

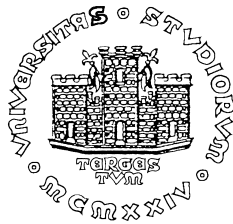
UNIVERSITY OF TRIESTE

INTERNATIONAL SCHOOL FOR ADVANCED STUDIES

THE ABDUS SALAM INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS

Algorithmic Design

LECTURES NOTES



Author:
Marco SCIORILLI

Monday 19th April, 2021

Abstract

This document contains my notes on the course of Algorithmic Design held by Prof. Alberto Casagrande for the Master Degree in Data Science and Scientific Computing at Trieste University in the year 2020/2021. As they are a work in progress, every correction and suggestion is welcomed. Please, write me at: marco.sciorilli@gmail.com .

Contents

1	Fundamentals	4
1.1	Algorithm and Computational Model	4
1.2	Time Complexity	6
1.3	Cost Criteria	8
1.4	Some Useful Notions	9
2	Strassen's Algorithm for Matrix Multiplication	11
2.1	Problem definition	11
2.2	Motivation	11
2.3	Naive Solution	12
2.4	Divide-and-Conquer Strategy	12
2.5	Strassen's Algorithm	13
3	Matrix Chain Multiplication Problem	15
3.1	Problem definition	15
3.2	A naive approach	16
3.3	A dynamic programming solution	17
4	Heaps	19
4.1	Binary Heaps	19
4.2	Finding the Minimum	21
4.3	Removing the minimum	22
4.4	Building a Heap	23
4.5	Decreasing a Key	24
4.6	Inserting a Value	25
5	Retrieving Data and Sorting	26
5.1	Retrieving Data	26
5.2	Sorting	27
5.3	Insertion Sort	27
5.4	Quick Sort	28
5.5	Finding the Maximum	31
5.6	Sorting in Linear Time	33
5.7	Select	36

6	Binary Search Trees and Red-Black Trees	40
6.1	Motivation	40
6.2	Binary Search Trees	41
6.3	Red-Black trees	49
7	Graphs	54
7.1	Basics	54

Chapter 1

Fundamentals

1.1 Algorithm and Computational Model

Definition 1. *Algorithm: a sequence of well-defined steps that transforms a set of inputs into a set of outputs in a finite amount of time.*

The output should be consistent with the given definition of the problem. In some specific times, there could be actually an algorithm that goes on indefinitely. But that case is not studied in this course.

Definition 2. *Computational model: a mathematical tool to perform computations.*

A function described by an algorithm is calculable (i.e. I can give an high level set of instructions to turn the input of the function into the output of the function, I can provide and algorithm for it).

A function is implementable in a computational model is computable (i.e. Given a formal model, it provides a set of instructions which are fixed, making me able to write a program that turn the input of the function in the output of the function using only that proposed rules).

Notion of calculable and computable are related? Algorithms would not guarantee implementability! An interesting example is the "Halting problem": we want to write a program that takes as input any other program and it establish whether that ends in a finite amount of time or not.

Example 1. *Halting Problem: Let h be the function that establish whether any program p eventually ends its execution (\downarrow) on an input i or runs forever (\uparrow)*

$$h(p, i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } p(i) \text{ never ends} \\ 1 & \text{otherwise} \end{cases} \quad (1.1)$$

It is not possible to implement h .

Proof. For any computable function $f(a, b)$ we can define the function

$$g_f(i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } f(i, i) = 0 (\text{if it ends}) \\ \uparrow & \text{otherwise (it goes on forever)} \end{cases} \quad (1.2)$$

If we assume that f is computable, so is g_f , cause the first part can be implemented using a conditional statement, while the second can be implemented using a "while" forever loop. If I can evaluate the former, I can evaluate the latter. Let's call G_f the program implementing the function g_f .

Can f be the Halting function? No

- If $f(G_f, G_f) = 0 \Rightarrow g_f(G_f) = 0$ and $h(G_f, G_f) = 1$: a.k.a. if G_f applied on G_f gives 0, then g_f ends, so the halting function return 1.
- If $f(G_f, G_f) \neq 0 \Rightarrow g_f(G_f) \uparrow$ and $h(G_f, G_f) = 0$: a.k.a. if G_f applied on G_f is not 0, then g_f goes on forever, so the halting function return 0.

So, any computable function must be different from h , because any f with a given result must be different from its associated h . □

Luckily, there is a thesis that gives a correlation between computable and calculable.

Thesis 1. *Church-Turing Thesis: Every effectively calculable function is a computable function.*

So, if we are able to describe an algorithm for a function f , then f can be formally computed. This has some direct consequences

- All the "reasonable" computation models are equivalent.(i.e. we are able to write a program in every different programming language)
- We can avoid "hard-to-be-programmed" models.(i.e. we can avoid models which are not easy to be handled)

Example 2. *Random-Access Machine: an easy computational model.*

It allows to

- *Variables to store data (no types)*
- *Arrays*
- *Integer and floating point constants*
- *Algebraic functions*
- *Assignments*
- *Pointers (no pointer arithmetic)*
- *Conditional and loop statements*
- *Procedure definitions and recursion*
- *Simple "reasonable" functions*

This is the computational model used in this course, not a real machine. Now let's look at an example of an algorithm, assuming to work on this computational model (so thanks to the Church-Turing Thesis every conclusion on this machinery will work on real life machine)

Algorithm 1: Algorithm example: find the maximum in an array

Input: An array A of numbers $\langle a_1, \dots, a_2 \rangle$

Output: The maximum among a_1, \dots, a_2

```
def find_max(A)
    max_value ← A[1]
    for i ← 2..|A| do
        if A[i] > max_value then
            max_value ← A[i]
        end
    end
    return max_value
enddef
```

Ram is not a real hardware: any variable in a ram algorithm can store any possible number. It doesn't have memory hierarchy, and no difference in instruction execution time (for example, same time for all the operations).

1.2 Time Complexity

We want to assess the efficiency of the algorithm. Execution time is not an effective way to do so, because it is not applicable in many cases. Counting the number of operations could be a good idea, but it depends on the size of the input.

Definition 3. *Scalability: effectiveness of a system in handling input growth.*

It means that the system has to keep the execution time low even if the input is big. We want to estimate the relation between the input size and the execution time, we use a function to do so.

The important part of this function is the asymptotic behavior (its exponent). Constants are not relevant. A theorem called "linear time speedup theorem", supports this idea, as it assesses that for every Turing Machine, it is possible to build an equivalent one scaled by a constant.

Big * notation

These are ways in which one can collect all the functions that behave in the same way.

Definition 4. *Big-O notation:*

$$O(f) \stackrel{\text{def}}{=} \{g | \exists c > 0 \exists n_0 > 0 \text{ s.t. } m \geq n_0 \Rightarrow g(m) \leq cf(m)\} \quad (1.3)$$

It gives the upper bound.

Example 3. *Examples of functions' $O(n)$ appartenance:*

- $n \in O(n)$
- $2n \in O(n)$
- $500n \in O(n)$
- $2n + 7 \in O(n)$
- $n \in O(n^2)$
- $n \in O(2n) = O(n)$

Definition 5. *Big- Ω notation:*

$$\Omega(f) \stackrel{\text{def}}{=} \{g | \exists c > 0 \exists n_0 > 0 \text{ s.t. } m \geq n_0 \Rightarrow cf(m) \leq g(m)\} \quad (1.4)$$

It gives a lower bound.

Example 4. *Examples of functions' $\Omega(n)$ appartenance:*

- $n \in \Omega(n)$
- $2n \in \Omega(n)$
- $500n \in \Omega(n)$
- $2n + 7 \in \Omega(n)$
- $n \notin \Omega(n^2)$
- $n \in \Omega(2n) = \Omega(n)$

Properties of Big-O notation, for any $c_1, c_2 \in \mathbb{N}$ and for any $k \in \mathbb{Z}$:

- $f(n) \in O(f(n))$
- $O(f(n)) = O(c_1 f(n) + k)$
- if $c_1 \geq c_2$ then $O(f(n)^{c_1} + k f(n)^{c_2}) = O(f(n)^{c_1})$
- $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$
- if $h(n) \subseteq O(f(n))$ and $h'(n) \in O(g(n))$, then
 - $h(n) + h'(n) \in O(g(n) + f(n))$
 - $h(n) \times h'(n) \in O(g(n) \times f(n))$

Definition 6. *Big- Θ notation:*

$$\Theta(f(n)) \stackrel{\text{def}}{=} \{g(n) | \exists c_1, c_2 > 0 \exists n_0 > 0 \text{ s.t. } m \geq n_0 \Rightarrow c_1 f(m) \leq g(m) \leq c_2 f(m)\} \quad (1.5)$$

Theorem 1.

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \cap \Omega(g(n)) \quad (1.6)$$

1.3 Cost Criteria

We are interested in how many time instructions are executed, not in the time of the single instruction.

Uniform cost criterion states that time cost is:

- 1 for any Boolean and algebraic expression evaluation.
- 0 for assignments and control instructions.

Any expression, no matter how difficult it is, cost the same. The aim is not to evaluate the execution time, but how many time an instruction is executed.

Example 5. *Uniform Cost Criterion Example*

Algorithm 2: Uniform Cost Criterion Example

```
def test(n):
    Z ← 2 // costs 1
    for i ← 1..n do // loop n time
        | Z ← Z * Z // costs 1
    end
    return Z
enddef
```

Cost functions is $T(n) = 1 + n * 1 \in \Theta(n)$

However, $test(n) = 2^{2^n}$, so the function in linear time, according to the uniform cost criterion, return a very large number, that could not be stored in a linear space.

Logarithmic Cost Criterion states that:

- $\max_{i \in [0, c]} (\log a_i)$ for any Boolean and algebraic expression involving a_0, \dots, a_c as operands.
- 0 for assignments and control instructions.

Example 6. *Logarithmic Cost Criterion Example*

Algorithm 3: Logarithmic Cost Criterion Example

```
def test(n):
    Z ← 2 // costs 2
    for i ← 1..n do // loop n time
        | Z ← Z * Z // costs log Z
    end
    return Z
enddef
```

Cost functions is $T(n) = \sum_{i=1}^n \log 2^{2^i} = \sum_{i=1}^n 2^i = 2 * (2^n - 1) \in \Theta(2^n)$

Logarithmic cost better represents big-number algorithms. If instead the representation space is bounded (as for CPU arithmetic), uniform cost is enough. Where not otherwise declared, we will use uniform cost criterion.

1.4 Some Useful Notions

Definition 7. *Arrays: indexed collections of values which is fixed in length.*

Definition 8. *Single-Linked Lists: sequences of values supporting **head** (the first element) and **next** operations.*

Definition 9. *Double-Linked Lists: sequences of values supporting **head** (the first element), **next** and **previous** operations.*

Definition 10. *Queues: Collections of values ruled according to the FIFO policy (First In First Out: the first element inserted in the list, should be the first one to be taken out). They support **head**, **is_empty**, **insert_back**, **extract_head** operations.*

Definition 11. *Stacks: Collections of values ruled according to the LIFO policy (Last In First Out: the last element inserted in the list, should be the first one to be taken out). They support **head**, **is_empty**, **insert_back**, **extract_head** operations.*

Definition 12. *Graphs: set of pairs (V, E) where:*

- *V is a set of nodes*
- *E is a set of edges*

If the edges are (un)directed, the graph itself is (un)directed. In directed graph, all edges has a direction: e.g. if we can go from one node to another, it may not be the case the other way around. In an undirected graph there are no such limitations.

Definition 13. *Path of length n between $a, b \in V$: a sequence e_1, \dots, e_n (sequence of edges) s.t.*

- *e_1 involves a*
- *e_n involves b*
- *e_i and e_{i+1} involve a common node n_i*

Definition 14. *Some definitions:*

- *A Cycle is a node in which the first node is also the last one.*
- *A graph is connected if there is a path between every pairs of nodes.*
- *A connected component of a undirected graph G is a maximum connected sub-graph of G .*
- *A graph is acyclic if it does not contains cycles.*
- *Directed Acyclic Graphs are also known as DAGs: i.e. a graph in which every edge is directed, and there is no cycle.*

A tree is a connected and acyclic undirected graph. Tree are useful to organize data, and store it in a smart way. The **root** is the node from which all the tree originate.

The **depth** the length of the path going from the node considered to the **root**.

A **level** is a set of nodes having the same **depth**.

The parent of a node is a node one step closer to the root. The children of a node have

the node as a parent. Two nodes are siblings if they have the same parent. Leaves are nodes without children, while internal nodes have children.

The height of a tree is the max depth among those of its leaves.

Every node of a n -ary tree can have up to n children. A n -ary tree is complete if the nodes in all the levels but the last one have n children.

Chapter 2

Strassen's Algorithm for Matrix Multiplication

2.1 Problem definition

Definition 15. *Row-Column Multiplication:* let A be a $n \times n$ matrix and let B be a $m \times l$ matrix. $A \times B$ is a $n \times l$ matrix s.t.

$$(A \times B)[i, j] = \sum_k A[i, k] * B[k, j] \quad (2.1)$$

Input: Two $n \times n$ matrices A and B (where n is a power of 2)

Output: The $n \times n$ matrix $A \times B$

2.2 Motivation

- In Deep Neural Network, both training and evaluation heavily rely on matrix multiplication.
- Good example to learn how to compute the complexity of an algorithm.

2.3 Naive Solution

Do exactly as you would do by hand.

Algorithm 4: Naive matrix multiplication

```
def naive_mult(C,A,B):
    for i ← 1..rows(A) do
        for j ← 1..cols(B) do
            a ← 0
            for k ← 1..cols(A) do
                a ← a + A[i, k] * B[k, j]
            end
            C[i, j] ← a
        end
    end
    return C
enddef
```

The algorithm is composed by 3 nested loop with indexes in $[1, n]$. The inner block takes $O(1)$, so the overall execution complexity takes time $\Theta(n^3)$. Is it possible to find a better algorithm using recursion?

2.4 Divide-and-Conquer Strategy

We try to use recursion by splitting the matrix into 4 quadrants, where

$$C_{ij} = (A_{i1} \times B_{1j}) + (A_{i2} \times B_{2j}) \quad (2.2)$$

We can then use the standard technique on the blocks. Let's find the complexity of this new matrix.

We can find the complexity required to find a quadrant by finding the complexity of the matrix product between matrices a quarter of the original size, plus the cost of the sum of the resulting matrices. Defining some symbols:

- $+$ is the element-wise matrix sum (with time complexity $\Theta(n^2)$)
- \times is the usual row-column multiplication
- A_{ik} and B_{kj} are $\frac{n}{2} \times \frac{n}{2}$ matrices

So it is possible to write the equation

$$T(n) = \begin{cases} 1 & n = 1 \\ 4(2 \cdot T(\frac{n}{2}) + \Theta(\frac{n^2}{4})) = 8 \cdot T(\frac{n}{2}) + \Theta(n^2) & \text{otherwise} \end{cases} \quad (2.3)$$

To solve it we can use a recursion tree. Each node is labeled by the cost of that specific call. In particular since we are saying that the cost at each step of the recursion is quadratic, we may select one specific constant such that the cost we are dealing with belong to $\Theta(n^2)$.

The first call cost $c \cdot n^2$. After the first call the matrix is split in four parts, and for each of the four new call we need to go through recursive call, so we have four child node of cost $c \cdot (\frac{n^2}{2})$ and we go on splitting in new children nodes until $n = 1$. The height of the tree is $\log_2 n$. So

$$\begin{aligned}
T(n) &= c \cdot n^2 + 2 \cdot c \cdot \frac{4}{2^2} \cdot n^2 + \dots \\
&= \sum_{i=0}^{\log_2 n} 8^i \cdot c \cdot \frac{n^2}{n^i} = \sum_{i=0}^{\log_2 n} \left(\frac{8}{4}\right)^i \cdot c \cdot n \\
&= c \cdot n^2 \cdot \frac{(2^{\log_2 n + 1}) - 1}{2 - 1} \\
&= c \cdot n^2 \cdot (2 \cdot n - 1) \\
&= 2 \cdot c \cdot n^3 - c \cdot n^2 \in O(n^3)
\end{aligned}$$

Which is the same as the naive algorithm. It would be nice to reduce the recursive calls. It is possible to do so.

2.5 Strassen's Algorithm

This algorithm consists of doing a bunch of sums that ends up in 10 different matrices, than perform on them 7 matrix multiplications, and from them, doing other sums, we are able to end up with the quadrants of the final matrix. Strassen achieve better results by doing more sums. The recursive equation associated with Strassen's algorithm

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 7 \cdot T(\frac{n}{2}) + \Theta(n^2) & \text{otherwise} \end{cases} \quad (2.4)$$

The recursion tree this time is So the final time complexity is

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_2 n} 7^i \cdot c \cdot \frac{n^2}{n^i} \\
&= c \cdot n^2 \cdot \sum_{i=0}^{\log_2 n} \left(\frac{7}{4}\right)^i \\
&= c \cdot n^2 \cdot \frac{\left(\left(\frac{7}{4}\right)^{\log_2 n + 1} - 1\right)}{\frac{7}{4} - 1} \\
&= c' \cdot n^2 \cdot \left(\left(\frac{7}{4}\right)^{\log_2 n + 1} - 1\right) \quad \text{for } c' = \frac{4}{3}c \\
&= c'' \cdot n^2 \cdot \left(\frac{7}{4}\right)^{\log_2 n} - \frac{4}{7} \\
&= c'' \cdot 4^{\log_2 n} \cdot \left(\frac{7}{4}\right)^{\log_2 n} - c' \cdot n^2 \\
&= c'' \cdot 7^{\log_2 n} - c' \cdot n^2 \\
&= c'' \cdot n^{\log_2 7} - c' \cdot n^2 \in \Theta(n^{\log_2 7})
\end{aligned}$$

So we are able to reduce the complexity to

$$O(n^{\log_2 7}) \subseteq O(n^3) \tag{2.5}$$

Chapter 3

Matrix Chain Multiplication Problem

The problem is focused on the parenthesization of the product, the order in which we want to apply the matrix multiplication.

3.1 Problem definition

We are dealing with a sequence of matrices. As the matrix product is associative, we can compute multiplication in the order we prefer. Even though the result is the same, the number of operations required by it can vary a lot.

Example 7. *Let's consider three matrices: A_1 , A_2 and A_3*

- A_1 has dimensions 50×5
- A_2 has dimensions 5×100
- A_3 has dimensions 100×10

The number of operations required for their product is:

- For $(A_1 \times A_2) \times A_3$

$$50 * 100 * 5 = 25000$$

$$50 * 10 * 100 = 50000$$

- For $A_1 \times (A_2 \times A_3)$

$$5 * 10 * 100 = 5000$$

$$50 * 10 * 5 = 2500$$

So the first case requires 75000 operations, the second 7500, 10 times less.

Consider a chain of matrices $\langle A_1, \dots, A_n \rangle$ where A_i has dimension $p_{i-1} \times p_i$ for all $i \in [1, n]$. The aim is to compute a parenthesization that minimizes the number of scalar product for the chain multiplication considered. This problem is usually faced during the design part of the coding.

Chain matrix multiplication are often used in Deep neural networks. It is also useful because it gives a relevant speedup in data preparation pipeline.

3.2 A naive approach

Build up every possible parenthesization and compute the cost of the parenthesization. As we are doing it at design time, there usually are no time constraints.

- If $n = 1$, the parenthesization is obvious
- If $n > 1$, the chain can be parenthesization as

$$(A_1 \times \dots A_k) \times (A_{k+1} \times \dots A_n) \quad (3.1)$$

for any $k \in [1, n - 1]$. I can then recursively produce the parenthesization for $\langle A_1, \dots, A_k \rangle$ and $\langle A_{k+1}, \dots, A_n \rangle$

But for a sequence $1..n$ the number of parenthesization is

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & \text{if } n > 1 \end{cases} \quad (3.2)$$

Trying to give a lower bound to the second term of the previous expression

$$\text{1st term } P(1) \cdot P(n-1) \quad (3.3)$$

$$\text{Last term } P(n-1) \cdot P(n-(n-1)) \quad (3.4)$$

So we can derive that

$$P(n) \geq 2 \cdot P(1) \cdot P(n-1) = 2 \cdot P(n-1) \quad (3.5)$$

So finally

$$P(n) \in \Omega(2^n) \quad (3.6)$$

And computing all of that is basically impossible.

We notice however that:

- If the given parenthesization $(A_1 \times \dots A_k) \times (A_{k+1} \times \dots A_n)$ is the optima chain, than we may conclude that
 - The first part is optimal for $\langle A_1 \times \dots A_k \rangle$
 - The second part is optimal for $\langle A_{k+1} \times \dots A_n \rangle$
- So many branches of the naive recursive approach perform the same computation (once the parenthesization of a section is done, there is no need to re-evaluate it in a different recursion of the algorithm)

Idea:

Recursively compute optimal parenthesization and use dynamic programming.

This means that starting from the smallest parenthesization then built it up to the top, ending up with a possible solution.

Definition 16. *Dynamic programming: a strategy in which the program store the partial result in a place, and use it any time it is needed.*

3.3 A dynamic programming solution

Store the minimum number of products for all the sub-chains in m .

To do so, we first consider the subset of the sequence of matrix that goes from i to j . At this stage we are not storing the parenthesization, i.e. the position in which we are going to place the parenthesis, but just evaluating the minimal number of product required. So we recursively compute $m[i, j]$ as:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{k \in [i, j-1]} \{m[j, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases} \quad (3.7)$$

The second term is well defined, and allow a minimum, because the two references to the matrix m are references in which the distances between the two indexes are smaller than the distance between i and j . So every time I am evaluating something in the matrix $m[i, j]$, I am reducing the distance between the indexes of the other two matrices $m[i, k]$ and $m[k+1, j]$, ending up in a situation in which $i = j$. For each i, j also store in $s[i, j]$ the k that minimizes

$$m[i, k] + m[k+1, j] + p_{i-1}p_kp_j \quad (3.8)$$

i.e. the parenthesization for the current level.

Example 8. Consider $A_1(3 \times 5), A_2(5 \times 10), A_3(10 \times 52)$ and $A_4(2 \times 53)$. Each color on the matrix on the left represents different steps of the evaluation of parenthesization cost of the matrix chain. The matrix on the right is populated with the parenthesization once the minimal cost of product for a parenthesization placement is found. Both m and s can be computed iteratively from the shortest sub-chain to the longest: we can proceed from the mid diagonal to the top right corner.

Algorithm 5: Dynamic Programming Solution

```
def MatrixChain(P):  
    m ← allocate (1..n, 1..n)  
    s ← allocate (1..n-1, 2..n)  
    for i ← 1..n do  
        | m[i, i] ← 0  
    end  
    for l ← 2..n do  
        | for i ← 1..(n-l+1) do  
            | | j ← i + l - 1  
            | | MatrixChainAux(P, m, s, i, j)  
        | end  
    end  
    return (m, s)  
enddef
```

Algorithm 6: Dynamic Programming Solution

```
def MatrixChainAux(P, m, s, i, j):  
    m[i,j] ← INFINITY  
    for k ← 1..(j-1) do  
        | q ← m[i,k] + m[k+1,j] + P[i-i] * P[k] + P[j]  
        | if q < m[i,j] then  
            | | m[i,j] ← q  
            | | s[i,j] ← k  
        | end  
    end  
    return m[i,j]  
enddef
```

The computation of $m[i, j]$ takes time:

$$\sum_{k=i}^{j-1} \Theta(1) = \Theta(j - i) \quad (3.9)$$

Since $i \in [1, n]$ and $j \in [i, n]$

$$T_C(n) = \sum_{i=1}^n \sum_{j=i}^n \Theta(j - i) = \Theta\left(\sum_{i=1}^n \left(\sum_{j=i}^n j\right) - n * i\right) \quad (3.10)$$

$$= \Theta\left(\sum_{i=1}^n \frac{n * (n + 1)}{2} - \frac{i * (i + 1)}{2} - n * i\right) = \Theta(n^3) \quad (3.11)$$

Chapter 4

Heaps

Definition 17. *Heaps: Abstract data types which store totally ordered values with respect to a given total order (\preceq)*

Example 9. *If I have a bunch of data regarding students, and I order it based on the students ID, that domain can be stored into a Heap*

Heaps efficiently support the following tasks:

- Built a heap from a set of data
- Finding the minimum with respect to \preceq
- Extracting the minimum with respect to \preceq
- Decreasing the one of the values with respect to \preceq
- Inserting a new value

Example 10. *A min-heap: a heap in which the relation between elements is "less" or "equal to" (\preceq is \leq).*

A max-heap: a heap in which the relation between elements is "more" or "equal to" (\preceq is \geq).

Heaps can be used to implement priority queues.

Queues elements respects the FIFO policy: the first object inserted in the queues will be the first object to exit from the queue. Sometimes however it is useful to establish a priority in the order to extract elements from a queue. To do so, the next element to be extracted from the queue is the one that minimizes a priority criterion.

Example 11. *In emergencies, more serious patient must be served first, and their condition may change and become more and more serious over time.*

4.1 Binary Heaps

Definition 18. *Binary heaps: A nearly completed binary tree. i.e. it is complete up to the second-last level and all leaves of the last level are on the left.*

The relation $\text{parent}(p) \preceq p$ holds for any node (head property).

A possible representation for a binary heap is using an array: the first position stores the root key.

Example 12. *Let's assume that the emergency room aforementioned can contain a maximum number of people. In that case we can organize the priority queues using an array, whose size corresponds to the number of places available.*

The values stored in the array would be the keys of the tree, while its position corresponds to the node. The first position of the array is the root.

The i -th position of the array represents a node whose:

- Left child has index $2 * i$
- Right child has index $2 * i + 1$
- Parent has index $i/2$

(In pseudo code the index starts from 1)

Following, the pseudo code for some useful function:

Algorithm 7: Array-based representation: useful functions

```
def LEFT(i):  
  | return 2* i  
enddef
```

Algorithm 8: Array-based representation: useful functions

```
def GET_ROOT():  
  | return 1  
enddef
```

Algorithm 9: Array-based representation: useful functions

```
def RIGHT(i):  
  | return 2 * i + 1  
enddef
```

Algorithm 10: Array-based representation: useful functions

```
def IS_ROOT(i):  
  | return i = 1  
enddef
```

Algorithm 11: Array-based representation: useful functions

```
def PARENT(i):  
  | return floor (i/2)  
enddef
```

Algorithm 12: Array-based representation: useful functions

```
def IS_VALID_NODE(H, i):  
  | return H.size ≤ i  
enddef
```

4.2 Finding the Minimum

For minimum we mean the minimum with relation to the relation which parameterize our heap.

The minimum with relation to the \preceq is in the root of the heap. If this is not the case, the heap property did not hold.

Algorithm 13: Array-based representation: minimum

```
def HEAP_MIN(H):  
  | return H.root.key  
enddef
```

Which for array based representation became

Algorithm 14: Array-based representation: minimum (array)

```
def HEAP_MIN(H):  
    | return H[ 1]  
enddef
```

In both case the complexity is $\Theta(1)$.

4.3 Removing the minimum

When removing the minimum, we have to

- Preserve the heap topological structure of the heap itself. To do so, we have to remove a leaf on the last level, the rightmost more.
- Preserve the property. Removing the minimum means to remove the key to the root, which could create some problem.

The solution to achieve the removal is to replace the root's key with the rightmost leaf of the last level, then delete the rightmost leaf of the least-level, as its key is already stored. Doing so, the heap property may be lost (but only in one point).

To restore the property, we use a procedure called heapify. It consists in

- Find the node n , among the root and its children, whose key is minimum with relation to \preceq
- If the root's key is the minimum, we are done
- Otherwise, swap n 's and root's keys
- Repeat on the sub-tree rooted on n

Correctness of HEAPIFY

Before the iteration: the heap property holds in T_1 and T_2 .

After the iteration:

- The heap property still holds on T_2 and between a and b .
- T_1 has been messed up, but it is shorter than the original tree and all the keys on T_1 are greater than a .

We are pushing the problem constantly deeper, and at some point we end up either solving the problem, or in a leaf (where the problem is also solved).

Replacing the root's key cost $\Theta(1)$, because finding the rightmost most node is simply finding the value which is in the last position of the array in this representation. Assigning the root key to that value is again easy, we just have to assign the value of the last position of the array, to the first position.

For each iteration of HEAPIFY:

- 2 comparison to find the minimum.

- 1 swap at most.

The distance from a leaf is decreased is decreased by one at each iteration. The total cost of HEAPIFY is the height of the heap $O(\log n)$

Algorithm 15: HEAPIFY

```

def HEAPIFY(H, i):
    m ← i
    for i in [LEFT(i), RIGHT(i)] do
        if IS_VALID_NODE(H, j) and H[j] ≤ H[m] then
            m ← j
        end
    end
    if i != m then
        swap(H, i, m)
        HEAPIFY(H, m)
    end
enddef

```

Algorithm 16: Array-based representation: remove minimum

```

def REMOVE_MIN(H):
    H[1] ← H[H.size]
    H.size ← H.size-1
    HEAPIFY(H, 1)
enddef

```

4.4 Building a Heap

Building the topology is easy because it can be done automatically once we decide to go for the array implementation of the heap. Once we have the heap topology, we can try to call the functions already implemented, which should preserve the heap topology. We cannot however call the **heapify** function on the root, because it can happen that the array is not yet ready to satisfy the precondition of the function.

Instead, we can call **heapify** from the bottom-up, starting from the leafs:

- Fix the heaps rooted on the second-last level with children.
- Fix the heaps rooted on the third-last level, and so on.

Complexity of Build_Heap

Let us focus on complete binary tree. How many nodes do they have at level i of the tree?

The answer is 2^i . If n is the number of nodes in our tree, then $n = \sum_{i=0}^l 2^i = 2^{l+1} - 1$. The number of leaves in a complete binary tree is then $\lceil \frac{n}{2} \rceil$ (ceiling of the fraction). This quantity holds also for nearly complete binary tree.

Let us consider the nodes at height h , how many h -heighted nodes has a nearly complete

binary tree having n nodes?

For height h we have $\lceil \frac{n}{2^{h+1}} \rceil$ nodes. What about the complexity of build_heap?

At each height h we have $\lceil \frac{n}{2^{h+1}} \rceil$ nodes. We want to fix the heap property by calling heapify on all the nodes of the tree. So the overall complexity, a.k.a. the complexity of invoking Build_Heap on n nodes

$$\begin{aligned}
 T_{BH}(n) &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot O(h) \\
 &\leq c \cdot n \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \\
 &\leq c \cdot n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} \\
 &\leq c \cdot n \cdot \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = 2 \cdot c \cdot n
 \end{aligned}$$

So

$$T_{BH}(n) \in O(n) \tag{4.1}$$

4.5 Decreasing a Key

In a Min-heap we are talking about a decrease in the key of a node, in a Max-heap we are talking about an increase in the key of a node.

The heap property doesn't hold anymore (in relation to the parent, for both cases): it preserve the heap property only on the sub-tree rooted to the node. Heapify solve the heap property on a sub-tree rooted on the node we are dealing with.

Swapping the keys of the node and its parent solves the problem on the sub-tree rooted on the parent (it preserve the relation with the sibling node). We could end up breaking the relation between the new parent node and its parent. In this case, we just need to reiterate the same procedure, repeating until the heap property is restored.

At each iteration we have to either:

- Ends the computation in time $\Theta(1)$ or
- Pushes the problem one step closer to the root in time $\Theta(1)$

Since the heap height is $\lfloor \log_2 n \rfloor$, the complexity is $O(\log n)$

Algorithm 17: Array-based representation: decreasing a key

```

def DECREASE_KEY(H, i, value):
    if H[i]  $\preceq$  value then
        | error(value+ "is not smaller than H["+i+"]")
    end
    H[i]  $\leftarrow$  value
    while not (IS_ROOT(i) or H[PARENT(i)]  $\preceq$  H[i]) do
        | swap(H, i, PARENT(i))
        | i  $\leftarrow$  PARENT(i)
    end
enddef

```

4.6 Inserting a Value

Add the new value as the rightmost node (where we would remove a node). A new node N has to be added preserving the heap topology. We could set the key of N to the maximum value with relation to \preceq (∞ or $-\infty$). Decrease than the key of N to the desired value.

Algorithm 18: Array-based representation: inserting a new value

```

def INSERT_VALUE(H, value):
    H.size  $\leftarrow$  H.size + 1
    H[H.size]  $\leftarrow$   $\infty_{\preceq}$ 
    DECREASE_KEY(H, H.size, value)
enddef

```

Has the same complexity of DECREASE_KEY: $O(\log n)$

Chapter 5

Retrieving Data and Sorting

5.1 Retrieving Data

If we have a batch of data, a very large database, and we want to extract data from it, because we are interested in only a specific instance, how can we do it?

Let's say we have an array $A = \langle a_1, \dots, a_n \rangle$ that contains some data. Each element of the array is associated to an **identifier**, that we will indicate as: $A[i].id$ (it is unique and different for all the values stored in the array). How can we find the data associated to the identifier id_1 ?

A Naive Solution

The easiest solution is to scan all the database searching for $A[i].id = id_1$. We start from the first value, we check if its id correspond to the one we are interested in, and if it is not, we go to the next element and so on.

Its asymptotic complexity in term of big-O is $O(n)$.

Dichotomic Search

If $A = \langle a_1, \dots, a_n \rangle$ is sorted with relation to the id 's (i.e. if $i < j$, then $A[i].id \leq A[j].id$), then we can look at the element in the middle $A[n/2]$.

- If $A[n/2].id = id_1$ then we have done
- If $A[n/2].id > id_1$ then we focus on the 1st half of $A = \langle a_1, \dots, a_{n/2-1} \rangle$
- If $A[n/2].id < id_1$ then we focus on the 2nd half of $A = \langle a_{n/2+1}, \dots, a_n \rangle$

Repeat until the array is empty, or the element is discovered.

Example 13. Search for 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$

Algorithm 19: Dichotomic Search

```
def di_find(A, a):  
    (l,r) ← (1, |A|)  
    while r ≥ l do  
        m ← (l+r)/2  
        if A[m] = a then  
            return m  
        end  
        if A[m] > a then  
            r ← m-1  
        end  
        else  
            l ← m + 1  
        end  
    end  
    return 0  
enddef
```

At each iteration, $l - r$ is halved. So, if $|A| = 2^m$, `di_find` ends after m iteration at most (we for sure reduce the array to one single element).

The while-block takes time $\Theta(1)$.

The `di_find`'s complexity is $O(\log n)$.

However, let's remember that the dichotomic approach only works on sorted data. So the problem of sorting becomes of great importance.

5.2 Sorting

Input: An array A of numbers.

Output: The array A sorted i.e. if $i < j$, then $A[i] \leq A[j]$

Example 14.

A naive solution would be checking all possible permutations of the elements of the array, with complexity $\Omega(n!)$. There exist however many strategies to solve this very same problem.

5.3 Insertion Sort

We assume that an initial part of the array is already sorted. We want to increase the number of sorted elements in the array, and to do so we pick the next value after the already sorted part of the array, and put it in the right position. It is not so easy to do it in an array, because it is an immutable object. We can implement the insertion by swapping the considered value with the one in the previous in the array, and continue to

do so until it reach the right position (comparing the value at each step).

Algorithm 20: Insertion Sort

```
def insertion_sort(A):
    for i in 2...|A| do
        j ← i
        while (j>1 and A[j] < A[j - 1]) do
            swap(A,j-1,j)
            j ← j-1
        end
    end
enddef
```

The while-loop block costs $\Theta(1)$, swapping constant, and the following operation, takes constant time. The while-loop however is repeated a number of time up to i . The maximum number of iteration is when the while stop at its first condition, while the minimum is when the value is already in its correct position. The algorithm iterates $O(i)$ and $\Omega(1)$ times for all $i \in [2, n]$, the complexity is:

- The upper bound: $\sum_{i=2}^n O(i) * O(1) = O\left(\sum_{i=2}^n i\right) = O(n^2)$
- The lower bound: $\sum_{i=2}^n \Omega(1) * \Omega(1) = \Omega\left(\sum_{i=2}^n 1\right) = \Omega(n)$

5.4 Quick Sort

Quick Sort aim to improve the asymptotic complexity of the Insertion sort. It exploit the same approach of the dichotomic search. It

- Select one element of the A : the pivot (randomly).
- Split the array in two part based on the pivot (partition):
 - Sub-array S of the values smaller or equal to the pivot.
 - The pivot itself.
 - Sub-array G of the values greater than the pivot.
- Repeat in the sub-arrays having more than 1 elements (recursively).
- At the end of the iterations above:
 - The values in S stay in S even after sorting A
 - The values in G stay in G even after sorting A
 - The pivot is in its "sorted" position
 - S and G are shorter than A

An iteration places at least one element in the correct position. It prepares A for two recursive calls on S and G .

Algorithm 21: Quick Sort

```
def QUICKSORT( $A, l = 1, r = |A|$ ):
    if  $l < r$  then
         $p \leftarrow \text{PARTITION}(A, l, r, l)$ 
        QUICKSORT( $A, l, p-1$ )
        QUICKSORT( $A, p+1, r$ )
    end
enddef
```

The last recursion call is a tail recursion (which can be substituted with a loop) An alternative version with less recursion is (with the same asymptotic complexity):

Algorithm 22: Quick Sort

```
def QUICKSORT( $A, l = 1, r = |A|$ ):
    while  $l < r$  do
         $p \leftarrow \text{PARTITION}(A, l, r, l)$ 
        QUICKSORT( $A, l, p-1$ )
         $l \leftarrow p+1$ 
    end
enddef
```

From the execution time point of view, if it is possible to avoid recursion, it is better to do so.

The time complexity T_Q of quick sort will be

$$T_Q(|A|) = \begin{cases} \Theta(1) & \text{if } |A| = 1 \\ T_Q(|S|) + T_Q(|G|) + T_P(|A|) & \text{otherwise} \end{cases} \quad (5.1)$$

Where T_P is the complexity of the partition.

Partition

An In-Place Algorithm to partition the array A is:

- Switch the pivot p and the first element in A , then (in a closing maneuver) :
 - If $A[i] > p$, swap $A[i]$ and $A[j]$ and decrease j
 - Else ($A[i] \leq p$), increase i by 1
- Repeat until $i \leq j$
- Swap p and $A[j]$

The lower bound on the Partition, as it has to pass through all the elements of the array, is $\Omega(n)$. The general complexity is $\Omega(|A|)$ (or of the slice of the array we are considering).

Algorithm 23: Partition

def PARTITION(A, i, j, p):

```
    swap( $A, i, p$ )
    ( $p, i$ )  $\leftarrow$  ( $i, i+1$ )
    while  $i \leq j$  do
        if  $A[i] > A[p]$  then
            swap( $A, i, j$ )
             $j \leftarrow j-1$ 
        end
        else
             $i \leftarrow i+1$ 
        end
    end
    swap( $A, p, j$ )
    return  $j$ 
```

enddef

Complexity

General Case:

$$T_Q(|A|) = T_Q(|S|) + T_Q(|G|) + \Omega(|A|) \quad (5.2)$$

But the complexity of the first two part depends on the size of S and G .

The worst case scenario is the one in which the two sub-arrays are unbalanced. After the partition S or G doesn't contain any value.

In that case $|G| = 0$ or $|S| = 0$ for all recursive call, so

$$\begin{aligned} T_Q(n) &= T_Q(n-1) + \Omega(n) \\ &= \sum_{i=0}^n \Omega(i) = \Omega\left(\sum_{i=0}^n i\right) \\ &= \Omega(n^2) \end{aligned}$$

Which is a telescopic equation.

The best case scenario is instead the one of a balanced partition a.k.a. the content of S and G preserve a given ration. To compute the complexity, we may use a recursion tree, in which every node, which represent a recursive call, specifying the cost of the call, taking into account the split ratio of the array. Each branch has different length, the smallest has length $\log_{ratio_1} n$ and the longest $\log_{ratio_2} n$ (with $ratio_2 > ratio_1$ and $ratio_1 + ratio_2 = 1$).

Every level until the length of the smallest branch has cost cn , while all the subsequent one has a decreasing cost $\leq cn$.

Example 15. *Recursive tree with fixed ratio:*

We want to sum up the cost of all levels.

The total is $\Theta(n \log n)$.

Finally, the average case depend on the ordering of A . If all the permutation are equally likely, the partition has the ration more balanced than $1/d$ with probability

$$\frac{d-1}{d+1} \tag{5.3}$$

However, if "good" and "bad" partition alternate, then we can conclude that the one consider is also a good scenario, because the average is $\Theta(n \log n)$

5.5 Finding the Maximum

In consist in:

- Find the maximum
- Move the maximum at the end of the array, then:
- If $|A| > 1$, repeat on the initial fragment of A

Example 16.

But how can we find the maximum? And how do we move it to the end of the array?

The complexity is $\sum_{i=1}^{|A|} (T_{\max}(i) + \Theta(1))$, and depends heavily on the complexity required by the algorithm which find the maximum, which by the way depend on the length of the array it is considering at the moment, which decrease by one at each iteration of the algorithm (that's why it T_{\max} is summed over i).

Bubble Sort

The first solution to the finding-the-maximum problem is the bubble sort algorithm. It work by pair-wise swapping the maximum to the right: it find the maximum by comparing each value with the one on its right, and swap the pair if the value on the right is smaller. Every iteration, bubble sort bring the highest value to the end of the array.

Algorithm 24: Bubble sort

```
def BUBBLE_SORT(A):
    for i in |A|..2 do
        for j in 1..i-1 do
            if A[j] > A[j + 1] then
                swap(A, j, j+1)
            end
        end
    end
enddef
```

One swap-block costs $\Theta(1)$, so the nested for-loop costs $\Theta(i)$.

So the total cost of this operation would be:

$$\begin{aligned} T_B(n) &= \sum_{i=2}^n \Theta(i) * \Theta(1) \\ &= \Theta\left(\sum_{i=2}^n i\right) = \Theta(n^2) \end{aligned}$$

Best and worst case scenario are the same, and require the same time-complexity.

Selection Sort

The second solution to the finding-the-maximum problem is the selection sort algorithm. It work by linear scanning the unsorted part: it find the maximum among all the values by doing a linear search (the naive algorithm shown at the beginning of the chapter), and perform one single swap.

Algorithm 25: Bubble sort

```
def SELECTION_SORT(A):
    for i in |A|..2 do
        max_j ← 1
        for j in 2..i do
            if A[j] > A[max_j] then
                max_j ← j
            end
        end
        swap(A, i, j+1)
    end
enddef
```

One if-block costs $\Theta(1)$, so the nested for-loop costs $\Theta(i)$.
So the total cost of this operation would be:

$$\begin{aligned} T_S(n) &= \Theta(1) \sum_{i=2}^n \Theta(i) * \Theta(1) \\ &= \Theta\left(1 + \sum_{i=2}^n i\right) = \Theta(n^2) \end{aligned}$$

Best and worst case scenario are the same, and require the same time-complexity. We decreased the number of swap necessary in a constant way, so overall selection sort has the same complexity of the bubble sort.

Heap Sort

The second solution to the finding-the-maximum problem is the heap sort algorithm. It works by using the max-heap H_{\max} :

- Store the elements of A in H_{\max} : building up an heap which stores all the values of the array (which has linear complexity).
- Extract the min(i.e. the maximum) and place it in A . It takes time constant respect to the heap itself.
- Repeat from 2 until H_{\max} is not empty.

The array-based representation of heaps is an in-place algorithm.

Algorithm 26: Heap sort

```

def HEAP_SORT(A):
    H ← BUILD_MAX_HEAP(A) // the root is the max
    for i in |A|..2 do
        A[i] ← EXTRACT_MIN(H)
    end
enddef

```

Let's now look at the heap sort complexity.

BUILD_MAX_HEAP procedure costs $\Theta(n)$. EXTRACT_MIN costs $O(\log i)$ per iteration, in which i is the number of elements in the heap (decreasing every time by 1). The total is

$$\begin{aligned}
 T_H(n) &= \Theta(n) + \sum_{i=2}^n O(\log i) \\
 &\leq O(n) + O\left(\sum_{i=2}^n \log n\right) = O(n \log n)
 \end{aligned}$$

This is an upper-bound, for the worst case scenario. The actual number of iteration is different for every case.

The overall complexity of heap sort is $O(n \log n)$.

Heap sort is overall faster than quick sort, because in its worst scenario quick sort takes a quadratic time.

5.6 Sorting in Linear Time

Is it possible to improve again the efficiency of sorting?

To find the lower bound, we need to consider all the possible computation made up by an algorithm. For this reason, the execution of a sorting-by-comparison algorithm can be modeled as a decision-tree model: given an algorithm and an array, a decision tree represent all the possible computation of the algorithm over that array. Any comparison between a and b elements of the array corresponds to a node, which branches the computation according to whether $a \leq b$ or $a > b$

Example 17.

In this way we can model every possible algorithm, dealing with every possible array. Since every leaves represent a permutation of the original array, the number of leaves in

a decision tree is always factorial with respect to the number of elements in the array. The height h is the maximum number of comparisons required by the algorithm. Since a binary tree has no more than 2^h leaves,

$$h \geq \log_2(n!) \in \Omega(n \log n) \quad (5.4)$$

The lower bound for comparison-based sorting is then $\Omega(n \log n)$. For this reason, there is no general algorithm to sort in linear time by using comparisons.

However, we can introduce some minor ad-hoc assumptions that allows us to lift this bound. Those are:

- Bounded domain for the array values.
- Uniform distribution of the array values.

Counting Sort

Counting Sort is an algorithm which doesn't use any comparison, but is nevertheless able to sort values in the interval $[1, k]$. The idea is

- Count the occurrences of A 's values and place them in C . This means: take the unsorted array A , and count the number of repetition of values in each array by counting an auxiliary matrix C , whose length is the difference between the first (min) value of the array A , and the last (max) value of the array A . Each of the indexes in C represent a value in A . To populate C we can just do a linear scan of the array A , updating every time we find a new value.
- Sums the values in C and get the number of elements smaller or equal to the C 's indexes. This means we are performing a pulling operation: accumulate in each entries of C the number of values smaller then the indexes. In this way, for each value in the array A , the we have the number of values in A which are smaller or equal to the value itself. From the update C we know the correct position of the value in the array.
- Use C to place the elements of A in the correct positions in B . To avoid change in position due to swapping, we start from the end of the array A , building a third array B (to maintain the relative order between equal values) . Also, every time we place a value of A in B , we decrease by 1 the corresponding value in the updated C .

Example 18.

Counting sort is not an in-place algorithm, the auxiliary space necessary is not constant, and we end up with a new array separated from the original A . Stability is the preservation of relative order of equivalent elements. We place the value backwardly to preserve it.

Generalizing the algorithm to deal with any interval $[k_1, k_2]$ domain is easy.

Algorithm 27: Counting sort

```

def COUNTING_SORT( $A, B, k$ ):
     $C \leftarrow \text{ALLOCATE\_ARRAY}(k, \text{default\_value} = 0)$ 
    for  $i \leftarrow 1$  upto  $|A|$  do
         $C[A[i]] \leftarrow C[A[i]] + 1$ 
    end
    //  $C[j]$  is now the number of  $j$  in  $A$ 
    for  $j \leftarrow 2$  upto  $|C|$  do
         $C[j] \leftarrow C[j-1] + C[j]$ 
    end
    //  $C[j]$  is now the number of  $j$  in  $A$ 's value  $\leq j$ 
    for  $i \leftarrow |A|$  downto 1 do
         $B[C[A[i]]] \leftarrow A[i]$ 
         $C[A[i]] \leftarrow C[A[i]] - 1$ 
    end
enddef

```

Let's consider the complexity of the algorithm:

- Allocating C and setting all its element to 0: $\Theta(k)$
- Counting the instances of A 's values: $\Theta(n)$
- Setting in $C[j]$ the number of A 's values $\leq j$: $\Theta(k)$
- Copying A 's values into B by using C : $\Theta(n)$

So the total is: $\Theta(n + k)$

Radix Sort

It perform the sorting of numbers which have the same number of digits. An array A of d -digit values can be sorted digit-by-digit:

- For each digit i from the rightmost down to the leftmost.
- Use a stable algorithm and sort A according to the digit i .

The way which you sort the number digit-by-digit is stable, i.e. assuming that the relative order of the equal values are not sorted during the sort.

This is the ideal case to use counting sort. If therefore the digit sorting is in $\Theta(|A| + k)$, radix sort takes time:

$$\Theta(d(|A| + k)) \tag{5.5}$$

where d is the number of digits in each of A 's values.

Bucket Sort

If we are able to provide the uniformity assumption to the distribution of the values in the array, i.e. the probability of getting a number is always the same, we can use the Bucket sort. The idea is to:

- Split $[0, 1]$ in n buckets: $\left[\frac{i-1}{n}, \frac{i}{n}\right)$ for $i \in [1, n]$. This means that we create an array of buckets which contains all the values of the given array in a given interval. We divide the original interval in some sub-interval, and place every value in the corresponding bucket.
- Add each value of A to the correct bucket.
- Sort the bucket: we have to sort all elements in the same buckets, but as we have assumed an uniform distribution, the probability of having more than one value in each bucket is very low.
- Reverse the content of buckets in bucket order on A .

The complexity is:

- Allocating an initializing B : $\Theta(n)$
- Filling the buckets: $\Theta(n)$
- Sorting each bucket(expected): $O(n)$
- Reversing buckets content into A : $\Theta(n)$

Total expected complexity: $O(n)$

5.7 Select

Let A be an array which may be unsorted. How could I find the value that, if A was sorted, would be in position:

- 1 i.e. the minimum value in A . We could scan all the array and search for the minimum. This requires to scan all the elements of A , so its minimum complexity is $\Theta(n)$.
- n i.e. the length of the array itself. In this case it means to search the maximum value, so as before the complexity is $\Theta(n)$.
- $i \in [1, n]$ i.e. any other value in the array. A straight forward strategy would be to sort the array A itself, and return the value in position i . Its complexity is then $O(n \log n)$.

Is it possible to deliver an algorithm with a better complexity?

Definition 19. *The Select Problem:*

Input: a potentially unsorted array A and an index $i \in [1, |A|]$

Output: the value $\overline{A}[i]$ where \overline{A} is the sorted version of A .

Which can be reformulated in a different, more easy to handle form:

Definition 20. *The Select Problem:*

Input: a potentially unsorted array A and an index $i \in [1, |A|]$

Output: an index j s.t. $\tilde{A}[j] = \overline{A}[i]$ where \tilde{A} and \overline{A} are A after the computation and the sorted version of A , respectively.

We do not want to build an index: we are considering an *ad-hoc* query (i.e. one specific position in A , not the case of answering the selection problem many consecutive times.)

We will also not assume that A does not contain multiple instances of the same value (this is not necessary, but simplifies things when dealing with the complexity of the algorithm).

Possible strategies

An approach could be using a **PARTITION** approach, inherited from the dichotomic approach. Let's say we want to find the value stored in position i , the algorithm consists in:

- Select a pivot $A[j]$
- Compute $k \leftarrow \text{PARTITION}(A, 1, |A|, j)$, and get S and G
- Compare i and k :
 - If $i = k$, then return k
 - If $i < k$, then $\overline{A}[i]$ is in S
 - If $i > k$, then $\overline{A}[i]$ is in G

Here anyway we are facing the same problem already seen in **QUICKSORT**: the pivot choice makes the difference. What if we end up in a situation in which the pivot ends up with a partition which is always the first or last element in the array?

In this case the complexity of the algorithm is $\Theta(n^2)$, that is not what we want to do. The best pivot is the one that is median in \overline{A} , but in our case A may be unsorted. But constant ratio partitions are **QUICKSORT**'s best scenarios as well. We want an alternative to the median of \overline{A} which guarantees us a constant ratio between the partition of A . We can find an almost-median in this way (getting the median of the medians):

- Split A in $\lceil n/5 \rceil$ chunks $C_1, \dots, C_{\lceil n/5 \rceil}$ each of size 5. We know that we can sort each of these arrays in constant time, since the length is constant.
- Find the median m_i of C_i , e.g., by sorting C_i itself. Sorting all of them takes a time linear to the number of chunks, which is linear to the length of the array. But once we have sorted each of the chunks, we can find the median of the chunk in a constant time. This operation is done in time $\Theta(n)$.
- Recursively compute the median m of the m_i 's. Let's assume we have now all the medians in an array, we want to find the median of them. We have decreased the size of the array to $\lceil n/5 \rceil$. We can apply recursion.

An algorithm which exploits this trick has a complexity which is below $n \log(n)$?
The worst case scenario is the one in which the median of the median is not a good enough pivot for PARTITION. Let's think the chunks as they were the columns of a matrix.
We can then sort the chunks according to the median. This representation let us to count how many values end up in G and in S .

- The number of chunks are of course $\lceil \frac{n}{5} \rceil$.
- How many medians are greater than the median of the medians? $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$
- How many chunks at least have 3 elements greater than m ? $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2$
- How many elements at least are greater than m ? $3 \left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 \right)$
- This is the number of values that for sure are greater than the median of the median:
 m ? $3 \left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 \right) \geq \frac{3n}{10} - 6$

The same calculations can be carried out for S . An upper bound for the number of elements smaller or equal to m (the worst case scenario) is: $3n - \left(\frac{3n}{10} - 6 \right) = \frac{7n}{10} + 6$.
Let's compute the complexity to get the pivot for the partition (the median of the median):

$$T_s(n) = \Theta(n) + T_s\left(\lceil \frac{n}{5} \rceil\right) + \Theta(n) + T_s\left(\frac{7n}{10} + 6\right) \quad (5.6)$$

The last two terms are the part account for the partition + recursion. We can solve $T_s(n)$ by using the Substitution method:

- We guess that $T_s(n) \in O(n)$
- Replace all the asymptotic classes by representative. So

$$T_s(n) = T_s\left(\lceil \frac{n}{5} \rceil\right) + T_s\left(\frac{7n}{10} + 6\right) + c' \cdot n \quad (5.7)$$

- Prove by induction:
 - Assume that $\exists c > 0$ such that

$$\forall m < n \quad T_s(m) \leq c \cdot \left\lceil \frac{m}{5} \right\rceil + c \cdot \left(\frac{7}{10}m + 6 \right) + c'm \quad (5.8)$$

- Need to prove that holds for m

$$\begin{aligned} T_s(n) &\leq T_s\left(\lceil \frac{m}{5} \rceil\right) + T_s\left(\frac{7}{10}n + 6\right) + c'm \\ &\text{if } n > \frac{7}{10}n + 6 \\ &\leq c \cdot \lceil \frac{n}{5} \rceil + c \cdot \left(\frac{7}{10}n + 6 \right) + c'n \\ &\leq c \cdot (n/5 + 1) + c \cdot \frac{7}{10}n + (c \cdot 7 + c'n) \\ &\leq \frac{9}{10} \cdot c \cdot n + c'n + 7 \cdot c \end{aligned}$$

So at the end

$$\left(\frac{9}{10} + c'\right) \cdot n + 7c \leq c \cdot n \tag{5.9}$$

And this holds for $c \geq 20c'$ and $n \geq 140$. So $T_s(n) \leq cn$

Chapter 6

Binary Search Trees and Red-Black Trees

6.1 Motivation

Example 19. *Let us consider the registry office, for which for every newborn we want to record a set of data (e.g. name, birthday, parents, etc.). The registry change quite often. What if we want to perform a search based on one of this data?*

If we use the array based scenario, but every time we update the data we have to build up a new array, and erase the precedent one. For searching we can:

- Scan all the data-set at each query, this takes time $\Theta(n)$
- Sort the data-set by the chosen data (e.g. birthday, in the example) at each insertion. This takes time $\Theta(n)$ per insertion (using the Radix Sort, because the birthday has a fixed number of digits) + $\Theta(\log n)$ (using a dichotomic search) per query. But what if we are interested in the order of a different data (e.g. search by name)?
- Sort the data-set by the chosen data (e.g. birthday, in the example) at each query. This takes time $\Theta(n + \log n) = \Theta(n)$ per query (using the Radix Sort and dichotomic search)

So there each strategies always lands on a complexity of approximately $\Theta(n)$. For this kind of registry, the array is not the best data structure. What we are looking for is a data structure that provides:

- Adding new data
- Searching data
- Removing data

We are aiming to build an index i.e. an auxiliary data structure to **efficiently** perform those operations.

6.2 Binary Search Trees

We already encountered trees

Definition 21. *Binary Search Trees (BST) are an abstract data type (ADT). The following notation holds*

- x is a node
- $x.left$ and $x.right$ are a node x left and right child respectively
- $x.parent$ is the parent of the node x
- $x.key$ is the value stored in x , i.e. its key
- $x.left = NIL$, then x misses the left child (i.e. the reference is not present)
- $T.root$ is the root of the tree T and $T.root.parent = NIL$ (i.e. the root has no parent, its reference is missing)

Utility function

Some useful functions with complexity $\Theta(1)$

Algorithm 28: BST: IS_ROOT

```
def IS_ROOT(x):  
    | return x.parent=NIL  
enddef
```

Algorithm 29: BST: IS_RIGHT_CHILD

```
def IS_RIGHT_CHILD(x):  
    | return  $\neg$  IS_ROOT and x.parent.right=x  
enddef
```

Algorithm 30: BST: SIBLING

```
def SIBLING(x):  
    | if IS_RIGHT_CHILD(x) then  
    | | return x.parent.left  
    | end  
    | return x.parent.right  
enddef
```

Algorithm 31: BST: CHILDHOOD_SIDE

```
def CHILDHOOD_SIDE(x): // get x's side w.r.t its parent  
    | if IS_RIGHT_CHILD(x) then  
    | | return RIGHT  
    | end  
    | return LEFT  
enddef
```

Algorithm 32: BST: REVERSE_SIDE

```
def REVERSE_SIDE(side): // reverse the side  
    | if side=LEFT then  
    | | return RIGHT  
    | end  
    | return LEFT  
enddef
```

Algorithm 33: BST: GET_CHILD

```
def GET_CHILD(x, side): // get x's child on side  
    | if side=LEFT then  
    | | return x.left  
    | end  
    | return x.right  
enddef
```

Algorithm 34: BST: SET_CHILD

```
def SET_CHILD(x, side, y): // set x's child
    if side=LEFT then
        | return x.left ← y
    end
    else
        | return x.right ← y
    end
    if y≠NIL then
        | y.parent ← x
    end
enddef
```

Algorithm 35: BST: UNCLE

```
def UNCLE(x): // get x's uncle
    | return SIBLING(x.parent)
enddef
```

Algorithm 36: BST: UNCLE

```
def GRANDPARENT(x): // get x's granpa
    | return x.parent.parent
enddef
```

Algorithm 37: BST: NEW_NODE

```
def NEW_NODE(v): // build a new node
    x.key ← v
    x.parent ← NIL
    x.right ← NIL
    x.left ← NIL
enddef
```

A BST is a tree s.t.:

- All the keys belong to a totally ordered set with relation to \preceq .
- If x_l is the left sub-tree of x , then $x_l.key \preceq x.key$: all the nodes on the left sub-tree of a node should be smaller or equal respect to the total order to the node itself .
- If x_r is the right sub-tree of x , then $x.key \preceq x_r.key$: all the nodes on the right sub-tree of a node should be greater or equal respect to the total order to the node itself.

Binary search tree are used to represent an index, so we usually avoid to have copies of the same key. To have multiple object to the same key, the best thing is to attach a list to every key with the corresponding object.

Printing, Finding Min/Max

How can we print in order all the nodes of a BST? Using a recursive algorithm: for every node, all keys which are smaller or equal then the key of the node considered at the moment belongs to the left subtree, so we first need to visit the left subtree, until you find the smaller. This goes on going right until all nodes are printed.

Algorithm 38: BST: INORDER.WALK:INORDER.WALK_AUX

```
def INORDER_WALK_AUX(x):  
    if x ≠ NIL then  
        INORDER_WALK_AUX(x.left)  
        print x.key  
        INORDER_WALK_AUX(x.right)  
    end  
enddef
```

Algorithm 39: BST: INORDER.WALK:INORDER.WALK

```
def INORDER_WALK(T):  
    INORDER_WALK_AUX(T.root)  
enddef
```

Due to the BST property:

- The minimum key is contained by the first node on the leftmost branch that has not a left child (we always have 2 child nodes on the tree, so we just has to follow, starting from the root, the left branch until no node is found).
- The maximum key is contained by the first node on the rightmost branch that has not a right child (we always have 2 child nodes on the tree, so we just has to follow, starting from the root, the right branch until no node is found).

Algorithm 40: BST: MINIMUM.IN.SUBTREE

```
def MINIMUM_IN_SUBTREE(x):  
    while x.left ≠ NIL do  
        x ← x.left  
    end  
    return x  
enddef
```

Algorithm 41: BST: MAXIMUM.IN.SUBTREE

```
def MINIMUM_IN_SUBTREE(x):  
    while x.right ≠ NIL do  
        x ← x.right  
    end  
    return x  
enddef
```

The complexity depends on the length of the tree itself: for both algorithms, the maximum number of steps we need to perform is equal to the length of the longest branch on the tree itself. So it is $O(h_T)$.

Find the successor of a node

We want find the node whose key is the next key in the total order respect to the key we are considering at the moment. Due to the BST property, the successor with relation to \preceq of node n is either

- The node containing the minimum in the right sub-tree of n .
- The nearest "right-ancestor" (i.e. go up until we end up with a root whose previous node was a right child of the node we are dealing with) of n , if n has not right child.

Algorithm 42: BST: SUCCESSOR

```
def SUCCESSOR(x):
    if x.right ≠ NIL then
        | return MINIMUM_IN_SUBTREE(x.right)
    end
    y ← x.parent
    while y ≠ NIL and IS_RIGHT_CHILD(x) do
        | x ← y
        | y ← x.parent
    end
    return y
enddef
```

The complexity is $O(h_T)$, for the worst case scenario, where we follow all the path up to the root.

Searching for a value

To search for a value v , we apply a dichotomic approach from the root x

- If n is *NIL* or $x.key = v$, return n : if the root has the value we are looking for, we are good.
- If $x.key \preceq v$, search on the right sub-tree
- If $x.key \not\preceq v$, search on the left sub-tree

Algorithm 43: BST: SEARCH_SUBTREE

def SEARCH_SUBTREE(x, v):

```
    while  $x \neq \text{NIL}$  do
        if  $x.\text{key} \preceq v$  then
            if  $v \preceq x.\text{key}$  then
                return  $x$ 
            end
             $x \leftarrow x.\text{right}$ 
        end
        else
             $x \leftarrow x.\text{left}$ 
        end
    end
```

enddef

Remark. The strategy is the same of the dichotomic search.

As before, the worst scenario is the one in which we follow the route of the longest branch. Each iteration performs $\Theta(1)$ operations. The number of iterations depends on the height h_T of T and on v , so the algorithm takes time $O(h_T)$.

What is the relations between the number of nodes in tree and its height? We first need to see how to insert a value in a BST.

Insert a value

Browse a branch of T and add a new leaf having v as a key. Once we discover the the corresponding node is absent, we insert the node with that key value.

Algorithm 44: BST: INSERT_BST

```
def INSERT_BST(T, v): // v is the new value
    x ← T.root
    y ← NIL // y is x's parent
    // search the right place for z
    while x ≠ NIL do
        y ← x
        if x.key ≤ v then
            if v ≤ x.key then
                return HANDLE_MULTIINSERT(x, v)
            end
            x ← x.right
        end
        else
            x ← x.left
        end
    end
    // attaching the new node
    x ← NEW_NODE(v)
    if T.root ≠ NIL then
        if v ≤ y.key then
            SET_CHILD(y, LEFT, x)
        end
        else
            SET_CHILD(y, RIGHT, x)
        end
    end
    else
        T.root ← x
    end
    return x
enddef
```

As before, the complexity is $O(h_T)$.

Removing a Key

Search the node x containing the key. We end up in two cases:

- x has one child at most. In this case we erase the node itself and replace with its only child. This operation is named transplant. The cost of this operation is constant, we simply need to re-assign the parent of the only child of the node, and

the child of the parent node.

- x has two children. In this case we can search for the successor of this node (the minimum on the right subtree). It surely does not have a left child. So we can substitute the value of the key of the new found node to the one we want to erase, and this preserve the order of the BST. We need then to erase the redundant node, and we can do it by using the transplant function.

Algorithm 45: BST: TRANSPLANT

```
def TRANSPLANT(T, x, y):// replace x by y
    if IS_ROOT(x) then
        | T.root ← y
    end
    if y ≠ NIL then
        | y.parent ← NIL
        | // else x has no parent
    end
    else
        | x_side ← CHILDHOOD_SIDE(x)
        | // attach y in place of x
        | SET_CHILD(x.parent, x_side, y)
    end
enddef
```

Algorithm 46: BST: REMOVE_BST

```
def REMOVE_BST(T, x):// remove x.key from T and return a removed
node
    if x.left = NIL // if x lacks of left child then
        | TRANSPLANT(T, x, x.right)
        | return x
    end
    if x.right = NIL // if x lacks of right child then
        | TRANSPLANT(T, x, x.left)
        | return x
    end
    y ← MINIMUM_IN_SUBTREE(x.right)
    x.key ← y.key
    return REMOVE_BST(T, y) // if y lacks of left child
enddef
```

As before, the complexity of removing a node x takes time $O(h_T)$.

So, again, what is the relations between the number of nodes in tree and its height? h_T may be equal to the number n of nodes (e.g. keeping inserting the maximum). So BSTs costs more than single-linked list (insertion $\Theta(1)$).

Are we able to provide some guidelines able to reduce the height of a binary search tree? Or better, are we able to balance the height of a binary search tree in a way in which

the ration between the longest and shortest branch in the tree is always below a given threshold?

Yes, with the use of Red-black trees.

6.3 Red-Black trees

Red-Black trees (RBTs) are a datatype meant to produce a dynamic index for a set of data, built as the BST, but balanced: the ratio between the longest and shortest branch in the tree is upper-bounded by 2.

Definition 22. *RBTs are BSTs satisfying the following conditions:*

- *Each node is either a RED or BLACK node.*
- *The tree's root is BLACK.*
- *All the leaves are BLACK NIL nodes.*
- *All the RED nodes must have BLACK children.*
- *For each node x , all the branches from x (down to a leaf) contain the same number of black nodes.*

Definition 23. *Black height ($BH(x)$): the number of BLACK nodes below x (down to a leaf) in any branch. Thanks to the last condition on RBTs, we do not need to specify anything on the branch.*

To have a totally balanced tree, we have to deal with a complete tree (at most 2^{h_T} nodes). But we also have to deal with the case in which the tree is not complete: the maximum length of a branch is always constrained from above by some linear bound on the logarithm of the number of the nodes. Let's compute the bound on the height of RBT:

- $n \geq 2^{BH(x)} - 1$: if we have a RBT rooted on the node x , and we consider exclusively the internal nodes (all but the leaves), the number of nodes n contained in the tree must be $n \geq 2^{BH(x)} - 1$. Let's consider the case in which we count all the nodes but the RED ones. Since all branches has the same BH, we end up with a complete tree, whose height is the one to the root. The height of the all-black tree is $2^{BH(x)} - 1$.
- $h(x) \leq 2 \cdot BH(x)$: this is caused by the definition. i.e. every time we end up on a red node, it has to have black children, but it might be the case we do not have any red tree.

So we deduce that

Theorem 2. *Any RBT with n internal nodes has height at most $2 \log_2(n + 1)$:*

$$\begin{aligned}
 n &\geq 2^{\frac{h(x)}{2}} - 1 \\
 \Leftrightarrow \log_2(n + 1) &\geq \frac{h(x)}{2} \\
 \Leftrightarrow h(x) &\leq 2 \cdot \log_2(n + 1)
 \end{aligned}$$

The height of any node $h(x) \in O(\log(n))$, because in the worst case scenario the ration between the shortest and longest branch is 2 (because BH is always equal).

Function to get the color of a node

Algorithm 47: RBT: COLOR

```
def COLOR(x)
    if x=NIL then
        return BLACK
    end
    return x.color
enddef
```

We want to insert nodes in the RBT preserving the properties of the tree.

We needs some useful operations which are meant to re-balance the tree after the insertion.

Definition 24. *Rotations: change the length of branches while preserving the BST properties.*

e.g. The cost of the rotation operation is always the same and take constant time $\Theta(1)$.

Algorithm 48: RBT: ROTATE

```
def ROTATE(T, x, side)
    r_side ← REVERSE_SIDE(side)
    y ← GET_CHILD(x, r_side)
    TRANSPLANT(T, x, y)
    beta ← GET_CHILD(y, side)
    TRANSPLANT(T, beta, x)
    SET_CHILD(x, r_side, beta)
enddef
```

Insertion requires:

- Inserting as in BST (cost depends on the height of the tree: logarithmic for a RBT).
- RED-coloring the node: we cannot insert directly black nodes because it would change the BH. But that could put a RED node as a child of another RED node, so we need to..
- Fixing up RB-Tree properties:
 - Case 1: x 's uncle (y) is RED, so we RED-color x 's granpa and BLACK-color x 's parent and y .
 - Case 2: x 's uncle (y) is BLACK and y and x are on the same side. In this case we rotate x 's parent on the opposite side. The new x is the former x 's parent.
 - Case 3: x 's uncle (y) is BLACK and y and x are on different side. In this case we invert x 's parent and grandparent colors and rotate on x 's granpa on y 's side.

All this case takes constant time. For the total complexity, let's consider all the possible situations:

- If x parent is BLACK: done.
- If x has no uncle, either x or $x.parent$ are the root: BLACK-color it.
- If x has an uncle, we can always choose between Cases 1,2, or 3.
 - Case 1 either removes the problem or pushes it towards the root (so we are upper bounding the cases in which we can apply Case 1).
 - Case 2 brings to Case 3.
 - Case 3 solves the problem

So in the worst case scenario the algorithm keeps repeating Case 1 steps along the insertion branch and the complexity is $O(\log n)$.

Algorithm 49: RBT: INSERT_RBTREE

```

def INSERT_RBTREE(T, v)
  x ← INSERT_BST(T, v)
  x.color ← RED
  FIX_INSERT_RBTREE(T, x)
enddef

```

Algorithm 50: RBT: FIX_INSERT_RBT_CASE1

```

def FIX_INSERT_RBT_CASE1(T, x)
  UNCLE(x).color ← BLACK
  x.parent.color ← BLACK
  GRANDPARENT(x).color ← RED
  return GRANDPARENT(x)
enddef

```

Algorithm 51: RBT: FIX_INSERT_RBT_CASE2

```

def FIX_INSERT_RBT_CASE2(T, x)
  x_side ← BCHILDHOOD_SIDE(x)
  p ← x.parent
  ROTATE(T, p, REVERSE_SIDE(x_side))
  return p
enddef

```

Algorithm 52: RBT: FIX_INSERT_RBT_CASE3

```

def FIX_INSERT_RBT_CASE3(T, x)
  g ← GRANDPARENT(x)
  x.parent.color ← COLOR
  g.color ← RED
  x_side ← CHILDHOOD_SIDE(x)
  ROTATE(T, g, REVERSE_SIDE(x_side))
enddef

```

Algorithm 53: RBT: FIX_INSERT_RBTREE

```
def FIX_INSERT_RBTREE(T, x)
  while  $\neg$ IS_ROOT(x) and ( $\neg$ IS_ROOT(x.parent) or x.parent.color=RED) do
    if COLOR(UNCLE(x))=RED then
      |  $z \leftarrow$  FIX_INSERT_RBT_CASE1(T,x)
    end
    else
      | if CHILDHOOD_SIDE(x)  $\neq$  CHILDHOOD_SIDE(x.parent) then
      | |  $z \leftarrow$  FIX_INSERT_RBT_CASE2(T,x)
      | end
      | FIX_INSERT_RBT_CASE3(T,x)
    end
  end
  T.root.color  $\leftarrow$  BLACK
enddef
```

Removing a key

Removing a key as in BST remove also a node y which is replaced by its former child x

- If y was RED, the RBT properties are preserved: not breaking any rules concerning both red and black key.
- If y was BLACK, the branches through x lost 1 BLACK node: $BH(x) = BH(w) - 1$ where w is the sibling of y before the removal. That is a problem, because we now have two siblings with different heights.

There are different cases:

- Case 0: x is RED. We just need to color x BLACK. $BH(x)$ is increased by 1 and the tree has been fixed.
- Case 1: x 's sibling is RED.
 - Invert colors in x 's parent and sibling.
 - Rotate x 's parent on x 's side.

$BH(x)$ does not change. By applying this transformation we moved the problem towards the leafs. We increased the distance from the root down to the root we are considering (the opposite of the insertion).

- Case 2: x 's sibling and nephews are BLACK. In this case we RED-color x 's sibling. $BH(x)$ does not change, while the BLACK height of both x 's parent and sibling are decreased by 1.
- Case 3: among x 's sibling and nephews, only the nephew on x 's side is RED. In this case:
 - Rotate x 's sibling on the opposite side with relation to x .

- Invert colors in both old and new siblings of x .

The BLACK height of both x and x 's parent does not change.

- Case 4: The x 's nephew on the opposite side with relation to x is RED. In this case:
 - Switch colors of x 's parent and sibling.
 - BLACK-color the x 's nephew on the opposite side with relation to x .
 - Rotate x 's parent on x 's side.

The BLACK height of both x and x 's parent does not change. In this case, we solved the problem.

All the case transformation procedure takes time $\Theta(1)$. All the transformation based on switching colors, applying a rotation, which takes constant times, are obviously constant. Times for different cases are:

- Both Case 0 and Case 4 transformation procedures fix the RBT.
- Case 1 cannot occur twice in-row: once we apply case 1, we are guaranteed that the sibling of the node we are interested in is BLACK. After it, we could apply either case 2, case 3 or case 4.
- Case 2 we move toward the root. We may end up again in case 1, but for sure after it we end up in a situation in which we can apply case 3.
- Worst case scenario: we keep applying case 2 without solving the issue until we push the problem to the root ($O(\log n)$ times).

This data-structure may be used to represent set. If we want to represent set of integers, the best way to do it is to use a RBT (depending on the application), because it is efficient both from the computational and space memory point of view.

Chapter 7

Graphs

7.1 Basics

Graph is an unique way to represent a lot of different things. We already defined them in section 1.4.

There are two ways to represent a graph:

- Adjacency list (usually for sparse graph, i.e. the number of edges is at most linear with the number of nodes). For each of the node in my data structure we have an index in our array which store the list of its neighbors. What if we want to know if there is an edge between two nodes? We can go to see the list of nodes reachable using one single edge from the one we are interested in, and look for our ending point node.

To detect if an edge is present in my graph, we are going to spend a time linear to the number of nodes in the graph (from the definition of sparse graph itself).

- Adjacency matrix (usually for dense graphs): a square matrix in which each row and each column represent the sources and the destination of edges. It is however less memory efficient.