

OMAR CHEHAIMI

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Notes based on the Udacity online course
January 2, 2021

NO COPYRIGHT

©© This book is released into the public domain using the CCo code. To the extent possible under law, I waive all copyright and related or neighbouring rights to this work.

To view a copy of the CCo code, visit:

<http://creativecommons.org/publicdomain/zero/1.0/>

COLOPHON

This document was written with L^AT_EX using ArsClassica, a reworking of the ClassicThesis style designed by André Miede, inspired to the masterpiece *The Elements of Typographic Style* by Robert Bringhurst.

Felix qui potuit rerum cognoscere causas.

– Publius Vergilius Maro

PREFACE

The following notes are based on the Udacity online course **Intro to Data Structures and Algorithms in Python**. These notes are not intended to be a book, nor a formal introduction to data structures and algorithms, they are just my personal attempt to have a small, yet comprehensive notes about the huge world on this topic.

Omar Chehaimi

CONTACTS

✉ omar.chehaimi@outlook.it

CONTENTS

Preface	iv
1 PRELIMINARIES CONCEPTS	1
1.1 Computational Complexity	1
1.2 Big O Notation	1
1.2.1 Properties	2
1.3 Time complexity evaluation example	3
1.4 Recursion	3
1.4.1 Python Implementation Examples	3
2 DATA STRUCTURES	5
2.1 Collection	5
2.2 List	5
2.3 Array	5
2.4 Linked List	6
2.4.1 Linked List Implementation	8
2.5 Stack	9
2.5.1 Stack Implementation	10
2.6 Queue	11
2.6.1 Queue Implementation	12
2.7 Set	12
2.8 Map	12
2.9 Hash Table	13
2.9.1 Hashing	13
2.10 String Keys	14
2.10.1 String Keys Implementation	15
3 SEARCHING AND SORTING	16
3.1 Binary Search	16
3.1.1 Efficiency of the Binary Search Algorithm	17
3.1.2 Binary Search Implementation	18
3.2 Bubble Sort	18
3.2.1 Efficiency of the Bubble Sort Algorithm	19
3.2.2 Bubble Sort Implementation	20
3.3 Merge Sort	20
3.3.1 Efficiency of the Merge Sort Algorithm	21
3.3.2 Merge Sort Implementation	22
3.4 Quicksort	23
3.4.1 Efficiency of the Quicksort Algorithm	24
3.4.2 Quicksort Implementation	25
4 TREES	28
4.1 General Definitions	28
4.2 Tree Traversal	29
4.2.1 Depth-first search	30
4.3 Binary Trees	31
4.3.1 Search	31
4.3.2 Delete	32

4.3.3	Insert	32
4.3.4	Perfect binary tree	33
4.3.5	Binary Tree Implementation	33
4.4	Binary Search Trees (BST)	34
4.4.1	Binary Search Tree Implementation	36
4.5	Heaps	37
4.5.1	Heapify	37
4.5.2	Heap Implementation	38
4.6	Self-balancing Binary Search Trees	38
4.7	Red-Black Trees	39
5	GRAPHS	42
5.1	General Definitions	42
5.1.1	Connectivity	42
5.2	Graph Representations	43
5.2.1	Edge List	43
5.2.2	Adjacency List	44
5.2.3	Adjacency Matrix	44
5.3	Graph Traversal	45
5.3.1	Depth-first Search (DFS)	45
5.3.2	Breadth-first Search (BFS)	45

Appendix

A	BINARY SEARCH COMPLETE IMPLEMENTATION	47
A.1	Pre-order Traversal	47
A.2	In-order Traversal	50
A.3	Post-order Traversal	52
B	COMPLEXITY OF DATA STRUCTURES FOR GRAPHS	56
C	IMPLEMENTATION OF GRAPH REPRESENTATION	57
	Bibliography	59

LIST OF FIGURES

Figure 1	Plots of the main functions and their evaluation for computational complexity.	2
Figure 2	Pseudocode of a recursive function and its internal execution.	3
Figure 3	Factorial recursive execution.	4
Figure 4	An example of an array with elements and indexes.	6
Figure 5	Removing or adding an element from an array and the indexes update.	6
Figure 6	An example of a linked list with the data and the reference to the next element.	7
Figure 7	Adding a new element to a linked list.	7
Figure 8	Removing an element to a linked list.	8
Figure 9	Doubly linked list.	8
Figure 10	In a stack only the element at the top is modified.	10
Figure 11	Allowed operations on queue elements.	11
Figure 12	In a deque the operations can be done on both head and tail.	11
Figure 13	A priority queue.	12
Figure 14	An example of a map.	13
Figure 15	An example of collision and a possible way to solve this issue by using the bucket method.	14
Figure 16	An array with numeric values ordered in an ascending order.	16
Figure 17	Binary search algorithms steps.	17
Figure 18	Array splitting in the implementation of the binary search algorithm.	18
Figure 19	Bubble sort algorithm.	19
Figure 20	The swapping process is repeated until the array is completely ordered.	19
Figure 21	Merge sort algorithm.	20
Figure 22	Merge sort algorithm implementation.	23
Figure 23	Quicksort algorithm part one.	23
Figure 24	Quicksort algorithm steps part two.	24
Figure 25	Quicksort algorithm worst case.	25
Figure 26	Quicksort algorithm best and average case.	25
Figure 27	Quicksort algorithm implementation.	27
Figure 28	Elements of a tree and linked list.	28
Figure 29	Possible structures of a tree.	28
Figure 30	A tree and its fundamentals elements.	29
Figure 31	The depth-first search and the breadth-first search.	29
Figure 32	Pre-order search.	30
Figure 33	In-order search.	30
Figure 34	Post-order search.	31

Figure 35	Example of tree search and traversal.	31
Figure 36	Delete an element of a tree.	32
Figure 37	Add an element of a tree.	33
Figure 38	Perfect binary tree results.	33
Figure 39	Search on an ordered binary tree.	35
Figure 40	Addition on an ordered binary tree.	35
Figure 41	An unbalanced binary tree.	36
Figure 42	Max heap (a) and min heap (b) example.	37
Figure 43	Add a new node to a heap.	38
Figure 44	Implementation example of a heap using an array.	38
Figure 45	Example of balanced (b) and unbalanced (a) tree.	39
Figure 46	A red-black tree.	40
Figure 47	Insertion of a new node and the following updating of the color nodes.	40
Figure 48	Left and right rotation for balancing the tree after an insertion.	41
Figure 49	Undirected (a) and directed (b) graphs and its elements.	42
Figure 50	A weakly connected graph.	43
Figure 51	Edge list.	44
Figure 52	Adjacency list.	44
Figure 53	Adjacency matrix.	44
Figure 54	Pre-order iterative implementation example.	49
Figure 55	In-order iterative implementation example.	52
Figure 56	Post-order iterative implementation example.	55

LIST OF TABLES

Table 1	Binary Search Efficiency.	17
Table 2	Merge Sort Efficiency.	21

LISTINGS

1.1	Sum of all elements of a matrix.	3
1.2	Implementation of the Fibonacci series with both iterative and recursive way.	4
1.3	Implementation of calculating the factorial of a number using the recursive way.	4
2.1	Linked List implementation.	8
2.2	Stack implementation.	10

2.3	Queue implementation.	12
2.4	String key implementation.	15
3.1	Binary search python implementation.	18
3.2	Bubble Sort python implementation.	20
3.3	Merge Sort python implementation.	22
3.4	Quicksort python implementation.	26
4.1	Class definition for a node and a tree.	33
4.2	Recursive pre-order traversal and search implementation. . . .	34
4.3	implementation of insert and search operation for a binary search tree.	36
A.1	Class definition for a node and a tree.	47
A.2	Recursive and iterative implementation of pre-order traversal. .	48
A.3	Recursive and iterative implementation of in-order traversal. .	50
A.4	Recursive and iterative implementation of post-order traversal.	53
C.1	Graph representation and fundamental operations.	57

ACRONYMS

ADT	Abstract data type
LIFO	Last In, First Out
FIFO	First In, First Out
DFS	Depth-first search
BFS	Breadth-first search
BST	Binary Search Trees

1

PRELIMINARIES CONCEPTS

1.1 COMPUTATIONAL COMPLEXITY

The **computational complexity**, or **complexity**, of an algorithm is the amount of resources (time and memory) needed for solving it, and it is the minimum complexity of all possible implementations for solving that given algorithm, included also the unknown ones [1] ([Computational complexity, Wikipedia](#)).

The amount of needed resources for solving an algorithm varies with the size of the input n . The computational complexity in general is a function $n \rightarrow f(n)$, and it represents the worst case complexity or the average complexity over all the inputs of size n .

When the kind of the complexity is not explicitly indicated, generally is meant to be the **time complexity**, which is different from computer to computer, and it is generally expressed as the number of elementary operations required to solve a given algorithm. It is assumed that these elementary operations take the same time for being solved. The computational complexity can be related also to the memory consumption.

1.2 BIG O NOTATION

The **big O notation** [2] ([Big O Notation, Wikipedia](#)) describe the behavior of a function when its argument tends to infinity or to a particular value.

Definition 1.1

Let f be a real or complex value function and let g be a real function. Let both functions be defined on the same positive and real unbounded interval, and let $g(x)$ be strictly positive for all large enough x values: $g(x) > 0, \forall x$ large enough. Thus: $f(x) = O(g(x))$ as $x \rightarrow \infty$ if $\exists M \in \mathbb{R}, M > 0$ and $x_0 \in \mathbb{R}$ such that $|f(x)| \leq Mg(x) \forall x \geq x_0$.

Usually $f(x) = O(g(x))$ is used as $x \rightarrow \infty$, but it can be also defined for the case $x \rightarrow a$, where a is a real number.

O notation is asymptotic for big x , so the important terms are the ones which grow faster than the others, which become irrelevant.

Example 1.1

In $f(x) = 6x^4 - 2x^3 + 5$ as $x \rightarrow \infty$, $6x^4$ is the fastest growing term. 6 is a constant and can be omitted, thus $f(x) = O(x^4)$.

1.2.1 Properties

Here are listed some simple properties about Big O notation.

Definition 1.2 Product-Sum-Multiplication by a constant

Product

$$f_1 = O(g_1) \text{ and } f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$$

Sum

$$f_1 = O(g_1) \text{ and } f_2 = O(g_2) \Rightarrow f_1 f_2 = O(\max(f_1, f_2))$$

Multiplication by a constant

Let k be a nonzero constant, then: $O(k|g) = O(g)$, $f = O(g) \Rightarrow kf = O(g)$

Definition 1.3 Logarithm and Exponential

Let c be a nonzero constant, then: $O(\log n^c) = O(\log n)$, because: $(\log n^c = c \log n)$.

$O(n^c)$ and $O(c^n)$ are very different, if $c > 1$ the latter grows much faster.

Let c be a nonzero constant. $(cn)^2 = c^2 n^2 = O(n^2)$, but 2^n and 3^n are not of the same order. In general $2^{cn} = (2^c)^n$ is not of the same order of 2^n .

Example 1.2

$$f = 9 \log n + 5(\log n)^4 + 3n^2 + 2n^2 = O(n^3) \text{ as } n \rightarrow \infty$$

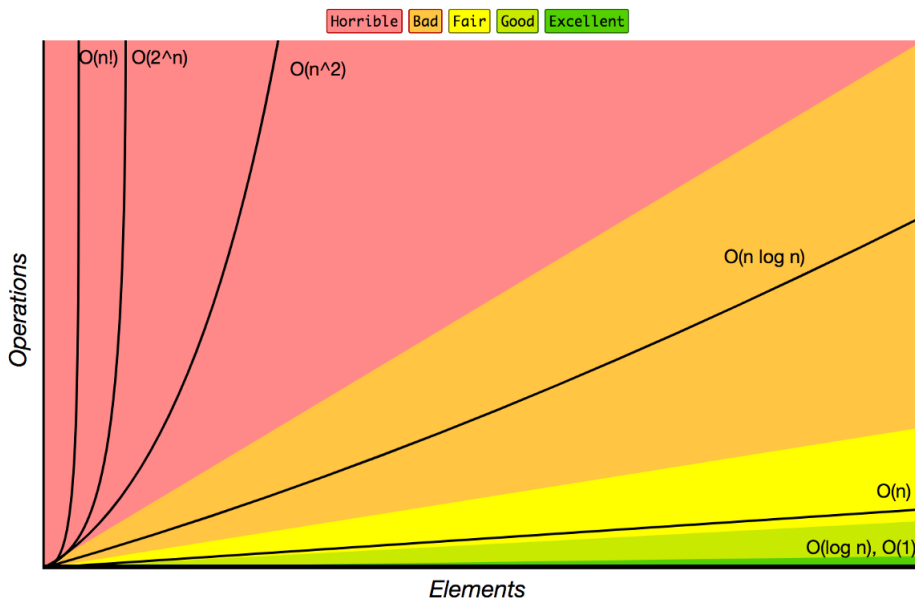


Figure 1: Plots of the main functions and their evaluation for computational complexity. Credits: bigocheatsheet.com.

1.3 TIME COMPLEXITY EVALUATION EXAMPLE

Let us suppose we want to calculate the sum of all elements of a $n \times n$ matrix. For evaluating the time complexity of this function we have to identify all the elementary operations, and evaluating their complexity based on how many times they are repeated. Here is the pseudocode for the function that calculate the sum of a matrix.

Listing 1.1: Sum of all elements of a matrix.

```

1 def find_sum_2d(array_2d):
2     totoal = 0 # -> O(1)
3     for each row in array_2d: # -> repeated n times
4         for each column in array_2d: # -> repeated n times
5             total += array_2d[column][row] # -> O(1)
6     return total # -> O(1)

```

The total time complexity is:

$$T = O(1) + n^2 O(1) + O(1) = O(n^2)$$

Where $O(1)$ is a constant value.

1.4 RECURSION

In recursion a function calls itself again on a smaller size input, until the exit condition stops this self calling (recursive calling) [3] ([Recursion, Wikipedia](#)). There are three fundamentals elements in a recursive function:

- 1 A function that calls itself.
- 2 Exit condition. Without this condition a recursive function would call itself forever without an end.
- 3 Input alteration. When the function is called again the input is changed to a smaller dimension than the previous call.

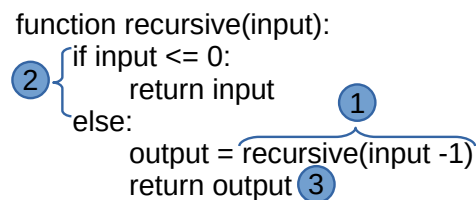


Figure 2: Pseudocode of a recursive function and its internal execution.

In recursion the exit condition is fundamental, because if it contains some errors the recursion will never end, resulting in an infinite process.

1.4.1 Python Implementation Examples

Listing 1.2: Implementation of the Fibonacci series with both iterative and recursive way.

```

1 def fibonacci_iterative(position):
2     if position == 0:
3         return 0
4     elif position == 1:
5         return 1
6     else:
7         first = 0
8         second = 1
9         next_value = first + second
10        for i in range(2, position):
11            first = second
12            second = next
13            next_value = first + second
14        return next_value
15
16 def fibonacci_recursive(position):
17     if position <= 1: # Exit condition
18         return position
19     return fibonacci(position - 1) + fibonacci(position - 2) # Calling
    the function on a smaller size input

```

Listing 1.3: Implementation of calculating the factorial of a number using the recursive way.

```

1 def factorial(n):
2     if n <= 1: # Exit condition
3         return n
4     else:
5         return n*factorial(n - 1) # Calling the function on a
    smaller input

```

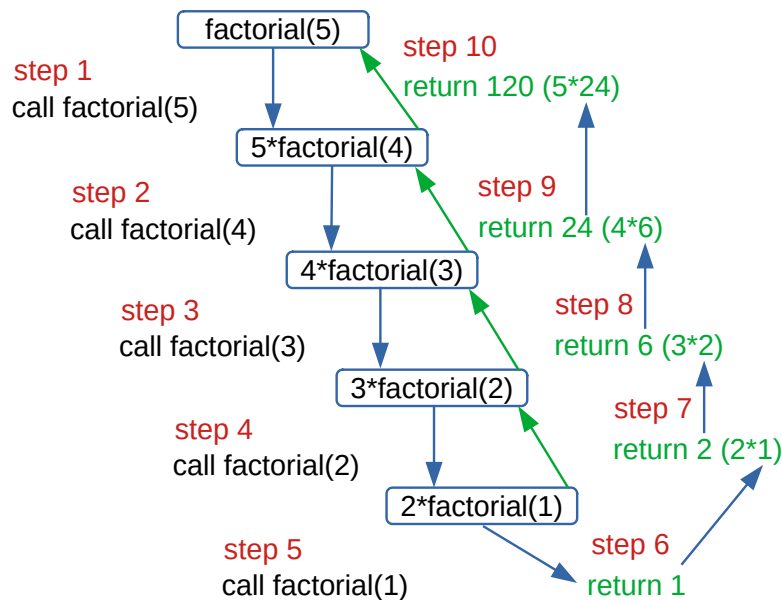


Figure 3: Factorial recursive execution.

2 | DATA STRUCTURES

A **data structure** is a data organization, management, and storage format that enables efficient access and modification of the collected data. The relationship among collected data, their properties, the operations that can be done on them, are all properties of the data structure [4] ([Data Structure, Wikipedia](#)). Data structure is the basis for an **abstract data type (ADT)**, a mathematical model for a **data type** [5] ([Data Type, Wikipedia](#)), which is defined by its behavior from the point of view of a user, its type of data, specifically in terms of possible values, by possible operations on these data, and by the behavior of these operations. This mathematical model contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not of a user [6] ([Abstract Data Type, Wikipedia](#)).

In this chapter the most important data structures like **collections**, **lists**, **arrays**, **linked lists**, **stacks**, and **queues** are introduced. For each data structure are shown all the most important properties, operations, and implementations.

2.1 COLLECTION

A **collection** is an object that groups several different elements in only single unit. Collections are used to save, to find, to manipulate, and to communicate grouped data [7] ([Collection \(abstract data type\), Wikipedia](#)). Usually the elements belonging to a collection are of the same type, such as a poker hand (a collection of playing cards), a folder containing emails (a collection of emails), or a phone book (a map $name \rightarrow phone\ number$).

2.2 LIST

A **list** is a collection which represents a set of **ordered** elements, which can be also of different type. Same value elements can be repeated several times. Lists do not have a fixed size, and it is possible to add, to remove, and to modify all the elements in the list. The complexity of adding or removing an element is constant ($O(1)$).

2.3 ARRAY

An **array** is a collection of elements of the same or also different type, in which each of them is identified with at least one **array index** or **key** [8]

([Array data structure, Wikipedia](#)). In some programming languages arrays have a fixed size, in others instead is fixed.



Figure 4: An example of an array with elements and indexes.

For doing any allowed operation to an element of an array it is enough to know its index.

Adding or removing an element of an array could be a very expensive operation. This is because when a change take place to an element all the following indexes must be updated (Figure 5). The worst case complexity is $O(n)$, where n is the size of the array.

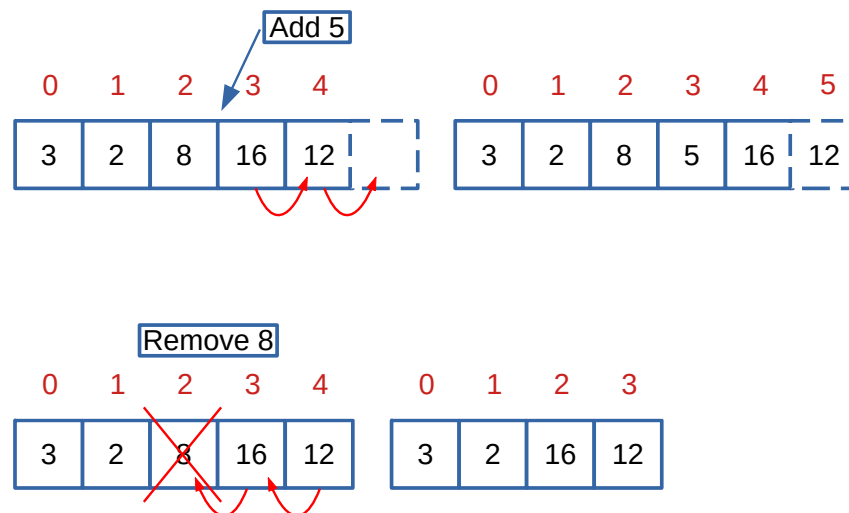


Figure 5: Removing or adding an element from an array and the indexes update.

2.4 LINKED LIST

A **linked list** is a linear collection in which each element has the data and a reference to the next element (a link) [9] ([Linked List, Wikipedia](#)).

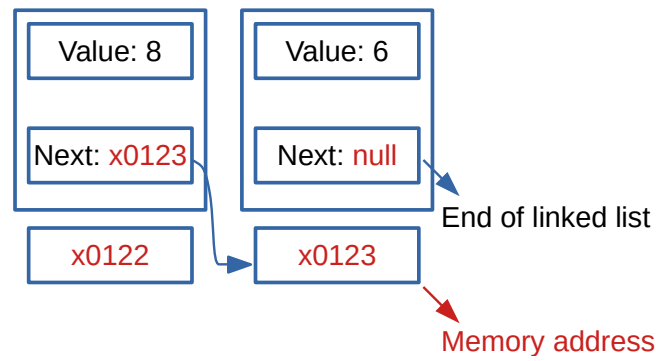


Figure 6: An example of a linked list with the data and the reference to the next element.

In a linked list the order of the elements is not assured. Operations like adding or removing an element in a linked list are very efficient, because it is enough to change the references of the elements involved in the operation. For example for adding a new element is enough to change the previous element reference to the just added element, and change the reference of the new element to the next element (Figure 7). In case of removing an element it is enough to update the previous element's reference to the next element of the removed one (Figure 8).

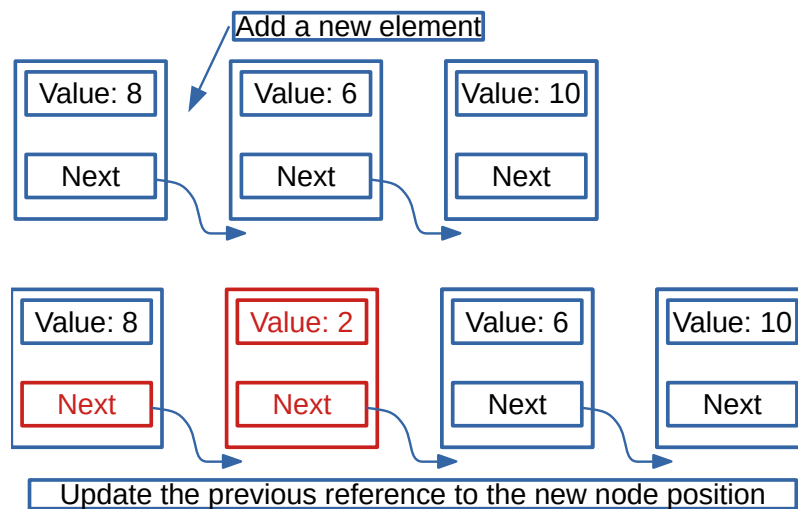


Figure 7: Adding a new element to the linked list. As explained before this operation is very efficient because it is enough to change the references of the new element and the previous element to the one just added.

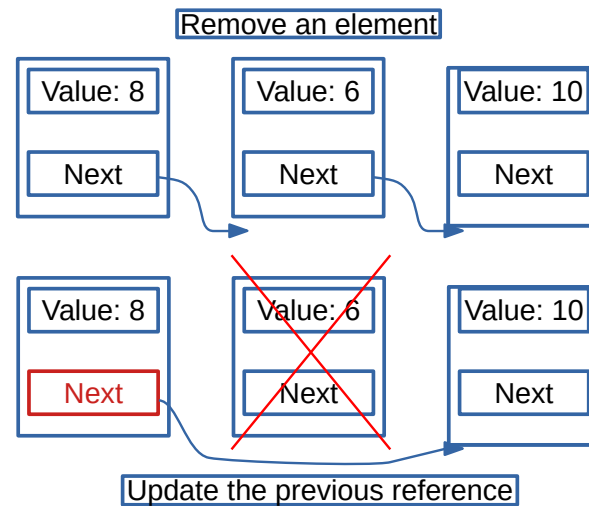


Figure 8: Removing an element to a linked list.

The complexity for adding or removing an element is constant $O(1)$. Linked lists can be also **doubly linked list** (Figure 9), which every element has the reference to the previous and next element of the collection [10] ([Doubly linked list, Wikipedia](#)).

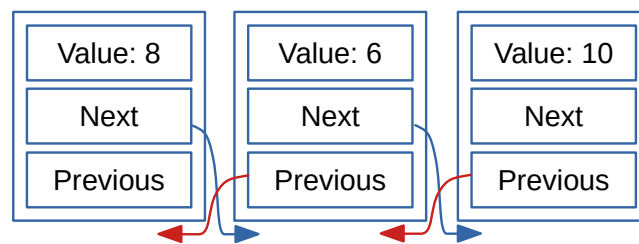


Figure 9: Doubly linked list.

Linked list are used for implementing other data structures such as lists, stacks, queues, and associative array.

2.4.1 Linked List Implementation

The following code is the implementation of a linked list in Python.

Listing 2.1: Linked List implementation.

```

1 class Element():
2
3     def __init__(self, value):
4         self.value = value
5         self.next = None
6
7 class LinkedList():
8
9     def __init__(self, head=None):
10         self.head = head
11

```

```

12     def append(self, new_element):
13         current = self.head
14         if self.head:
15             while current.next:
16                 current = current.next
17             current.next = new_element
18         else:
19             self.head = new_element
20
21     def get_position(self, position):
22         counter = 0
23         current = self.head
24         if position < 1:
25             return None
26         while current and counter <= position:
27             if counter == position:
28                 return current
29             current = current.next
30             counter += 1
31         return None
32
33     def insert(self, new_element, position):
34         counter = 1
35         current = self.head
36         if position > 1:
37             while current and counter < position:
38                 if counter == position - 1:
39                     new_element.next = current.next
40                     current.next = new_element
41                     current = current.next
42                     counter +=1
43         elif position == 1:
44             new_element.next = self.head
45             self.head = new_element

```

2.5 STACK

A **stack** is abstract data type belonging to collections, in which only two operations are allowed [11] ([Stack, Wikipedia](#)):

- **Push:** add an element at the top of the stack
- **Pop:** remove the newest element of the stack

In the stacks only the element at the top of the stack can be modified. The complexity remains constant for adding and removing operations. The stacks are also called **Last In, First Out (LIFO)**.

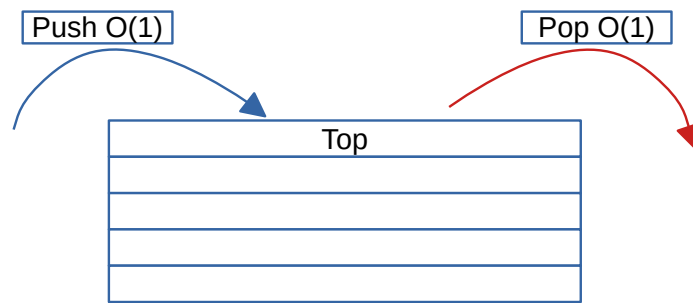


Figure 10: In a stack only the element at the top is modified.

2.5.1 Stack Implementation

The following is the implementation in Python of a stack using the linked list.

Listing 2.2: Stack implementation.

```

1 class Element():
2     ...
3
4 class LinkedList():
5     ...
6
7     def insert_first(self, new_element):
8         new_element.next = self.head
9         self.head = new_element
10
11     def delete_first(self):
12         if self.head:
13             delete_element = self.head
14             temp = delete_element.next
15             self.head = temp
16             return delete_element
17         else:
18             return None
19
20 class Stack():
21
22     def __init__(self, top=None):
23         self.ll = LinkedList(top)
24
25     def push(self, new_element):
26         self.ll.insert_first(new_element)
27
28     def pop(self):
29         self.ll.delete_first()

```

2.6 QUEUE

A **queue** is abstract data type belonging to collections very similar to the stacks. For a queue the admitted operations are only on the oldest element, thus the first element to be added [12] ([Queue, Wikipedia](#)). The allowed operations on a queue are:

- **Enqueue:** adding an element at the bottom of the queue
- **Dequeue:** removing the element at the head of the queue
- **Pick:** observing the element at the head of the queue

For adding and removing operations the complexity remain constant. Queues are also called **First In, First Out (FIFO)**.

The most efficient way to implement a queue is using linked lists.

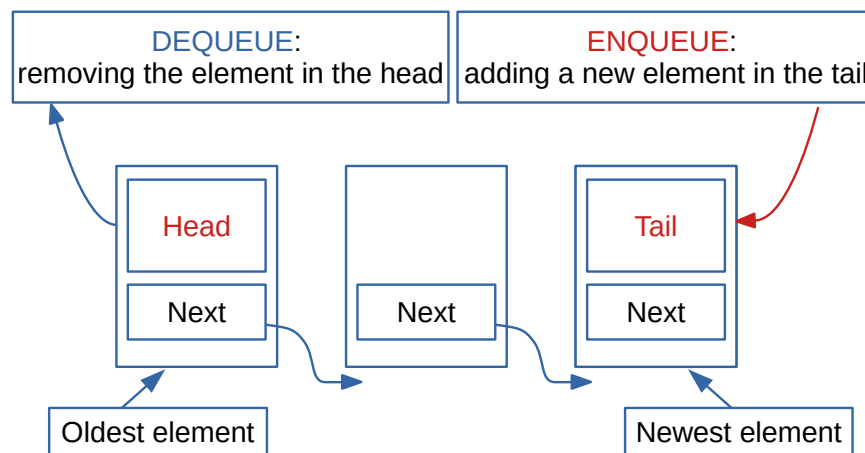


Figure 11: In a queue only the first element to be added can be removed or where a new element can be added.

A generalization of queues and linked lists are **dequeues**, in which is possible to perform deque and enqueue operations on both the **head** and **tail**.

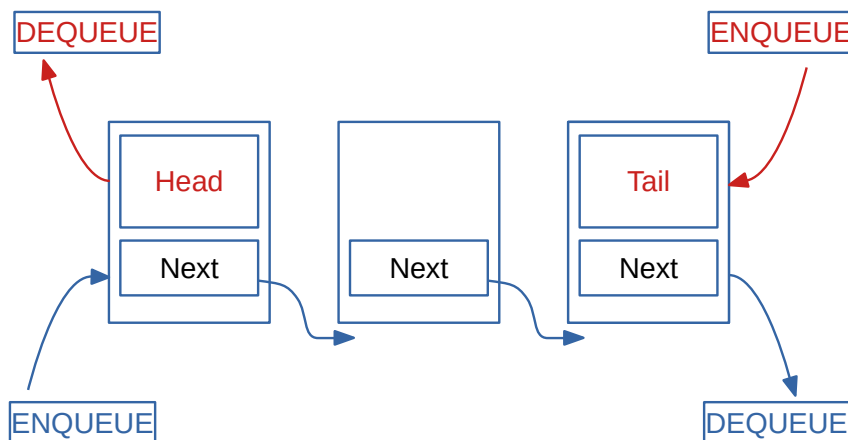


Figure 12: In a deque the operations can be done on both head and tail.

Another modification of queue are **priority queue**, in which each element has a priority (a numeric value that indicates its importance). When an element of the queue is removed (deque), the element to be removed is the one which has the highest priority. In case two or more elements have the same priority the oldest element is removed.

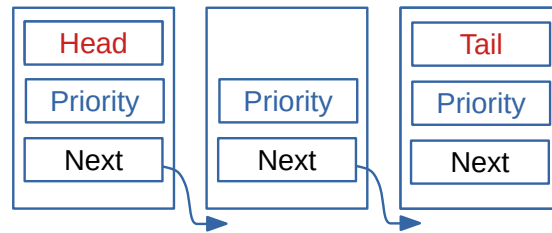


Figure 13: A priority queue.

2.6.1 Queue Implementation

The following is the implementation of a queue using the list of Python.

Listing 2.3: Queue implementation.

```

1 class Queue():
2     def __init__(self, head=None):
3         self.storage = [head]
4
5     def enqueue(self, new_element):
6         self.storage.append(new_element)
7
8     def peek(self):
9         return self.storage[0]
10
11    def dequeue(self):
12        return self.storage.pop(0)

```

2.7 SET

A **set** is an abstract data type in which the unique values are stored without a particular order [13] ([Set](#), [Wikipedia](#)).

2.8 MAP

A **map**, **associative array**, **symbol table**, or **dictionary** is an abstract data type composed of a collection of (key, value) pairs [14] ([Hash Map](#), [Wikipedia](#)). The group of the key is a set, in which each element has a unique value. Map is a very useful data type, used in a lot of situations.

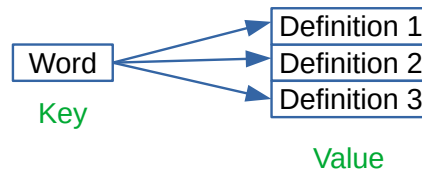


Figure 14: An example of a map.

2.9 HASH TABLE

A **hash table** or **hash map** is a data structure that implements the map abstract data type [15] ([Hash Table, Wikipedia](#)). Hash table uses the **hash function** for working [16] ([Hash Function, Wikipedia](#)). Using the hash function is called hashing.

2.9.1 Hashing

Hashing is an operation that allows to transform the value of a variable to another one which is much easier to find within a collection. If we want to find a value inside a collection (list, stack, queue, etc) the time requested for the search is linear with the size of the collection. In fact we have to check all the elements until the searched one is found (in case of stack and queue this is not true is only the last or the first element respectively is checked). For solving this problem and thus to find an element in a constant time hash function are used.

Let us consider the follow example, where we have an array in which we would like to store big random numbers. A simple way for hashing these numbers is to consider only the last numbers (56 and 17) and divide them for a fixed number. The reminder of the division is used as the new index of the array associated to the number (Figure).

What happen when the hash function transforms two different number in the same? In this case we have a **collision**. In case of a collision there are several strategies for solving this issue: one way could be to change the the hash function, another one could be to use an array for storing different values associated to the same key (**bucket**). For the last occurrence the worst case complexity for a search is $O(n)$ (Figure 15).

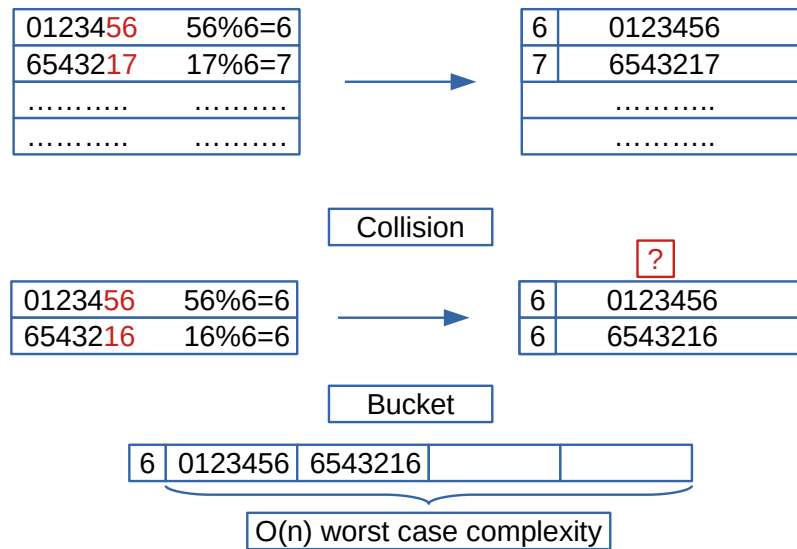


Figure 15: An example of collision and a possible way to solve this issue by using the bucket method.

It does not exist a perfect hash function and a trade off must always be reached. A way to analyze a hash function is by using the **load factor** defined as follow: $\text{Load Factor} = \# \text{ of entries} / \# \text{ of buckets}$. For example, if we want to save 10 values using 1000 buckets the Load Factor is equal to 0.01, and most of the buckets is empty. In this case is convenient to change the hash function by using less buckets. More the Load Factor is closed to 0 more the hash table will be empty (**sparse**), and more the Load Factor is closed to 1 more the has table will be full and efficient. In case the Load Factor is bigger than 1 there is the certainty there will happen some collisions. Another example: let us consider a hash function that divide the numbers by 100. If we consider 100 values all multiple of 5, the Load Factor will be: $\text{Load Factor} = \# \text{ of entries} / \# \text{ of buckets} = 100/100 = 1$. But this way is very slow and a different configuration should be used to make this more efficient. For example if we use more bucket, for example 107, we still avoid collisions and we do not have the hash map too much sparse.

2.10 STRING KEYS

Let us consider a hash function that associates a word to a numerical value. For example we can use the ASCII character encoding of the first two letters of the string as numerical value. Thus, in case of *UDACIY* we have $U=85$ and $D=68$. For hash function we can use the following: $S[0] * 31^{(n-1)} + S[1] * 31^{(n-2)} + \dots + S[n-1]$, where n is the length of the string. In this way for the word in the previous example, in which only the first two letters are encoded in a numerical value for simplicity, we have $85 * 31^1 + 68 = 2703$.

This hash function works very well because it assures a very low probability of collision. 31 is a number empirically obtained by several researches and showed good results in hashing strings.

2.10.1 String Keys Implementation

The following is the implementation of the string key map in Python.

The hash function of this implementation is: Hash Function = (ASCII[0] * 100) + ASCII[1]. In this code *ord()* and *char()* functions are used. *ord()* takes a char as an argument and return the respective ASCII code (ord('U') = 85), and *char()* takes a numeric value as ASCII code and return the respective char (char(85) = 'U').

Listing 2.4: String key implementation.

```

1 class HashTable():
2     def __init__(self):
3         self.table = [None]*10000
4
5     def store(self, string):
6         hv = self.calculate_hash_value(string)
7         if hv != -1:
8             if self.table[hv] != None:
9                 self.table[hv].append(string)
10            else:
11                self.table[hv] = [string]
12
13    def lookup(self, string):
14        hv = self.calculate_hash_value(string)
15        if hv != -1:
16            if self.table[hv] != None:
17                if string in self.table[hv]:
18                    return hv
19        return -1
20
21    def calculate_hash_value(self, string):
22        value = ord(string[0]*100) + ord(string[1])
23        return value

```


3 | SEARCHING AND SORTING

In this chapter are introduced the most important and used algorithms about **searching**, which retrieve some data stored in a particular data structure [17] ([Search algorithm, Wikipedia](#)), and **sorting**, which put in a certain order some data stored in a list [18] ([Sorting algorithm, Wikipedia](#)).

3.1 BINARY SEARCH

Let us consider the problem of finding a number stored in an array with an **ascending** order (Figure 16).

1	3	9	11	15	19	29
---	---	---	----	----	----	----

25?

Figure 16: An array with numeric values ordered in an ascending order.

A first way to tackle the problem could be to check all the numbers of the array, in others words this method consists in performing a loop all over the elements of the array and check one by one if the number where is the element we are looking for. The complexity of this method is $O(n)$ because in the worst case we have to look at all the elements of the array.

There is a more efficient way to search an element in an ascending ordered array then the method showed previously. This method is called **binary search** [19] ([Binary Search Algorithm, Wikipedia](#)). Let us start to check as the first element the central value of the array (in case the array has an even number of elements there are two central values, and one can choose the bigger or the lower). If the central element is the one we are looking for the search ends, otherwise, because the array is ascending ordered, we can ignore one half of the array. If the central number is bigger than the number we are looking for thus we will ignore the right half of the array, if the central number is lower than the number we are looking for thus we will ignore the left half of the array. This procedure is repeated: we consider the central value of the new array, which has half of the size of the previous step, and we check is that value is the one we are looking for, and we repeat the procedure until or we find or we do not find the number we are looking for (Figure 17).

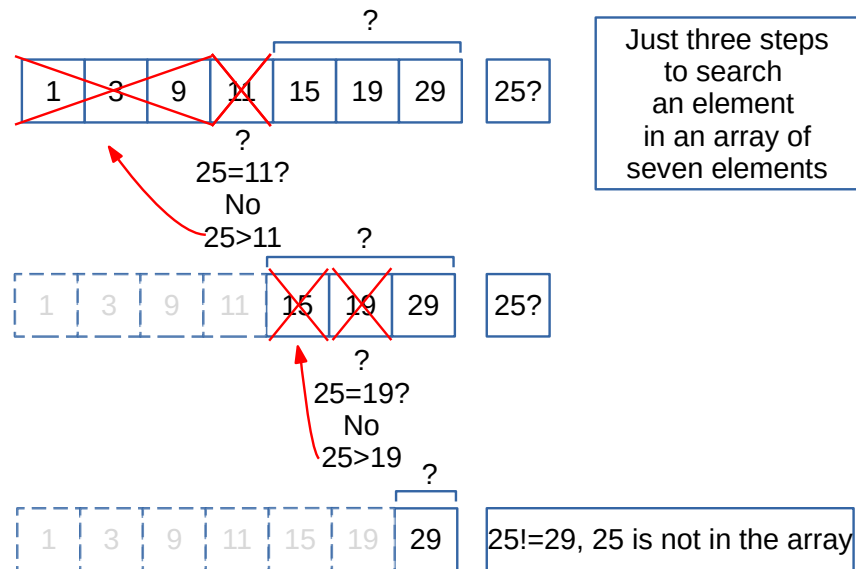


Figure 17: Binary search algorithms steps.

3.1.1 Efficiency of the Binary Search Algorithm

For evaluating the efficiency of this algorithm we have to evaluate the number of steps required for solving the problem. We already know that the efficiency of this algorithm will not be $O(n)$, because in this algorithm not all the elements of the array are checked. The way for evaluating the complexity of an algorithm is to execute the algorithms varying the size of the input, and evaluating how many operations are done in the worst case. In the binary search algorithm the worst case is when the size of the array becomes one, so we find or we do not find the searched element at the end of the array splitting process.

Table 1: The number of iterations grows of one every power of two, in others words it grows as $\log(n)$.

	2^0		2^1		2^2			2^3	
Array Size	0	1	2	3	4	5	6	7	8
Iterations (worst case)	0	1	2	2	3	3	3	3	4

We observe that the exponent of the power of two is the number of iteration minus one, or at the contrary the number of iteration is the exponent of a power plus one.

$$\log(\text{power of exponent } 2 + 1) = \log_2(n) + 1$$

In general it is used \log and is said that the binary search algorithm has a complexity of $\log(n)$.

3.1.2 Binary Search Implementation

The following code is the python implementation of the binary search algorithm.

Listing 3.1: Binary search python implementation.

```

1 def binary_search(array_input, value):
2     low = 0
3     high = len(array_input) - 1
4     while low <= high:
5         mid = (low + high) // 2
6         if array_input[mid] == value:
7             return mid
8         elif value > array_input[mid]:
9             new_low = mid + 1
10        else:
11            new_high = mid - 1
12    return -1

```

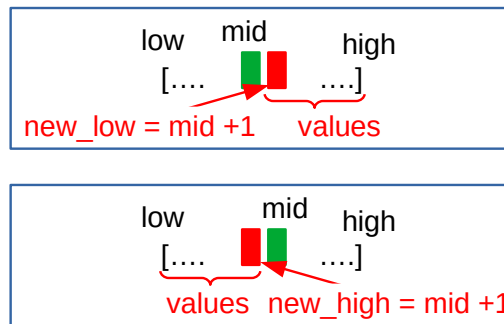


Figure 18: Array splitting in the implementation of the binary search algorithm.

3.2 BUBBLE SORT

bubble sort is the easiest sorting algorithm working on arrays. Bubble sort works by swapping two element at each step if they are in the wrong order, repeating this process until all the array is not completely ordered [20] ([Bubble Sort, Wikipedia](#)). In this algorithm the bigger elements tend to move at the bottom of the array, like bubbles that move at the top of a water bottle.

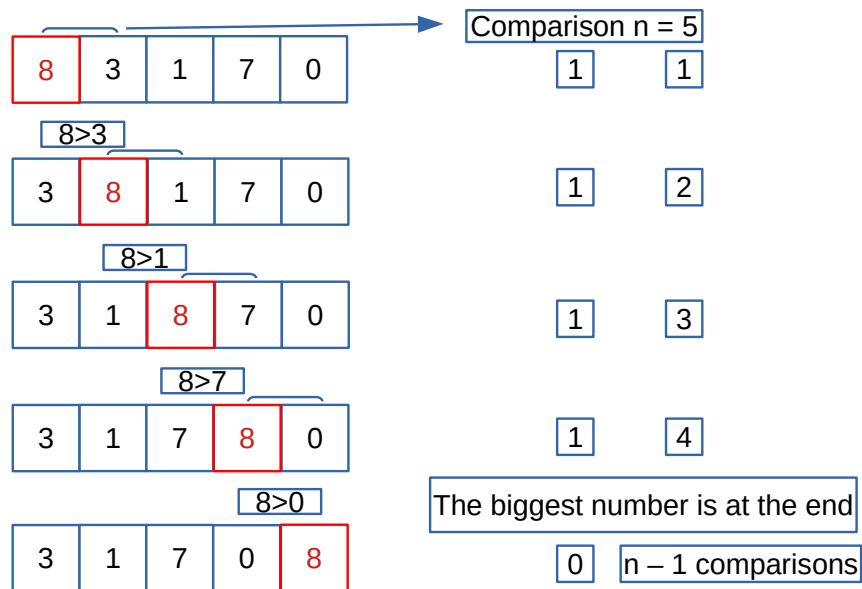


Figure 19: In bubble sort the biggest element goes at the end of the array.

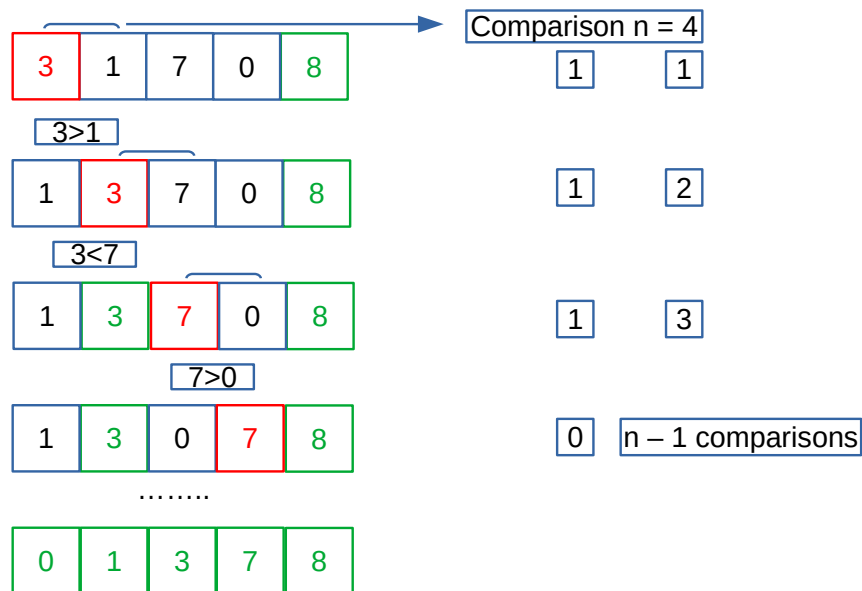


Figure 20: The swapping process is repeated until the array is completely ordered.

3.2.1 Efficiency of the Bubble Sort Algorithm

For ordering an array using the bubble sort $n - 1$ iterations are required, every step. The number of total steps are $n - 1$, thus the total number of operations to be executed for ordering an array is $(n - 1) * (n - 1) = n^2 - 2n + 1 = O(n^2)$. In summary:

- **Worst Case:** $O(n^2)$
- **Average Case:** $O(n^2)$

- **Best Case:** $O(n)$. The array is already completely ordered and it is enough to cycle all the elements.

3.2.2 Bubble Sort Implementation

The following code is the python implementation of the bubble sort algorithm.

Listing 3.2: Bubble Sort python implementation.

```

1 def bubble_sort(array_input):
2     index = len(array_input) - 1
3     sorted = False
4
5     while not sorted:
6         sorted = True
7         for i in range(0, index):
8             if array_input[i] > array_input[i + 1]:
9                 sorted = False
10                array_input[i], array_input[i + 1] = array_input
11                [i + 1], array_input[i]
12
13     return array_input

```

3.3 MERGE SORT

The **merge sort** algorithm works by dividing the array in single elements at first, grouping and ordering all the elements two by two. After the first step we will have a lot of subarrays of two elements. The next step is to merge all these subarrays and to order the elements. The merging and ordering process is repeated until the array is unified again [21] ([Merge Sort, Wikipedia](#)). This way of reducing a big problem in several smaller is called **Divide et impera** (Divide and conquer).

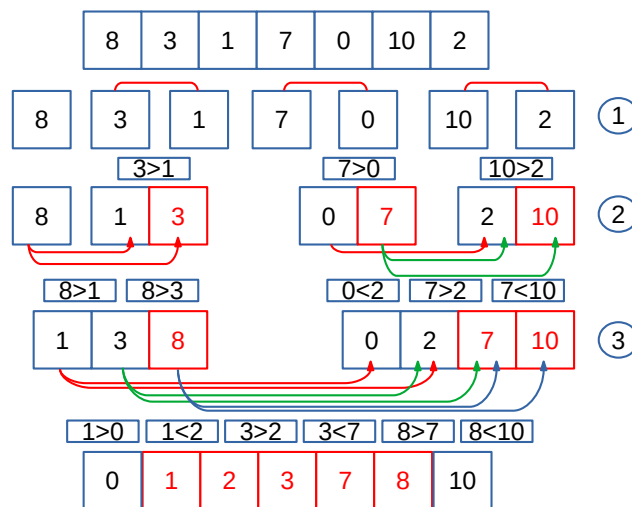


Figure 21: In merge sort merging and sorting process is repeated until the array is again unified with all the elements sorted.

The steps are (Figure 21):

- 1 Divide the array in subarrays of one element. Merge two by two and order the elements. The number of the subarrays is odd, so one array at this step is not merged. **The number of comparison for this step is 3.**
- 2 Merge and order again the new subarrays. For ordering in this case we start from the first element of the array on the left, and we compare this value with all the elements of the array on the right. If the first element is bigger than the picked one from the array on the right is moved, otherwise is not moved and we go to the next element. **The number of comparison for this step is 5.**
- 3 Merge and order again as at the previous step. **The number of comparison for this step is 6.**

3.3.1 Efficiency of the Merge Sort Algorithm

For evaluating the efficiency of this algorithms we have to count the number of iterations and comparisons are being done. By using the example showed in Figure 21 we will try to extrapolate a general pattern for an array of dimension n .

The number of comparisons depends by the array size. For an array of two elements the number of comparisons is one, for one of three elements are two, for one of four are three, and for one of seven are six. It is impossible to calculate in general the number of comparisons, but it is possible to calculate the worst case given the array dimension. From the previous example we see that for each step the maximum number of comparisons is seven, the size of the array. The reason is that the sum of all subarrays is seven. In general the sum of all subarrays is always the size of the array. Thus the total efficiency is $O(\# \text{ of comparison} * \# \text{ of iterations})$.

How many iterations are required? In our example for an array of seven elements, the iterations required are three. From the subprocess of our example we observe that for an array of size four the number of iterations are two, for one of size three are two, and for one of size two is one. Thus we can create the following table:

Table 2: The number of iterations grows of one every power of two, in others words it grows as $\log(n)$.

	2^0		2^1		2^2			2^3	
Array Size	1	2	3	4	5	6	7	8	9
Iterations (worst case)	0	1	2	2	3	3	3	3	4

In conclusion the efficiency is $O(n \log(n))$, which is better than $O(n^2)$ of the bubble sort.

The memory efficiency in this case is bigger than the bubble sort algorithm. For the merge sort some subarrays (in the worst case are n) are used and they need to be stored in the memory.

3.3.2 Merge Sort Implementation

The following code is the python implementation of the merge sort algorithm.

Listing 3.3: Merge Sort python implementation.

```

1 def merge_sort(array_input):
2
3     if len(array_input) > 1:
4         mid = len(array_input)//2
5         left_side = array_input[:mid]
6         right_side = array_input[mid:]
7
8         merge_sort(left_side)
9         merge_sort(right_side)
10
11        i = 0 # Left side index
12        j = 0 # Right side index
13        k = 0 # Sorted array index
14
15        while i < len(left_side) and j < len(right_side):
16            if left_side[i] < right_side[j]:
17                array_input[k] = left_side[i]
18                i+= 1
19            else:
20                array_input[k] = right_side[j]
21                j+= 1
22            k+= 1
23
24            # Adding all elements if some of
25            # them have been left behind
26        while i < len(left_side):
27            array_input[k] = left_side[i]
28            i+= 1
29            k+= 1
30
31        while j < len(right_side):
32            array_input[k] = right_side[j]
33            j+= 1
34            k+= 1
35
36        return array_input

```

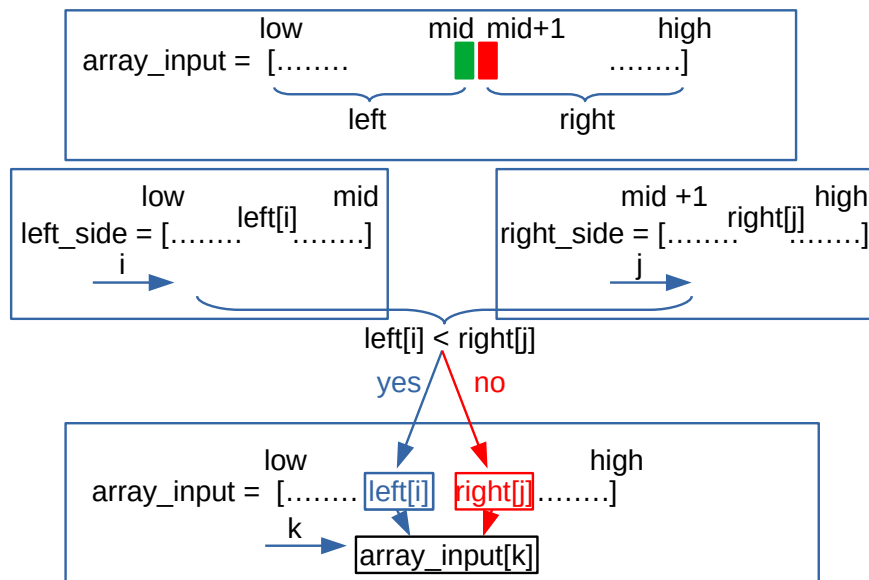


Figure 22: Merge sort algorithm implementation.

3.4 QUICKSORT

The **quicksort** is a sorting algorithm of the divide et impera type. It works by randomly choosing an element of the array, called pivot, and putting all the bigger and lower values on its left or on its right respectively. This procedure is repeated recursively on the two new subarrays until all the elements have been a pivot [22] ([Quicksort, Wikipedia](#)).

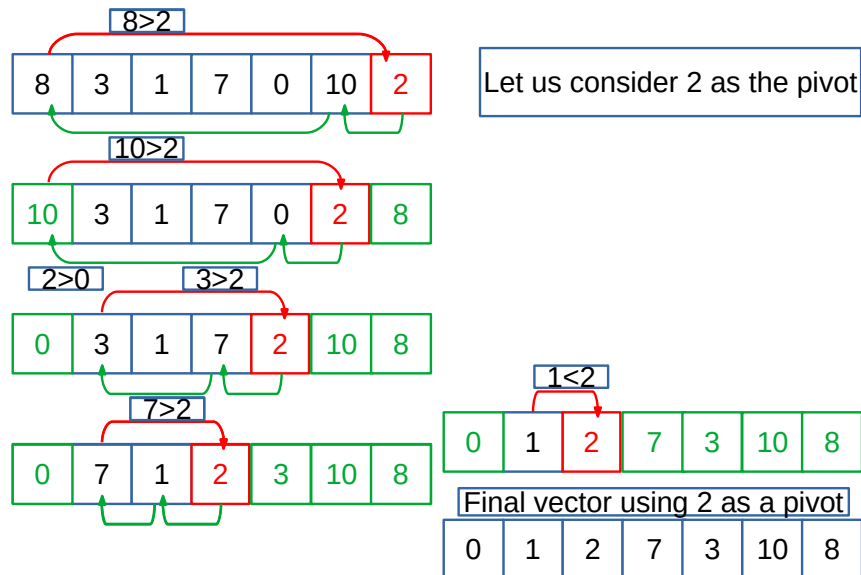


Figure 23: Quicksort algorithm part one.

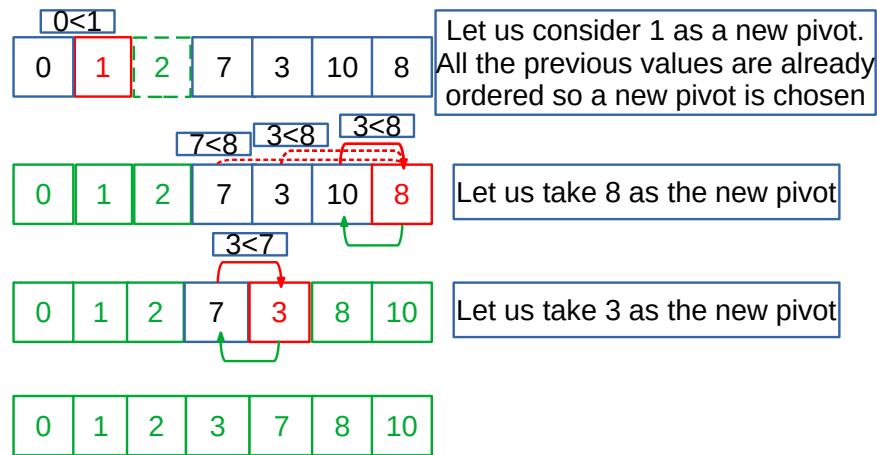


Figure 24: Quicksort algorithm steps part two.

Here are the steps of quicksort algorithm based on the example of Figure 23 and Figure 24:

- 1 Let us consider the last element as pivot, two in this case, and let us compare it with all the elements to its left. Let us start from the first element of the array, and let us compare their values. In this case $8 > 2$, so 8 is moved on the position of the pivot (2) which is moved of one position to the left. The number to be removed, 10 in this case, is moved in the first position.
- 2 Let us repeat the process. Now we have to compare the pivot (2) in the new position with the first element (10), and because $10 > 2$ we repeat the previous step of moving the elements.
- 3 In this case $0 < 2$ so we do not have to do anything, but going to the next element, 3 in this case.
- 4 For the pivot 2 all the elements to the left are less than it, and all the element to the right are bigger than it. 2 is not moved anymore. We can change the pivot and repeat all the steps for the new pivots until all the elements have been a pivot.

3.4.1 Efficiency of the Quicksort Algorithm

Evaluating the efficiency of quicksort is very hard. In the following there are some justifications for the worst case, and for the best and average complexity.

Let us consider first the worst case. In this situation the last elements of the array are the bigger ones, so it is necessary to check all the previous elements, by doing n^2 comparisons Figure 25.

1	8	2	5	3	9	13
1<13	8<13	2<13	5<13	3<13	9<13	
1<9	8<9	2<9	5<9	3<9		

Figure 25: Quicksort algorithm worst case.

In the best and in the average case the complexity is $O(n \log(n))$. The reason is because the first pivot tends to move at the center of the array, having in this way two subarrays. The pivots of these two subarrays will tend to move at their center and the process is repeated until all the elements have been a pivot, having in this way an ordered array.

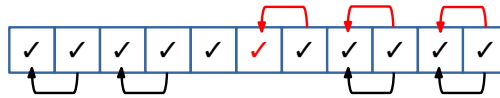


Figure 26: Quicksort algorithm best and average case.

The space complexity of quicksort is constant, $O(1)$.

This algorithm can be optimized in several ways. For example it is possible to order at the same time two half of the array, or considering as pivot always the last elements.

QUICKSORT AND MERGE SORT COMPARISON Quicksort is often a better solution than merge sort, because even if its worst case performance is $O(n^2)$, this problem can be solved by using the randomized quicksort. If the right pivot is chosen the problem related to having a worst case performance is solved. Moreover the quicksort algorithm dose not require an auxiliary memory, which is a big advance in a lot of situations.

On the other hand merge sort is a better solution than quicksort and heap-sort [23] ([Heapsort, Wikipedia](#)) when the sorting is done on linked lists that do not require big auxiliary space and on very large data sets stored on slow-to-access media, such as disk storage or network-attached storage [22].

In summary:

- **Worst Case:** $O(n^2)$
- **Average Case:** $O(n \log(n))$
- **Best Case:** $O(n \log(n))$
- **Space:** $O(1)$

3.4.2 Quicksort Implementation

The following code is the python implementation of the quicksort algorithm [24] ([Quicksort Python Implementation](#)).

Listing 3.4: Quicksort python implementation.

```

1 def merge_sort(array_input):
2
3     elements = len(array_input)
4
5     # Base case
6     if elements < 2:
7         return array_input
8
9     # Position of the partitioning element
10    current_position = 0
11
12    # Partitioning loop
13    for i in range(1, elements):
14        if array_input[i] <= array_input[0]:
15            current_position += 1
16            temp = array_input[i]
17            array_input[i] = array_input[current_position]
18            array_input[current_position] = temp
19
20    # Brings pivot to its appropriate position
21    temp = array_input[0]
22    array_input[0] = array_input[current_position]
23    array_input[current_position] = temp
24
25    # Sorts the elements to the left of pivot
26    left = SortingAlgorithms.quicksort(array_input[0:current_position])
27    # Sorts the elements to the right of pivot
28    right = SortingAlgorithms.quicksort(array_input[current_position+1:
29                                         elements])
30
31    # Merging everything together
32    array_input = left + [array_input[current_position]] + right
33
34    return array_input

```

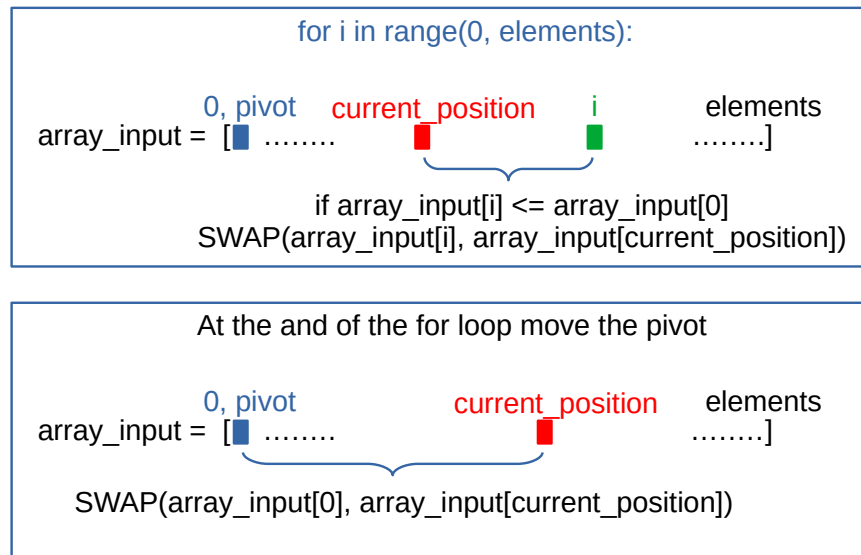


Figure 27: Quicksort algorithm implementation.

4 | TREES

In this chapter are introduced the fundamentals concepts of tree as abstract data type, and the most used algorithms related to **trees**.

4.1 GENERAL DEFINITIONS

A **tree** is an abstract datatype that simulate a hierarchical structure [25] ([Trees, Wikipedia](#)). A tree is a particular kind of a linked list where there are more next elements. The base element of a tree is called **node** while the first element is called **root**.

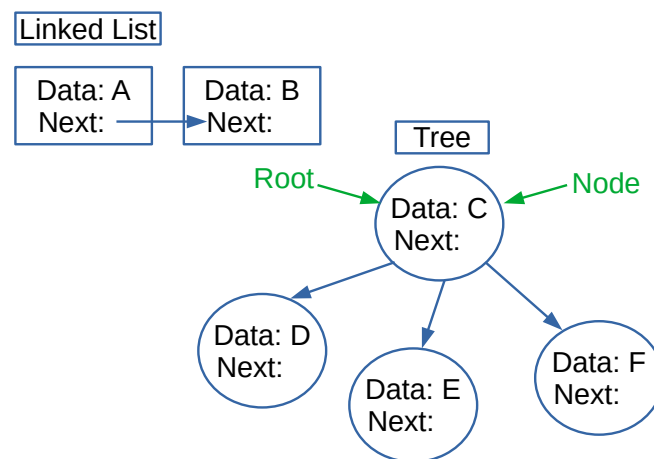


Figure 28: Elements of a tree and linked list.

Trees must be completed connected structures, this means that there are not any nodes which are not connected to anything (Figure 29 case (a)), and there must not be present any cycles (Figure 29 case (c)).

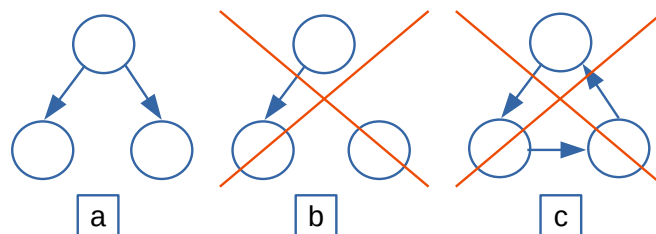


Figure 29: Completed connected structure (a), a non completed connected structure (b), and a cycle (c).

Trees are a hierarchical structures divided into layers: the first layer is the one belonging to the root node, the first node of a tree. The next element

of the root are called children, which became parents in case they have next node connected as well. The last nodes of a trees are the nodes which do not have any children and they are called **leaf**. The numbers of connections is called **height**. A set of connections creates a **path**. The numbers of edges starting from the root to a node is called **depth**. All this definitions are defined in following figure (Figure 30).

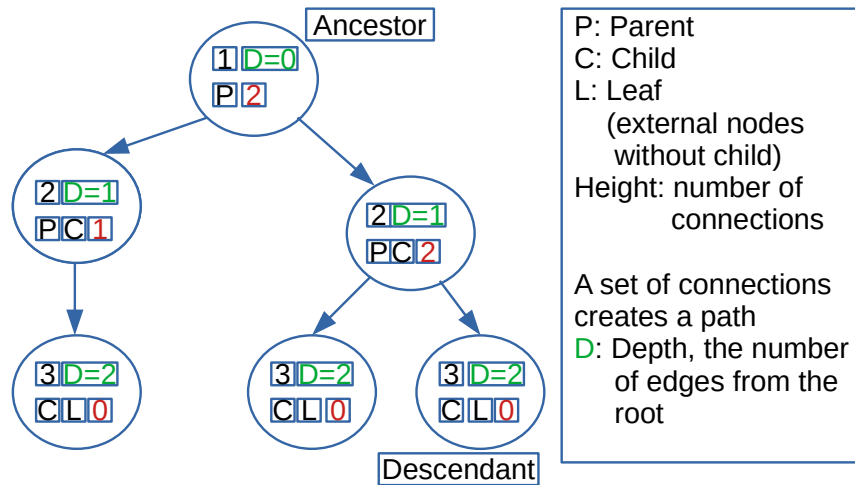


Figure 30: A tree and its fundamentals elements.

4.2 TREE TRAVERSAL

Which way is the most efficient for visiting all the nodes of a tree? Is it more efficient looking layer by layer or looking at subtrees? There are two different approach to traverse a tree: the **depth-first search (DFS)** and the **breadth-first search (BFS)**. In the first one the priority is to look at the children of a node, instead in the second one the priority is to look at the node of the same layer 31.

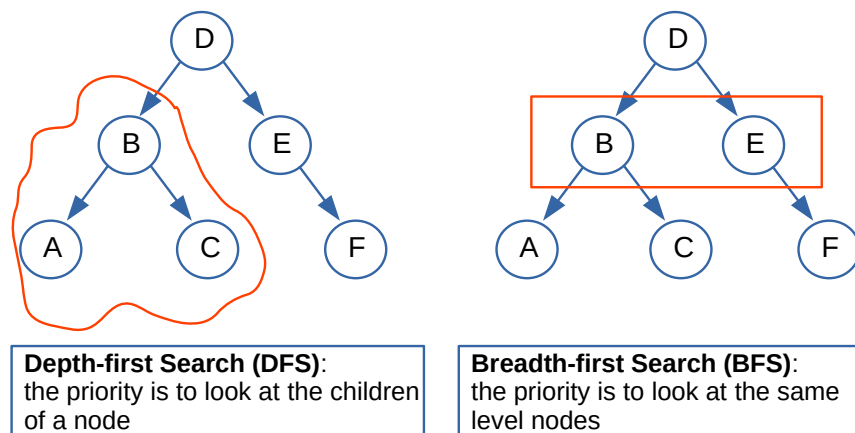


Figure 31: The depth-first search and the breadth-first search.

4.2.1 Depth-first search

In the depth-first search there are several different ways to perform a search on a tree.

PRE-ORDER SEARCH In the **pre-order search** the first node to be checked as visited is the root. The following node is the left child by convention. Once checked the left child the process is repeated until the first node without any children is reached. At that point the same process is applied to the right side: all the nodes on the right are checked until all the nodes are checked as visited 32.

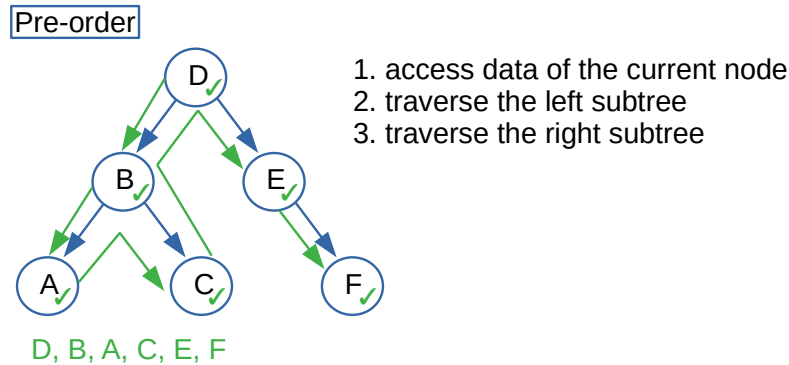


Figure 32: Pre-order search.

IN-ORDER SEARCH In the **in-order search** the first node to be checked is the first node without children on the left side. Once checked off this node the next one to be checked off is its parent, and the process is repeated again on the right side of the nodes, until all the nodes are checked off.

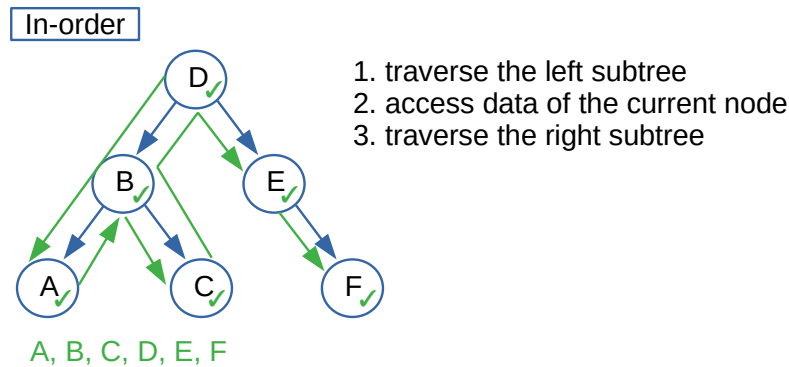


Figure 33: In-order search.

POST-ORDER SEARCH In the **post-order search** the first node to be checked off is the first node without children on the left side. Once checked off this node the next one to be checked off is the one which does not have any children. Once all the nodes of the current left subtree without any children

are checked off, the parents can be checked off, and the whole process is repeated until all the nodes are checked off.

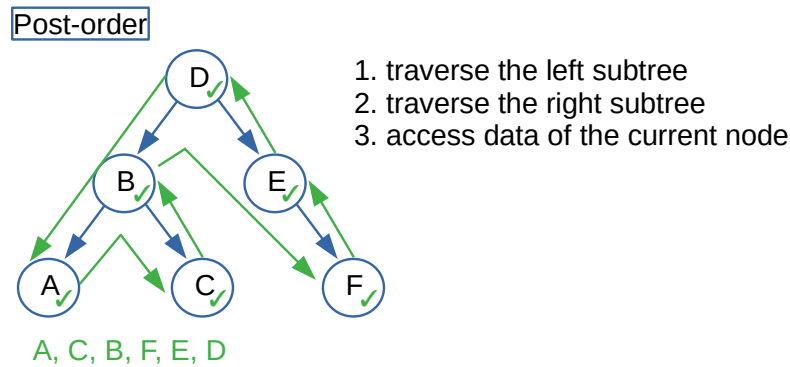


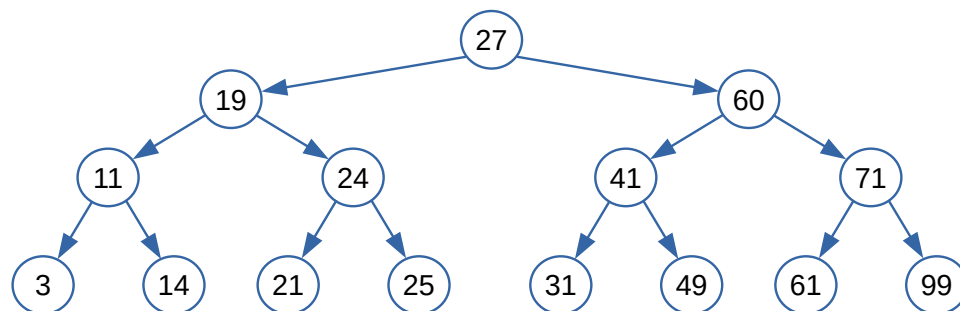
Figure 34: Post-order search.

4.3 BINARY TREES

Binary trees are trees in which the parent has at most two children (0, 1, 2 are the only number of admitted children). On binary trees some operations like searching, deleting, and inserting can be easily and efficiently done.

4.3.1 Search

To search an element in the tree one of the previous methods can be used. Because in a general tree there is no ordering this means that potentially all the elements of the tree can be visited, then the complexity results in $O(n)$.



DFS

Pre-order traversal: [27, 19, 11, 3, 14, 24, 21, 25, 60, 41, 31, 49, 71, 61, 99]

In-order traversal: [3, 11, 14, 19, 21, 24, 25, 27, 31, 41, 49, 60, 61, 71, 99]

Post-order traversal: [3, 14, 11, 21, 25, 24, 19, 31, 49, 41, 61, 99, 71, 60, 27]

BFS: [27, 19, 60, 11, 24, 41, 71, 3, 14, 21, 25, 31, 49, 61, 99]

Figure 35: Example of tree search and traversal.

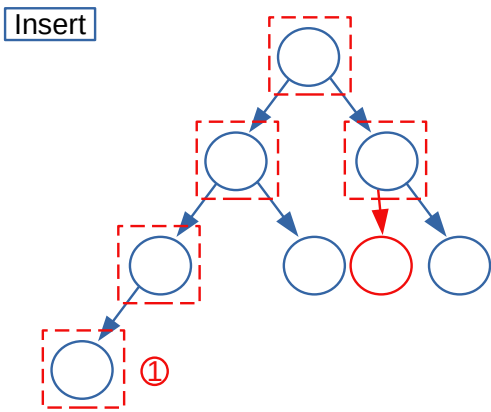


Figure 37: Add an element of a tree.

AGGIUNGERE IL LIBRO DI ALGORITMI CITANDO CHE SIA UNA LETTURA APPROFONDATA PER MAGGIORI DETTAGLI.

4.3.4 Perfect binary tree

A perfect binary tree is a binary tree in which all the nodes have two children except the leaf which do not have any children. In this case the following results are valid




Level		Nodes in level		Total nodes
1		1	2^0	1
2		2	2^1	3
3		4	2^2	7

Figure 38: Perfect binary tree results.

When a new row is added, the number of nodes double. Given a level n the total number of nodes is $2n + 1$.

4.3.5 Binary Tree Implementation

The following code is the recursive Python implementation of the pre-order search and traversal (Figure 32). The recursive and iterative implementations of the other ways of search and traverse a tree are in appendix A.

Listing 4.1: Class definition for a node and a tree.

```
1 class Node():
2
```

```

3     def __init__(self, value):
4         self.value = value
5         self.left = None
6         self.right = None
7
8 class BinaryTree():
9
10    def __init__(self, root):
11        self.root = Node(root)

```

Listing 4.2: Recursive pre-order traversal and search implementation.

```

1 class Node():
2     ...
3
4 class BinaryTree():
5     ...
6
7     def print_tree(self):
8         return self.preorder_print(tree.root, "")[:-1]
9
10    def preorder_search_recursive(self, start, find_val):
11        if start:
12            if start.value == find_val:
13                return True
14            self.preorder_search_recursive(start.left, find_val)
15            self.preorder_search_recursive(start.right, find_val)
16
17    def preorder_print(self, start, traversal):
18        if start:
19            traversal += (str(start.value) + "-")
20            traversal = self.preorder_print(start.left, traversal)
21            traversal = self.preorder_print(start.right, traversal)
22        return traversal

```

4.4 BINARY SEARCH TREES (BST)

Performing operations such as search, addition, or removal of an element it is a very efficient process if the binary tree is ordered. A binary tree is ordered if for all nodes the left child has a minor value, and the right child has a greater value than the considered node [26]([Binary search tree, Wikipedia](#)).

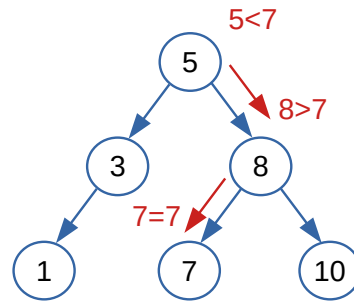


Figure 39: Search on an ordered binary tree.

For example in Figure 39 is an ordered tree. Let us consider we would like to find the node which value is 7. Because the tree is ordered for efficiently finding 7 is enough to compare 7 with the value of the node and choose every time if to search on the right or on the left. In this way for finding a value is not mandatory to look at all the nodes, and the complexity for this operation is $\log(n)$.

Let us consider instead we would like to add a new node of value 4 to the tree of Figure 39. Because the tree is ordered, there is only one place in which the node can be added. In the case of the tree in Figure 39 the node can be easily added because there is an empty space that can correctly take it (Figure 40). In this case the complexity for this operation is again $\log(n)$.

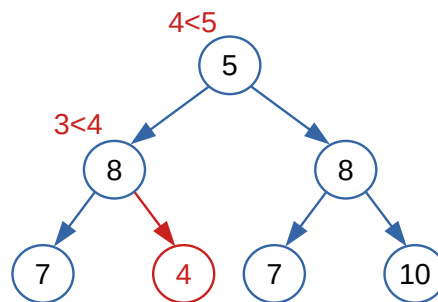


Figure 40: Addition on an ordered binary tree.

In case the position of the node is already occupied, adding the node could be a very difficult task. For a more complete and formal description of this topic refer to [27] in the chapter about trees.

When all the possible nodes are present in each level the tree is said **balanced**. In Figure 40 there is an example of a balanced tree. When instead a tree has all its children nodes which have always a bigger value, all the tree is on the right, and it is said to be **unbalanced**. In this case the search of an element has a $O(n)$ complexity in the worst case.

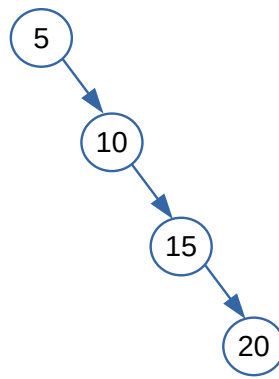


Figure 41: An unbalanced binary tree.

4.4.1 Binary Search Tree Implementation

The following code is the python implementation of the search and addition operations performed on a ordered binary tree.

Listing 4.3: implementation of insert and search operation for a binary search tree.

```

1  class BST():
2
3      def __init__(self, root):
4          self.root = Node(root)
5
6      def insert(self, new_val):
7          self.insert_helper(self.root, new_val)
8
9      def insert_helper(self, current, new_val):
10         if current.value < new_val:
11             if current.right:
12                 self.insert_helper(current.right, new_val)
13             else:
14                 self.current.right = Node(new_val)
15         else:
16             if current.left:
17                 self.insert_helper(current.left, new_val)
18             else:
19                 current.left = Node(new_val)
20
21     def search(self, find_val):
22         return self.search_helper(self.root, find_val)
23
24     def search_helper(self, current, find_val):
25         if current:
26             if current.value == find_val:
27                 return True
28             elif current.value < find_val:
29                 return self.search_helper(current.right,
30                                         find_val)
31             else:
32                 return self.search_helper(current.left, find_val
33                                         )
  
```

32

`return False`

4.5 HEAPS

Heaps are particular tree-based data structure in which the elements are ordered in increasing or decreasing order [28] ([Heap, Wikipedia](#)). Heaps are a very efficient implementation of the abstract data type (Chapter 2) priority queue (Section 2.6). The value with the highest (or lowest) priority is always stored at the root. Heaps are very useful when an element with a high (or low) priority must be repeatedly removed from the heap. Moreover, heaps are not sorted structure, but are regarded as partially ordered.

We say the *max heap* (Figure 42 (a)) if for every node its children have the values less or equal than the one of the parent (the root has always the highest value). We say instead *min heap* (Figure 42 (b)) if for every node its children have the values higher or equal than the one of the parent (the root has the lowest value).

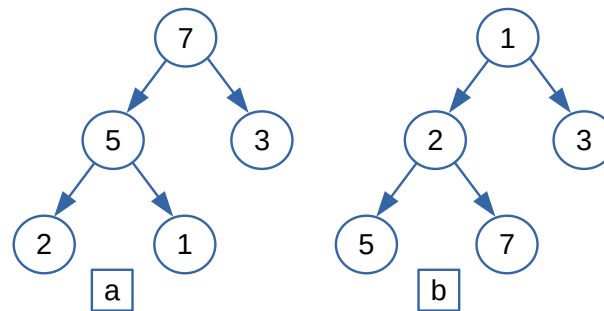


Figure 42: Max heap (a) and min heap (b) example.

In general heaps are not binary trees so they can have more than two nodes.

In a max heap tree the search of the biggest value is called **peek** and it has a constant complexity ($O(1)$). For searching other values instead we have to check node by node as the ordering of the left and right side is not guarantee. The average case is $O(\frac{n}{2}) = O(n)$.

4.5.1 Heapify

Let us consider we would like to add a new node to a heap. In this case the node to be add must respect the heap condition. For doing so the node is added in the first available position, and later the node of the heap are swapped in order to keep true the heap condition.

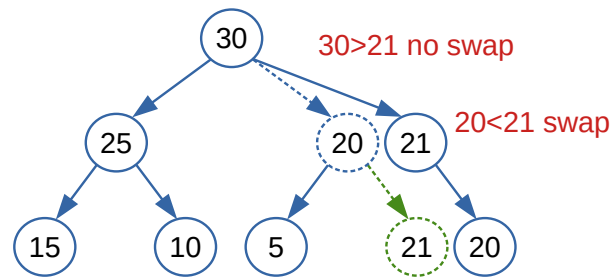


Figure 43: Add a new node to a heap.

In the example of Figure 43 only one swap is necessary. The complexity of adding a new element and swap all the node, in order to keep valid the heap condition, is $\log(n)$, and in the worst case the highest number of operations correspond to the height of the tree.

4.5.2 Heap Implementation

To implement a heap we can use an array. An example of this application is shown in the following figure (Figure 44).

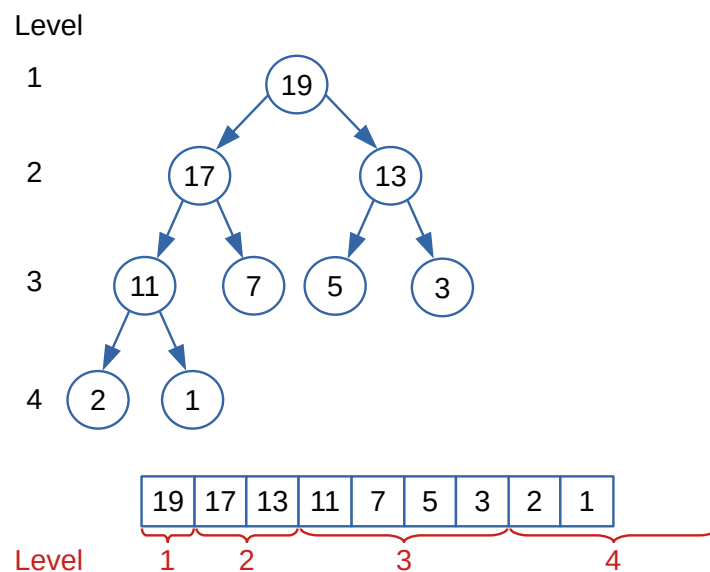


Figure 44: Implementation example of a heap using an array.

4.6 SELF-BALANCING BINARY SEARCH TREES

Self-balancing binary trees are binary tree-based structure that automatically maintain the lowest height (number of levels below the root) when an insertion or deletion is done [29] ([Self-balancing binary search tree, Wikipedia](#)). This structure is an efficient way to implement abstract data type like lists, priority queue, and sets.

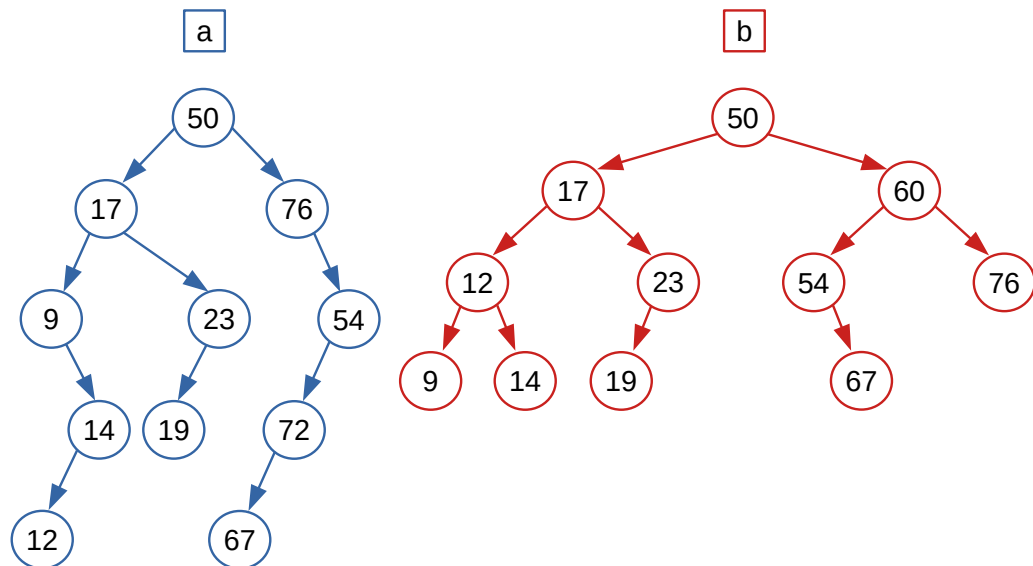


Figure 45: Example of balanced (b) and unbalanced (a) tree. Credits: [Self-balancing binary search tree, Wikipedia](#)

In Figure 45 is show an unbalanced tree (case a) in which the average path to reach the leaf is 3.27 nodes. If the same tree is balanced (case b) the average path to reach any leaf from the root decreased to 3.

4.7 RED-BLACK TREES

The **red-black trees** are particular kind of self balancing trees. An extra bit is used to store the color of the node, which can be only **red** or **black**. Using a color for each node assures that the tree remains balanced during insertion and deletion operations [30] ([Red-black tree, Wikipedia](#)).

The rules that a red-black tree must respect are:

- 1 Nodes can be only of two types: **red** or **black**.
- 2 There must exist the **null leaf nodes**. Every node which does not have two leaf must have the children **null**.
- 3 If a node is **red** all its children must be **black**.
- 4 Usually the root is **black**, but this is not mandatory.
- 5 Every path from a the upper nodes to the lower ones must contain the same number of **black** nodes. This property is very important when inserting new nodes.

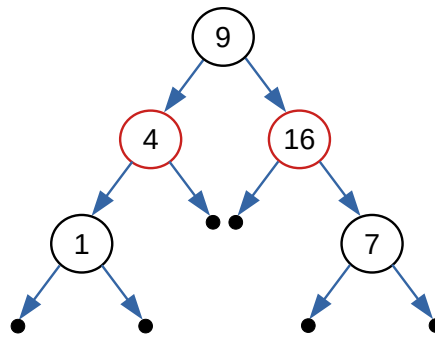


Figure 46: A red-black tree.

Usually only red nodes are added and the color of the other nodes are changed accordingly in order to respect the rule for the red-black tree.

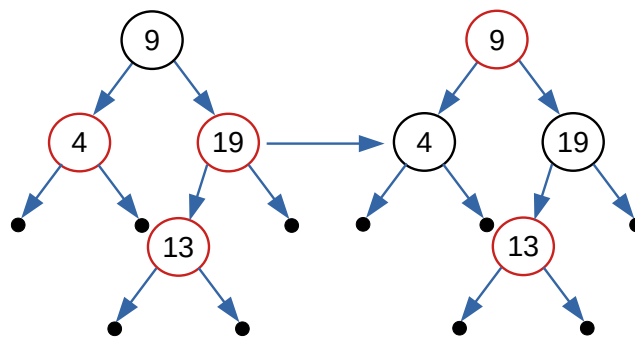


Figure 47: Insertion of a new node and the following updating of the color nodes.

In case the insertion or the removal does not respect the rules of the red-black tree and BST some more complex operations must be done as in the following figure.

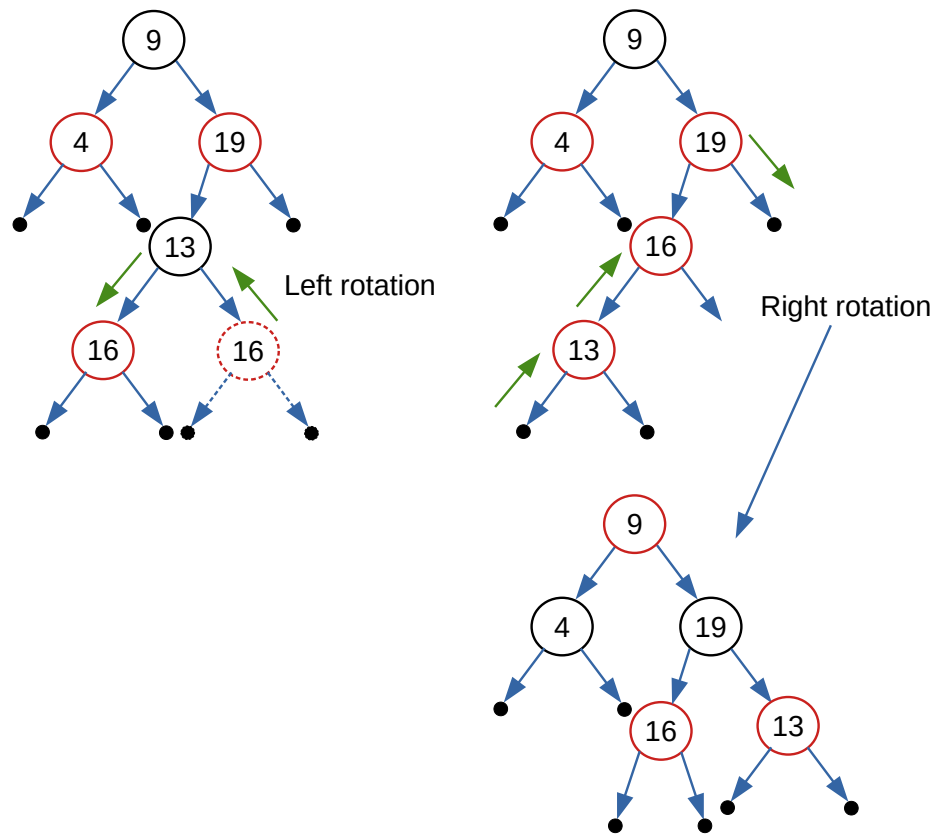


Figure 48: Left and right rotation for balancing the tree after an insertion.

5 | GRAPHS

In this chapter are introduced the fundamentals concepts of **graphs** as mathematical objects, and as an abstract data type, its applications in computer science, and the most important and notable algorithms with their implementation.

5.1 GENERAL DEFINITIONS

A graph is a discrete mathematical structure in which the connections between its elements, and their relationship are highlighted. A graph is made up by two different elements: **nodes** or **vertices**, and **links** or **edges**. In case the links do not have any direction the graph is called **undirected graph** (Figure), in case instead the links have a direction the graph is called **directed graph**.

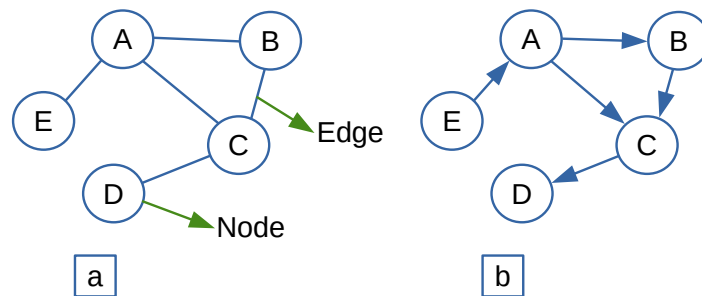


Figure 49: Undirected (a) and directed (b) graphs and its elements.

Formally a graph is the pair $G = (V, E)$, where V is the set of all vertices, and E is the set of paired vertices, whose elements are called edges [31] (Graph, Wikipedia).

A **tree** (Chapter 4) is a special kind of graph.

Graphs are very useful to describe a lot of real situations like: connections between people, computers (internet), web pages (world wide web), airports, cities, and gene inside the DNA.

In graphs, differently to trees, closed loops can exist. These kind of closed loops can be dangerous for the algorithms because they could lead to infinite executions.

5.1.1 Connectivity

Connectivity is a measure that describe how much the nodes of a graph are connected. It is defined as the minimum number of elements (nodes or

edges) that need to be removed to separate the remaining nodes into isolated subgraphs [32] ([Connectivity, Wikipedia](#)).

A graph is said to be **connected** if every pairs of nodes are connected. Thus it always exists at least one path that connects every pairs of nodes. If an undirected graphs is not connected then it is **disconnected**: in this case there is one or more nodes that can not be reached by any paths.

STRONGLY CONNECTED A directed graph is said to be **strongly connected** if every pairs of nodes can be reached by one or more path.

WEAKLY CONNECTED A directed graph is said to be **weakly connected** if by replacing all the direct edges with undirectd ones, the new graph is connected. in a directed graph some nodes can not be reached if all the edges exit or enter from them. The graph in Figure 50 is weakly connected because the node U has only entering edges.

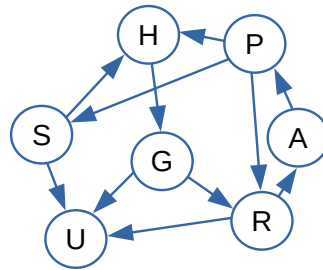


Figure 50: A weakly connected graph.

5.2 GRAPH REPRESENTATIONS

There are several ways to represent graphs using data structures. For example in an object oriented programming language a way could be to define a type for the vertex, and a type for the edges.

The most common data structures used for representing a graph are: **edge list**, **adjacency list**, and **adjacency matrix** [27].

In appendix B there is a summary of the complexities for the most common operations performed on graphs for the different data structures.

5.2.1 Edge List

The **edge list** is an unordered list of all pairs of nodes that form an edge. This representation is minimal but it does not allow to locate a specific edge, or the set of edges incident to a particular node [27].

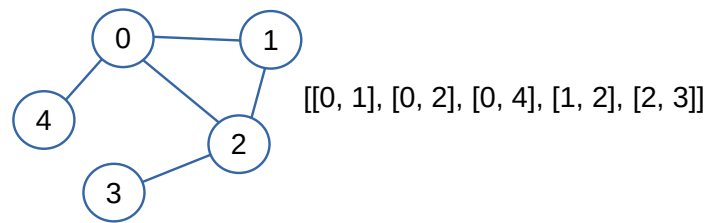


Figure 51: Edge list.

5.2.2 Adjacency List

The **adjacency list** is a list containing a separate list for each node containing all the incident edges of that node. In this representation identifying all the edges incident to a node is easy [27]. In case of directed graphs for each node there are two different lists: one for entering edges, and another one for the edges that go out.

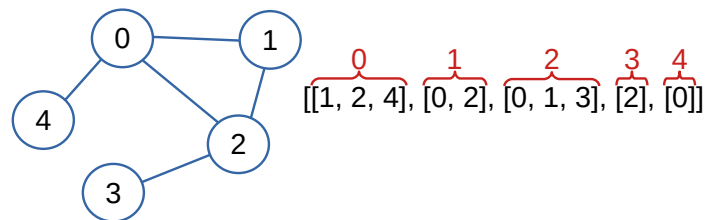


Figure 52: Adjacency list.

5.2.3 Adjacency Matrix

The **adjacency matrix** is a matrix A in which each element $A[i, j]$ represent the relationship between the edge i with the edge j . If between i and j there is an edge $A[i, j] = 1$, otherwise 0. In case the an ed go out and enter in the same edge $A[i, i] = 1$. For undirected graphs this matrix is symmetric, and in case there are not edges going in and out the same node the principal diagonal is all of 0.

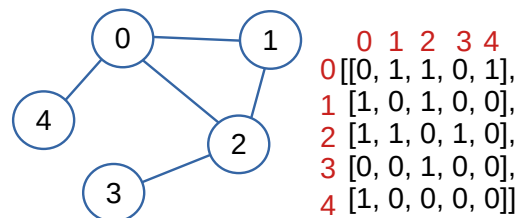


Figure 53: Adjacency matrix.

In the appendix C there is the full python implementation of the graph representation. There are defined the classes for nodes, edges, and graphs objects. Moreover, all the main operations on graphs like: insert a new node,

a new edge, get the edge list, the adjacency list, the adjacency matrix, and find the max index are also implemented.

5.3 GRAPH TRAVERSAL

Like in the case of trees (Chapter 4), also for graphs there are two different way to traverse the structure: **depth-first search**, and **breadth-first search**. But for graphs, differently from trees, there is not a privileged way to traverse the structure. Then it is arbitrarily choose a node where to start the search.

5.3.1 Depth-first Search (DFS)

5.3.2 Breadth-first Search (BFS)

APPENDIX

A

BINARY SEARCH COMPLETE IMPLEMENTATION

The best way to implement the tree data structure is to create a class for the nodes (**Node**) and a class for the binary tree (**BinaryTree**). As all the nodes of a binary trees has at most two children, in the class **Node** is enough to have three attributes: the value of the node, the left, and the right child, in turn of **Node** type as well. This way of implement tree is very similar to the one used for the linked list (2.4).

Listing A.1: Class definition for a node and a tree.

```
1 class Node():
2
3     def __init__(self, value):
4         self.value = value
5         self.left = None
6         self.right = None
7
8 class BinaryTree():
9
10     def __init__(self, root):
11         self.root = Node(root)
```

In this appendix there is a detailed implementation, both recursive and iterative, of all possible ways to perform a tree traversal in the case of depth-first search: pre-order traversal A.1, in-order traversal A.2, and post-order traversal A.3. The convention followed here is to look up first at the left child always, and later look up the right one. The opposite approach is equivalent.

The pseudocode of the iterative implementations is taken from [33]([Tree traversal, Wikipedia](#)).

A.1 PRE-ORDER TRAVERSAL

In the pre-order traversal every new node is first checked as visited, and the left subtree is traversed first, and later the right one. This process is repeated until all the nodes of the tree are visited (4.2.1).

For implementing the pre-order traversal iteratively a stack (2.5) is used. The steps are [33]:

Algorithm 1: Pre-order iterative implementation pseudocode.

```

1 Function Recursive-Preorder(node):
2   if (node == null) then
3     return
4   visit(node)
5   Recursive-Preorder(node.left)
6   Recursive-Preorder(node.right)
7
8 Function Iterative-Preorder(node):
9   if (node == null) then
10    return
11  s ← empty stack
12  s.push(node)
13  while (not s.isEmpty()) do
14    node ← s.pop()
15    visit(node)
16    /* Right child is pushed first so that left is
17       processed first (LIFO) */
17    if (node.right ≠ null) then
18      s.push(node.right)
19    if (node.left ≠ null) then
20      s.push(node.left)

```

Listing A.2: Recursive and iterative implementation of pre-order traversal.

```

1 class BinaryTree():
2   ...
3
4   def recursive_print_tree(self):
5       return self.preorder_recursive_print(tree.root, "")[:-1]
6
7   def iterative_print_tree(self):
8       return self.preorder_search_iterative(self, tree.root, "")
9
10  def preorder_search_recursive(self, start, find_val):
11      if start:
12          if start.value == find_val:
13              return True
14          self.preorder_search_recursive(start.left, find_val)
15          self.preorder_search_recursive(start.right, find_val)
16
17  def preorder_search_iterative(self, start, find_val):
18      if start == None:
19          return None
20      visited = []
21      stack = []
22      stack.push(start)
23      while not stack: # Keep cycle until the stack is empty

```

```

24         node = stack.pop()
25         visited.append(node)
26         if node == find_val:
27             return True
28         # Right child is pushed first so that the left one is
           processed first
29         if node.right:
30             stack.append(node.right)
31         if node.left:
32             stack.append(node.left)
33         return visited
34
35     def preorder_recursive_print(self, start, traversal):
36         if start:
37             traversal += (str(start.value) + "-")
38             traversal = self.preorder_recursive_print(start.left,
39                                                         traversal)
40             traversal = self.preorder_recursive_print(start.right,
41                                                         traversal)
42         return traversal

```

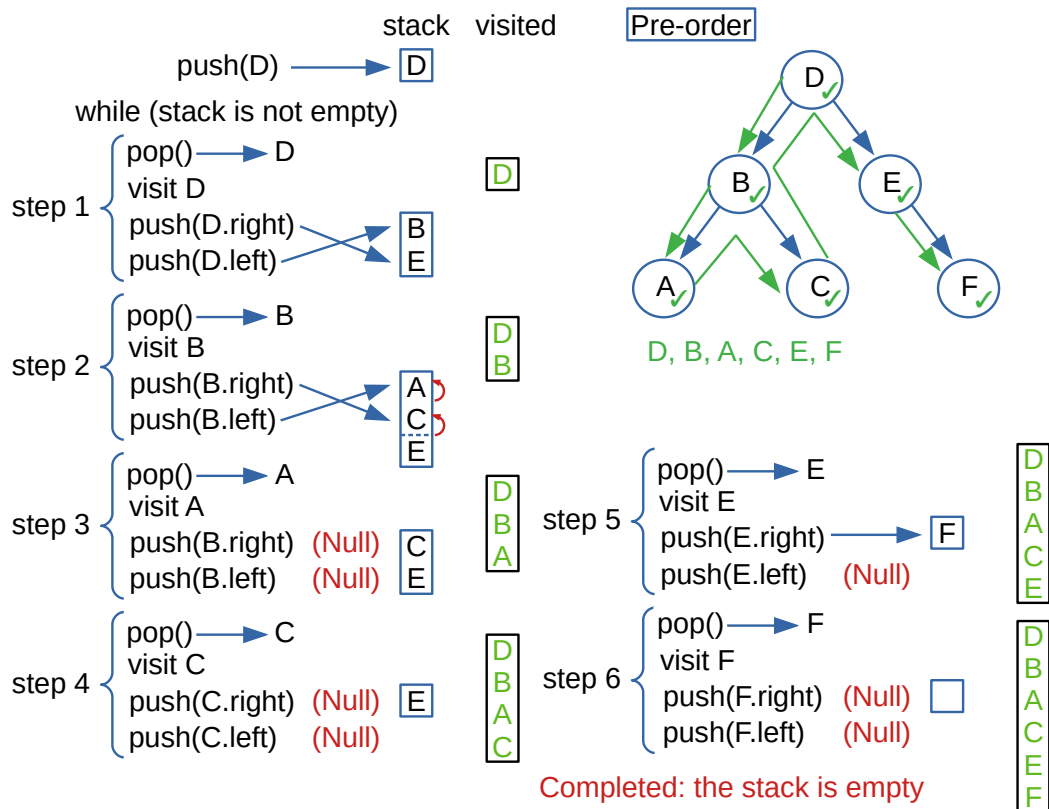


Figure 54: Pre-order iterative implementation example.

A.2 IN-ORDER TRAVERSAL

In the in-order traversal the first node to be checked is the first leaf of the left subtree. The second one is the parent of the first checked node, and then the right subtree is checked repeating the previous process. All this process is repeated until all the nodes are visited or the value is found [4.2.1](#).

For implementing the in-order traversal iteratively a stack is used. The steps are [\[33\]](#):

Algorithm 2: In-order iterative implementation pseudocode.

```

1 Function Recursive-Inorder(node):
2   if (node == null) then
3     return
4   Recursive-Inorder(node.left)
5   visit(node)
6   Recursive-Inorder(node.right)
7
8 Function Iterative-Inorder(node):
9   s ← empty stack
10  while (not s.isEmpty() or node ≠ null) do
11    if (node ≠ null) then
12      s.push(node)
13      node ← node.left
14    else
15      node ← s.pop()
16      visit(node)
17      node ← node.right

```

Listing A.3: Recursive and iterative implementation of in-order traversal.

```

1 class BinaryTree():
2   ...
3
4   def recursive_print_tree(self):
5     return self.inorder_recursive_print(tree.root, "")[:-1]
6
7   def iterative_print_tree(self):
8     return self.inorder_search_iterative(self, tree.root, "")
9
10  def inorder_search_recursive(self, start, find_val):
11    if start:
12      self.inorder_search_recursive(start.left, find_val)
13      if start.value == find_val:
14        return True
15      self.inorder_search_recursive(start.right, find_val)
16
17  def inorder_search_iterative(self, start, find_val):
18    if start == None:
19      return None
20    stack = []
21    stack.append(start)

```

[illegible]

Also in this case for implementing the post-order traversal iteratively a stack is used. The steps are [33]:

Algorithm 3: Post-order iterative implementation pseudocode.

```

1 Function Recursive-Postorder(node):
2   if (node == null) then
3     return
4   Recursive-Postorder(node.left)
5   Recursive-Postorder(node.right)
6   visit(node)
7
8 Function Iterative-Postorder(node):
9   s ← empty stack
10  lastNodeVisited ← null
11  while (not s.isEmpty() or node ≠ null) do
12    if (node ≠ null) then
13      s.push(node)
14      node ← node.left
15    else
16      peekNode ← s.peek()
17      /* If right child exists and traversing node from
18         left child, then move right */
19      if (peekNode.right ≠ null and lastNodeVisited ≠
20         peekNode.right) then
21        node ← peekNode.right
22      else
23        visit(peekNode)
24        lastVisitedNode ← s.pop()

```

Listing A.4: Recursive and iterative implementation of post-order traversal.

```

1 class BinaryTree():
2   ...
3
4   def recursive_print_tree(self):
5     return self.postorder_recursive_print(tree.root, "")[:-1]
6
7   def iterative_print_tree(self):
8     return self.postorder_search_iterative(tree.root, "")
9
10  def postorder_search_recursive(self, start, find_val):
11    if start:
12      self.postorder_search_recursive(start.left, find_val)
13      self.postorder_search_recursive(start.right, find_val)
14      if start.value == find_val:
15        return True
16
17  def postorder_search_iterative(self, start, find_val):
18    if start == None:
19      return None

```

```

20     stack = []
21     stack.push(start)
22     last_visited_node = None
23     visited = []
24     current = start
25     while (not stack) or (current != None):
26         if current != None:
27             s.push(current)
28             current = current.left
29         else:
30             # Take the last element of the stack
31             peeked_node = stack[-1]
32             # If right child exists and traversing node from
33             # left child, then move right
34             if (peeked_node.right != null and
35                 last_visited_node != peeked_node.right):
36                 current = peeked_node.right
37             else:
38                 if peeked_node == find_val:
39                     return True
40                 visited.append(peeked_node)
41                 last_visited_node = stack.pop()
42     return visited
43
44 def postorder_recursive_print(self, start, traversal):
45     if start:
46         traversal = self.postorder_recursive_print(start.left,
47                                                     traversal)
48         traversal = self.postorder_recursive_print(start.right,
49                                                     traversal)
50         traversal += (str(start.value) + "-")
51     return traversal

```

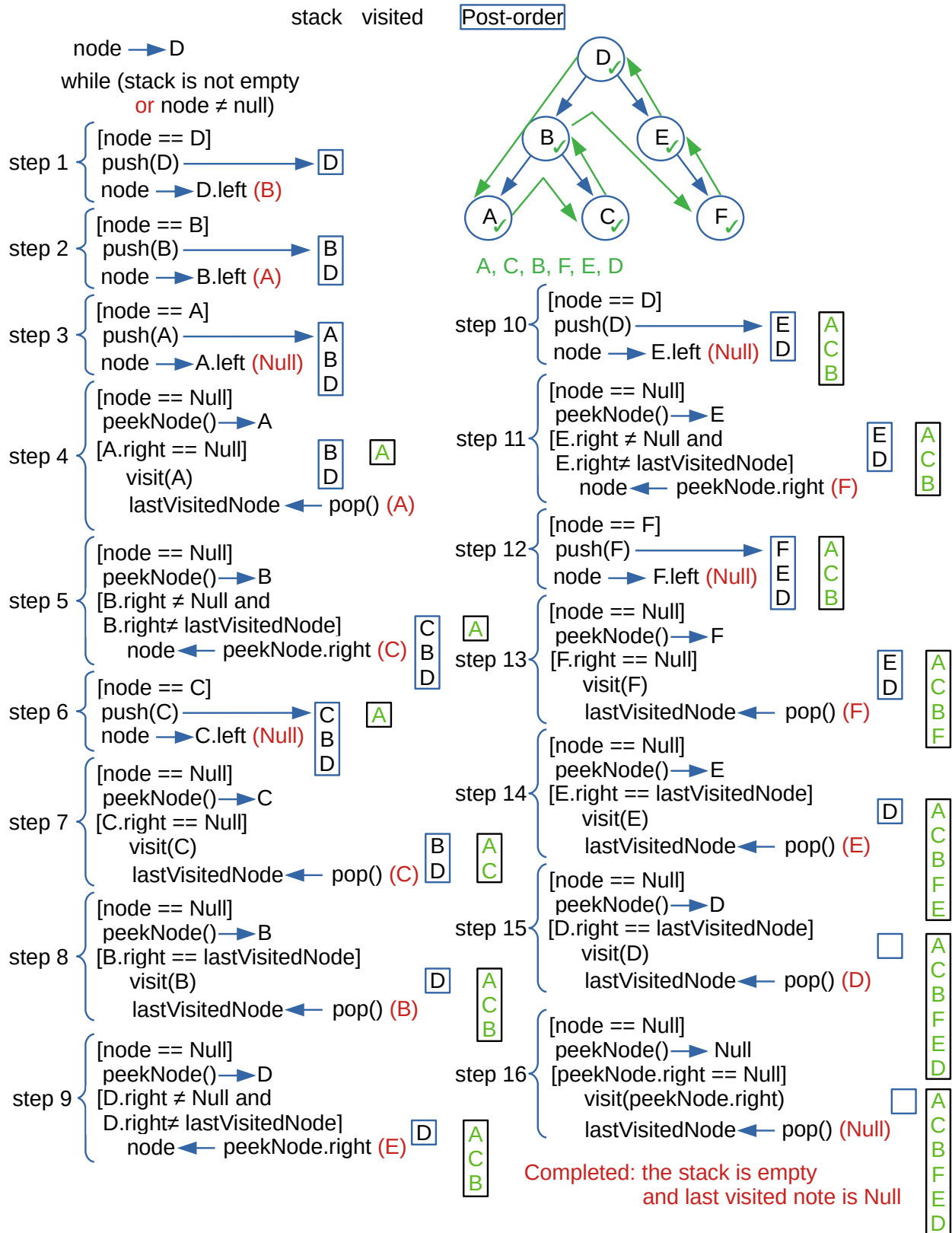


Figure 56: Post-order iterative implementation example.

B | COMPLEXITY OF DATA STRUCTURES FOR GRAPHS

In the following table are summarized the complexities of the most common operations on graphs for the different data structures [27].

C

IMPLEMENTATION OF GRAPH REPRESENTATION

In the following python code there is the implementation for the all the different graph representation, and the operation of insertion of nodes and edges.

Listing C.1: Graph representation and fundamental operations.

```
1 class Node():
2
3     def __init__(self, value):
4         self.value = value
5         self.edges = []
6
7 class Edge():
8
9     def __init__(self, value, node_from, node_to):
10         self.value = value
11         self.node_from = node_from
12         self.node_to = node_to
13
14 class Graph():
15
16     def __init__(self, nodes=[], edges=[]):
17         self.nodes = nodes
18         self.edges = edges
19
20     def insert_node(self, new_node_value):
21         new_node = Node(new_node_value)
22         self.nodes.append(new_node)
23
24     def insert_edge(self, new_edge_val, node_from, val, node_to_val):
25         from_found = None
26         for node in self.nodes:
27             if node_from_val == node.value:
28                 from_found = node
29             if node_to_val == node.value:
30                 to_found = node
31             if from_found == None:
32                 from_found = Node(node_from_val)
33                 self.nodes.append(from_found)
34             if to_found == None:
35                 to_found = Node(node_to_val)
36                 self.nodes.append(to_found)
37         new_edge = Edge(new_edge_val, from_found, to_found)
38         from_found.edges.append(new_edge)
39         to_found.edges.append(new_edge)
40         self.edges.append(new_edge)
41
42     def get_edge_list(self):
43         get_list = []
```

```

44         for edge_object in self.edges:
45             edge = (edge_object.value, edge_object.node_from.value,
46                     edge_object.node_to.value)
47             edge_list.append(edge)
48         return edge_list
49
50     def get_adjacency_list(self):
51         max_index = self.find_max_index()
52         adjacency_list = [None]*(max_index + 1)
53         for edge_object in self.edges:
54             if adjacency_list[edge_object.node_from.value]:
55                 adjacency_list[edge_object.node_from.value].
56                     append((edge_object.node_to.value,
57                             edge_object.value))
58             else:
59                 adjacency_list[edge_object.node_from.value] = [(
60                     edge_object.node_to.value, edge_object.value
61                     )]
62         return adjacency_list
63
64     def get_adjacency_matrix(self):
65         max_index = self.find_max_index()
66         adjacency_matrix = [[0 for i in range(max_index + 1)] for j
67                             in range(max_index + 1)]
68         for edge_object in self.edges:
69             adjacency_matrix[edge_object.node_from.value][
70                 edge_object.node_to.value] = edge_object.value
71         return adjacency_matrix
72
73     def find_max_index(self):
74         max_index = 1
75         if len(self.nodes):
76             for node in self.nodes:
77                 if node.value > max_index:
78                     max_index = node.value
79         return max_index

```

BIBLIOGRAPHY

- [1] Wikipedia. *Computational complexity*. URL: https://en.wikipedia.org/wiki/Computational_complexity.
- [2] Wikipedia. *Big O notation*. URL: https://en.wikipedia.org/wiki/Big_O_notation.
- [3] Wikipedia. *Recursion (computer science)*. URL: [https://en.wikipedia.org/wiki/Recursion_\(computer_science\)](https://en.wikipedia.org/wiki/Recursion_(computer_science)).
- [4] Wikipedia. *Data structure*. URL: https://en.wikipedia.org/wiki/Data_structure.
- [5] Wikipedia. *Data type*. URL: https://en.wikipedia.org/wiki/Data_type.
- [6] Wikipedia. *Abstract data type*. URL: https://en.wikipedia.org/wiki/Abstract_data_type.
- [7] Wikipedia. *Collection (abstract data type)*. URL: [https://en.wikipedia.org/wiki/Collection_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Collection_(abstract_data_type)).
- [8] Wikipedia. *Array data structure*. URL: https://en.wikipedia.org/wiki/Array_data_structure.
- [9] Wikipedia. *Linked list*. URL: https://en.wikipedia.org/wiki/Linked_list.
- [10] Wikipedia. *Doubly linked list*. URL: https://en.wikipedia.org/wiki/Doubly_linked_list.
- [11] Wikipedia. *Stack (abstract data type)*. URL: [https://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)).
- [12] Wikipedia. *Queue (abstract data type)*. URL: [https://en.wikipedia.org/wiki/Queue_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Queue_(abstract_data_type)).
- [13] Wikipedia. *Set*. URL: [https://en.wikipedia.org/wiki/Set_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Set_(abstract_data_type)).
- [14] Wikipedia. *Hash map*. URL: https://en.wikipedia.org/wiki/Associative_array.
- [15] Wikipedia. *Hash table*. URL: https://en.wikipedia.org/wiki/Hash_table.
- [16] Wikipedia. *Hash function*. URL: https://en.wikipedia.org/wiki/Hash_function.
- [17] Wikipedia. *Search algorithm*. URL: https://en.wikipedia.org/wiki/Search_algorithm.
- [18] Wikipedia. *Sorting algorithm*. URL: https://en.wikipedia.org/wiki/Sorting_algorithm.

- [19] Wikipedia. *Binary search algorithm*. URL: https://en.wikipedia.org/wiki/Binary_search_algorithm.
- [20] Wikipedia. *Bubble sort*. URL: https://en.wikipedia.org/wiki/Bubble_sort.
- [21] Wikipedia. *Merge sort*. URL: https://en.wikipedia.org/wiki/Merge_sort.
- [22] Wikipedia. *Quicksort*. URL: <https://en.wikipedia.org/wiki/Quicksort>.
- [23] Wikipedia. *Heapsort*. URL: <https://en.wikipedia.org/wiki/Heapsort>.
- [24] Wikipedia. *Quicksort python implementation*. URL: <https://www.educative.io/edpresso/how-to-implement-quicksort-in-python>.
- [25] Wikipedia. *Trees*. URL: [https://en.wikipedia.org/wiki/Tree_\(data_structure\)](https://en.wikipedia.org/wiki/Tree_(data_structure)).
- [26] Wikipedia. *Binary search tree*. URL: https://en.wikipedia.org/wiki/Binary_search_tree.
- [27] M.T. Goodrich, R. Tamassia, and M.H. Goldwasser. *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated, 2013. ISBN: 9781118476734. URL: <https://books.google.it/books?id=2UccAAAAQBAJ>.
- [28] Wikipedia. *Heap*. URL: [https://en.wikipedia.org/wiki/Heap_\(data_structure\)](https://en.wikipedia.org/wiki/Heap_(data_structure)).
- [29] Wikipedia. *Self-balancing binary search tree*. URL: https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree.
- [30] Wikipedia. *Red-Black tree*. URL: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree.
- [31] Wikipedia. *Graph*. URL: [https://en.wikipedia.org/wiki/Graph_\(discrete_mathematics\)](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)).
- [32] Wikipedia. *Connectivity*. URL: [https://en.wikipedia.org/wiki/Connectivity_\(graph_theory\)](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)).
- [33] Wikipedia. *Trees traversal*. URL: https://en.wikipedia.org/wiki/Tree_traversal.