that $T(n) = \Theta(n \log n)$.

Remark 9.21 (Space complexity of QuickSort). The recursion part takes $\mathcal{O}(1)$ space for local variables in each level of the recursion tree. We have seen in Remark 9.18 that in the worst-case scenario the recursion tree has n levels, so in total the space complexity is $\mathcal{O}(n)$. Asymptotically this isn't really a problem ($\mathcal{O}(n)$ space is good enough), but since recursion is very costly in time and space, we still use a way to limit the height of the recursion tree to $\log n$. We can do this by a method called *tail recursion elimination*.

Definition 9.11 (Tail recursion). A tail recursion is a recursive program in which the recursive call is the last command executed in the function.

Example 9.5 (Tail recursion). The following snippet of code is an example of tail recursion:

```

1 static void print(int n) {
2     if (n < 0)
3         return;
4     cout << " " << n;
5     print(n - 1);
6 }

```

Example 9.6 (Tail recursion nonexample). The following snippet of code is a nonexample of tail recursion:

```

1 unsigned int fact(unsigned int n) {
2     if (n <= 0)
3         return 1;
4     return n * fact(n - 1);
5 }

```

The reason this is not a tail recursion, is that the result of `func(n-1)` is still used in `func(n)`, so clearly calling `func(n-1)` is not the last operation in `func(n)`.

Remark 9.22 (Why tail recursion is good). The reason tail recursion is good, is because it allows us (or the compiler), to optimize the space complexity of the program. The reason is that in the case of tail recursive functions, when we follow the recursion into the next level of the recursion tree, by definition of tail recursion we don't need to store the information from the function that called us anymore. Also, even when the original recursive function is not tail recursive, it is usually possible to turn it into a tail recursive function by increasing the number of parameters. For example, if we consider the nonexample from Example 9.6, we could turn it into a tail recursive function as follows:

```

1 unsigned factTR(unsigned int n, unsigned int a) {
2     if (n <= 1)
3         return a;
4     return factTR(n - 1, n * a);
5 }
6 unsigned int fact(unsigned int n) { return factTR(n, 1); }
```

Here the first function is tail recursive, and the second function is simply a wrapper for the factorial function in order to make it easier to use.

Remark 9.23 (Back to QuickSort). Recall Algorithm 7. We can modify the algorithm as follows:

Algorithm 8: Quick sort, recursion on left part only

Input: Array $A[1 \dots n]$
Output: Sorted array A

```

1 Function Quicksort( $A, l, r$ ):
2     while  $l < r$  do
3          $p \leftarrow \text{ChoosePivot}(A, l, r)$ ;
4          $i \leftarrow \text{Partition}(A, l, r, p)$ ;
5         Quicksort( $A, l, i - 1$ );
6          $l \leftarrow i + 1$ ;
7     return;
```

In this way, we only enter recursion on the left part, and update l to handle the right part iteratively in-place. In this way, the recursion tree will only have left children. In practice, this is sometimes an improvement, but in theory, in the worst case scenario (if the pivot is always the rightmost element) then the height of the recursion tree will remain $\mathcal{O}(n)$, and on average it is still $\mathcal{O}(\log n)$. We could also eliminate the left part of the recursion by

Algorithm 9: Quick sort, recursion on right part only

Input: Array $A[1 \dots n]$
Output: Sorted array A

```

1 Function Quicksort( $A, l, r$ ):
2     while  $l < r$  do
3          $p \leftarrow \text{ChoosePivot}(A, l, r)$ ;
4          $i \leftarrow \text{Partition}(A, l, r, p)$ ;
5         Quicksort( $A, i + 1, r$ );
6          $r \leftarrow i - 1$ ;
7     return;
```

However, it is clear that this still isn't an improvement, and we need to optimize the algorithm further.

Remark 9.24 (Optimized QuickSort). The optimized version of quick sort (which is the standard implementation of quicksort in most programming languages) is as follows:

Algorithm 10: Quick sort, optimized

Input: Array $A[1 \dots n]$
Output: Sorted array A

```

1 Function Quicksort( $A, l, r$ ):
2   while  $l < r$  do
3      $p \leftarrow \text{ChoosePivot}(A, l, r)$ ;
4      $i \leftarrow \text{Partition}(A, l, r, p)$ ;
5     if  $i - l < r - i$  then
6       Quicksort( $A, l, i - 1$ );
7        $l \leftarrow i + 1$ ;
8     else
9       Quicksort( $A, i + 1, r$ );
10       $r \leftarrow i - 1$ ;
11  return;
```

In this way, we enter recursion on the left side if the subarray to the left of the pivot is smaller than the subarray to the right of the pivot, and enter recursion on the right otherwise. It follows that in each recursion step the size of the array is at most half of its size in the previous step, and thus the height of the recursion tree is bounded by $\log_2 n$. As mentioned in Remark 9.21, this improves the space complexity of the quick sort algorithm from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$.

9.3 Finding the i th largest element

The goal of this section is to create an algorithm that find the i th largest element in an array $A[n]$. This process is called **selection**.

We know that we can find the minimal/maximal element in an array in $\mathcal{O}(n)$ time by scanning the array linearly. A natural question to ask is whether we can do the same for the i th largest element. Of course, we can always sort the array first and then find it in $\mathcal{O}(n)$, but as we will see soon, every general sorting algorithm takes $\Omega(n \log n)$ time.

Intuitively, finding the i th largest element is harder and more complex than finding the minimal or maximal element in an array, however, this is not the case. First, we will show a solution for the problem with expected complexity $\mathcal{O}(n)$, and then we will show the best solution, which solves the problem in worst-case $\mathcal{O}(n)$ time.

Definition 9.12 (First selection algorithm). Our first selection algorithm is based on an idea similar to QuickSort and it is implemented as follows:

Algorithm 11: Randomized Selection Algorithm

```

1 Function RandomizedSelect( $A, l, r, i$ ):
2    $p \leftarrow$  random pivot index in  $[l, r]$ ;
3    $q \leftarrow$  Partition( $A, l, r, p$ );
4   if  $q = i$  then
5     return  $A[q]$ ;
6   end
7   else if  $q > i$  then
8     return RandomizedSelect( $A, l, q - 1, i$ );
9   end
10  else
11    return RandomizedSelect( $A, q + 1, r, i$ );
12  end

```

Example 9.7 (Randomized selection). Suppose we need to find the 5th largest element in the following array:

7	4	2	1	5	9	8
---	---	---	---	---	---	---

Suppose the first randomized pivot is **pivot** = 4, so after **Partition** we get the array:

2	1	4	8	5	9	7
---	---	---	---	---	---	---

↑
q

Since the index of the pivot element denoted **q** is 3 (remember, we are using the **QuickSort** conventions in which indices start from 1) and we are looking for the 5th largest element, we need the find the 2nd largest element in:

			8	5	9	7
--	--	--	---	---	---	---

Suppose that the randomized pivot is **pivot** = 8, so after **Partition** we get the array:

			7	5	8	9
--	--	--	---	---	---	---

↑
q

Since the index of the pivot element in the updated array is 3 and we are looking for the 2nd largest element we need the find the 2nd largest element in:

			7	5		
--	--	--	---	---	--	--

Suppose that the randomized pivot is **pivot** = 7, so after **Partition** we get the array:

			5	7		
--	--	--	---	---	--	--

↑
q

Since the index of the pivot element in the updated array is 2 and we are looking for the 2nd largest element, we return the pivot, which is now 7. Indeed, 7 is the 5th largest element in the original array.

Remark 9.25 (Randomized selection, best-case analysis). If the pivot element is always the median element, since **Partition** can be implemented in $\Theta(n)$ we get that

$$T(n) = \Theta(n) + T(n/2) \leq c \cdot (n + n/2 + n/4 + \cdots + 1) \leq 2c \cdot n,$$

so in this case $T(n) = \Theta(n)$. This is only the ideal case, and in practice it is usually very hard to find the median.

Remark 9.26 (Randomized selection, worst-case analysis). In the worst case, in each step the size of the array decreases by exactly 1 and then

$$T(n) = \Theta(n) + T(n-1) \leq c \sum_{i=1}^n i = \Theta(n^2).$$

Remark 9.27 (Randomized selection, average-case analysis). If the pivot element is chosen uniformly at random we get that the *expected* time is:

$$T(n) \leq \frac{1}{n} \left(\sum_{k=1}^n T(\max(k-1, n-k)) \right) + \mathcal{O}(n),$$

and thus in particular,

$$T(n) \leq \frac{1}{n} \left(2 \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) \right) + \mathcal{O}(n),$$

Let $d > 0$ be such that

$$T(n) \leq \frac{1}{n} \left(2 \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) \right) + d \cdot n.$$

We will show that $T(n) = \mathcal{O}(n)$ by finding a constant $c > 0$ and showing that $T(n) \leq c \cdot n$ by induction on n . That is, we will show that for any natural $n \in \mathbb{N}$ (larger than zero) the expected time satisfies $T(n) \leq c \cdot n$.

Let $c > 0$ be sufficiently large such that $\max(d+2, 8d, T(1)) \leq c$. For $n = 1$ it is clear that

$$T(1) \leq 2T(0) + d \cdot 1 = 2 + d \cdot 1 \leq c \cdot 1.$$

Next suppose that $T(k) \leq c \cdot k$ for all $1 \leq k < n$. We will show that $T(n) \leq c \cdot n$. Indeed, by using the induction hypothesis and the last inequality on $T(n)$, we get that

$$\begin{aligned} T(n) &\leq \frac{2}{n} \left(c \sum_{k=\lfloor n/2 \rfloor}^{n-1} k \right) + d \cdot n = \frac{2c}{n} \cdot \left(\frac{1}{2} \cdot \left\lceil \frac{n}{2} \right\rceil \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor + n - 1 \right) \right) + d \cdot n \\ &\leq \frac{c}{n} \cdot \left(\frac{n}{2} + 1 \right) \cdot \frac{3n}{2} + \frac{cn}{8} \leq \frac{3cn}{4} + \frac{3c}{2} + \frac{cn}{8} \leq \frac{7cn}{8} + \frac{3c}{2} \leq c \cdot n, \end{aligned}$$

where the first equality follows from the formula for a finite arithmetic series, and the last inequality holds for $n \geq 12$. This is fine because just like we added the specific condition for $d+2 \leq c$ in $\max(d+2, 8d, T(1)) \leq c$, we could add another 12 conditions so that c is sufficiently large so that $T(k) \leq c \cdot k$ for all $1 \leq k \leq 12$.

Definition 9.13 (Median element). In a set S such that $|S| = n$, the median element is the $\lfloor (n+1)/2 \rfloor$ th largest element in S .

Remark 9.28. We can use the selection algorithm we developed in Line 11 with a random pivot in order to find the $\lfloor (n+1)/2 \rfloor$ th largest element (the median) in *expected* time $\mathcal{O}(n)$.

Exercise 9.1 (K problem). Find the most optimized algorithm that returns the k largest elements in an arbitrary array.

Solution. There are many solutions to this problem, some more trivial than others. For example:

- Linearly scan the array k times. This algorithm takes $\mathcal{O}(nk)$.
- Sort the array and return the first k elements. This algorithm takes $\mathcal{O}(n \log n)$.
- Construct a max-heap ($\mathcal{O}(n)$). Return the root and delete it ($\mathcal{O}(\log n)$). Repeat this k times ($\mathcal{O}(k \log n)$). This algorithm takes $\mathcal{O}(n + k \log n)$.
- There is another heap based algorithm which takes $\mathcal{O}(n + k \log k)$ which we will see in the homework.
- (Best solution) Find k th largest element in expected time $\mathcal{O}(n)$, then linearly scan the array and return elements smaller than the k th greatest element. This algorithm takes expected time $\mathcal{O}(n)$.

Note that the selection problem and the K problem are very closely related. Every solution to the K problem gives a solution for the selection problem (selecting i th largest element) with the additional cost of $\mathcal{O}(i)$. Similarly, every solution to the selection problem gives a solution to the K problem with the additional cost of $\mathcal{O}(n)$ (by scanning the array the returning elements lower than the i th largest element). In a general case, these problem are equivalent since the additional costs are negligible, however, if the input is given as a sorted array, then the additional cost may not be negligible anymore. Similarly, if the input is given in the form of a heap or a rank tree the problems may not be equivalent asymptotically.

9.4 Exercises

Exercise 9.2. Suppose we have a linked list and the median of the list is given. Implement a data structure that supports the following operations:

- | | |
|-------------------|--|
| Init(Data) | Initializes the structure with m elements.
Time complexity: $\mathcal{O}(m)$. |
| Insert(x) | Insert x to the structure.
Time complexity: $\mathcal{O}(\log n)$ where n is the current number of elements in the structure. |
| FindMed() | Returns the current median element in the array.
Time complexity: $\mathcal{O}(1)$. |
| DelMed() | Deletes the median element.
Time complexity: $\mathcal{O}(\log n)$ where n is the current number of elements in the structure. |

Solution. The trick in this exercise is to initialize two different heaps. We create a minimum heap L with the $\lceil n/2 \rceil$ largest elements in the structure, and a maximum heap S with the $\lfloor n/2 \rfloor$ smallest elements in the structure. Note that by the construction we either have $|L| = |S|$ or

$|L| = |S| + 1$. The implementation details are as follows:

- Init(Data)** Partition the data into the $\lceil m/2 \rceil$ largest elements and the $\lfloor m/2 \rfloor$ smallest elements in the structure. Then use **MakeHeap** in order to initialize L as S as described above.
- Insert(x)** If x is smaller than the median we insert it to S , and otherwise we insert it to L . In case this creates an imbalance in the sizes of the heaps so that $|L| = |S| - 1$ we transfer the root of S to L , and in case $|L| = |S| + 2$ we transfer the root of L to S .
- FindMed()** Return the root of L .
- DelMed()** Delete the root of L . In case we now have $|L| = |S| - 1$ we transfer the root of S to L .

It is not hard to verify that these operations satisfy the complexity requirements, which completes the exercise.

Exercise 9.3. Implement a data structure that supports the following operations:

- Init(Data)** Initializes the structure with n elements.
Time complexity: $\mathcal{O}(n)$.
- Insert(x)** Insert x to the structure.
Time complexity: $\mathcal{O}(\log n)$ where n is the current number of elements in the structure.
- FindMin(), FindMax()** Returns the minimal/maximal element in the array respectively.
Time complexity: $\mathcal{O}(1)$.
- DelMin(), DelMax()** Deletes the minimal/maximal element from the structure.
Time complexity: $\mathcal{O}(\log n)$ where n is the current number of elements in the structure.

Solution. The trick is very similar to the previous trick. We maintain a minimum heap and a maximum heap at the same time. This time all elements in the structure have a copy in both heaps.

Question. How can we have constant time access between two copies of the same element in the structure in the different heaps?

Answer. We simply use double pointers to connect between every pair of copies.

Question. How can we delete a node p from the minimum heap, which is not necessarily the root?

Answer. We use **DecKey(p, -inf)** and then **DelMin()**.

Note that maintaining both heaps simultaneously solves the exercise. However, we can also solve the problem using a single list.

In order to implement the structure with a single heap, we start by implementing a minimum heap, and then compute a **max** field for each node. The field represents the maximal value of the subtree rooted at the node. We will then maintain and use this field in order to solve the exercise.

The implementation details of the solution with one list are as follows:

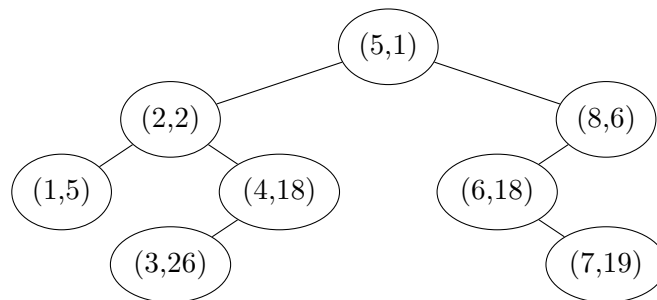
Init(Data)	Initialize a minimum heap, then perform a postorder traversal in order to compute the max fields.
Insert(x)	Insert x to the structure, and update the max fields of the nodes on the insertion path from the bottom to the top.
FindMin()	Return the value of the root.
FindMax()	Return the max value of the root.
DelMin()	Delete the root and replace it with the rightmost leaf (recall that this is how we always delete the root in heaps). Then update the max fields along the path from that leaf to the root, and also update the max fields along the sift-down path from the root downwards.
DelMax()	Note that we can access the maximal element by starting a search from the root and checking which child node has a larger max value and going in that direction until eventually we reach the maximal element. We then delete this leaf in the same way we described in the algorithm of DelMin() by updating the max values in both paths.

Exercise 9.4. Recall the following invariants:

Binary search tree invariant: For any given node, its key is greater than all keys in its left subtree and less than all keys in its right subtree.

Heap invariant: For any given node (except the root), its key is greater than or equal to the key of its parent.

We define a **mixed binary tree** to be a binary tree where each node stores a pair of keys (i, X) , such that the tree satisfies the BST invariant with respect to the first key, and the (min) heap invariant with respect to the second tree. For example, the following tree is a mixed binary tree:



Solve the following:

- (1) Let $\{(i_j, X_j)\}_{j=1}^n$ be a sequence of pairs so that $|\{i_j\}_{j=1}^n| = |\{X_j\}_{j=1}^n| = n$. Does any mixed binary tree with these values has the same root? Prove your answer.
- (2) Describe an algorithm that takes pairs $\{(i_j, X_j)\}_{j=1}^n$ as input, and constructs a mixed binary trees for these pairs in average-case time complexity $\mathcal{O}(n \log n)$.

Solution.

- (1) Yes. The root would always have the key (i_j, X_j) for $X_j = \min\{X_j\}_{j=1}^n$.
- (2) When we see problems with average-case complexity we immediately think about hash tables, but in this case we need to remember the other section in the course in which we encountered average-case calculations, and that is when we see that the average height of a (non balanced) binary tree is $\mathcal{O}(\log n)$ in the average case over the input.

We can construct the tree as required by executing the following steps:

- (a) Sort the pairs $\{(i_j, X_j)\}_{j=1}^n$ with respect to the second coordinate. Put the sorted pairs in an array X .
- (b) Create an empty binary search tree (not an AVL tree), and insert pairs into the tree by their order in X , where the BST invariant respects the first coordinate of the pairs.

Step (a) takes worst-case $\mathcal{O}(n \log n)$ time. Step (b) takes $\mathcal{O}(h)$ time for each insertion, where h is the height of the tree at the time of the insertion, hence step (b) takes $\mathcal{O}(n \cdot h)$ where h is the height of the tree after all insertions. As we saw in class, the final height of the tree h is $\mathcal{O}(\log n)$ in the average-case, hence step (b) takes average-case $\mathcal{O}(n \log n)$ time. In total the complexity of the algorithm is $\mathcal{O}(n \log n)$ as wanted.

The BST invariant is satisfied since we inserted the elements into the tree according to it. The heap invariant is satisfied since insertion of a pair (i_j, X_j) into the tree results in it having no children and having a parent node which was inserted earlier in the tree, hence by order of insertion (given by X) the children node has greater second coordinate value from his parent.

10 Bucket sort and radix sort

10.1 Selection

In the previous section, we have seen how to implement a selection algorithm that outputs the i th largest element in expected complexity $\mathcal{O}(n)$. In this subsection, our goal is to construct an algorithm that finds the i th largest element in *worst-case* complexity $\mathcal{O}(n)$.

Intuitively, if we use the same logic from the quick sort algorithm, a good worst-case $\mathcal{O}(n)$ selection algorithm would look like this:

Algorithm 12: SELECT(A , first, last, i), intuition

Input: Array A [first..last], integer i
Output: The i th smallest element in A

```

1 Find  $x$  that approximates the median;
2  $s \leftarrow \text{PARTITION}(A, \text{first}, \text{last}, x)$  ;           /* Partition  $A$  with  $x$  */
3 if  $s - \text{first} = i$  then
4   | return  $x$ 
5 else if  $s - \text{first} > i$  then
6   | return SELECT( $A$ , first,  $s - 1$ ,  $i$ )
7 else
8   | return SELECT( $A$ ,  $s$ , last,  $i - (s - \text{first})$ )

```

One suggestion could be to let x just be the median. However, we don't have an algorithm that finds the median in worst-case complexity $\mathcal{O}(n)$. Therefore we would have to find an approximation to the median (a good approximation is an element which is bigger than $\alpha\%$ of elements and smaller than $\alpha\%$ of elements, where $0 < \alpha < 50$).

Definition 10.1 (SELECT). The selection algorithm is as follows:

Algorithm 13: SELECT(A , first, last, i)

Input: Array A [first..last], integer i
Output: The i th smallest element in A

```

1 if  $|A| < 20$  then
2   | Sort  $A$  and return the  $i$ th largest element;
3 Partition  $A$  into groups of 5 consecutive elements;
4 Find the median of each group of 5;
5 Store these medians into an array  $B$  of size about  $|A|/5$ ;
6  $x \leftarrow \text{SELECT}(B, 1, |B|, \lfloor (|B| + 1)/2 \rfloor)$  ;           /* Median of medians */
7  $s \leftarrow \text{PARTITION}(A, \text{first}, \text{last}, x)$  ;           /* Partition  $A$  with  $x$  */
8 if  $s - \text{first} = i$  then
9   | return  $x$ 
10 else if  $s - \text{first} > i$  then
11   | return SELECT( $A$ , first,  $s - 1$ ,  $i$ )
12 else
13   | return SELECT( $A$ ,  $s$ , last,  $i - (s - \text{first})$ )

```

Remark 10.1. For simplicity, we assume that the elements in the array are unique. If there are repetitions, we can modify PARTITION to return two pointers so that elements equal to the pivot are between them.

Proposition 10.1. The number of elements in the recursive calls to *Select* in steps 11/13 in Definition 10.1 is at most $7n/10 + 3$, where n is the number of elements of the current array.

Proof. Since x is the median of medians, and there are at most $\lceil n/5 \rceil$ medians, we get that at least $\left\lceil \frac{1}{2} \cdot \lceil n/5 \rceil \right\rceil$ medians are smaller than or equal to x . For every group of 5 elements, if their median is smaller than or equal to x , then there are at least 3 elements in that group which are smaller than or equal to x (except for possibly the last group which could have less than 5 elements). Therefore the number of elements smaller than or equal to x in the array is at least

$$\left(\left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{5} \right\rceil \right\rceil - 1 \right) \cdot 3 \geq \frac{3}{10}n - 3.$$

Similarly the number of elements bigger than or equal to x is also at least

$$\left(\left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{5} \right\rceil \right\rceil - 1 \right) \cdot 3 \geq \frac{3}{10}n - 3.$$

Therefore, in every recursive call the number of elements is at most $7n/10 + 3$ where n is the number of elements of the current array. $\square$

Proposition 10.2. *In the `Select` algorithm we have $T(n) = \mathcal{O}(n)$.*

Proof. By Theorem 10.1 we get that the running time satisfies the inequality

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + \mathcal{O}(n).$$

In fact it satisfies a stricter inequality, but this less strict inequality makes the calculations slightly easier. Let $d > 0$ be the constant so that

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + d \cdot n,$$

and let $c > 0$ be such that $T(k) \leq c \cdot k$ for $k \leq 80$, and such that $c > 80d$ (we can do this because these are finite conditions). We will show by induction that for any $n > 0$ we have $T(n) \leq c \cdot n$. For $n = 1$ this is obvious by definition of c . Next assume that $T(k) \leq c \cdot k$ for all $k < n$, and we will show that $T(n) \leq c \cdot n$. By the original inequality and the constructions so far we get:

$$\begin{aligned} T(n) &\leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T(7n/10 + 6) + d \cdot n \\ &\leq c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot (7n/10 + 6) + d \cdot n \\ &\leq c \cdot \frac{n}{5} + c + c \cdot (7n/10 + 6) + d \cdot n \\ &\leq c \cdot \left(\frac{9n}{10}\right) + 7c + d \cdot n \\ (\text{since } c > 80d) \quad &\leq c \cdot \left(\frac{9n}{10}\right) + 7c + \frac{c}{80} \cdot n \\ &= \frac{73}{80}cn + 7c \\ &= cn + (-7c \frac{n}{80} + 7c) \leq cn. \end{aligned}$$

$\square$

Remark 10.2 (Why groups of 5?). It may seem that the choice of partitioning the array into groups of 5 in Line 13 is arbitrary, and that we could have chosen groups of 3 or 7 without affecting the complexity/performance of the algorithm. However, if we partition the array into groups of 3 we would get the following recurrence relation:

$$T(n) \leq T(\lceil n/3 \rceil) + T(2n/3 + 2) + \mathcal{O}(n).$$

Here the coefficients sum of to $\frac{1}{3} + \frac{2}{3} = 1$, so by the master theorem the complexity of $T(n)$ could potentially be superlinear, and indeed, it turns out to be $\mathcal{O}(n \log n)$. If we partition the array into groups of 7 we would get the following recurrence relation:

$$T(n) \leq T(\lceil n/7 \rceil) + T(5n/7 + 4) + \mathcal{O}(n).$$

Here the coefficients sum of to $\frac{1}{7} + \frac{5}{7} = \frac{6}{7}$, so by the master theorem the complexity of $T(n)$ is linear $\mathcal{O}(n)$, however, finding medians of groups of 7 elements is slower in practice, which affects performance.

Remark 10.3 (Deterministic quick sort). A natural question that arises with this new **Select** algorithm is whether we can now implement quick sort as a deterministic algorithm with complexity $\mathcal{O}(n \log n)$. The answer to that question is yes, yes we can.

Algorithm 14: Deterministic quick sort using median-of-medians

```

1 Function QuickSort( $A, l, r$ ):
2   if  $l \geq r$  then
3     return
4    $m \leftarrow \lfloor (r - l + 1)/2 \rfloor$ 
5    $p \leftarrow \text{Select}(A, l, r, m)$            // Find median (deterministic pivot)
6    $i \leftarrow \text{Partition}(A, l, r, p)$ 
7   QuickSort( $A, l, i - 1$ )
8   QuickSort( $A, i + 1, r$ )
```

In this case since we find the median in deterministic worst-case $\mathcal{O}(n)$ complexity, the recurrence relation of the algorithm is now $T(n) = 2 \cdot T(n/2) + \Theta(n)$, and therefore $T(n) = \Theta(n \log n)$.

10.2 Comparison-based sorting lower bound

We have seen many sorting algorithms so far, but when it comes to a general array, we have not seen any sorting algorithm that sorts it in worst-case $\mathcal{O}(n)$ time or in fact, any complexity under $\mathcal{O}(n \log n)$. In this subsection, we will show that indeed, there is a lower bound for the complexity of comparison-based sorting algorithms which is $\Omega(n \log n)$.

Remark 10.4 (Comparison-based sorting algorithms). A comparison-based sorting algorithm is an algorithm that sorts the input by comparing the elements to each other. For example if k_1 and k_2 are both keys in the input, a comparison-based algorithm determines whether $k_1 < k_2$, $k_1 = k_2$, $k_1 \geq k_2$, et cetera. A non-comparison-based algorithm may check whether k_1 is even or odd, what is the value of the third to the left bit in k_2 , or what is the difference between k_1 and k_2 . Note that all sorting algorithms we have seen so far (bubble sort, merge sort, heap sort, quick sort) are all comparison-based.

Remark 10.5 (Comparison of (non)comparison-based algorithms). The greatest advantage of comparison-based sorting algorithms over non-comparison-based algorithms lies in their generality. Non-comparison-based sorting algorithms only work for specific types of input, and the input usually has to satisfy some range conditions, while comparison-based sorting algorithms

work for any type of input that has a linear (total) order on it. However, we will soon see that while there is a lower bound for the complexity of comparison-based sorting algorithms which is $\Omega(n \log n)$, this is not the case for non-comparison-based sorting algorithms, and we will later discuss some of these algorithms and see that they are indeed faster (but remember, they only work on specific well behaving input).

Remark 10.6 (Finding permutations and sorting arrays). In general, each unsorted array can be thought of a sorted array and a permutation that mixed its elements. For example if we define $\pi \in S_6$ by:

$$\begin{aligned}\sigma(1) &= 5 & \sigma(2) &= 2 & \sigma(3) &= 3 \\ \sigma(4) &= 6 & \sigma(5) &= 4 & \sigma(6) &= 1\end{aligned}$$

and an arbitrary sorted array $\mathbf{A} = [28, 32, 47, 53, 71, 85]$ we get the unsorted array:

1	28	$\xrightarrow{\sigma}$	1	85
2	32		2	32
3	47		3	47
4	53		4	71
5	71		5	28
6	85		6	53

If we are only given the unsorted array $\sigma(A)$ the sorting the array is equivalent becomes equivalent to finding $\pi := \sigma^{-1}$. In this case:

$$\begin{aligned}\pi(1) &= 6 & \pi(2) &= 2 & \pi(3) &= 3 \\ \pi(4) &= 5 & \pi(5) &= 1 & \pi(6) &= 4\end{aligned}$$

and indeed:

1	85	$\xrightarrow{\pi}$	1	28
2	32		2	32
3	47		3	47
4	71		4	53
5	28		5	71
6	53		6	85

sorts the array perfectly.

Theorem 10.3. *Every comparison-based sorting algorithm is $\Omega(n \log n)$.*

Remark 10.7. First, in order to tackle this theorem we need to have a better understanding of all comparison-based sorting algorithms. Every comparison-based sorting algorithm is of the following form:

- (1) Execute some operations not on the keys.
- (2) Compare the keys $A[i_1]$ and $A[j_1]$.
- (3) Execute some operations not on the keys.
- (4) Compare the keys $A[i_2]$ and $A[j_2]$.
- ...
- (2m) Compare the keys $A[i_m]$ and $A[j_m]$.

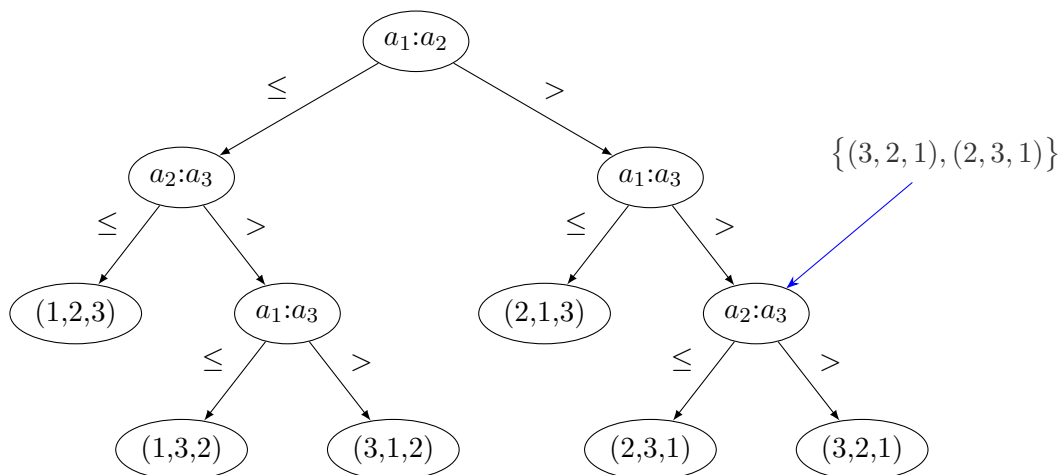
(2m+1) Execute some operations not on the keys.

(2m+2) Return (or output) the sorted array or the inversing permutation $\pi := \sigma^{-1}$.

Note that even if the algorithm itself moves positions of keys we could still translate this into comparing the keys in their original positions.

Definition 10.2 (Decision tree). In the context of comparison-based sorting algorithms, a decision tree is a binary tree in which each node represents a comparison between two elements in the input $(a_1, \dots, a_n)$. Each edge represents a result of the comparison, and the leaves represent a certain permutation of the elements.

Example 10.1. Suppose that the input is (a_1, a_2, a_3) , and the sorting algorithm is a bunch of if-else blocks. Then the associated decision tree would look something like this:



Each leaf holds a single permutation σ which represents the permutation σ from Remark 10.6. This leaf could easily hold $\pi = \sigma^{-1}$ in order to return the sorted array immediately. Note that we can think of inner nodes such as $a_2 : a_3$ as holding all possible permutations up to the current state. For example, before comparing a_2 and a_3 there were two possibilities for π that satisfy the previous conditions which are $\{(3, 2, 1), (2, 3, 1)\}$.

Remark 10.8 (Height of decision tree). Since each leaf in the decision tree holds exactly one possible permutation, it follows that the decision tree of any comparison-based algorithm has exactly $n!$ leaves. By properties of binary tree it follows that the height of the tree is at least $\log_2(n!)$. Remember, our goal is to analyze the lower bound for complexity of comparison-based sorting algorithms. Since the height of a decision tree for every comparison-based algorithm has height at least $\log_2(n!)$, it follows that in every comparison-based algorithm there is a route which takes $\log_2(n!)$ steps. Therefore the complexity of each comparison-based sorting algorithm is $\Omega(\log_2(n!))$.

Proof of Theorem 10.3. We already know from Remark 10.8 that $T(n) = \Omega(\log_2(n!))$. In fact, we know that in the worst case each comparison-based sorting algorithm executes $\log_2(n!)$ commands. By Stirling's approximation we know that

$$n! \geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \geq \left(\frac{n}{e}\right)^n,$$

for any natural $n \in \mathbb{N}$. It follows that $\log_2(n!) \geq n \log_2(n/e)$, or in other words:

$$\log_2(n!) \geq n \log_2(n) - n \log_2(e).$$

Note that for $n > 9$ we have

$$\log_2(e) < \log_2(\sqrt{9}) \leq \log_2(\sqrt{n}) = \frac{1}{2} \log_2(n),$$

and thus for $n > 9$ we get

$$\log_2(n!) > n \log_2(n) - n \log_2(e) > n \log_2(n) - \frac{1}{2}n \log_2(n) = \frac{1}{2}n \log_2(n) = \Omega(n \log n).$$

It follows that every comparison-based algorithm executes at least $\frac{1}{2}n \log n$ commands and is thus $\Omega(n \log n)$, which completes the proof. $\square$

10.3 Bucket sort

Bucket sort is the first of two examples of non-comparison-based sorting algorithms which we will encounter in this section.

Definition 10.3 (Bucket sort, simplified). Here is an example of a simplified bucket sort algorithm:

Algorithm 15: Bucket sort for n unique integers in $\{1, 2, \dots, k\}$.

```

1 Function BucketSort( $A[1..n]$ ,  $k$ ):
2   create boolean array  $C[1..k]$  initialized to 0;
3   for  $i \leftarrow 1$  to  $n$  do
4      $C[A[i]] \leftarrow 1$ ;
5    $j \leftarrow 1$ ;
6   for  $i \leftarrow 1$  to  $k$  do
7     if  $C[i] = 1$  then
8        $A[j] \leftarrow i$ ;
9        $j \leftarrow j + 1$ ;

```

Remark 10.9 (Bucket sort intuition). The simplified bucket sort algorithm creates a boolean array which acts as a histogram for the input. Then it linearly scans the histogram from the smallest possible value in the range to the largest value in the range, and whenever it sees a 1 in the histogram it adds it to the array in the order of the scan. Note that it is only possible to create such a histogram under the specific conditions we assumed the input satisfies.

Definition 10.4 (Bucket sort, less simplified). Here is an example of a less simplified bucket sort algorithm:

Algorithm 16: Bucket sort for n integers in $\{1, 2, \dots, k\}$.

```

1 Function BucketSort( $A[1..n]$ ,  $k$ ):
2   create array  $C[1..k]$  initialized to 0;
3   for  $i \leftarrow 1$  to  $n$  do
4      $C[A[i]] \leftarrow C[A[i]] + 1$ ;
5    $j \leftarrow 1$ ;
6   for  $i \leftarrow 1$  to  $k$  do
7     while  $C[i] > 0$  do
8        $A[j] \leftarrow i$ ;
9        $C[i] \leftarrow C[i] - 1$ ;
10     $j \leftarrow j + 1$ ;

```

Remark 10.10 (What makes this non-comparison-based?). The reason that bucket sort is not considered a comparison-based algorithm is not only because it never compares elements of $A[1..n]$, but also because it uses the values we sort as indices. Remember, in a comparison-based sorting algorithm we are only allowed to compare between elements, and we are specifically not allowed to use them as indices. More generally, comparison-based sorting algorithms are general sorting algorithms that should work on any totally ordered type, in particular, we are not allowed to make assumptions on our input, even more particularly, we are not allowed to assume our input can be used for indexing.

Definition 10.5 (Stable sort). A sorting algorithm is called stable if whenever $A[i] == A[j]$ and $i < j$ then $A[i]$ comes before $A[j]$ in the output.

Remark 10.11 (Is bucket sort stable?). It is not too difficult to see that in its simplified implementation (no repetitions) it is stable. More generally, if the input has no repetitions the sorting is always stable. In its less simplified version, there isn't any direct relation between the numbers in the array and the numbers in the output. In fact, if the numbers in the array are fields of a node with a pointer to some other object, which is usually the case, our algorithm does not work at all. However, we can make it stable by making each entry of the histogram a linked list, and then appending the pointers according to their order in the original array. In this way, the less simplified algorithm is also stable.

Remark 10.12 (Bucket sort complexity). In each of its implementations we can see that its complexity is $\Theta(k)$ for initializing the array $C[1..k]$, plus $\Theta(n)$ for scanning the input, plus $\Theta(k)$ for scanning the histogram. In total, the complexity of bucket sort is $\Theta(k + n)$, and in case $k = \mathcal{O}(n)$ we even get a linear sorting algorithm in $\Theta(n)$.

Remark 10.13. We can also use bucket sort more generally for k elements in a continuous range $[0, 1)$, in which case the buckets in the histogram could potentially store two different elements (or more) inside them. In this case we are left with no choice but to sort the buckets with insertion sort (which is what it sounds), or some other way.

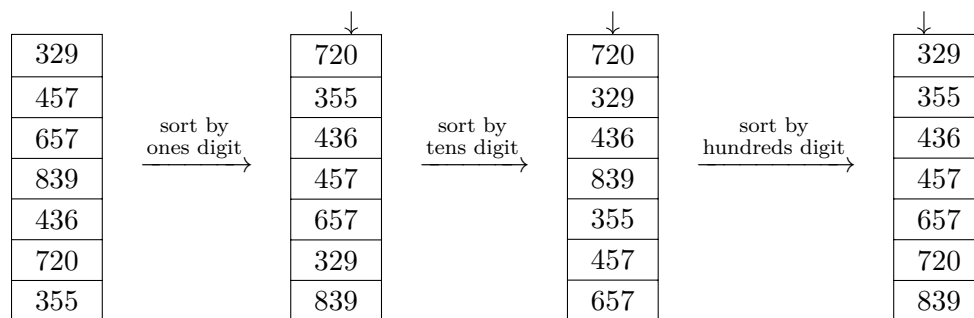
10.4 Radix sort

One important observation is that Bucket sort is not efficient when $n \ll k$. For example, sorting just 3 elements from $\{1, 2, \dots, 10^9\}$ would require an array of size 10^9 while $n = 3$, and clearly this is not efficient (this is similar to the reason we introduced hash tables).

Radix sort is a different non-comparison-based sorting algorithm that solves this problem by assuming each element of the input is made up of d digits from 0 to 9, then sorting by the digits in the least significant place to the digits in the most significant place using some stable sort (such as bucket sort).

Intuitively we may think it's better to start sorting the input from the most significant digits to the least significant ones. Indeed that kind of radix sort exists and it is called an MSD (most significant digit) radix sort, but it does not work as smoothly as LSD (least significant digit) radix sort. In this subsection (and in fact in the entirety of this course) when we say radix sort we always mean LSD radix sort, and we will not get into MSD radix sort.

Example 10.2. Here is a simple example of radix sort in action:



Note that after the i th step if we truncate each entry to its first i digits we get an ordered array.

Definition 10.6 (Radix sort). Denote the least significant digit (LSD) in 1 and the most significant digit (MSD) by d . Then we can implement radix sort as follows:

Algorithm 17: Radix sort

Input: Array A with n natural numbers in the range $[0, 10^d)$

Output: Sorted array

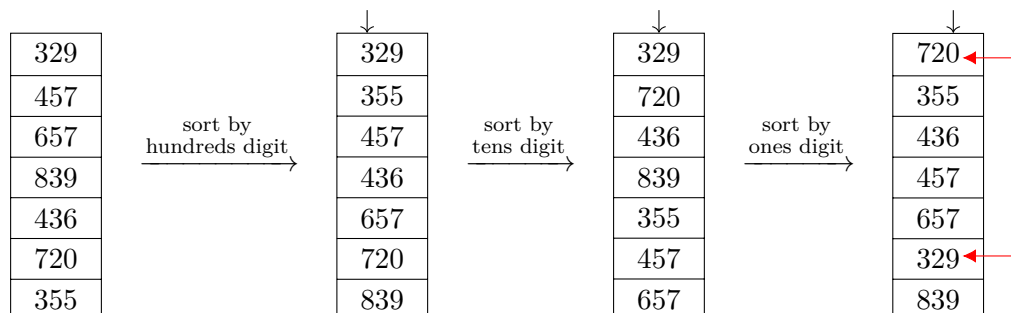
```

1 for  $i \leftarrow 1$  to  $d$  do
2   | call stable BucketSort to sort array  $A$  on digit  $i$ .
3 end

```

Exercise 10.1. Prove correctness of Line 17 by using induction on i . Prove that this implementation of radix sort is stable. Pinpoint why MSD radix sort is unstable without additional modification to the algorithm.

Example 10.3 (Example of MSD radix sort malfunction). Here is an example of incorrect output of a naïve implementation of the MSD radix sort:



where the red arrows show where the sorting is violated.

Remark 10.14 (Radix sort complexity). By Remark 10.12 we know that each bucket sort in base- k is $\Theta(n + k)$, so for d digits the complexity of radix sort is $\Theta(d(n + k))$, and in particular for base-10 we get $\Theta(d(n + 10))$. If the number of digits d does not depend on n then we can conclude that the complexity is $\Theta(n)$. For example, for n natural numbers in base-10 in a constant range $[0, 10^5 - 1)$ the algorithm is $\Theta(n)$.

Remark 10.15 (Radix sort complexity, continued). If we know that the maximal element in sorting is M , then its number of digits in base k is $\log_k M$, and therefore the complexity of the algorithm is $\Theta(\log_k M \cdot (n + k))$. For example, if $M = \Theta(n^c)$ for some constant c (in particular we may choose $n = 100$ numbers in range $[0, M)$ where M is the maximal number and $M \approx n^c = 100^2$ for $c = 2$), then the complexity is $\Omega(n \log n)$, which isn't really better than a comparison based sorting algorithm.

Question. In general we see that the complexity is $\Theta(\log_k M \cdot (n + k))$ and that as k is larger $\log_k M$ gets smaller and $(n + k)$ gets larger. This raises the question: how can we find the optimal base?

Answer. If we know the value of n we choose the base $k = n$, and then the complexity is $\Theta(n \log_n M)$. In particular, if $M = \Theta(n^c)$ then radix sort will run in linear time $\Theta(n \log_n M) = \Theta(c \cdot n) = \Theta(n)$.

Remark 10.16 (Counting sort). Additionally to the two main non-comparison-based algorithms we have introduced, you may also encounter an algorithm called “counting sort”. Its correctness is based on arithmetic and counting arguments. It is a stable sort for n integers in $\{0, 1, \dots, k\}$, and just like bucket sort its complexity is $\mathcal{O}(n + k)$. The implementation is as follows:

Algorithm 18: Stable Counting Sort for n integers in $\{0, 1, \dots, k\}$.

```

1 Function CountingSort( $A[1..n], B[1..n], k$ ):
2   create array  $C[0..k]$  initialized to 0;
3   for  $j \leftarrow 1$  to  $n$  do
4      $C[A[j]] \leftarrow C[A[j]] + 1$ ;
      //  $C[i]$  now contains the number of elements equal to  $i$ 
5   for  $i \leftarrow 1$  to  $k$  do
6      $C[i] \leftarrow C[i] + C[i - 1]$ ;
      //  $C[i]$  now contains the number of elements  $\leq i$ 
7   for  $j \leftarrow n$  down to 1 do
8      $B[C[A[j]]] \leftarrow A[j]$ ;
9      $C[A[j]] \leftarrow C[A[j]] - 1$ ;
```

10.5 Exercises

Exercise 10.2. Describe an efficient algorithm for sorting n numbers from the range $[0, n^2 - 1]$.

Solution. We will describe four such algorithms, some better than others.

- (1) Use a standard comparison-based sorting algorithm with time complexity $\Omega(n \log n)$.
- (2) We can use counting sort, or bucket sort, and get time complexity $\mathcal{O}(n + k) = \mathcal{O}(n + n^2) = \mathcal{O}(n^2)$.
- (3) We can use radix sort with numbers represented in base 10. In this way the complexity is $\mathcal{O}(d \cdot (n + b)) = \mathcal{O}((\log_{10} n^2) \cdot (n + 10)) = \mathcal{O}(n \log n)$.
- (4) We can use radix sort with numbers represented in base n . Under the assumption that arithmetic operations such as division take constant time a change of base for a number with $\log n$ digits takes constant time, hence in total the change of base for all inputs takes $\mathcal{O}(n)$ time. In this way the complexity is $\mathcal{O}(d \cdot (n + b)) = \mathcal{O}((\log_n n^2) \cdot (n + n)) = \mathcal{O}(2n) = \mathcal{O}(n)$, where $d = 2$ since each number in $[0, n^2 - 1]$ has at most 2 digits in base n .

Exercise 10.3. Describe an efficient algorithm for sorting an array of size n with $\log n$ distinct elements.

Solution. We will solve this exercise in three increasingly efficient ways:

- (1) We can insert all elements to an AVL tree, where each node would contain the value of an element, and also maintain a counter of how many times it appears in the input (if

an element's value is not present in the AVL tree we add a new node and otherwise we increment the counter of the node with that value by one). After that process we can perform an inorder traversal on the tree and print each element in the traversal as many times as its counter shows.

Note that since there are $\log n$ distinct elements the height of the tree is bounded by $\log \log n$, and so the total insertion process takes $\mathcal{O}(n \log \log n)$ time. The printing process takes $\mathcal{O}(n)$ time since we print n elements, and so in total the time complexity of the algorithm is $\mathcal{O}(n \log \log n)$.

- (2) Create a hash table T of size $\log n$. Again, we insert all elements into the hash table, and maintain counters of how many times they appeared as we do so. By hash table complexity the total time complexity of this process is average-case $\mathcal{O}(n)$. We can then sort the hash table in worst-case complexity $\mathcal{O}(\log n \cdot \log \log n)$ (since it is of size $\log n$), and then print all elements of the sorted hash table while respecting the counters in worst-case $\mathcal{O}(n)$ time. In total, this solution has average-case $\mathcal{O}(n)$ complexity.

In the worst case all elements would go into the same entry in the hash table (implemented with chain hashing based on linked lists), and so each insertion would take $\mathcal{O}(\log n)$ time, we get that the worst-case complexity of the algorithm is now $\mathcal{O}(n \log n)$, which is worst than what we got in (1).

- (3) We simply modify the solution in (2), so that instead of using hash tables implemented with chain hashing based on linked lists, we use chain hashing based on AVL trees. In this way the average-case complexity remains $\mathcal{O}(n)$, but even in the worst case each insertion would take $\mathcal{O}(\log \log n)$ (since this is the maximal height of an AVL tree with $\log n$ elements), and so the worst-case complexity would be $\mathcal{O}(n \log \log n)$. Clearly this is the best solution.

Exercise 10.4. Prove or disprove the existence of a data structure that supports the following operations:

Insert(x) If x is not in the structure already insert x to the structure.

Succ(x) Return the smallest element in the structure which is greater than x .

so that the time complexity of these operations is $\mathcal{O}(1)$ amortized.

Solution. The idea in all of these type of exercises is almost always to reach a contradiction to the comparison-based sorting algorithm theorem, which generally states that sorting general n inputs in $\mathcal{O}(n \log n)$ is impossible.

Here the solution is quite straightforward. Assume, by way of contradiction, that such a data structure exists. We first find the minimal element in the input and print it in $\mathcal{O}(n)$ time. We then proceed to compute the successor of this element and print it, and once we repeat this process n times we have sorted n elements in $\mathcal{O}(n)$ time. This contradiction completes the proof.

Exercise 10.5. Let A be an array of size $n = 2^k - 1$ with $\log n$ distinct elements. Moreover, suppose that for all $0 \leq i < k$ there exists an element in the array that appears 2^i times. For example, the array $[5, 3, 8, 5, 3, 5, 5]$ is such an array for $n = 7$ and $k = 3$. Describe an algorithm that sorts this array in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space complexity.

Solution. The trick in this exercise is to notice that the mode (most frequent element) appears 2^{k-1} times in the array, which is more than half the size of the array. Hence the mode and the median are equal. We can now find the median in $\mathcal{O}(n)$ by using the selection algorithm we discussed in the lectures, and solve the question recursively. More precisely the algorithm is a recursive algorithm on k and an array of size $n = 2^{k-1} - 1$ described as follows:

- (1) If $k = 1$ return the array.
- (2) Find the median using selection on the array in $\mathcal{O}(n)$, put all appearances of the median in an array B of size 2^{k-1} , put the other elements in A , and return the concatenation of $\text{Recursion}(A, k-1)$ and B , where **Recursion** is the name of this recursive algorithm.

It follows that the time complexity of this recursive algorithm can be described by the following recursive equation:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \mathcal{O}(n).$$

By the master theorem we get that $T(n) = \mathcal{O}(n)$ which completes the exercise.

Exercise 10.6. Prove that every algorithm that prints the elements of a minimum heap in a sorted, increasing order, takes at least $\Omega(n \log n)$ time.

Solution. Assume, by way of contradiction, that there exists an algorithm that prints the elements of a minimum heap in a sorted, increasing order, in less than $\Omega(n \log n)$ time, or equivalently $o(n \log n)$ time.

We simply construct a minimum heap from the input in $\mathcal{O}(n)$ time using the known algorithm from class, and then use the algorithm in $o(n \log n)$ time print the input in a sorted order. This is a sorting algorithm that runs in $o(n \log n)$ time, which contradicts the theorem on the minimal complexity of comparison-based sorting algorithms. This contradiction completes the proof.

Exercise 10.7 (Spring 20, B, 4).

- (1) Suppose we have an array with n numbers, so that the number 9 appears at least $n - \frac{n}{100}$ times in the array (at least 99% of the array are the number 9). Prove or disprove the existence of a comparison-based sorting algorithm for such an array in worst-case time complexity $\mathcal{O}(n)$.
- (2) Suppose we have an array with n numbers, so that the number 9 appears exactly $n - n^{\frac{99}{100}}$ times in the array. Prove or disprove the existence of a comparison-based sorting algorithm for such an array in worst-case time complexity $\mathcal{O}(n)$.

Solution.

- (1) There does not exist such an algorithm. We will show this by finding a contradiction to the theorem on the minimal complexity of comparison-based sorting algorithms.

Assume, by way of contradiction that such an algorithm exists. Let A be an arbitrary array of size n . Scan A and change all 9's in the array to 9' (these are just 9s with a special mark). Concatenate an array of size $99n$ that only contains 9 (this takes $\mathcal{O}(n)$). We then use the algorithm to sort the array in worst-case $\mathcal{O}(n)$ time, and then scan the array again and delete all the 9s that do not have a mark, and delete the mark from the original mark (without deleting them). The result is that we sorted the array in $\mathcal{O}(n)$ time, which is a contradiction to the theorem on the minimal complexity of comparison-based sorting algorithms. This contradiction completes the proof.

- (2) This is, in fact, a proof. The algorithm is as follows:

- (I) Delete the $n - n^{\frac{99}{100}}$ appearances of 9 from the array, and put them in another array. We now have a sorted, constant array of 9s of size $n^{\frac{1}{100}}$, and an unsorted array of size $n^{\frac{99}{100}}$.
- (II) Sort the unsorted array from (I) using a standard sorting algorithm in $\mathcal{O}(n^{\frac{99}{100}} \log(n^{\frac{99}{100}}))$ time.

(III) Merge both sorted arrays.

Step (I) takes $\mathcal{O}(n)$ time since we just scan the arrays. Step (II) takes $\mathcal{O}(n^{\frac{99}{100}} \log(n^{\frac{99}{100}}))$ time, which is in particular $o(n)$. Step (III) is a merge so it takes $\mathcal{O}(n)$ time. In total, the complexity of this sorting algorithm is worst-case $\mathcal{O}(n)$ time.

11 Strings

In Section 5 we have already discussed basic data structures that support management of strings efficiently such as tries and radix trees (which we could also call a “tries with compressions”). In this section we will talk about these data structures in more depth, and also introduce new data structures made specifically for storing strings.

Some problems that will guide us through this section include:

- Finding a substring in a long string. For example, does the string `janteloven` appear in the Oxford dictionary? If so on what page?
- In human-made strings there are many repetitions of words or sounds or even collocations. How can we use that fact in order to compress the strings?
- How can we find the longest common substring of two strings?

11.1 Sorting using tries

Exercise 11.1. Given n strings $S_1, \dots, S_n$ with a total length $m = \sum_{i=1}^n |S_i|$, describe an algorithm that prints the strings in lexicographical order.

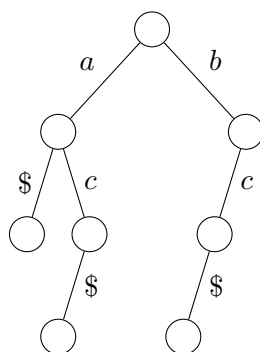
Solution (Comparison based solution). In this solution we assume for simplicity that all strings are of the same length m/n . We know that a comparison-based sorting algorithm can sort n elements in $\mathcal{O}(n \log n)$ time. However, each comparison takes $\mathcal{O}(m/n)$ time, so that total complexity of an algorithm that uses a standard comparison-based sorting is $\mathcal{O}((m/n)n \log n) = \mathcal{O}(m \log n)$.

Remark 11.1. We can notice that strings have a very specific structure. They are finite sequences of letters from an alphabet Σ of constant size. We can use this additional information in order to find a non-comparison-based sorting algorithm that works in $\mathcal{O}(m)$. This is very similar to how we use other assumptions on the input, for example that it is in range $[0, 10^4]$ in radix sort in order to achieve complexity $\mathcal{O}(n)$.

Definition 11.1 (Linear string sorting algorithm). This is another solution to Exercise 11.1. The algorithm is very simple:

- (1) Insert $S_1, \dots, S_n$ into a trie.
- (2) Perform a preorder traversal on the trie and output the path to each leaf. We push the edges into the stack as we follow them, and when we reach a leaf we output the stack. Once we “travel the edges backwards” we also pop them from the stack.

Example 11.1. Consider the trie:



By the algorithm defined in Definition 11.1, we get the output `a$, ac$, and bc$` in that order.

Remark 11.2 (Linear string sorting complexity). Each insertion of a string S_i into a tree requires $|S_i|$ time, so the insertion of all strings takes $\mathcal{O}(\sum |S_i|) = \mathcal{O}(m)$ time. The preorder traversal takes $\mathcal{O}(\#nodes)$ time, and since by properties of trees we get $\mathcal{O}(\#nodes) = \mathcal{O}(\#edges)$, and $\mathcal{O}(\#edges) = \mathcal{O}(m)$, we conclude that the traversal is $\mathcal{O}(m)$. The printing of the strings also takes $\mathcal{O}(m)$ time and therefore the worst-case time complexity of the linear string sorting algorithm is linear.

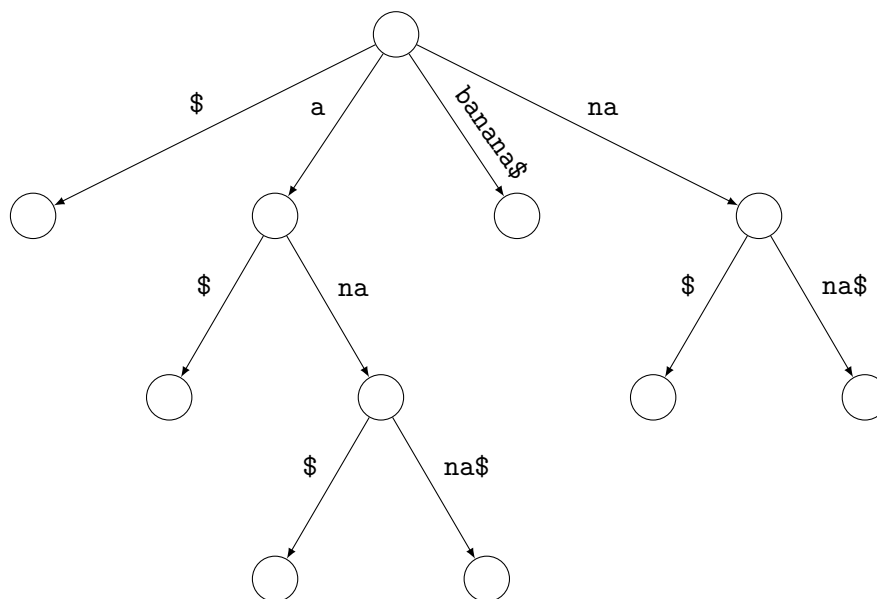
Exercise 11.2. Prove or disprove the existence of a data structure that supports insertion, deletion, and median lookup in an amortized complexity $\mathcal{O}(\log \log n)$.

Solution. The idea is to assume by way of contradiction that such a structure exists, use it as a “black box” in order to create a comparison-based sorting algorithm, prove correctness of the algorithm and compute its complexity, then get that its less than the lower bound of complexity $\Omega(n \log n)$, for example $\mathcal{O}(n \log \log n)$. This contradiction completes the proof.

11.2 Suffix trees

Exercise 11.3. Construct a compressed trie (radix tree) for all suffixes of the word **banana**.

Solution. The suffixes are $\{\text{banana}\$, \text{anana}\$, \text{nana}\$, \text{ana}\$, \text{na}\$, \text{a}\$, \$\}$, and the if we insert them at that order we get the following compressed trie:



Remark 11.3. When inserting the suffixes into the compressed trie, it is easier to insert the suffixes from the longest suffix to the shortest one.

Remark 11.4. From now on we assume that all tries are compressed tries.

Definition 11.2 (Suffix tree). A suffix tree for a string S is a trie with all suffixes of S .

Example 11.2. The tree from Exercise 11.3 is a suffix tree for the word **banana**.

Remark 11.5 (Motivation for suffix trees). Let S be a string of length n and let $S[i..j]$ represent any substring of S . Then $S[i..j]$ is a prefix of the suffix $S[i..n]$, that includes its first $j - i + 1$ letters.

It follows that all substrings of S are “hiding” inside its suffix tree, possibly ending mid-edge (for example the substring **an** in the suffix tree from Exercise 11.3). This is incredible motivation considering the number of suffixes of a string is linear in the length of the string.

Remark 11.6 (Notation). Given a node v we denote the string that it represents in the tree as $\text{str}(v)$.

Remark 11.7. The number of leaves in the subtree of v is equal to the number of suffixes of S that contain $\text{str}(v)$ as a prefix. Equivalently (although this may be more difficult to see), the number of leaves in the subtree of v is equal to the number of occurrences of the substring $\text{str}(v)$ in the entire string S .

Remark 11.8. Paths that don't start at the root also correspond to substrings in the tree, so it is possible to find more paths corresponding to a substring than the number of its occurrences in S .

Remark 11.9. A suffix tree has plenty of applications in algorithms that analyze strings. We will focus on the following three applications:

- Finding a substring inside a string.
- Implementing Ziv–Lempel compression.
- Find the longest common substring of two strings.

Definition 11.3 (Suffix tree generation algorithm). The suffix tree generation algorithm for a string S of length m is very simple:

- (1) Insert the suffixes $S[m..m], \dots, S[2..m]$, and $S[1..m]$ in that order into a (non compressed) trie.
- (2) Compress the trie into a radix tree.

Remark 11.10 (Suffix tree generation complexity). The complexity of the algorithm described in Definition 11.3 is

$$\text{time}(m) = cm + c(m-1) + \dots + c = \mathcal{O}(m^2).$$

Remark 11.11 (Black box). There exist multiple complex algorithms that construct a suffix tree for a string S of length m in worst-case $\mathcal{O}(m)$ time. However, we will not learn any of them in this course. That said, we will still use this algorithm as a “black box”, which means that given a string S of length m we are allowed to say “construct a suffix tree for S in $\mathcal{O}(m)$ time”. It is important to note that we are not allowed to modify this algorithm since we do not actually know how it works.

Exercise 11.4. Find a sequence of strings S_m of length $|S_m| = m$, such that the total length of string labels on the edges of the suffix tree of S_m is $\omega(n)$.

Solution. Denote the total length of the strings on the edges of the suffix tree of S_m by $L_m := \sum_{e \in E} |\text{label}(e)|$. Recall that $L_m = \omega(m)$ if and only if

$$\lim_{m \rightarrow \infty} \frac{L_m}{m} = \infty.$$

Note that if $S = 000\ 001\ 010\ 011\ 100\ 101\ 110\ 111$ is the strings that contains the numbers 0-7 in binary, then its suffix tree (which contains 25 suffixes) has at least 8 edges going out from the root. Next consider the string $S_m = x_1x_2 \dots x_m$. It is clear that $|S_m| = m$, and similarly to the previous string S , it is clear that the suffix tree of S_m has m edges going out from the root, each of length $m - i + 1$. It follows that

$$L_m = \sum_{i=1}^m (m - i + 1) = \sum_{t=1}^m t = \frac{m(m+1)}{2},$$

and thus

$$\lim_{m \rightarrow \infty} \frac{L_m}{m} = \lim_{m \rightarrow \infty} \frac{m+1}{2} = \infty,$$

so $L_m = \omega(m)$ which completes the proof.

Remark 11.12 (Space constraints). By Exercise 11.4 it follows that if we store the labels on the edges directly then any algorithm for creating the suffix tree will take $\omega(n)$ time. Thus, every algorithm that creates a suffix tree for a string S in linear time cannot store the labels of the edges directly.

Question. We have seen in Theorem 5.4 that the amount of nodes is in a compressed suffix tree of S of length n is $\mathcal{O}(n)$. However, a suffix tree requires $\Theta(m^2)$ space for the labels on the edges, and therefore its space complexity is $\Omega(m^2)$. Is there a way optimize the space complexity to be linear?

Answer. Yes! We notice that each label on an edge is a (continuous) substring ² $S[i..j]$, and therefore instead of storing the entire substring in all label, we can simply store a copy of the string somewhere else, and in the labels we only store two pointers to the indices i and j . Since now each edge takes $\mathcal{O}(1)$ space and there are $\mathcal{O}(n)$ edges, the space complexity of this modified suffix tree is $\mathcal{O}(m)$.

Definition 11.4 (The value $c(v)$ for a node v). Let S be a string of length n . Then for each node v in the (modified compressed) suffix tree of v we define $c(v)$ to be the index (counting from 1) of the first appearance of $\text{str}(v)$ in S .

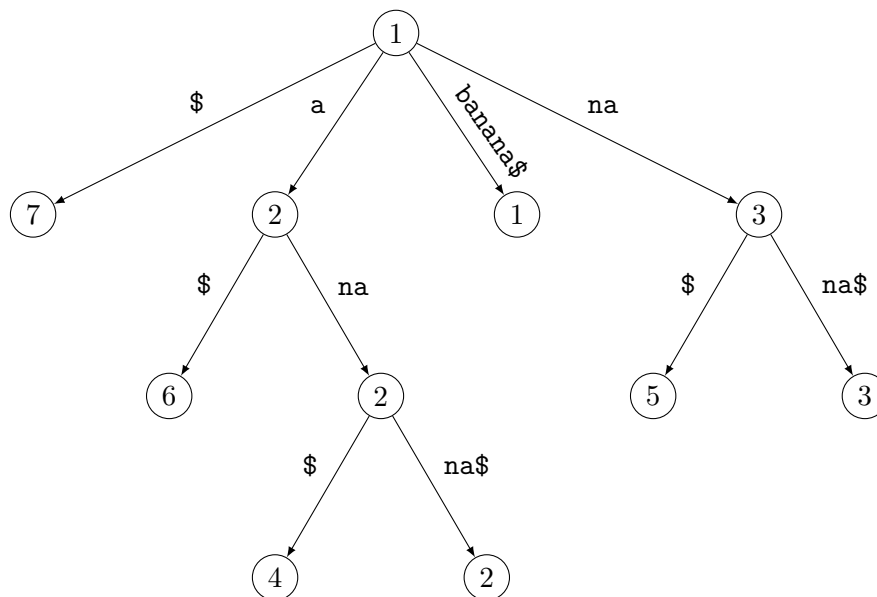
Exercise 11.5. Find an algorithm that computes $c(v)$ for each node v in a suffix tree of a string S of length n in worst-case complexity $\mathcal{O}(n)$.

Solution. We perform a postorder traversal on the tree, and when we reach a leaf we can compute $c(v)$ by first finding the total length of labels (computed in $\mathcal{O}(1)$ by $j - i + 1$ for each node in the modified suffix tree) on the path to v , then subtracting it from the total length of the string n (where n includes $\$$ as part of the string). By properties of postorder, when we get to an inner node v the values $c(v \rightarrow \text{right})$ and $c(v \rightarrow \text{left})$ are already well-defined, and thus we define $c(v)$ to be the minimum of these value.

Remark 11.13. Make sure you understand why the solution of Exercise 11.5 works, and in particular why if r is the root then $c(r)$ is promised to be 1.

Example 11.3. In the tree from Exercise 11.3 we can use the algorithm from Exercise 11.5 and get that:

²All substrings in this course are required to be continuous



where the value in each node represents its c value.

Remark 11.14. One may want to find the index of the first instance of **ban** in **banana**, and it may seem impossible at first, since no search path leads exactly to **ban**, and in particular there are multiple substrings that have the same search path as **ban** (for example **bana** and **ba**). However, note that really, there is no problem with our algorithm at all.

11.3 Finding substrings in a long text

In this subsection we are given a string T of length m called the *text*. We will show a method to process the text in $\mathcal{O}(m)$ time such that we will then be able to receive any string S of unknown length n , and return the first occurrence (if exists) of S in the text.

Remark 11.15. Note that any solution to this problem that does not involve a processing of the text beforehand must require at least $\Theta(m)$ complexity.

Remark 11.16 (Applications). Here are some applications to this problem:

- (1) The text is a corpus or a dictionary for a foreign language given in `.txt` format and the string we search for is a certain word or phrase.
- (2) The text is the genome of your family member, and the string we search for is a certain sequence that encodes a specific harmful gene.

Remark 11.17 (Generalizations). Note that we can easily generalize this problem in order to solve problems such as:

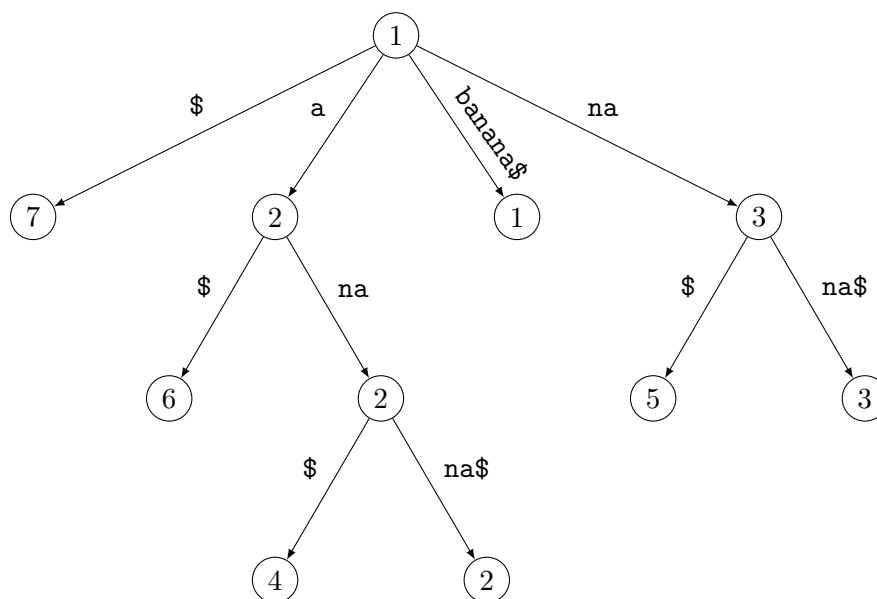
- Finding all occurrences of S in T .
- Finding the n th occurrence of S between multiple texts.

Remark 11.18 (Assumption). We assume that $m \ll n$. That is, the length of the string we are searching is significantly shorter than the length of the text.

Definition 11.5 (The algorithm). Use Remark 11.11 in order to construct a suffix tree for the text T in $\mathcal{O}(m)$ time, then compute $c(v)$ (from Definition 11.4) in $\mathcal{O}(m)$ as shown in Exercise 11.5.

Given a string S , we follow the path it defines on the suffix tree from the root. If S is found, then the first occurrence of S in the string is $c(v)$ where v is the last node in its search path.

Example 11.4. Consider the suffix tree for the text **banana**



If we use the algorithm from Definition 11.5 on the string **ban** we get that its first appearance is at the first index, and if we search for **nan** we get that its first appearance is at the third index.

Remark 11.19 (Multiple occurrences). In order to find all occurrences of a string **S** in the text, we first remember that every substring of **T** is a prefix of some suffix, that is, it is a prefix of a path from the root to some leaf. It follows that after finding the node v from Definition 11.5, the occurrences of **S** in **T** corresponds with a leaf in the subtree rooted at v and vice versa. Thus, we simply need to return the value of $c(u)$ for each node in the subtree rooted at v .

Example 11.5. Recall Example 11.4. We want to find all occurrences of the substring **na** in **banana**. By the algorithm described in Definition 11.5 we find that its first appearance is in index 3, and then by Remark 11.19 we get that its appearances are in indices 3 and 5.

11.4 Text compression

In this subsection, our goal is to find a way to encode all the information of a long string **S** in a significantly shorter string **S''**.

There are many applications to an efficient text compression algorithm, for example when transferring text files from one computer to another, or when trying to save disc space on your computer.

Remark 11.20 (The main idea). We scan the string **S** from left to right, and whenever we see a substring **z** that we have seen before, we replace it with an ordered pair that contains the starting index and length of its first occurrence in the string.

This algorithm is based on a concept called “Ziv–Lempel compression” first described in two papers from 1977–1978. In the papers it is also shown that this algorithm is optimal asymptotically.

Example 11.6 (Compressing a short string). Consider the string **S** = **x a b a c aba xab y** (without the whitespace characters). After compressing **S** using the idea described in Remark 11.20 we get the compressed string **S''** = **x a b (2,1) c (2,3) (1,3) y**.

We notice that the compressed string **S''** is not really shorter than the original string **S**. This is a common situation when compressing short strings.

Example 11.7 (Compressing a long string). Consider the string:

$S = a \ b \ ab \ abab \ abababab \ abababababababab,$

then the compressed string is:

$S'' = a \ b \ (1,2) \ (1,4) \ (1,8) \ (1,16).$

In a more general case, if S is expanded to be of length $m = 2^{k-1}$ where k is the number of “blocks” it has (for example the original S has 6 blocks), then the compressed string would be of length $2(k-1) = 2\log_2 m$. In other words, if the noncompressed string S is of length m , then the compressed string S'' is of length $\Theta(\log m)$. This shows that in this case the compression is extremely useful.

Example 11.8 (Banana compression). If $S = \text{banana}$ then $S'' = \text{ban } (2,2) \ (2,1)$.

Definition 11.6 (Prior). Let $S[1..m]$ be a string of length m . Then for each index $1 \leq i \leq m$ we define the substring $\text{Prior}(i)$ to be the longest prefix of $S[i..m]$ that appears as a substring in $S[1..(i-1)]$.

Remark 11.21 (More on Prior). We denote the length of $\text{Prior}(i)$ by $L(i)$. When $\text{Prior}(i)$ is the empty string we define $L(i) = 0$. When $L(i) > 0$ we also denote the index of the first occurrence of $\text{Prior}(i)$ by $s(i)$.

Example 11.9. Let $S = a \ b \ a \ x \ c \ a \ b \ a \ x \ a \ b \ y$. Then since the letter at the 7th index of S is b we get that $\text{Prior}(7) = b \ a \ x$. We also get that $L(7) = \text{length}(\text{“bax”}) = 3$ and $s(3) = 2$ since the first occurrence of $b \ a \ x$ is at the second index of S .

Definition 11.7 (The compression algorithm). The compression algorithm for a string $S[1..m]$ is:

```

1  for (i = 1; i <= m;) {
2      compute(s(i), L(i)) ;
3      if (L(i) > 0) {
4          output(s(i), L(i)); i = i + L(i)
5      };
6      else {
7          output(S[i]); i = i + 1
8      }
9  }
```

It is clear that the complexity of this algorithm depends on the computation of $(s(i), L(i))$. We will soon see that we can compute this pair in $\mathcal{O}(L(i))$ time. Given that, we can see that the time complexity of this algorithm is $\mathcal{O}(m)$.

Definition 11.8 (Efficient implementation of Ziv–Lempel). For a string $S[1..m]$ use the black box from Remark 11.11 in order to construct a suffix tree for S in $\mathcal{O}(m)$ time. Then compute $c(v)$ for all nodes in the suffix tree in $\mathcal{O}(m)$ time. Then follow the algorithm described in Definition 11.7, and in order to compute $(s(i), L(i))$, follow the path from the root according to the suffix $S[i..m]$ as long as $c(v) + t - 1 < i$, where t is the current length of the path (this condition implies we are in $S[1..(i-1)]$). When the search is terminated, denote the terminal node by u , then $L(i)$ is the number of letter we have followed, and $s(i)$ is equal to $c(u)$. The complexity of this routine for finding is $\mathcal{O}(L(i))$ time.

Remark 11.22 (Decoding compressed strings). Given an encoded string S'' we can easily decode it by reading it from left to right, when we reach a letter, we copy it to S (which starts as an empty string), and when we reach a pair $(s(i), L(i))$, we go to index $s(i)$ and copy the first $L(i)$ letters from there.

11.5 Finding longest common substring

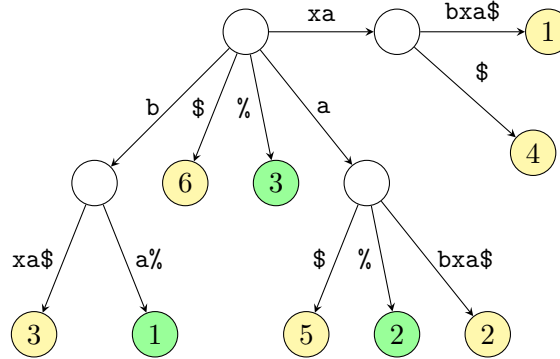
In this subsection, our goal is to find an efficient algorithm for finding the longest common substring of two strings.

Example 11.10. Consider the strings $S1 = \text{superiorcalifornialives}$ and $S2 = \text{sealiver}$. Their longest common substring is the string `alive`.

Remark 11.23. We can solve this problem by brute-force. We simply check for every substring of $S2$ if it is a substring of $S1$ and take the longest substring we find. The complexity of this algorithm is $\mathcal{O}(|S1| \cdot |S2|^2)$. We will show a solution using suffix trees in $\mathcal{O}(|S1| + |S2|)$.

Definition 11.9 (Generalized suffix tree). A generalized suffix tree is a trie to which we insert all suffixes of all elements in a set of one or more strings $D = \{S_1, \dots, S_n\}$, and we end the suffix of a string S_i with a special terminating character $\$i$. That is, we can determine which suffix came from which string by its terminating character. It is also compressed.

Example 11.11 (Generalized suffix tree). Here is an example of a generalized suffix tree for the strings $D = \{\text{xabxa}\$, \text{ba}\%\}$.



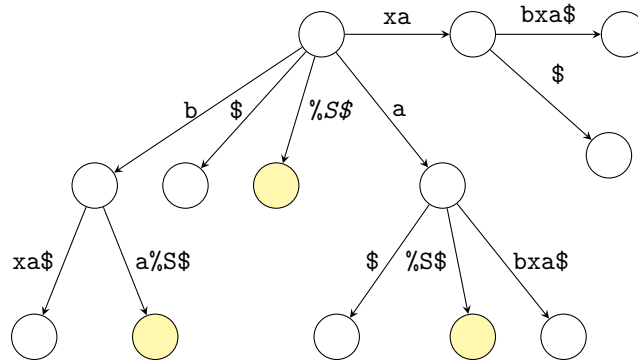
Remark 11.24. In order to use a generalized suffix tree we first need to create it. We could create it simply by inserting each of the suffixes into the tree which would make the running time at least the sum of all suffixes of both strings, which is more than $\mathcal{O}(|S1| + |S2|)$. The following algorithm will show us how to construct a generalized suffix tree (for two strings) in $\mathcal{O}(|S1| + |S2|)$.

Definition 11.10 (Linear generalized suffix tree construction algorithm). Given strings $S1$ and $S2$ the algorithm is as follows:

- (1) We construct a suffix tree for $S'' = S2\%S1\$$ (the concatenation of both strings) in $\mathcal{O}(|S1| + |S2|)$ using the black box algorithm from Remark 11.11.
- (2) Traverse the tree (for example in postorder) and prune all subtrees under edges whose label contains the terminal character $\%$.

The time complexity for this algorithm is $\mathcal{O}(|S1| + |S2|)$ as required.

Example 11.12. The construction for the generalized suffix tree for $\{S = \text{xabxa}\$, T = \text{ba}\%\}$ starts by constructing a suffix tree for $\text{ba}\%\text{xabxa}\$$ which looks as follows:



We can see that the on edges that lead to yellow nodes the substring `%S$` appears on the labels. According to the algorithm we must prune their subtrees (in the diagram they are already pruned). We do this since any node in those subtrees represents a suffix that includes `%` inside it. Such suffixes do not exist in the generalized suffix tree of `ba%xabxa$`, so it makes sense to remove them. We then proceed to remove the `S$` from the label leading to the yellow nodes, and get the generalized suffix tree in linear time as desired.

Remark 11.25 (Technical remark). Since an edges contains a sequence of characters and not just a single character we need to check if this string contains a terminal node such as `%` inside it (not as the last character). In order to check this in $\mathcal{O}(1)$ we simply store the length of `T` (in the previous example $|\text{ba}\%| = 3$), and since each label contains pointers to indices of their string (starting and ending indices of the substring on the label), we just need to check if the starting index is less than or equal to 3 (or generally $|T|$), and the ending index is more than or equal to 4 (or generally $|T| + 1$).

Definition 11.11 (Algorithm for longest common substring). In this definition when we say “linear time”, we mean linear in the sum of the length of the strings. That is: $\mathcal{O}(|S1| + |S2|)$.

Given two substrings `S1` and `S2`, in order to find their longest common substring we first construct a generalized suffix tree for $D = \{S1, S2\}$ in linear time (by Definition 11.10), then mark each leaf by 1 if its a suffix of `S1`, and by 2 if its a suffix of `S2`, and propagate these values up the tree the the inner nodes in the following way: an inner node gets the union of the values of its children (this can be done in linear time as well if we use a postorder traversal). Then, the longest common substring is represented by the longest path from the root whose nodes are all marked with $\{1, 2\}$, which we can find with a smart preorder traversal (explicit details are left as an exercise), and since the smart preorder traversal is also done in linear time, the complexity of the algorithm overall is linear $\mathcal{O}(|S1| + |S2|)$.

Remark 11.26 (Applications). Suppose that we are given two strings `S1` and `S2` which are the genomes of two different living organisms, and the longest common substring in this genome helps us identify their mutual phenotypes, and advance our understanding of evolution.

Remark 11.27. We can also generalize the algorithms we have seen in this section with two strings to k strings.

11.6 Exercises

Exercise 11.6. Implement a data structure that supports the following operations:

- | | |
|--|--|
| <code>Init(s₁, ..., s_n)</code> | Initializes the structure given n strings.
Time complexity: $\mathcal{O}(m)$ where $m = \sum_{i=1}^n s_i $. |
| <code>FindPermutation(s)</code> | Returns if <code>s</code> is a permutation of one of the strings <code>s₁, ..., s_n</code> .
Time complexity: $\mathcal{O}(s)$. |

Solution. The trick in this question is to first bring all strings into a normal form. More precisely, we will sort each string internally so that its characters are sorted in a lexicographical order. For example `cat` would become `act`. This would make the rest of the computations trivial. The exact implementation details are as follows:

`Init(s1, ..., sn)` Sort each string internally using bucket sort in $\mathcal{O}(|s_i| + |\Sigma|) = \mathcal{O}(|s_i|)$ time complexity. Next insert all sorted strings into a trie. In total the complexity is $\mathcal{O}(m + m) = \mathcal{O}(m)$.

`FindPermutation(s)` Sort s using bucket sort in $\mathcal{O}(|s| + |\Sigma|) = \mathcal{O}(s)$ time complexity, then return true if and only if the sorted version of s is found in the trie. In total this takes $\mathcal{O}(|s| + |s|) = \mathcal{O}(|s|)$ time complexity.

The proof for correctness follows from the uniqueness of the normal form we introduced (the sorted form of the strings), and transitivity of permutations.

Exercise 11.7. A suffix array of a string S of length m is an array of size m that contains the m suffixes of s sorted in lexicographical order. For example, if $s = \text{“easy”}$ then the suffix array is $A = [\text{asy}, \text{easy}, \text{sy}, \text{y}]$. Solve the following:

- (1) Alice suggested to use linked lists in order to store the suffixes in the array. More precisely, each element in the array would be a linked list with each node in the list representing a single character of the suffix.
 - (a) What is the space complexity of this implementation?
 - (b) Suggest an alternative implementation that requires $\mathcal{O}(m)$ space with constant time access to the i th suffix in the array.
- (2) Describe an algorithm that takes a string of size m as input, and generates its suffix array in $\mathcal{O}(m)$ time.
- (3) Implement the operation `IsSubstring(r)` that takes a string r as input, and returns whether it is a substring of s , in both a suffix tree and a suffix array. Compare both implementations.
- (4) Implement the operation `CountSubstring(r)` that takes a string r as input, and returns how many times r appears as a substring in s , in both a suffix tree and a suffix array. Compare both implementations.

Solution.

- (1a) The sum of the sizes of the list is:

$$1 + 2 + \dots + m = \Theta(m^2),$$

hence the space complexity of this implementation is $\Theta(m^2)$.

- (1b) We can store a single copy of s , and an array B of size $|s|$ where $B[i]$ would be a pointer to the place in the string where the suffix begins. For example if the string is $s = \text{easy}$ then since $A = [\text{asy}, \text{easy}, \text{sy}, \text{y}]$ we get that $B = [2, 1, 3, 4]$. The space complexity is clearly $\mathcal{O}(m)$.
- (2) Given a string s , we can generate its suffix array by using the following algorithm:
- (I) Construct a (compressed) suffix array for s using the black box algorithm in $\mathcal{O}(m)$.
 - (II) Perform a perorder traversal on the tree, where the order is the lexicographical order on the values on the edges, so indeed the traversal visits the leaves in the order the suffixes they represent appear in the suffix tree.

- (III) When performing (II) we also maintain a counter l that counts the length of the path from the root to the current node (in characters), and when we reach a leaf we perform $A[i] = |s| + 1 - l$.

The time complexity of this algorithm is $\mathcal{O}(m)$, since both the black box algorithm and a preorder traversal take $\mathcal{O}(m)$.

- (3) In a suffix tree we have already discussed this, we simply search for r in the suffix tree, and it is a substring if and only if it is found since another characterization of substrings is “prefixes of suffixes”. This takes $\mathcal{O}(|r|)$.

In a suffix array based implementation we also want to check if r is a suffix of one of the suffixes, so we perform a binary search for r in the array. Since each comparison takes $\mathcal{O}(|r|)$ time and there are $\mathcal{O}(\log m)$ comparisons, the total time complexity is $\mathcal{O}(|r| \cdot \log m)$.

In this case, clearly the suffix tree implementation is more efficient.

- (4) The problem is equivalent to finding how many suffixes have r as a prefix so in a suffix tree we can simply search the tree for r , then count how many leaves are in the subtree of the node where r is found. If r is not found we return 0. In the worst case we may count all leaves and so the time complexity is $\mathcal{O}(m)$.

In a suffix array we do a binary search in order to find the first and last appearances of r in s (verify you understand how to find them with a logarithmic amount of comparison). If r is not found at all return 0. If the indices of those first and last appearances are F and L respectively, then we return $L - F + 1$. The time complexity of this algorithm is $\mathcal{O}(|r| \cdot \log m)$.

In this case, there is no clear winner, but if we work with many long strings, the suffix array implementation would be more efficient.

Note that in the suffix tree implementation after generating the tree we could perform a postorder traversal and write in each node how many leaves the tree rooted at it has, we could perform `CountSubstring(r)` in $\mathcal{O}(|r|)$ time instead of $\mathcal{O}(m)$.

Exercise 11.8. A pair of strings s_1 and s_2 are called k -similar if at least one of the following conditions holds:

- (1) The prefix of length k of s_1 is identical to the suffix of length k of s_2 .
- (2) The prefix of length k of s_2 is identical to the suffix of length k of s_1 .

Note that if s_1 and s_2 are k -similar, it does not follow that they are m -similar for $1 \leq m < k$. For example, the strings `abc` and `abcdef` are 3-similar but they are not 2-similar. However, if the strings are `efxyzabc` and `abcdef` then they are both 2-similar and 3-similar (but they are not 1-similar).

- (1) Given two strings s_1 and s_2 , describe an algorithm that finds the maximal value of $k \in \mathbb{N}$ so that s_1 and s_2 are k -similar in $\mathcal{O}(|s_1| + |s_2|)$ time complexity.
- (2) Describe an algorithm that solves the problem from (1) in $\mathcal{O}(\min(|s_1|, |s_2|))$ time complexity.

Solution.

- (1) The algorithm is described for a non-compressed suffix tree implementation, but since we will use a compressed suffix tree for complexity, there is a need to treat the compressed tree as a non-compressed tree throughout the algorithm. It is as follows:

- (I) Construct a suffix tree for s_1 .
- (II) Initialize $k = 1$ and $l = 1$.
- (III) Initialize a pointer p to the root of the tree.
- (IV) For every character c in s_2 from beginning to end:
 - (a) Advance p to the node corresponding to c . If that is not possible, exit the for loop and go to step (V).
 - (b) Increment l by 1.
 - (c) If there exists a node corresponding to $\$,$ update k to be l .
- (V) Write k somewhere and repeat steps (I) to (IV) for $s_1 \leftarrow s_2$ and $s_2 \leftarrow s_1$, then return the maximal k value from the two we have.

The complexity of step (I) is $\mathcal{O}(|s_1|)$ by the black box algorithm. The complexity of steps (II) and (III) are both $\mathcal{O}(1)$. The complexity of step (IV) is $\mathcal{O}(\min(|s_1|, |s_2|))$, since each iteration is $\mathcal{O}(1)$ and the amount of iterations is bounded both by $|s_1|$ and by $|s_2|$. Step (V) repeats everything with the swapped strings, hence the total complexity of the algorithm is:

$$(\mathcal{O}(|s_1|) + \mathcal{O}(\min(|s_1|, |s_2|))) + (\mathcal{O}(|s_2|) + \mathcal{O}(\min(|s_2|, |s_1|))) = \mathcal{O}(|s_1| + |s_2|).$$

- (2) Without loss of generality, assume that $|s_1| < |s_2|$ (we can always find which string is the shorter string in $\mathcal{O}(\min(|s_1|, |s_2|))$ time so this does not effect the complexity). We can solve the problem more efficiently by modifying step (V) so that it repeats the process with $s_1 \leftarrow \text{reverse}(s_1)$ and $s_2 \leftarrow \text{reverseIterator}(s_2)$, where the **reverse** function reverses the string s_1 , and the **reverseIterator** function means that in step (IV) we go over the characters of s_2 from end to beginning instead of beginning to end.

The algorithm is still correct since in steps (I) to (IV) we first find the length of the longest suffix of s_2 that is a prefix of s_1 , and in the modified step (V) we find the length of the longest prefix of s_2 (since the for loop is from beginning to end) that is a suffix of s_2 (since we consider the suffix tree of **reversed**(s_1)).

The complexity analysis now shows that the complexity of step (V) is:

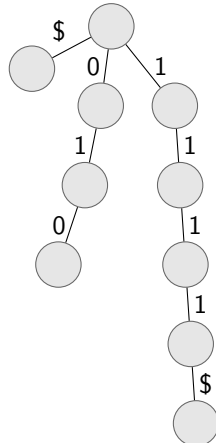
$$\mathcal{O}(|\text{reverse}(s_1)|) + \mathcal{O}(\min(|\text{reverse}(s_1)|, |\text{reverseIterator}(s_2)|)),$$

which is simply $\mathcal{O}(|s_1|) + \mathcal{O}(\min(|s_1|, |s_2|))$, hence the total complexity of the algorithm is:

$$2(\mathcal{O}(|s_1|) + \mathcal{O}(\min(|s_1|, |s_2|))) = \mathcal{O}(|s_1|),$$

and this is under the assumption that $|s_1| < |s_2|$, so this completes the exercise.

Exercise 11.9. Find the minimal string s so that the following tree is a subtree (in this case simply a part of the tree, not necessarily a proper subtree) of its suffix tree:



Solution. From the subtree, we can infer that s must end in 1111, and must have 010 as a substring. Hence the minimal such s is $s = 0101111\$$.

Exercise 11.10. Let s be a string over a finite (constant) alphabet Σ . Describe an algorithm that prints a string t that satisfies the following properties:

- (1) t is the longest substring of s that appears at least 3 times in s .
- (2) t is the (lexicographically) smallest string that satisfies (1).

If s does not have a substring that appears in s more than twice, the algorithm should not print anything. For example, if $s = \text{ababcbac}$ then the algorithm would return **ab**, and if $s = \text{ababababa}$ the algorithm would return **ababa** since it appears three times: **ababab**aba, ab**abab**aba, abab**ab**aba. The algorithm should run in $\mathcal{O}(|s|)$ time where $|s|$ is the length of the string.

Solution. We start by constructing a suffix tree for s in $\mathcal{O}(|s|)$ time using the black box algorithm. We then proceed to perform a postorder traversal on the suffix tree, and write at each node the amount of leaves in its subtree (we did a similar thing in Exercise 11.7 when we implemented the **CountSubstring** operation). This is a standard process when we need to understand the amount of times a substring appears in a string, since the amount of appearances of a substring in a string is equal to the amount of leaves in the subtree rooted at the node that represents that substring. This node is unique since in a suffix tree every path from the root is unique, or equivalently, every node represents a unique substring, and in a non-compressed suffix tree we can even say that the amount of unique substrings in the string is exactly the amount of nodes in the tree.

Since we need to find the (lexicographically) smallest substring t that appears at least 3 times, we need to perform a preorder traversal on our tree and maintain a variable that tells us the length of each substring in order to also take the longest such substring. We do this as follows:

- (1) Traverse the suffix tree in preorder in order to go over the substrings lexicographically.
- (2) Maintain a variable **max** that keeps track of the longest substring we have seen so far that appears at least three times, and also store a pointer that node.
- (3) Update **max** and the pointer only if we see a substring that appears at least three times of length strictly greater than **max**.

At the end of this process we can simply return the string the pointer points to. We used the implicit assumption that we always know the distance of the current node from the root (the length of the substring that node represents), and indeed we can do it without changing the complexity of the algorithm, by using the same trick we saw in Exercise 11.7 part (2).

12 Graphs

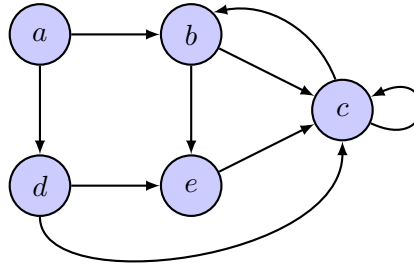
In this section we will talk about graphs. Graphs are mathematical object that represent a plethora of real-life structures. They are used everywhere in computer science, and in science in general. From biological processes to civil engineering, and even in social media apps.

Remark 12.1. We have even already seen some examples of graphs when we talked about trees.

12.1 Introduction

Definition 12.1 (Directed graph). A directed graph is an ordered pair (V, E) , where V is a set called the *vertices*, and $E \subseteq V \times V$ is a set of ordered pairs of vertices called *directed edges*.

Example 12.1. The following directed graph:



is represented by the sets

$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, d), (b, c), (b, e), (c, b), (c, c), (d, c), (d, e), (e, c)\}$$

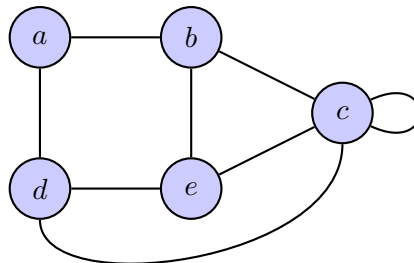
In general, we denote $|V| = n$ and $|E| = m$. In this case $n = 5$ and $m = 9$.

Remark 12.2. The number of edges in a directed graph (V, E) is at most n^2 .

Definition 12.2 (Undirected graph). An undirected graph is an ordered pair (V, E) , where V is a set called the *vertices*, and $E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V\}$ is a set of unordered pairs of vertices called *undirected edges*, or simply edges.

Remark 12.3. In the context of undirected graphs, we may denote an edge $\{i, j\}$ by (i, j) , even though this notation is less accurate.

Example 12.2. The following undirected graph:



is represented by the sets

$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, d), (b, c), (b, e), (c, c), (d, c), (d, e), (e, c)\}$$

In this case $n = 5$ and $m = 8$.

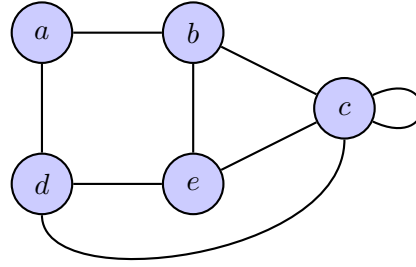
Remark 12.4. The number of edges in an undirected graph is at most $n(n+1)/2$. That is because

$$m \leq \binom{n}{2} + \binom{n}{1} = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}.$$

Definition 12.3 (Adjacency matrix). Let (V, E) be a finite graph with some order on its vertices. Then we define its adjacency matrix to be a boolean $n \times n$ matrix such that $A_{ij} = 1$ if and only if $(i, j) \in E$, where i and j represent the i th and j th vertices in the order respectively.

Remark 12.5. An undirected graph has a symmetric adjacency matrix.

Example 12.3. Consider the undirected graph



If we give the vertices V the order (a, b, c, d, e) then its adjacency matrix is:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Question. Here are a few question to develop intuition on adjacency matrices:

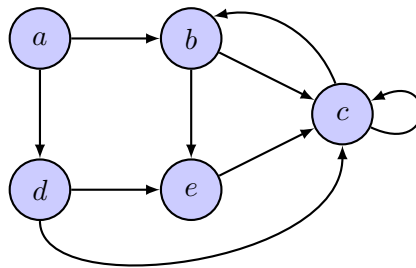
- (1) What is the meaning of 1 on the prime diagonal?
- (2) What is the meaning of the sum of elements in the matrix?
- (3) What is the meaning of the sum of elements in the i th row?
- (4) What is the meaning of the sum of elements in the i th column?

Answer.

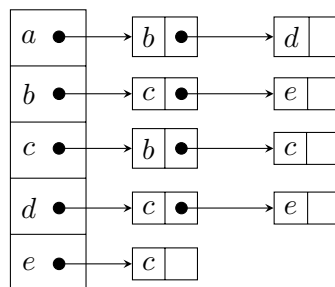
- (1) It represents an edge from a vertex to itself.
- (2) It represents the total number of edges in an undirected matrix. In a directed matrix, we can deduce the number of edegs only if we know how many 1s are on the diagonal.
- (3) It represents the number of edges outgoing from the i th vertex.
- (4) It represents the number of edges incoming to the i th vertex.

Definition 12.4 (Adjacency list). Let (V, E) be a finite graph with $V = \{v_1, \dots, v_n\}$. Then we define its adjacency list to be an array of size n where the i th entry contains a list of the vertices v_i “points to”. In other words, it contains u if and only if $(v_i, u) \in E$.

Example 12.4. Recall the graph:



Its adjacency list is:



Question. Here are a few question to develop intuition on adjacency lists:

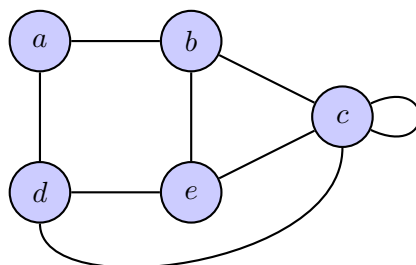
- (1) What is the space complexity of an adjacency list?
- (2) How long does it take to check if b points to e ?
- (3) What is the meaning of the number of nodes in the list of c ?
- (4) What is the meaning of the number of nodes in all lists combined?

Answer.

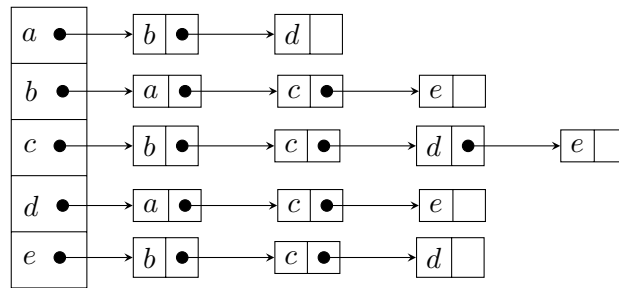
- (1) The space complexity is $\mathcal{O}(|V| + |E|) = \mathcal{O}(n + m)$.
- (2) Checking whether b points to e takes $\mathcal{O}(\deg(b))$ time in the worst case, since we may need to scan all neighbours of b .
- (3) The number of nodes in the list of c represents the outdegree of vertex c (the number of edges going out of c).
- (4) The number of nodes in all lists combined represents the total number of edges in the graph.

Example 12.5 (Representing undirected graphs with adjacency lists). Given an undirected graph (V, E) we can represent it by its adjacency list, so that a list representing the node v contains exactly the nodes connected directly to v .

For example, we can represent the undirected graph:



by the following adjacency list:



Note that every node in the adjacency list corresponds to a unique entry of the adjacency matrix.

Exercise 12.1. Let (V, E) be a graph where the vertices are the numbers $\{1, \dots, n\}$ and E is given in an unsorted array of size m . Find an algorithm that constructs an adjacency list for the graph in $\mathcal{O}(n + m)$.

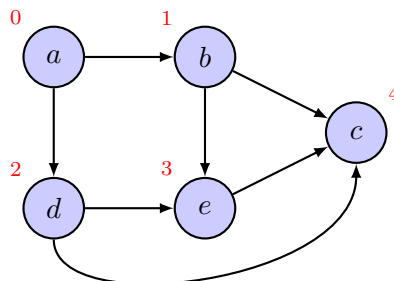
Solution. Since we can treat each edge as a two digit number of base n , we can use radix sort in order to sort the array in $\mathcal{O}(n + m)$.

12.2 Topological sorting

Definition 12.5 (Topological sorting). A topological sorting of a directed graph (V, E) is a numbering of the vertices $N: V \rightarrow \mathbb{N}$ given by $N: v \rightarrow N(v)$, such that $(v, u) \in E$ implies $N(v) < N(u)$. Equivalently, a topological sorting is a graph traversal in which each node v is visited only after all nodes u such that $(u, v) \in E$ have already been visited.

Remark 12.6. The function $N: V \rightarrow \mathbb{N}$ can be implemented as an array $N[1..n]$.

Example 12.6. The following is an example of a topological sorting:



where the red numbers represent the value of the vertex: $N(v)$.

Remark 12.7. A topological sorting does not always exist! It is clear that if a directed graph contains a cycle³ then there cannot exist a topological sorting. We will soon show that if a directed graph is acyclic (such graphs are called DAGs, for **D**irected **A**cylic **G**raphs) then there exists an algorithm which produces a topological sorting in linear time $\mathcal{O}(|V| + |E|)$. Beyond topological sorting, the family of DAGs is one of the most important families of graphs in computer science.

Remark 12.8. Recall that a source is a vertex that has no incoming edges. We get that every DAG has at least one source, otherwise we could choose a random vertex and start going backwards ad infinitum without visiting the same vertex twice, which is a contradiction.⁴

³In a directed graph, unless stated otherwise, a cycle is always a directed cycle.

⁴Note that we have made a tacit assumption that the graph is finite. Don't panic! In this course everything is finite and discrete.

Definition 12.6 (Topological sorting algorithm). The process is as follows:

Algorithm 19: Topological Sorting

Input: A directed graph $G = (V, E)$

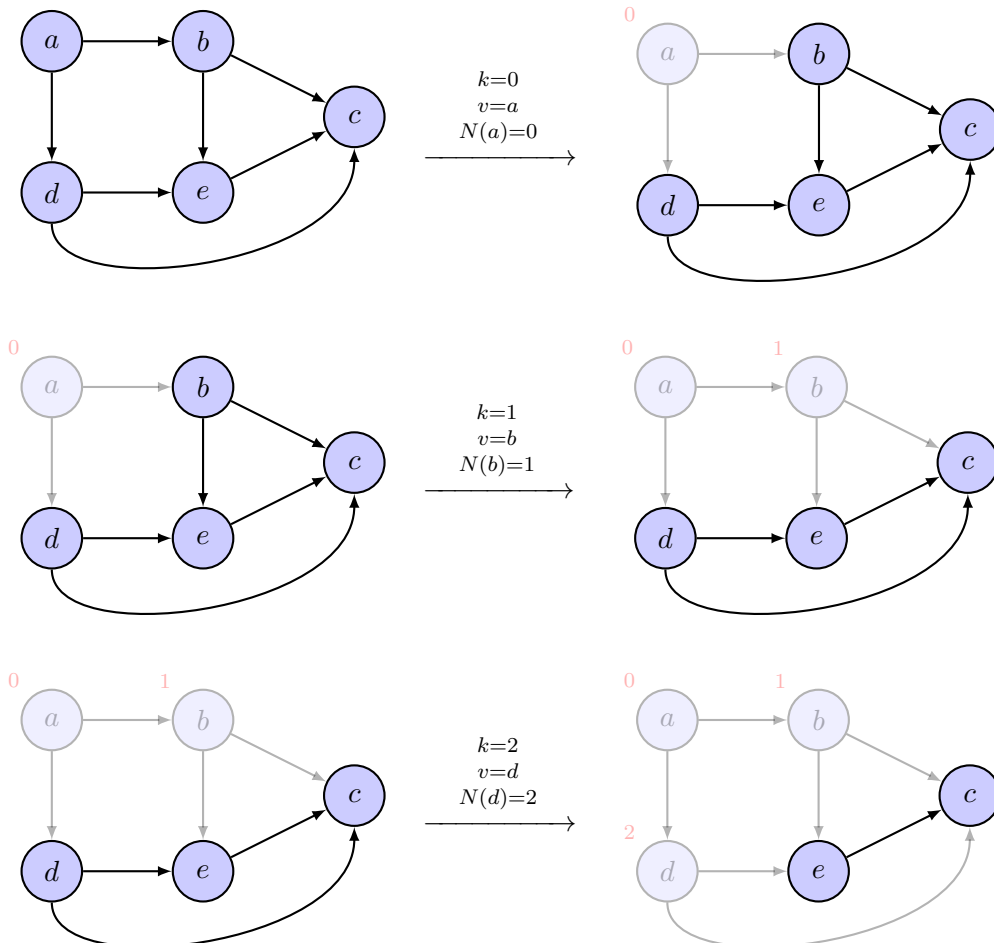
Output: A topological numbering $N(v)$ for all $v \in V$, or report that G has a cycle.

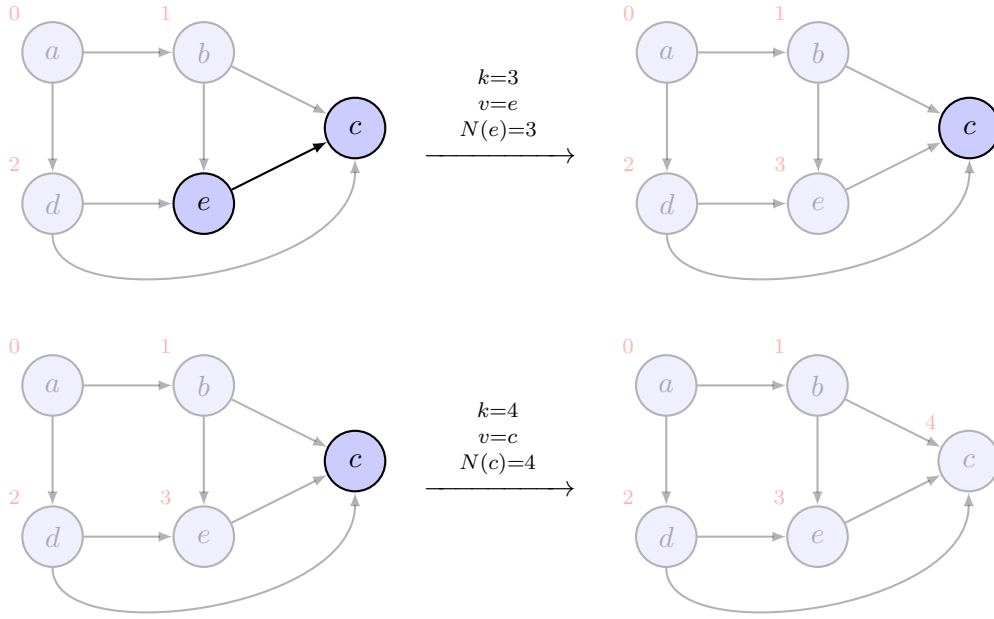
```

1  $k \leftarrow 0$ ;
2 while a source exists in  $G$  do
3    $v \leftarrow \text{FindSource}(G)$ ;
4    $N(v) \leftarrow k$ ;
5    $\text{RemoveSource}(v)$  ;           // Removes  $v$  and all its outgoing edges from  $G$ 
6    $k \leftarrow k + 1$ ;
7 if  $k = |V|$  then
8   return  $N(v)$  for all  $v \in V$  (a topological order)
9 else
10  return " $G$  contains a cycle"

```

Example 12.7. Here is an example for the topological sorting process of the graph from Example 12.6:





Remark 12.9. In order to compute the complexity of Algorithm 19 we must first implement lines 3 and 5 in the code. We will call these functions **FindSource** and **RemoveSource** respectively.

Example 12.8 (**FindSource** and **RemoveSource** using the adjacency matrix). Given a directed graph and its adjacency matrix we can implement **FindSource** by scanning the lines of the matrix until we find a column (of a vertex not sorted yet) that only contains 0s. This implementation for **FindSource** takes $\mathcal{O}(n^2)$ time. In order to implement **RemoveSource** we simply have to set all entries in the source's row to 0 (its column only contains zero already because it is a source), and denote it as “sorted”. This implementation for **RemoveSource** takes $\mathcal{O}(n)$ time.

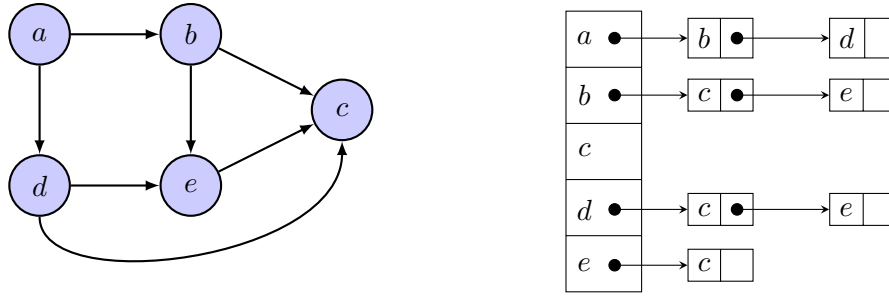
Since in the worst case we execute **FindSource** and **RemoveSource** n times, the total complexity for a topological sorting using this implementation is $\mathcal{O}(n^3)$.

Example 12.9 (**FindSource** and **RemoveSource** using the adjacency matrix, improved). Given a directed graph and its adjacency matrix, we first *preprocess* the input. This means we perform some manipulations on the input before starting the meaningful part of the algorithm. In this case, we construct an array called **InDegree**[1.. n] by scanning the adjacency matrix and storing the in-degree (the amount of edges going into a vertex) of the i th vertex in the i th entry in the array. The time complexity of this preprocessing is $\mathcal{O}(n^2)$. In order to implement **FindSource** we now simply scan the **InDegree** array in order to find a vertex (not sorted yet) with in-degree 0 (a source). The time complexity of this implementation of **FindSource** is $\mathcal{O}(n)$. In order to implement **RemoveSource** we set all entries in the source's row to 0, and denote it as “sorted”. Also, if we denote the source by i , we now also have to decrement **InDegree**[j] by 1 for each $A[i, j] = 1$ in order to maintain the correctness of the **InDegree** array. The time complexity of this implementation of **RemoveSource** is $\mathcal{O}(n)$.

Since in the worst case we execute **FindSource** and **RemoveSource** n times, the total complexity for a topological sorting using this implementation is $\mathcal{O}(n^2)$.

Example 12.10 (**FindSource** and **RemoveSource** using the adjacency list). Given a directed

graph and its adjacency list, for example consider the following graph and adjacency lists:



We can implement **FindSource** by scanning all lists in the adjacency list n times in order to find which vertices are not in any list, and then choose one such vertex (which is necessarily a source). The time complexity of this implementation of **FindSource** is $\mathcal{O}(n \cdot m + n^2)$. In order to implement **RemoveSource** we simply denote it as removed in the adjacency list, which takes $\mathcal{O}(1)$ time.

Since in the worst case we execute **FindSource** and **RemoveSource** n times, the total complexity for a topological sorting using this implementation is $\mathcal{O}(n^2m + n^3)$.

Note that we can improve the algorithm by improving the implementation of **FindSource** a little. We maintain a boolean array initialized to 0 in all entries, and then we simply scan the lists in the adjacency list and set the entries for the vertices we visit to 1. This implements **FindSource** in $\mathcal{O}(n + m)$ time, so that total complexity of the algorithm is now $\mathcal{O}(n \cdot m + n^2)$.

Example 12.11 (Topological sorting using the adjacency list, modified). We can further improve Example 12.10 by maintaining an **InDegree**[1..n] array. Its construction time is $\mathcal{O}(n + m)$. We can now implement **FindSource** by scanning the **InDegree** array which takes $\mathcal{O}(n)$ time. In order to implement **RemoveSource**(i) we first denoted it as “removed” and then scan its list in the adjacency list and update the **InDegree** array accordingly. This implementation of **RemoveSource** takes $\mathcal{O}(\text{OutDegree}(i))$ time.

In total, the algorithm executes $\mathcal{O}(n + m)$ commands for the preprocessing, then it executes the **FindSource** and **RemoveSource** methods n times (once for each vertex). It follows that the total complexity of topological sorting with the above implementations is:

$$\mathcal{O}(n + m) + \sum_{i=1}^n \mathcal{O}(n + \text{OutDegree}(i)) = \mathcal{O}(n + m) + \mathcal{O}(n^2 + m) = \mathcal{O}(n^2),$$

where the last transition follows from the fact $m \leq n^2$.

Example 12.12 (Topological sorting using the adjacency list, optimal). We can further improve Example 12.11 by additionally maintaining a stack/queue/list denoted **Sources** for nodes with in-degree 0. The preprocessing includes creating the **InDegree** array like before, and additionally creating a list for vertices with in-degree 0. This can be done in $\mathcal{O}(n + m)$ time. We can now implement **FindSource** by choosing the first element in **Sources** (the head), which takes $\mathcal{O}(1)$ time. In order to implement **RemoveSource**(i) we now have to scan its list in the adjacency list and update the **InDegree** array. Moreover, if some element in the updated **InDegree** array now has in-degree 0, we add it to the list **Sources** in $\mathcal{O}(1)$ time. This implementation of **RemoveSource** takes $\mathcal{O}(\text{OutDegree}(i))$ time.

In total, the algorithm executes $\mathcal{O}(n + m)$ commands for the preprocessing, then it executes the **FindSource** and **RemoveSource** methods n times (once for each vertex). It follows that the total complexity of topological sorting with the optimal implementations is:

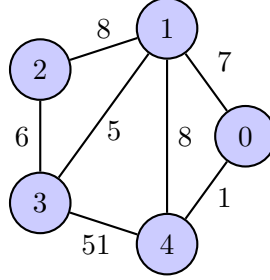
$$\mathcal{O}(n + m) + \sum_{i=1}^n \mathcal{O}(1 + \text{OutDegree}(i)) = \mathcal{O}(n + m) + \mathcal{O}(n + m) = \mathcal{O}(n + m).$$

12.3 Minimum spanning tree

Definition 12.7 (Weighted graph). A graph (V, E) is said to be weighted, if there exists a weight function $w: E \rightarrow \mathbb{R}_{>0}$, which assigns each edge $e \in E$ a positive weight $w(e)$.

Remark 12.10. In this subsection, we focus on weighted undirected graphs, since we can represent them using adjacency matrices.

Example 12.13. Consider the following undirected graph:



We can represent it using a matrix similar to the adjacency matrix in the following way:

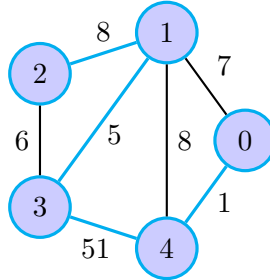
$$\begin{pmatrix} 0 & 7 & 0 & 0 & 1 \\ & 0 & 8 & 5 & 8 \\ & & 0 & 6 & 0 \\ & & & 0 & 51 \\ & & & & 0 \end{pmatrix}.$$

This matrix is given by an array based implementation of a symmetric matrix defined by:

$$A[i, j] := \begin{cases} 0 & (i, j) \in E, \\ w(i, j) & \text{otherwise.} \end{cases}$$

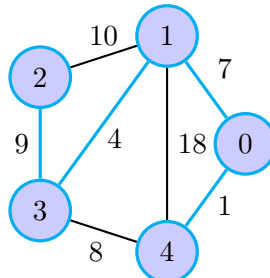
Definition 12.8 (Spanning tree). A spanning tree of an undirected graph $G = (V, E)$ is a subgraph $T = (V', E')$ of G which is a tree (not a forest), such that $V' = V$.

Example 12.14. The following highlighted subgraph is a spanning tree:



Definition 12.9 (Minimum spanning tree). A minimum spanning tree (MST) of a weighted undirected graph $G = (V, E)$, is a spanning tree of G whose sum of edges is minimal.

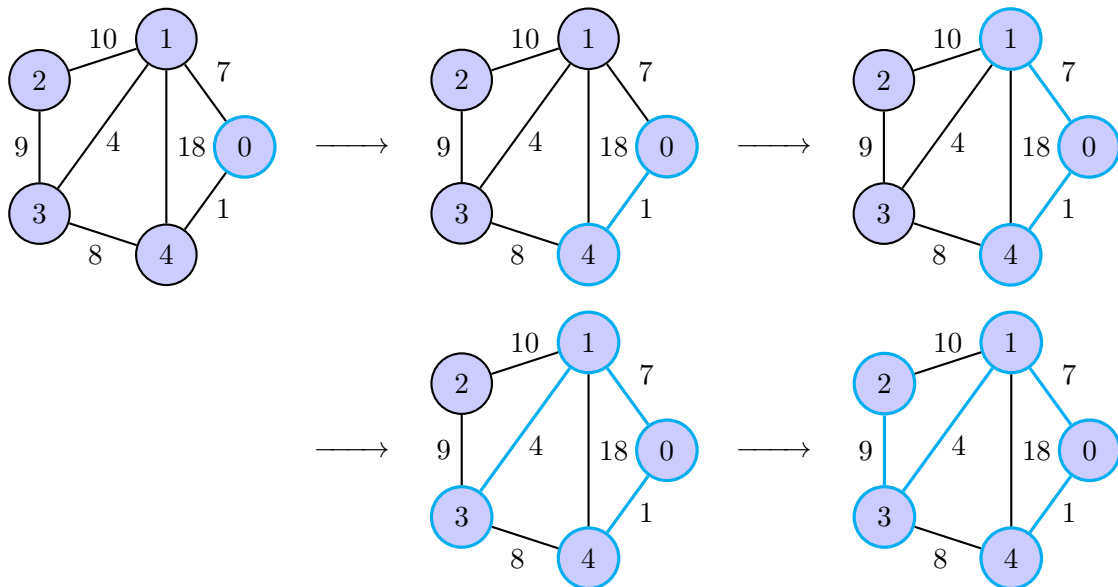
Example 12.15. The following highlighted subgraph is a minimum spanning tree:



Remark 12.11. In the rest of this subsection, we will introduce an algorithm called Prim’s algorithm and show multiple ways to implement it. We will focus on the implementation of the algorithm, the proof for its correctness will not be discussed in this course, but it can be found in CLRS.

Remark 12.12 (Prim’s algorithm). The idea behind Prim’s algorithm is to start by choosing a random vertex and inserting it to an empty tree T . Then at every iteration we find the minimal edge that connects a vertex from T to a vertex not in T , and insert it to T .

Example 12.16. Here is an example of Prim’s algorithm in action, where the first vertex chosen is the 0 vertex:



Definition 12.10 (Prim’s algorithm). In Prim’s algorithm we store the value $d(v)$ for every vertex $v \notin T$, which represents its “distance” from the tree T . If there exists at least one edge (v, u) such that $v \notin T$ and $u \in T$, we let $d(v)$ be the minimal weight of such edges, and otherwise we let $d(v) = \infty$. The implementation of the algorithm is as follows:

Algorithm 20: Prim’s Algorithm

Input: A connected weighted graph $G = (V, E)$

Output: A minimum spanning tree T

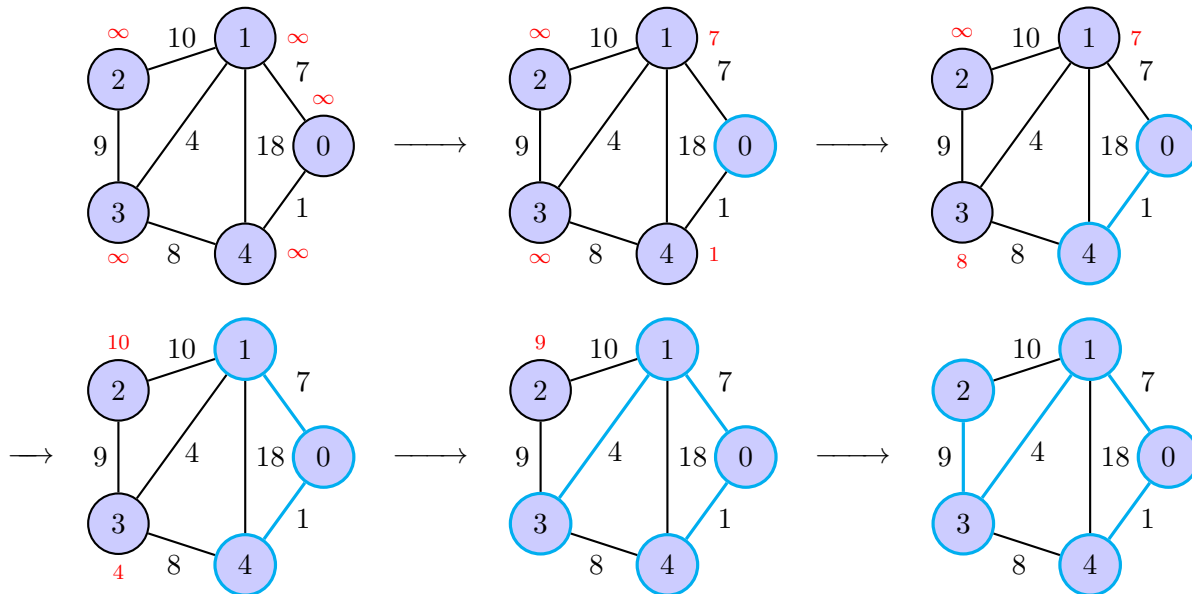
```

1 foreach  $v \in V$  do
2    $d(v) \leftarrow \infty$ ;
3  $T \leftarrow \emptyset$ ;
4 while  $T$  does not contain all vertices do
5   choose any  $i \notin T$  with minimal  $d(i)$ ;
6   add  $i$  to  $T$ ;
7   add edge  $(j, i)$  stored for  $i$  to  $T$ ;           /* Unless in first iteration */
8   foreach outgoing edge  $(i, k)$  do
9     if  $k \notin T$  and  $d(k) > w(i, k)$  then
10       $d(k) \leftarrow w(i, k)$ ;
11      vertex  $k$  stores  $(i, k)$ ;

```

Remark 12.13. Note that there may be multiple vertices with minimal $d(i)$. In this case the algorithm can choose any one of them. In particular, in the first iteration all vertices have $d(i) = \infty$, so any vertex can be the first vertex. Also, note that after adding a vertex i to the tree, its value $d(i)$ becomes meaningless, so we don’t have to track it anymore.

Example 12.17. Here is an example of Prim’s algorithm in action, where the first vertex chosen is the 0 vertex, and we also track the value $d(i)$ of every vertex outside the tree, written in red besides it:



Note that at every step we update the values $d(i)$ of the relevant vertices outside the tree.

Remark 12.14. In order to compute the complexity of Prim’s algorithm (Algorithm 20), we first have to implement its two main high-level parts (finding a vertex i with minimal $d(i)$; scanning the edges (i, k)) which we will conveniently call **FindMinVertex** and **ScanningEdges** respectively.

Example 12.18 (First implementation). In order to implement Prim’s algorithm using an adjacency matrix, we need to create an empty list that stores vertices not in T , which takes $\mathcal{O}(n)$ time in preprocessing. In order to implement **FindMinVertex** we scan the auxiliary list, find the minimal element with respect to $d(i)$ in $\mathcal{O}(n)$, and remove it from the list. In order to implement **ScanningEdges** we need to scan the i th row in the adjacency matrix, and update $d(k)$ for each vertex in the edges (i, k) . The total complexity of this routine is $\mathcal{O}(n)$.

Since **FindMinVertex** and **ScanningEdges** both take $\mathcal{O}(n)$ time, the worst-case time complexity of Prim’s algorithm in this implementation is $\mathcal{O}(n^2)$.

Example 12.19 (Second implementation). If we implement Prim’s algorithm using an adjacency list, we can implement **FindMinVertex** by simply scanning the adjacency list. When we find the vertex with minimal $d(i)$ value we label it as “in T ”, in order to prevent returning the same vertex every time. This implementation of **FindMinVertex** takes $\mathcal{O}(n)$ time. In order to implement **ScanningEdges** we can simply scan the list of the minimal vertex we found and update their $d(k)$ values which takes $\mathcal{O}(1)$ for each vertex. In total the time complexity of this implementation of **ScanningEdges** is $\mathcal{O}(\text{degree}(i))$.

In total for every vertex we do $\mathcal{O}(n + \text{degree}(i))$ operations so that total worst-time complexity of this implementation of Prim’s algorithm is:

$$\sum_{i=1}^n \mathcal{O}(n + \text{degree}(i)) = \mathcal{O}(n^2 + m).$$

Remark 12.15. We notice that the bottleneck in the implementation is the implementation of **FindMinVertex**. Recall that we have previously seen a data structure that allows finding the minimum element in $\mathcal{O}(1)$. This data structure is the “min-heap” data structure.

Example 12.20 (Third implementation). In this implementation, we additionally maintain a min-heap that stores all vertices in the graph that are not in T , and we also modify the vertices in the adjacency list another field which contains a pointer to their associated node in the min-heap.

First, we need to construct the heap with all nodes which takes $\mathcal{O}(n)$ time. Then in order to implement **FindMinVertex** and **ScanningEdges** we need to find the minimum ($\mathcal{O}(1)$), delete it ($\mathcal{O}(\log n)$), and decrease the value $d(i)$ of the vertices connected to the minimum using the function **DecKey** introduced in Remark 9.2 (this takes $\mathcal{O}(\log n)$ time for each vertex).

Since we do these operations for each vertex in the graph, the total complexity of Prim's algorithm in this implementation is thus:

$$\sum_{i=1}^n (\mathcal{O}(1) + \mathcal{O}(\log n)) + \sum_{i=1}^n \text{degree}(i) \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n) + \mathcal{O}(m \log n) = \mathcal{O}(n \log n + m \log n).$$

Question. When the the third implementation better than the second implementation?

Solution. The third implementation is better when $m \log n = o(n^2)$. More reasonably, since the computation is not asymptotic, it is better when the number of edges satisfies the inequality $m < n^2 / \log n$.

Example 12.21 (Fourth implementation). The last and best implementation of Prim's algorithm uses a data structure we have not introduced yet called a *Fibonacci heap*.

We will not cover the implementation of a Fibonacci heap in this course, but the (amortized) complexity of its initialization and standard operations is constant, just like in a min-heap, except that in a Fibonacci heap we can implement the operation **DecKey** in $\mathcal{O}(1)$ instead of $\mathcal{O}(\log n)$.

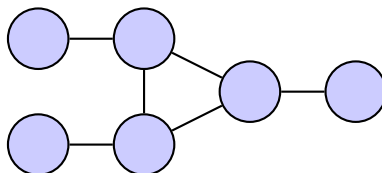
In total, the complexity of Prim's algorithm in this implementation is thus:

$$\sum_{i=1}^n (\mathcal{O}(1) + \mathcal{O}(\log n)) + \sum_{i=1}^n \text{degree}(i) \cdot \mathcal{O}(1) = \mathcal{O}(n \log n) + \mathcal{O}(m) = \mathcal{O}(m + n \log n).$$

Remark 12.16. When the Fibonacci heap data structure was first constructed and introduced, it improved the asymptotic complexity of many algorithms, and in particular it broke the asymptotic lower bound of Prim's algorithm at the time.

It is important to remember that the operations in a Fibonacci heap have constant *amortized* complexity. This means that the performance of a Fibonacci heap is better than that of a min-heap, even though it has worse worst-case complexity.

Exercise 12.2. We say that a graph is k -dense if every vertex in the graph has at least k neighbours. We want to find an algorithm that takes a graph as input, and returns whether it has a k -dense subgraph. For example the graph:



has a 2-dense subgraph, but it does not have a 3-dense subgraph. Find the most optimal algorithm you can in order to implement such an algorithm when:

- (1) The graph is implemented with an adjacency matrix.
- (2) The graph is implemented with an adjacency list.

Denote the number of vertices in the graph by n , the number of edges by m , and the number k can be any natural number.

Solution. Note that when $n < k$ the algorithm would always return **False** since there cannot be any vertex with more neighbours than n . This allows us to assume that $0 < k < n$.

- (1) We know that if a k -dense subgraph, then any vertex in it has degree at least k . This means that we can start by scanning all vertices in the graph, and delete those with degree strictly less than k . We then need to update the degrees of its neighbouring vertices (decrease them by one), and now again, if their updated degree is strictly less than k , we need to remove them as well. After we finish with this type of algorithm, the remaining graph is necessarily a k -dense subgraph.

First, we preprocess the input and create a `degree[1..n]` array containing the degrees of all vertices in the graph ($\mathcal{O}(n^2)$). Then, we insert all vertices with degree less than k into a queue of vertices to be removed ($\mathcal{O}(n)$). Then, until the queue is empty we perform the following routine: We remove the vertex from the graph (mark it `removed`) and decrease the degree of each of its neighbours by one, and in case their updated degree is now strictly less than k we add them to the queue of vertices to be removed ($\mathcal{O}(n)$). Since we do this for all vertices, the total complexity of this algorithm is $\mathcal{O}(n^2)$.

- (2) When the graph is implemented with an adjacency list, we can still apply the same principles and logic. We can compute the degree of all vertices in $\mathcal{O}(n+m)$, and each time we remove the neighbouring vertices of a vertex and update the degrees of its neighbours we make $\mathcal{O}(\text{degree}(i))$ operations. Thus, the total complexity of the algorithm with this implementation is $\mathcal{O}(n+m)$.

12.4 Exercises

Before getting into the exercises, we will start by discussing some more definitions from graph theory. After that we discuss some optimizations for implementing a graph using an adjacency list.

Definition 12.11 (Multigraph). A multigraph is a graph $G = (V, E)$ that is allowed to have multiple edges (also called parallel edges). These are edges that connect the same nodes in the same way but are still different elements of E .

Remark 12.17. If $G = (V, E)$ is a multigraph, the set of edges E can be thought of as a multiset.

Definition 12.12 (Pseudograph). A multigraph that is allowed to have loops (edges from a vertex to itself) are called pseudographs.

Remark 12.18 (Connectivity in directed graphs). In the case of undirected graphs, we had a notion of connectivity which means that any two vertices have a path between them.

A direct graph is called weakly connected if the underlying undirected graph is connected, and strongly connected if there is a directed path between any two nodes in both directions. A directed graph where every two nodes have a directed path between them in at least one direction is called semiconncted (or unilaterally connected).

Remark 12.19 (Optimization for adjacency lists). In some situations there are multiple optimizations we can make to an adjacency list.

Sorted lists - This means that the lists of neighbours would be sorted. With this optimization, if the first edge we see in a list is $(v, 4)$ and we search for $(v, 2)$ we can immediately return **false** since $2 < 4$ at the cost of insertion time. This does not change the worst-case complexity.

AVL trees - Instead of using lists, each element in the array would be an AVL tree. This makes the operations search/insert/delete worst-case $\mathcal{O}(\log d_i)$ time. Note that there are cases where this does not improve the running-time. For example, in many graphs in the real world a bound of the number of neighbours is known, and hence $\mathcal{O}(d_i) = \mathcal{O}(\log d_i) = \mathcal{O}(1)$. Moreover, even when we are not given such a bound, if the graph is a multigraph we don't check whether an edge is already present in the graph before we insert it so insertion is already $\mathcal{O}(1)$. This optimization does not make any operation run in worse time asymptotically, but in reality, an AVL tree can perform more $\mathcal{O}(1)$ operations than necessary (for example in multigraph insertion), so it could technically be worst, but we only care for asymptotic complexity in this course so you may assume it is always better to use this optimization.

Mixed implementation - We can add to the adjacency list implementation an $n \times n$ matrix where each entry in the matrix would contain a pointer to the respective edge in the adjacency list. This makes the search/insert/delete operations $\mathcal{O}(1)$ at the cost of $\Theta(n^2)$ space complexity. Note that even though we use an adjacency matrix type object here, scanning the neighbours of an edge still takes $\mathcal{O}(d_i)$ since we have the adjacency list as well.

Exercise 12.3. Find an effective way to implement an undirected graph so that we can check whether it is connected or not efficiently. Assume that the initialization process is not part of the complexity analysis.

Solution. Assume the graph is $G = (V, E)$ and $|V| = n$ and $|E| = m$. If $n = 0$ the graph is always connected so we assume $n > 0$. We will present multiple solutions.

- (1) Initialize a union-find structure where each vertex represents a set, then run the following algorithm:

Algorithm 21: Graph Connectivity using Union-Find.

```

1 Function CheckConnectivityUF( $V, E$ ):
2    $x \leftarrow n$ ;
3   Initialize Union-Find structure for each  $v \in V$ ;
4   foreach  $(u, v) \in E$  do
5     if Find( $u$ )  $\neq$  Find( $v$ ) then
6       Union(Find( $u$ ), Find( $v$ ));
7        $x \leftarrow x - 1$ ;
8   if  $x = 1$  then
9     return "Connected";
10  else
11    return "Disconnected";
```

We perform a constant amount of union find operations in each operation of the for loop, and since there are m edges the loop process takes $\mathcal{O}(m \log^* n)$ time. However, the for loop iterates over all edges we need to consider the implementation of the graph. If the graph is implemented using an adjacency matrix then scanning all edges takes $\mathcal{O}(n^2)$, so in that case the total complexity is $\mathcal{O}(n^2 + m \log^* n)$. If the graph is implemented using an adjacency list then scanning all edges takes $\mathcal{O}(n + m)$, so in that case the total complexity is $\mathcal{O}(n + m + m \log^* n) = \mathcal{O}(n + m \log^* n)$ time.

- (2) We can notice that an undirected graph is connected if and only if there is a path from any arbitrary $v \in V$ to any $u \in V$. Fix $v \in V$. We can now solve the problem in $\mathcal{O}(m)$ using the idea of an algorithm called "Breadth First Search" (or BFS for short) which we

will learn more about in the algorithms course. The idea is to go over all edges connected to v and check if we can reach all nodes from v . The algorithm is as follows:

Algorithm 22: Connectivity check via neighbor list concatenation.

```

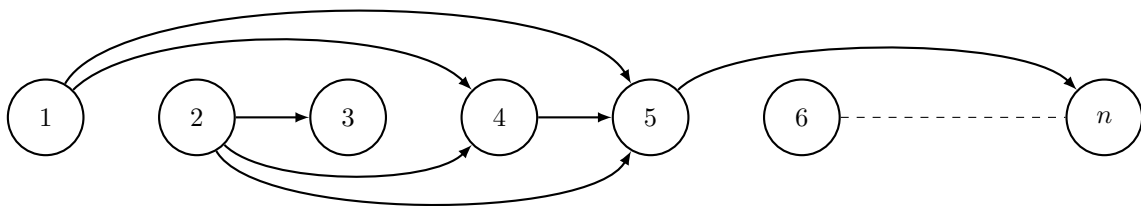
1 Function CheckConnectivityList( $V, \text{Adj}$ ):
2    $x \leftarrow n$ ;
3   Initialize array  $A[1..n]$  to false;
4   Pick an arbitrary start vertex  $v \in V$ ;
5    $A[v] \leftarrow \text{true}$ ;
6    $p \leftarrow$  pointer to head of  $\text{Adj}[v]$ ;
7    $x \leftarrow x - 1$ ;
8   while  $p \neq \text{null}$  do
9      $u \leftarrow p.\text{vertex}$ ;
10    if  $A[u] = \text{false}$  then
11      Concatenate list  $\text{Adj}[u]$  to the end of the current traversal list;
12       $A[u] \leftarrow \text{true}$ ;
13       $x \leftarrow x - 1$ ;
14    Increment  $p$  one step forward;
15  if  $x = 0$  then
16    return "Connected";
17  else
18    return "Not Connected";

```

In this algorithm Adj represents the adjacency list and $\text{Adj}[v]$ represents the list of neighbours of v . The array A documents which nodes have already been visited.

Exercise 12.4. Let $G = (V, E)$ be a directed graph with nodes $V = \{1, 2, \dots, n\}$ such that for all $(i, j) \in E$ we have $i < j$. Find an efficient algorithm for finding the length of the longest path in the graph.

Solution. We have a topologically sorted graph. Here is one example for visualization:



Our strategy is to compute for each node the longest path that starts at that node. We will compute these values and store them in an array L , so that $L[i]$ contains the length of the longest path that starts at the i th vertex. Note that $L[n] = 0$ and similarly $L[i] = 0$ for each vertex i that is a sink. We can easily compute the array L since if we scan the vertices of G from the $n - 1$ th to the 1st vertex, we can use the formula:

$$L[i] = 1 + \max \{L[j_1], L[j_2], \dots, L[j_k]\},$$

where $j_1, \dots, j_k$ are all the neighbours of i . By properties of a topologically sorted graph, all of i th neighbours are to its right hence $\max \{L[j_1], L[j_2], \dots, L[j_k]\}$ and consequently $L[i]$ can be computed in the same time it takes to scan its neighbours.

In case the graph is implemented using an adjacency matrix, scanning the neighbours of each vertex takes $\Theta(n)$ and hence the algorithm would take $\Theta(n^2)$ total time.

In case the graph is implemented using an adjacency list, scanning the neighbours of each vertex takes $\Theta(d_i)$ where d_i is the degree of the vertex i , and hence the algorithm would take $\Theta(n + \sum_{i=1}^n d_i) = \Theta(n + m)$ total time.

Question. Can Exercise 12.4 be solved for a general directed graph?

Answer. If the graph contains a directed cycle, then there exists a path (technically a walk) of length t for all $t \in \mathbb{N}$.

If the graph has no directed cycles, then it is a directed acyclic graph (DAG), and hence by an algorithm we saw in the lectures we can perform a topological sort on it in $\mathcal{O}(n + m)$, and then find the longest path in the graph in the same way and the same complexity as we had in the original solution.

13 Bloom filters

The motivation for Bloom filters comes from the real-life need to have (partial) quick access to dictionary-type structures quickly.

For example, if we want to store 500,000,000 URLs of malicious sites in order to discern dangerous links from safe ones, we would need to maintain a really large data structure such as a B+ tree, and such a B+ tree (assuming each URL is approximately 20 characters) costs (without compressions) 10GB of memory.

Since it is only possible to store such large data structures on the disk, and since each access to the disk is very expensive, a need for a less space-heavy data structure that supports at least some operations of a dictionary such as lookups (for example when inserting an element to the dictionary we cannot avoid accessing the disk). As usual, in order to get better (space) complexity, we have to make the structure probabilistic and compromise on accuracy.

Definition 13.1 (Approximate membership queue). An approximate membership queue (also known as AMQ) is a probabilistic ADT that supports the following operations:

Create() Creates an empty data structure S .
Add(S, x) Add element x to S .
IsMember(S, x) Returns **True** if x is in S and returns **False** with probability $1 - \epsilon$ otherwise.

Remark 13.1. In an AMQ, a false positive is possible, but a false negative is not. This means that if an AMQ returns **True** for a search query, then the element is most likely in the list, but possibly it is not, but when it returns **False**, then it is definitely not on the list. It will be easier to understand why this is the case when we see the implementation of Bloom filters. Note that in the example of storing 500,000,000 malicious URLs it makes sense to allow a small amount of false positives but avoid false negatives at all costs.

Remark 13.2 (How are things better now?). We are not avoiding storing the dictionary key-value pairs in the disk, this is unavoidable. However, when creating the dictionary we additionally create an approximate membership queue F , such that:

- Whenever we insert an item to the dictionary we also insert it to F .
- Before accessing the disk for a lookup of some element x , we first check the truth value of **IsMember(F, x)**. If it is false, then we do not access the disk (this saves a lot of time), and if it returns **True**, we still have to access the disk to make sure this is not a false positive.

Remark 13.3. As you can see in Definition 13.1, an approximate membership queue does not support deletions. Indeed, we use AMQs only when the dictionary does not require deletions. For example, if the dictionary stores malicious URLs, there is (usually) no reason to delete a URL from the list.

Example 13.1 (Naïve implementation of an AMQ). We can implement an AMQ using a hash table with a hash function h . The size of the hash table must still be linear in the size of the dictionary, but instead of storing the large keys, we will simply initialize the table to be all zeros, which saves a large amount of space. We implement the rest of the operations of an AMQ as follows:

Add(S, x) Set the bit in index $h(x)$ to 1.
IsMember(S, x) Return **True** if and only if the bit $h[x]$ in the table is 1.

Example 13.2. Suppose that the size of the table is 5, and the hash function is $h: n \mapsto n \pmod{5}$. After inserting the elements 7, 13, 23, 42, 11 to the AMQ, the table would look like this:

0	1	1	1	0
0	1	2	3	4

Now we get that `IsMemeber(F, 30)` returns `False`, `IsMemeber(F, 23)` returns `True`, however, `IsMemeber(F, 17)` also returns `True`, which is a false positive.

Remark 13.4 (Correctness and complexity). It is fairly clear that the implementation is correct (there aren't any false negatives), however, there may be false positives, as we saw in Example 13.2. The time complexity of the algorithm is $\mathcal{O}(1)$, and the space algorithm is $\mathcal{O}(m)$ where m is the size of the boolean hash table.

Question. Why is this implementation of an AMQ does not support deletions?

Answer. Consider the hash table and function from Example 13.2, and suppose we insert 11 and 21, then delete 11. We consider the two possible alternatives to implementing deletions. If we change the bit in the $h(11) = 1$ index from 1 to 0, then `IsMemeber(F, 21)` returns `False`, which is a false negative. Since false negatives are not allowed in an AMQ, this is not a viable implementation. On the other hand, if we keep `F[1]` as 1, then over time as we insert and delete more and more elements, the rate of false positives will increase more and more, which eventually renders the AMQ useless.

Remark 13.5 (False positive rate analysis). Denote the size of the AMQ hash table by m . Given an element x which has not been inserted to the AMQ, what is the probability of a false positive under the assumption n elements have been inserted so far. We also assume that h satisfies the simple uniform hash assumption.

For every inserted element the chance of a non-collision with x is $(1 - 1/m)$, and thus the chance that not a single element collided with x is $(1 - 1/m)^n$ (in which case the bit associated with x is still zero). Therefore, by a mathematical approximation, we get that the probability of a false positive is $1 - (1 - 1/m)^n \approx 1 - e^{(-n/m)}$.

Example 13.3 (Bloom filter). A Bloom filter is another, improved implementation of an AMQ, that is very similar to the naïve implementation introduced in Example 13.1. The main difference is that instead of using a single hash function, we use multiple hash function $h_1, \dots, h_k$. The implementation of the operations is as follows:

`Add(S, x)` Set the bits in indices $h_1(x), \dots, h_k(x)$ to 1.

`IsMember(S, x)` Return `True` if and only if *all the bits* $h_1(x), \dots, h_k(x)$ are 1.

Remark 13.6. The naïve implementation is a special case of a Bloom filter where $k = 1$. Also, although this is an improved version of the naïve implementation, notice that false positives are very much possible (and false negatives are impossible).

Remark 13.7 (Complexity). From similar calculation to the calculations done in Remark 13.5, we get that the probability of a false positive is $(1 - e^{(-kn/m)})^k$. Given n and m this allows us to find the optimal value of k , which turns out to be $k_{\min} = \ln(2)m/n$. The probability for a false positive using $k_{\min}$ is approximately $0.6185^{m/n}$.

Remark 13.8. Suppose $m/n = 8$. In this case, by allocating 8 bits per key on average ($m/n = 8$ is the average case), and choosing $k = \lfloor k_{\min} \rfloor = \lfloor 8 \ln 2 \rfloor = 5$, we get a probability of not more than 3% for a false positive.

If we use such a Bloom filter for the example from the very beginning of this section, we would only need 0.5GB to maintain it. This allows us to store the Bloom filter on the working memory (RAM) for fast access.

Remark 13.9. We say that the optimal state for a Bloom filter is one where half of the bits in array (hash table) are turned on. This is because with $k := k_{\min}$, we get that the probability that a bit is 1 (or similarly 0) is:

$$e^{-km/n} = e^{-\ln 2} = \frac{1}{e^{\ln 2}} = \frac{1}{2}.$$

Intuitively, that is because if there are too many 1s the rate of false positives explodes, and when there are too few 1s, the number of false positives is small enough so that we could insert more elements to the Bloom filter without affecting the false positive rate too much.

Remark 13.10. Bloom filters are still commonly used in the industry. They were invented in 1970 by Burton Howard Bloom, and had a resurgence of attention in the 2000s when we found out about more of their generalizations and applications.

Example 13.4 (Counting Bloom filter). One of the generalizations for Bloom filters is a counting Bloom filter, which was introduced in the year 2000. This variation of a Bloom filter supports deletions by modifying the table into holding a counter instead of a single bit at each entry. Whenever we add an element x we increase the counter at index $h_i(x)$ by 1 for all $1 \leq i \leq k$, and whenever we delete an element y , we decrease the counter at index $h_i(y)$ for all $1 \leq i \leq k$.