Data Structures & Algorithms with C++
for Students in Computer Science

Hubert Cecotti

Fall 2020 (v0.04)

Department of Computer Science
College of Science and Mathematics
FRESNO STATE

Contents

Abstract

The goal of this document is to provide some notes about the main definitions and the main algorithms related to the key data structures that are used in computer science. It includes: arrays, queues, stacks, lists (simple chained, double chained, circular, skip), hash tables, binary trees, ...
This document contains examples in C++ for the development of the main data structures that can be covered during an undergraduate course related to data structures in computer science. The chosen implementation for the different algorithms stresses the readability aspect. It may not represent proper practices for industrial codes, but it includes relevant algorithmic practices with some software engineering practices. In addition, the presented code is aimed at being presented in a document, therefore the choice of the name for the variables should be changed to respect industrial standards and conventions. It is worth noting that data structures such as queues and stacks are readily available in C++. Therefore, this document is for **instructional purposes**, highlighting some key functionalities of C++. It does not aim at presenting all the existing data structures can be found in the Standard Template Library (STL).
Warning: This book uses C++ to present state of the art data structures. It does NOT focus on object oriented programming concepts.
You should keep this document to get prepared for job interviews.

Contact: hcecotti@csufresno.edu

CSCi 115 - Course outline

The outline corresponds to the list topics covered in each class, with 3 classes per week. The order of the classes does not represent the order of the chapters and sessions in the book. Some advanced data structures are covered at the end of the course, so important elements that are necessary for the projects are covered in advance.

1. Syllabus introduction - outline and advice
2. C++ implementation guideline for labs and classes.
3. Examples in C++
4. Complexity definitions + Hospital rule
5. Recursivity (Fibonacci + Hanoi tower)
6. Divide and conquer + Master Theorem
7. Time measurements in C++ for performance evaluation in labs.
8. Arrays: insertion/deletion/creation/search
9. Arrays: Selection sort, insert sort, bubble sort
10. Arrays: mergesort and quicksort
11. Queues and Stacks
12. Lists: definition + simple chained
13. Lists: double chained
14. Lists: circular lists
15. Skip lists and applications
16. Hash tables - Definition + Keys
17. Hash Tables Part - Probe functions
18. Sets
19. Trees - Definitions
20. BST: invert tree, insert, delete, search
21. Midterm 1 - revision
22. AVL trees (rotations)
23. AVL trees (insert/delete)
24. 2-3 trees
25. Heaps + Heap sort

26. Definition of Graphs
27. Main graphs problems
28. Graph traversals (BFS)
29. Graph traversals (DFS + Parenthesis)
30. Dynamic programming
31. Greedy algorithms
32. Shortest paths methods - Single source
33. Shortest paths methods - All pairs
34. Minimum spanning tree (Kruskal and Primm)
35. Minimum spanning tree (implementation) + Discussion about Midterm 2
36. B-trees (part 1)
37. B-trees (part 2)
38. Quadtree in images, Octree in 3D
39. Read and Black trees (part 1)
40. Read and Black trees (part 2)
41. Fibionacci heap
42. Questions and answers related to the project
43. Preparation for the finals

Chapter 1

Introduction

Contents

1.1 Symbols

Symbols are used in mathematical and logical expressions. They can also be used in pseudo-code to represent variables. For a better understanding, it is critical to know how to pronounce these different symbols, to establish a direct connection in your mind between their visual representation and their sound when you read them.

Greek letters

Lower case symbols

Greek symbol	α	β	δ	ϵ	ϕ	φ	γ	η	ι	κ
English	alpha	beta	delta	epsilon	phi	phi	gamma	eta	iota	kappa
Greek symbol	λ	μ	ν	π	θ	ρ	σ	τ	υ	ω
English	lambda	mu	nu	pi	theta	rho	sigma	tau	upsilon	omega
Greek symbol	ξ	ψ	ζ							
English	xi	psi	zeta							

Upper case symbols

Greek symbol	Δ	Φ	Γ	Λ	Π	Θ	Σ	Υ	Ω	Ξ	Ψ
English	delta	phi	gamma	lambda	pi	theta	sigma	upsilon	omega	xi	psi

Some useful symbols

- $\exists$ There exists...
- $\exists!$ There exists a unique ...
- $\nexists$ There does not exist ...
- $\forall$ For all...
- $\in$ belongs (example: $x \in X$, x belongs to X).
- $\notin$ does not belong (example: $x \notin X$, x does not belong to X).
- ∞ infinity
- $\emptyset$ empty set (the shape can be more rounded).

1.2 Sets

To define the elements in a set, we use curly brackets: { }. These brackets should not be confused with the beginning and end of an instruction block in C++, or the definition of a static array in C++. Example: $A = \{1, 2, 3\}$. The set A contains the elements 1, 2, and 3.

We consider 2 sets A and B.

- A is a subset of B: $A \subseteq B$

$$\forall a(a \in A \rightarrow a \in B) \tag{1.1}$$

- A is not a subset of B: $A \nsubseteq B$.

$$\exists a | a \in A \wedge a \notin B \tag{1.2}$$

- B is a superset of A: $B \supseteq A$

$$\forall a(a \in A \rightarrow a \in B) \tag{1.3}$$

- B is a not superset of A: $B \nsupseteq A$

$$\exists a | a \in A \wedge a \notin B \tag{1.4}$$

- If A is a subset of B but A is not equal to B, then A is a proper subset of B: $A \subset B$.

$$(\forall a\ ,\ a \in A \rightarrow a \in B) \wedge (\exists b \in B | b \notin A) \tag{1.5}$$

- If B is a superset of A but B is not equal to A, then B is a proper superset of A: $B \supset A$.

$$(\forall a\ ,\ a \in A \rightarrow a \in B) \wedge (\exists b \in B | b \notin A) \tag{1.6}$$

- A is not a proper subset of B: $A \not\subset B$

$$(\exists a \in A | a \notin B) \vee (\forall b \in B | b \in A) \tag{1.7}$$

- B is not a proper superset of A: $B \not\supset A$.

$$(\exists a \in A | a \notin B) \vee (\forall b \in B | b \in A) \tag{1.8}$$

- Equality, both sets of the same elements: $A = B$

$$(\forall a\ ,\ a \in A \rightarrow a \in B) \wedge (a \notin A \rightarrow a \notin B) \tag{1.9}$$

- Complement, the set of elements that do not belong to A: $C = A'$

$$C \quad = \quad \{c | c \notin A\} \tag{1.10}$$

- Union: $C = A \cup B$ (or)

$$C \quad = \quad \{c | c \in A \vee c \in B\} \tag{1.11}$$

- Intersection: $C = A \cap B$ (and)

$$C = \{c | c \in A \wedge c \in B\} \quad (1.12)$$

- Relative complement, objects that belong to A and not to B: $C = A - B$

$$C = \{a | a \in A \wedge a \notin B\} \quad (1.13)$$

- Symmetric difference, objects that belong to A or B but not to their intersection: $C = A \Delta B$

$$C = \{c | (c \in A \wedge c \notin B) \vee (c \notin A \wedge c \in B)\} \quad (1.14)$$

- Cardinality, number of elements in the set A: $|A|$

Here, we define some common sets:

- Natural numbers with 0: $\mathbb{N}_0 = \{0, 1, 2, 3, 4, ...\}$
- Natural numbers without zero: $\mathbb{N}_1 = \{1, 2, 3, 4, 5, ...\}$
- Integer numbers set: $\mathbb{Z} = \{... - 3, -2, -1, 0, 1, 2, 3, ...\}$
- Rational numbers set: $\mathbb{Q} = \{x | x = a/b, (a, b) \in \mathbb{Z}^2 \text{ and } b \neq 0\}$
- Real numbers set: $\mathbb{R} = \{x | -\infty < x < +\infty\}$
- Complex numbers set: $\mathbb{C} = \{z | z = a + bi, -\infty < a < +\infty, -\infty < b < +\infty, i = \sqrt{-1}\}$

1.3 Definitions

1.3.1 Asymptotic notation

Let $f(n)$ and $g(n)$ be two functions that map positive integers to positive real numbers. n is the main parameter of the function. Example: f is an algorithm to sort an array of size n, $f(n)$ represents the number of "operations" required to sort the array of size n.

- Θ: Big Theta, asymptotically tight bound. $f(n)$ is $\Theta(g(n))$ (or $f(n) \in \Theta(g(n))$) if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

$$\Theta(g(n)) = \{f(n) : \exists\{c_1, c_2, n_0\} | 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0\} \tag{1.15}$$

- O: Big O, asymptotic upper bound.

$$O(g(n)) = \{f(n) : \exists\{c, n_0\} | 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\} \tag{1.16}$$

- Ω: Big omega, asymptotic lower bound

$$\Omega(g(n)) = \{f(n) : \exists\{c, n_0\} | 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\} \tag{1.17}$$

- o: little o, upper bound, not asymptotically tight.

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 | 0 \leq f(n) < c \cdot g(n) \forall n \geq n_0\} \tag{1.18}$$

- ω: little omega, lower bound, not asymptotically tight.

$$\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 | 0 \leq c \cdot g(n) < f(n) \forall n \geq n_0\} \tag{1.19}$$

Given the definitions of Big Θ, Big O, small o, Big Ω, and little ω, it is possible to resume their relationship in the following table:

Table 1.1: Definitions summary.

Notation	? $c > 0$	? $n_0 \geq 1$	$f(n)$? $c \cdot g(n)$
$O()$	$\exists$	$\exists$	$\leq$
$o()$	$\forall$	$\exists$	$<$
$\Omega()$	$\exists$	$\exists$	$\geq$
$\omega()$	$\forall$	$\exists$	$>$

Examples

It is important to know the meaning of the Big O notations in relation to their meaning. It is important to separate the notion of complexity from the notion of "how long it takes" because O(1) that is constant may take a very long time but independent of n: no matter how much time it takes to do it, it n increases, it will still be the same amount of time.

- $O(1)$: constant.
- $O(log(n))$: logarithmic (examples: finding an item in a sorted array with a binary search or a balanced search tree).

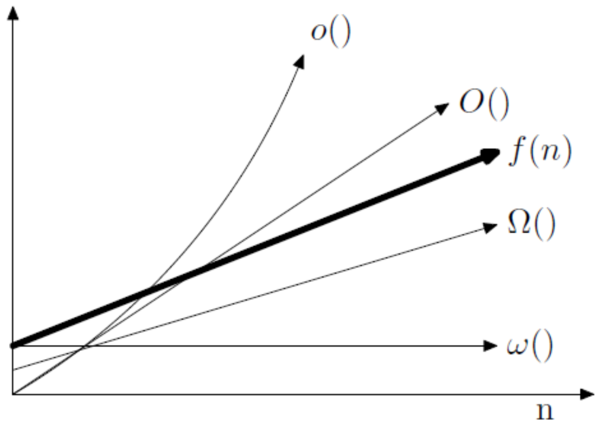

Figure 1.1: Relationships between the notations.

- $O(n)$: linear (examples: finding an item in an unsorted list or in an unsorted array).
- $O(n \cdot log(n))$: log-linear (example: mergesort).
- $O(n^2)$: quadratic (examples: selection sort and insertion sort).

$\forall n > 0$, and $c > 0$ we have: $n^{c+1} > n^c$, $n > log(n)$, $n \cdot log(n) > log(n) \cdot log(n)$.

$$\begin{aligned} T(n) &= 5n^3 + 3n \cdot log(n) + 2n \\ &\leq 5n^3 + 5n \cdot log(n) + 5n \\ &\leq 5n^3 + 5n^3 + 5n^3 \\ &\leq 15n^3 \\ &= O(n^3) \end{aligned}$$

1.3.2 Divide and conquer

The divide and conquer principle has 3 main steps that are resumed here:

1. Divide the problem into a number of sub-problems that are smaller instances of the same problem.
2. Conquer the sub-problems by solving them recursively.
   - If the sub-problems are large enough to solve recursively, solve the recursive case.
   - If the sub-problem sizes are small enough, solve the base case (solve the sub-problems in a straightforward manner).
3. Combine the solutions to the sub-problems into the solution for the original problem.

1.3.3 Loop invariant

Determining loop invariant is critical for your understanding of the algorithms and to select wisely where the code should be placed. While loops are basic structures in computer science, they must remain well thought and their goals should be clear. For sorting algorithms and other algorithms with nested loops, it is necessary to determine what will change after the execution of a loop and what set of statements will remain true.

1. Initialization: It is true prior to the first iteration of the loop.
2. Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
3. Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

1.3.4 Master theorem

The Master theorem is defined as follows:

$$T(n) \quad = \quad aT(n/b) + f(n) \tag{1.20}$$

with

- n: size of an input problem (e.g. number of elements in an array).
- a: number of sub-problems, with $a \geq 1$.
- n/b: size of the sub problem, with $b > 1$.
- $f(n)$: Divide + Combine operations. $f(n)$ is an asymptotically positive function.

It means that for $T(n)$, we will do a times T with an input of size n/b, and each time we will run T we will have to run the function f as well.

The critical exponent is defined by:

$$c_{crit} \quad = \quad log_b(a) \tag{1.21}$$

In relation to the balance between $aT(n/b)$ and $f(n)$, we can distinguish 3 cases. In the first case, the first part will be more important, in the second case they have equal importance, and in the third case, the seconde part will be more important.

Case 1

Recursion tree is leaf heavy: $aT(n/b) > f(n)$.
If $f(n) = O(n^c)$ where $c < c_{crit}$ then $T(n) = \Omega(n^{c_{crit}})$.

Case 2

Split/recombine, same as sub-problems: $aT(n/b) = f(n)$.
If $f(n) = \Omega(n^{c_{crit}} \cdot log^k(n)) \forall k \geq 0$ then $T(n) = \Omega(n^{c_{crit}} \cdot log^{k+1}(n))$

Case 3

Recursion tree is root heavy: $aT(n/b) < f(n)$.
When $f(n) = \Omega(n^c)$ where $c > c_{crit}$ and $a \cdot f(n/b) \leq k \cdot f(n)$ for a constant $k < 1$ and n large enough then it is dominated by the splitting term $f(n)$, so $T(n) = \Omega(f(n))$.

Remark: the following expression must remain polynomial

$$\frac{f(n)}{n^{c_{crit}}} \tag{1.22}$$

Therefore:

$$\frac{n/log(n)}{n} \quad = \quad \frac{1}{log(n)} \tag{1.23}$$

will not work with the Master Theorem.

Examples

Table 1.2 presents examples related to the master theorem with the 3 cases, and examples where any of the 3 cases can be applied.

Table 1.2: Examples for case 1, 2, and 3.

$T(n)$	a	b	Case	Notation
$16T(n/4) + n$	16	4	1	$\Theta(n^2)$
$3T(n/2) + n$	3	2	1	$\Theta(n^{log_2(3)})$
$3T(n/3) + \sqrt{n}$	3	3	1	$\Theta(n)$
$4T(n/2) + cn$	4	2	1	$\Theta(n^2)$
$4T(n/2) + n/logn$	4	2	1	$\Theta(n^2)$
$4T(n/2) + logn$	4	2	1	$\Theta(n^2)$
$\sqrt{2}T(n/2) + logn$	$\sqrt{2}$	2	1	$\Theta(\sqrt{n})$
$4T(n/2) + n^2$	4	2	2	$\Theta(n^2 \cdot log(n))$
$2T(n/2) + nlogn$	2	2	2	$\Theta(n \cdot log^2(n))$
$3T(n/3) + n/2$	3	3	2	$\Theta(n \cdot log(n))$
$T(n/2) + 2^n$	1	2	3	$\Theta(2^n)$
$3T(n/2) + n^2$	3	2	3	$\Theta(n^2)$
$2T(n/4) + n^{0.51}$	2	4	3	$\Theta(n^{0.51})$
$3T(n/4) + nlogn$	4	4	3	$\Theta(n \cdot log(n))$
$6T(n/3) + n^2logn$	6	3	3	$\Theta(n^2 \cdot log(n))$
$7T(n/3) + n^2$	7	3	3	$\Theta(n^2)$
$16T(n/4) + n!$	16	4	3	$\Theta(n!)$
$2^nT(n/2) + n^n$	2^n	2	NO	a: not a constant
$2T(n/2) + n/logn$	2	2	NO	$f(n)$: not polynomial
$0.5T(n/2) + 1/n$	0.5	2	NO	$a < 1$
$64T(n/8) - n^2logn$	64	8	NO	$f(n)$ not positive

1.3.5 Time complexity

This subsection presents the time complexity of the main algorithms that are present in this book. The time complexity using the Big O notation for the main sorting algorithms is presented in Table 1.3. It is worth noting the bad performance in the worst case of the Quicksort as well as the good performance in the best case for the insertion sort and the bubble sort. The comparison between dynamic arrays and linked lists is given in Table 1.4. The time complexity for the different types of tree is given in Table 1.5. Finally, the time complexity for the Fibonacci heap is given in Table 1.6. These algorithms will be presented in the subsequent chapters.

Table 1.3: Sorting algorithms

Algorithm	Best case	Average case	Worst case
Quicksort	$n \cdot log(n)$	$n \cdot log(n)$	n^2
Mergesort	$n \cdot log(n)$	$n \cdot log(n)$	$n \cdot log(n)$
Insertion	n	n^2	n^2
Selection	n^2	n^2	n^2
Bubble sort	n	n^2	n^2
Heap sort	$n \cdot log(n)$	$n \cdot log(n)$	$n \cdot log(n)$

Table 1.4: Array vs. List

Data structure	Indexing	Insert/Delete		
		Beginning	Middle	End
Dynamic array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	search + $\Theta(1)$

Table 1.5: Trees in O notation

Data structure	Space		Search		Insert		Delete	
	Average	Worst	Average	Worst	Average	Worst	Average	Worst
Skip list	n	$n \cdot log(n))$	$log(n)$	n	$log(n)$	n	$log(n)$	n
Binary Search Tree	n	n	$log(n)$	n	$log(n)$	n	$log(n)$	n
AVL Tree	n	n	$log(n)$	$log(n)$	$log(n)$	$log(n)$	$log(n)$	$log(n)$
B Tree	n	n	$log(n)$	$log(n)$	$log(n)$	$log(n)$	$log(n)$	$log(n)$
Red-Black Tree	n	n	$log(n)$	$log(n)$	$log(n)$	$log(n)$	$log(n)$	$log(n)$

Table 1.6: Heap (average case)

Data structure	Insert	Find min	Delete min	Decrease key	Decrease key
Fibonacci heap	$\Theta(1)$	$\Theta(1)$	$O(log(n))$	$\Theta(1)$	$\Theta(1)$

1.4 C++ examples

This section is about the C++ programming language. It will certainly not replace a book about C++ and all its specifications. The book defining the specifications of C++ 17 is around 1600 pages. It is worth noting that the present document stresses implementation using C++ and not C. Many algorithms are present in the Standard Template Library (STL), in algorithms, which defines a collection of functions especially designed to be used on ranges of elements. The present document will just highlight a few elements that are used in the next chapters to better understand the implementation of the data structures.

In the next chapters, we consider the definition of typedef double MyType. The goal is to avoid the definition of template in the code that is heavy, and to stress the type of the values that are manipulated in the data structures.

1.4.1 Expression value categories

The C++17 standard defines the following expression value categories:

- A **glvalue** is an expression whose evaluation determines the identity of an object, bit-field, or function.
- A **prvalue** is an expression whose evaluation initializes an object or a bit-field, or computes the value of the operand of an operator, as specified by the context in which it appears.
- An **xvalue** is a glvalue that denotes an object or bit-field whose resources can be reused, usually because it is near the end of its lifetime. For instance, some kinds of expressions involving rvalue references yield xvalues, such as a call to a function whose return type is an rvalue reference or a cast to an rvalue reference type.
- An **lvalue** is a glvalue that is not an xvalue.
- An **rvalue** is a prvalue or an xvalue.

Algorithm 1.1: Expression values.

```
void main() {
  const int i = 12;
    int j,l = 12;
    // i = 13; // cannot change i because it was defined as a const
    j = i; // j can change its value as it s not a const
    // a literal such as 56 is a prvalue
    // 56 = i; // it does not work
    j = 56; // it works
    // left side is an expression BUT it returns an lvalue ( j or l)
    ((j < 30) ? j : l) = 7;
    // It does not work as it  returns an evaluated expression j*2 or l*2
    // ((j < 30) ? j*2 : l*2) = 7;
}
```

1.4.2 NULL vs. nullptr

The C++09 nullptr keyword represents an rvalue constant that serves as a universal null pointer literal. It replaces the weakly-typed literal 0 and the infamous NULL macro. 0x or NULL is therefore replaced by nullptr. nullptr is a pointer literal of type std::nullptr_t, it is a prvalue.

1.4.3 Include

The using namespace std is to not use std:: in front of cout and other methods of classes that are in the std namespace. Namespaces allow to cluster named entities that would have global scope into narrower scopes otherwise, giving them namespace scope. It provides a way of organizing the elements of programs into different logical scopes referred to by names. A namespace is a declarative region that gives a scope to the identifiers (names of the types, function, variables,...) inside it.

Algorithm 1.2: Includes.

```
using namespace std;
#include <chrono>    // to use functions related to the time
#include <string>    // to use strings
#include <tuple>     // to use tuple
#include <iostream>  // to use cout, cin
#include <fstream>   // to use files
#include <algorithm> // to use max, min
#include <stdlib.h>  // to use rand
#include <memory>    // to use unique_ptr
```

1.4.4 Types

Algorithm 1.3: Type conversion and promotion.

```
void main() {
    int i = 5;
    int j = 2;
    double d = 2.1;
    cout << i + d << endl; // 7.1 (double)
    cout << i* d << endl;  // 10.5 (double)
    cout << d* i << endl;  // 10.5 (double)
    cout << i / d << endl; // 2.38095 (double)
    cout << d / i << endl; // 0.42 (double)
    cout << i / j << endl; // 2 (int)
}
```

1.4.5 Arrays

For static arrays, you can specify the maximum size between square brackets. You can determine the elements contained in the array using curly brackets. For dynamic arrays, you must use the keyword **new** to allocate the memory corresponding to the number of elements that are needed. When the array is not needed anymore you must use the keyword **delete[]** to deallocate the array from the memory. In the example, the array primes contains 7 elements and has the memory allocated for these elements, while the array primes1 has only 5 elements in it but 10 elements are allocated.

Algorithm 1.4: Static and dynamic arrays.

```
void main() {
  // Static array
    int s[5]; // 5 elements only, cannot be changed
    // primes: 7 elements only
    int primes[] = { 1, 2, 3, 5, 7, 11, 13 };
    // primes1: 10 elements, 5 first elements are initialized
    int primes1[10] = { 1, 2, 3, 5, 7 };
    // Dynamic array
    int n = 10;
    int *x = new int[n]; // an array of n int
    cout << "Number of elements in primes: ";
    cout << sizeof(primes) / sizeof(int) << endl; // 7
    cout << "Number of elements in primes1: ";
    cout << sizeof(primes1) / sizeof(int) << endl; // 10
    cout << "Number of elements in x: ";
    cout << sizeof(*x)/sizeof(double) << endl; // 1
    delete[] x;  // delete the array x
}
```

1.4.6 Functions

Variables

In this example, you can see different types of function. f1 and f2 are functions that return nothing (void), while f3 is a function that return an int*. In f1, a and n are just as input. In f2, a** and n are inputs, but modify the link to *a which is **a, so *a is an output. In f3, n is the input and the function returns a as an output.

Algorithm 1.5: C++ examples.

```
// void: no return keyword
void f1(int* a,int n) {
    a = new int[n];
    for (int i = 0; i < n; i++)
        a[i] = i;
}
// void: no return keyword
void f2(int** a, int n) {
    *a = new int[n];
    for (int i = 0; i < n; i++)
        (*a)[i] = i;
}
// f3 must return an int*
int* f3(int n) {
    int *a = new int[n];
    for (int i = 0; i < n; i++)
        a[i] = i;
    return a;
}
void DisplayArray(int* a, int n) {
    for (int i = 0; i < n; i++)
        cout << a[i] << ",";
    cout << endl;
}
```

Algorithm 1.6: Main test functions.

```
void main() {
    int n = 10;
    int* a =  nullptr;
    cout << "Evaluate f1" << endl;
    f1(a,n);
    //DisplayArray(a,n); // not working because a has nothing !
    cout << "Evaluate f2" << endl;
    f2(&a,n);
    DisplayArray(a, n);
    cout << "Evaluate f3" << endl;
    a =  nullptr;
    a=f3(n);
    DisplayArray(a, n);
}
```

Arrays

Reference versus Pointers for arrays. The following examples illustrate the differences between x++ and ++x.

Algorithm 1.7: C++ functions.

```
int f0(int x) {
    return x++; // done after the return
}
int f1(int x) {
    return ++x; // done before the return
}
void f2(int* x) {
    cout << x << " " << *x << endl;
    // 00AFFB3C 6
    *x++;
    cout << x << " " << *x << endl;
    // 00AFFB40 -858993460
}
void f3(int* x) { // Passing by pointer

    (*x)++;
}
void f4(int& x) { // Passing by reference
    x++;
}
```

Algorithm 1.8: C++ function calls.

```
int main() {
    int x = 6;
    cout << f0(x) << endl; // x=6, f0(x) returns 6
    cout << f1(x) << endl; // x=6, f1(x) returns 7
    f2(&x); // x=6, after f2, x=6
    cout << x << endl; // x=6
    f3(&x); // x=6, after f3, x=7
    cout << x << endl; // x=7
    f4(x); // x=7, after f4, x=8
    cout << x << endl; // x=8
}
```

It is not because you have a * that you pass the variable as input/output, you need to consider the type of the variable you are manipulating. In the following examples, we consider an array of int v of size n. In the function f8, we consider v as an integer.

Algorithm 1.9: C++ functions.

```cpp
void f5(int** v, int n) { // Passing by pointer a variable v of type int*
    *v = new int[n];
    for (int i = 0; i < n; i++)
        (*v)[i] = i;
}
void f6(int* &v,int n) { // Passing by reference a variable v of type int*
    v = new int[n];
    for (int i = 0; i < n; i++)
        v[i] = i;
}
void f7(int* v, int n) { // Passing by value a variable v of type int*
    v = new int[n];
    for (int i = 0; i < n; i++)
        v[i] = i;
}
void f8(int* v) { // Passing by pointer a variable v of type int
    *v = 12;
}
void f9(int &v) { // Passing by reference a variable v of type int
    v = 15;
}
```

The functions f5, f6, f7, and f8 are used with different inputs. You can notice that in f7, v as an int* has not been modified within the function.

Algorithm 1.10: C++ function calls.

```cpp
void main() {
    int *v1= nullptr,*v2= nullptr,*v3= nullptr;
    int v4,v5;
    f5(&v2, 5);
    f6(v1,5);
    f7(v3,5);
    f8(&v4,5);
    f9(v4,6);
    cout << v1[3] << endl; // 3
    cout << v2[3] << endl; // 3
    cout << v3[3] << endl; // read access violation
    cout << v4; // 12
    cout << v5; // 15
}
```

1.4.7 Sum and product

It is important to implement computations and to represent formally a piece of code. In the following examples, you have the description of equations and the corresponding evaluation for a particular value of n. Note the initialization of x to 0 for a sum ($\sum$) and to 1 for a product ($\prod$). As the iterator i is an integer, you have to change its type when it is evaluated when the expected output is a real number, i.e. a variable of type float or double. Summing the values in an array in C++, the values will range from $i = 0$ (index of the first element) to $i = n - 1$ (index of the last element, where n is the size of the array).

Algorithm 1.11: Variables declaration.

```
int n = 10;
  double x;
  double* a = new double[n];
  double* b = new double[n];
  for (int i = 0; i < n; i++) {
      a[i] = (2*(double)i+1)/n;
      b[i] = (5* (double)i+2)/n;
  }
```

Sum 1: The sum of all the i, from $i = 1$ to $i = n$.

$$x = \sum_{i=1}^{n} i \tag{1.24}$$

Sum 2: The sum of all the i/n, from $i = 1$ to $i = n$.

$$x = \sum_{i=1}^{n} 1/i \tag{1.25}$$

Sum 3: The sum of all the $a(i) * b(i)$, with values of i ranging from $i = 0$ to $i = n - 1$.

$$x = \sum_{i=0}^{n-1} a(i) \cdot b(i) \tag{1.26}$$

Product 1: The product of all the i, from $i = 1$ to $i = n$.

$$x = \prod_{i=1}^{n} i \tag{1.27}$$

Product 1: The product of all the $a(i) * b(i)$, from $i = 0$ to $i = n - 1$.

$$x = \prod_{i=0}^{n-1} a(i) \cdot b(i) \tag{1.28}$$

Algorithm 1.12: Sum and product.

```
    // Sum 1
    x = 0.0;
    for (int i = 1; i <= n; i++)
        x += i;
    cout << x << endl; // 55
    // Sum 2a
    x = 0.0;
    for (int i = 1; i <= n; i++)
        x += 1/i;
    cout << x << endl; // 1
    // Sum 2b
    x = 0.0;
    for (int i = 1; i <= n; i++)
        x += 1.0 / i;
    cout << x << endl; // 2.93
    // Sum 2c
    x = 0.0;
    for (int i = 1; i <= n; i++)
        x += 1/(double)i;
    cout << x << endl; // 2.93
    // Sum 3
    x = 0.0;
    for (int i = 0; i < n; i++)
        x += a[i]*b[i];
    cout << x << endl; // 32.8
    // Product 1
    x = 1.0;
    for (int i = 1; i <= n; i++)
        x *= i;
    cout << x << endl; // 3.63e+06
    // Product 2
    x = 1.0;
    for (int i = 0; i < n; i++)
        x *= a[i]*b[i];
    cout << x << endl; // 26
}
```

This example illustrates the notion of reference, and pointer arithmetic.

Algorithm 1.13: C++ examples.

```
void main() {
  // Access by the reference
    int x = 25;
    int &y = x;
    y = 12;
    cout << "x:" << x << endl;
    cout << "y:" << x << endl;
    // Increment example
    int i = 3;
    cout << i++ << endl;
    int j = 3;
    cout << ++j << endl;
    // Pointer shifting example
    int n = 10;
    int* a1 = new int[n];
    for (int i = 0; i < n; i++)
        a1[i] = i * 2;
    int* b1 = a1;
    cout << b1[2] << endl;
    cout << *(b1+3) << endl;
}
```

1.4.8 No break, no continue!

This examples illustrates different ways to find an element in an array, with a for loop using a break, a while loop, and a do while loop. In the remaining part of this document, there will be no break or continue.

Algorithm 1.14: For loop with break.

```
void Search_v01(int* a, int n,int x) {
    bool find = false;
    int argx = -1;
    for (int i = 0; i < n; i++) {
        if (a[i] > x) {
            find = true;
            argx = i;
            break;
        }
    }
    if (find)
        cout << "Found " << a[argx] << endl;
    else
        cout << "Not found" << endl;
}
```

Algorithm 1.15: While loop.

```
void Search_v02(int* a, int n, int x) {
    bool find = false;
    int argx = -1, i = 0;
    while ((!find) && (i < n)) {
        if (a[i] > x) {
            find = true;
            argx = i;
        }
        i++;
    }
    if (find)
        cout << "Found " << a[argx] << endl;
    else
        cout << "Not found" << endl;
}
```

Algorithm 1.16: Do While loop.

```
void Search_v03(int* a, int n, int x) {
    bool find = false;
    int argx = -1, i = 0;
    do {
        if (a[i] > x) {
            find = true;
            argx = i;
        }
        i++;
    } while ((!find) && (i < n));
    if (find)
        cout << "Found " << a[argx] << endl;
    else
        cout << "Not found" << endl;

}
```

Algorithm 1.17: Main.

```
void main() {
    int n = 10, x = 100;
    int* a = new int[n];
    for (int i = 0; i < n; i++)
        a[i] = rand() % 200;
    Search_v01(a, n, x);
    Search_v02(a, n, x);
    Search_v03(a, n, x);
}
```

1.4.9 No new, no delete!

In this section, we highlight different possibilities for the creation of objects. In particular, the use of unique_ptr allow to forget the delete once the function is finished.

Algorithm 1.18: Class definition.

```
class MyClass {
public:
    MyClass() {
        x = 0;
        cout << "Constructor " << x << endl;
    }
    MyClass(int x1) {
        x = x1;
        cout << "Constructor " << x << endl;
    }
    ~MyClass() {
        cout << "Destructor " << x << endl;
    }
    int x;
};
```

In this example, the constructor functions are called in the order for Object1, Object2, and Object3. For the destructor function, the calls are for Object2 (heap), Object3 (heap), and then Object1 (stack).

Algorithm 1.19: Example.

```
void main() {
    MyClass Object1(1); // Memory allocated on the stack
    MyClass *Object2 = new MyClass(2); // Memory allocated on the heap
    unique_ptr<MyClass> Object3(new MyClass(3));
    cout << "Object1:" << Object1.x << endl;
    cout << "Object2:" << Object2->x << endl; // pointer
    cout << "Object3:" << Object3->x << endl; // pointer
    delete Object2; // Must use delete free the memory
    // Object3 is free automatically thanks to the use of unique_ptr
    // Object1 is free automatically at the end the function
}
```

1.5 Recursivity

Recursivity is a key concept in mathematic and computer science. Depending on the problems, some algorithms can be better formulated using recursivity while some others are better with an iterative version.

- **A simple base case (or simple cases)**: It corresponds to a terminating scenario that does not use recursion to produce an output. If you get a stack overflow error in an algorithm, it may be because you have not properly defined the simple case, which cannot be reached, leading to an infinite recursion.
- **A recursive step**: It corresponds to a set of rules that reduces all other cases toward the base case.

1.5.1 Factorial

The factorial function is defined as follows:

$$Factorial(1) = 1 \tag{1.29}$$

$$Factorial(x) = x \cdot Factorial(x-1) \tag{1.30}$$

It is possible to implement the factorial function in various ways. Factorial1 uses a for loop while Factorial2 uses a while loop, and Factorial3 uses a recursive approach in which the function Factorial3 is called.

Algorithm 1.20: Factorial functions - iterative version.

```
// iterative function with for a loop
int Factorial1(int x) {
    int result = 1;
    for (int i = 2; i <= x; i++)
        result *= i;
    return result;
}

// iterative function with for a while loop
int Factorial2(int x) {
    int result = 1;
    int i = 2;
    while (i <= x) {
        result *= i;
        i++;
    }
    return result;
}
```

Algorithm 1.21: Factorial functions - recursive version.

```
// recursive function
int Factorial3(int x) {
    if (x <= 1)
        return 1; // base case
    else
        return x*Factorial3(x - 1); // recursive call
}

// recursive procedure, passing the input/output variable by pointer
```

```cpp
void Factorial4(int x, int* y) {
    if (x <= 1)
        *y = 1; // base case
    else {
        int y1;
        Factorial4(x - 1, &y1); // recursive call
        *y = x*y1;
    }
}

// recursive procedure, passing the input/output variable by reference
void Factorial5(int x, int &y) {
    if (x <= 1)
        y = 1; // base case
    else {
        int y1;
        Factorial5(x - 1, y1); // recursive call
        y = x*y1;
    }
}
```

Algorithm 1.22: Factorial examples.

```cpp
void main() {
    cout << Factorial1(6) << endl; // 720
    cout << Factorial2(6) << endl; // 720
    cout << Factorial3(6) << endl; // 720
    int y1;
    Factorial4(6, &y1);
    cout << y1 << endl; // 720
    int y2;
    Factorial5(6, y2);
    cout << y2 << endl; // 720
}
```

1.5.2 Fibonacci

This famous sequence is from Leonardo Bonacci, Italian mathematician from the Republic of Pisa, commonly named Fibonacci. In this sequence, each number is the sum of the two preceding ones.

The sequence is defined as follows:

$$Fibonacci(0) = 0 \tag{1.31}$$

$$Fibonacci(1) = 1 \tag{1.32}$$

$$Fibonacci(x) = Fibonacci(x-1) + Fibonacci(x-2) \tag{1.33}$$

There are many ways to implement the Fibonacci function.

Algorithm 1.23: Fibonacci function - Recursive (ternary operator).

```
int FibonacciREC1(int n)
{
    return (n==0) ? 0 : (n==1) ? 1 : FibonacciREC1(n - 1) + FibonacciREC1(n - 2);
}
```

Algorithm 1.24: Fibonacci function - Recursive (switch).

```
int FibonacciREC2(int n)
{
    switch (n)
    {
    case 0: return 0; // base case
    case 1:return 1; // base case
    default: return FibonacciREC2(n - 1) + FibonacciREC2(n - 2); // any other case
    }
}
```

Algorithm 1.25: Fibonacci function - Recursive (if).

```
int FibonacciREC(int n)
{
    if (n == 0)
        return 0; // base case
    else if (n == 1)
        return 1; // base case
    else
        return FibonacciREC(n - 1) + FibonacciREC(n - 2); // recursive call
}
```

Algorithm 1.26: Fibonacci function - Recursive (output as a reference).

```
void FibonacciREC(int n, int &output)
{
    if (n == 0)
        output=0; // base case
    else if (n == 1)
        output=1; // base case
    else
    {
        int output1;
```

```
        int output2;
        FibonacciREC(n - 1, output1);
        FibonacciREC(n - 2, output2);
        output = output1 + output2;
    }
}
```

Algorithm 1.27: Fibonacci function - Iterative.

```
int FibonacciITE(int n)
{
    if (n == 0)
        return 0; // base case
    else if (n == 1)
        return 1; // base case
    else
    {
        int a, b,f=0;
        a = 0; b = 1;
        for (int i = 2; i <= n; i++)
        {
            f = b + a; // f(n)=f(n-1)+f(n-2)
            // Update the values for the next iteration
            a = b;
            b = f;
        }
        return f;
    }
}
```

Algorithm 1.28: Fibonacci function - Iterative (output as a reference).

```
void FibonacciITE(int n,int &output)
{
    if (n == 0)
        output=0; // base case
    else if (n == 1)
        output = 1; // base case
    else
    {
        int a, b, f=0;
        a = 0; b = 1;
        for (int i = 2; i <= n; i++)
        {
            f = b + a; // f(n)=f(n-1)+f(n-2)
            // Update the values for the next iteration
            a = b;
            b = f;
        }
        output = f;
    }
}
```

1.5.3 Anagram

Algorithm 1.29: Rotate.

```
//  Method to rotate left all characters from position to end
void Rotate(char* str, int size, int newsize) {
    int position = size - newsize;
    char temp = str[position];
    int i;
    for (i = position + 1; i < size; i++) {
        str[i-1] = str[i];
    }
    str[i-1] = temp;
}
```

Algorithm 1.30: Do anagram.

```
void DoAnagram(char* str, int size, int newsize) {
    if (newsize > 1) {
        for (int loop = 0; loop < newsize; loop++) {
            DoAnagram(str, size, (newsize - 1));
            if (newsize == 2) {
                for (int i = 0; i < size; i++) {
                    cout << str[i];
                }
                cout << endl;
            }
            Rotate(str, size, newsize);
        }
    }
}
```

Algorithm 1.31: Anagram example.

```
void main() {
    char mystr[] = { 'R','A','T','S' };
    DoAnagram(mystr, 4, 4);
// RATS, RAST, RTSA, RTAS, RSAT, RSTA, ATSR, ATRS, ASRT, ASTR,
// ARTS, ARST, TSRA, TSAR, TRAS, TRSA, TASR, TARS, SRAT, SRTA,
// SATR, SART, STRA, STAR
```

1.5.4 Hanoi Tower

The Hanoi Tower is a famous puzzle game. It consists of three rods and a number of disks of different sizes. The disks can be placed onto any rod. The puzzle starts with the disks in a stack in ascending order of size on one stick, the smallest at the top. The goal is to move the whole stack to another rod, following these rules:

- One disk can be moved at a time.
- Each action consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- It is possible to place disk of diameter d_1 on top of disk d_2 only if $d_1 < d_2$.

Thinking of the Hanoi Tower solution in an iterative way is a difficult problem. It is a perfect example of thinking: "what if this part is already done, what else would I need to do", where the base case, i.e. the easy case, has to be determined first. It is an example in which it is better to think about the last steps than the first steps to execute.

Algorithm 1.32: Solve Hanoi Tower.

```
void HanoiTower(int n, string start, string auxiliary, string end) {
    if (n == 1)
        cout << start << " -> " << end << endl;
    else {
        HanoiTower(n - 1, start, end, auxiliary);
        cout << start << " -> " << end << endl;
        HanoiTower(n - 1, auxiliary, start, end);
    }
}
```

Algorithm 1.33: Hanoi tower example.

```
void main() {
    int n = 5; // number of disks
    HanoiTower(n, "A", "B", "C");s
// A->C, B->A, B->C, A->C, A->B, C->B, C->A, B->A, C->B, A->C,
// A->B, C->B, A->C, B->A, B->C, A->C, B->A, C->B, C->A, B->A,
// B->C, A->C, A->B, C->B, A->C, B->A, B->C, A->C.
}
```

Chapter 2

Queues and Stacks

Contents

2.1 Stacks

A stack

Algorithm 2.1: Stack - class definition

```
typedef double MyType;
class MyStack {
public:
    MyStack();
    MyStack(int capacity1);
    ~MyStack();
    bool IsFull();
    bool IsEmpty();
    MyType Pop();
    MyType Top();
    void Push(MyType x);
    void Display();
private:
    MyType* s;
    int capacity;
    int size;
};
```

Algorithm 2.2: Stack - functions

```
MyStack::MyStack() {
    s =  nullptr;
    capacity = 0;
    size = 0;
}
MyStack::MyStack(int capacity1) {
    capacity = capacity1;
    s = new MyType[capacity];
    size = 0;
}
MyStack::~MyStack() {
    delete[] s;
}
bool MyStack::IsFull() {
    return (size == capacity);
}
bool MyStack::IsEmpty() {
    return (size == 0);
}
MyType MyStack::Pop() {
    if (!IsEmpty()) {
        size--;
        return s[size];
    }
}
MyType MyStack::Top() {
    return s[size - 1];
}
void MyStack::Push(MyType x) {
    if (size < capacity) {
        s[size] = x;
        size++;
    }
}
void MyStack::Display() {
    cout << "Max capacity: " << capacity << endl;
    cout << "Size: " << size << endl;
    for (int i = 0; i < size; i++)
        cout << "Element: " << s[i] << " at position " << i << endl;
    cout << endl;
}
```

Algorithm 2.3: Example

```
void main() {
        MyStack* S = new MyStack(5);
        S->Push(4);
        S->Push(6);
        S->Push(8);
        S->Push(10);
        S->Push(12);
        S->Push(16);
        S->Push(18);
        S->Display();
        cout << "Pop: " << S->Pop() << endl;
        cout << "Pop: " << S->Pop() << endl;
        cout << "Top: " << S->Top() << endl;
        S->Push(20);
        S->Display();
        delete s;
}
```

2.2 Queues

Algorithm 2.4: Circular queue - class definition.

```
typedef double MyType;
class MyQueue {
public:
    MyQueue();
    MyQueue(int capacity1);
    ~MyQueue();
    bool IsFull();
    bool IsEmpty();
    void Enqueue(MyType x);
    MyType Dequeue();
    MyType Front();
    MyType Rear();
    void DisplayQueue();
    void DisplayAll();
private:
    int front, rear, size;
    int capacity;
    MyType* q;
};
```

Notice how the beginning of the queue is at position front and not at position 0 of the array.

Algorithm 2.5: Display Queue.

```
// Display the elements in the queue in the order they should be dequeued
void MyQueue::DisplayQueue() {
    cout << "Size: " << size << endl;
    for (int i = 0; i < size; i++)
        cout << "Element: " << q[(front+i)%capacity] << " at position " << i << endl;
    cout << endl;
}

// Display all the elements in the array sorting the values of the queue
void MyQueue::DisplayAll() {
    cout << "Max capacity: " << capacity << endl;
    for (int i = 0; i < capacity; i++)
        cout << "Element: " << q[i] << " at position " << i << endl;
    cout << endl;
}
```

Algorithm 2.6: Circular queue - functions.

```
MyQueue::MyQueue() {
    capacity = 0;
    front=size=0;
    rear=capacity-1;
    q= nullptr;
}
MyQueue::~MyQueue() {
    delete[] q;
}
MyQueue::MyQueue(int capacity1) {
    capacity=capacity1;
    front=size=0;
    rear=capacity-1;  // important, see the enqueue
    q=new MyType[capacity];
}
bool MyQueue::IsFull() {
    return (size==capacity);
}
bool MyQueue::IsEmpty() {
    return (size==0);
}
void MyQueue::Enqueue(MyType x) {
    if (!IsFull()) {
        rear = (rear + 1) % capacity;
        q[rear] = x;
        size = size + 1;
    }
}
MyType MyQueue::Dequeue() {
    if (IsEmpty())
        return INT_MIN;
    MyType item = q[front];
    front=(front+1)%capacity;
    size=size-1;
    return item;
}
MyType MyQueue::Front() {
    if (IsEmpty())
        return INT_MIN;
    return q[front];
}
MyType MyQueue::Rear() {
    if (IsEmpty())
        return INT_MIN;
    return q[rear];
}
```

Algorithm 2.7: Example

```
void main() {
        MyQueue* Q = new MyQueue(4);
        Q->Enqueue(4);
        Q->Enqueue(6);
        Q->Enqueue(8);
        Q->Enqueue(10);
        Q->Enqueue(12);
        Q->Enqueue(16);
        Q->Enqueue(18);
        Q->Display();
        cout << "Dequeue: " << Q->Dequeue() << endl;
        cout << "Dequeue: " << Q->Dequeue() << endl;
        Q->Enqueue(20);
        Q->Enqueue(22);
        Q->Enqueue(24);
        Q->Enqueue(26);
        Q->Display();
        cout << "Dequeue: " << Q->Dequeue() << endl;
        Q->Enqueue(28);
        Q->Display();
        delete Q;
}
```

Chapter 3

Prototype pattern

Contents

The goal of this section is to show how you can create the prototype pattern. It is one of the 23 well-known Gang of Four design patterns. The first part is the main program, which creates dynamically a data structure. In the present case, the code 1 indicates that we want to create a data structure array.

3.1 Main program

Algorithm 3.1: Main program

```
#include <iostream>
#include <tuple>
using namespace std;
#include "MyDataStructure.h"
#include "MyArray.h"
#include "MySCList.h"
#include "MyDCList.h"
#include "DataStructureFactory.h"

int main() {
    MyDataStructure* ds;
    ds = DataStructureFactory::makeDataStructure(1); // 1 = MyArray
    int n = 10;
    for (int i = 0; i < n; i++) {
        ds->Insert(i * 10 + 2);
    }
    ds->Display();
    delete ds;
    return 0;
}
```

3.2 Parent class

This part corresponds to the class MyDataStructure, which is the parent class of the classes related to each data structure.

Algorithm 3.2: MyDataStructure.h

```
#include <tuple>
#include <iostream>
typedef double MyType;
extern void Swap(MyType *r, MyType *s);
class MyDataStructure {
public:
    MyDataStructure();
    ~MyDataStructure();
    virtual MyDataStructure* clone() = 0;
    virtual void MyDataStructure::Insert(MyType x) { }
    virtual void MyDataStructure::Delete(MyType x) { }
    virtual bool MyDataStructure::Search(MyType x) { return false; }
    virtual void MyDataStructure::Display() {}
};
```

Algorithm 3.3: MyDataStructure.cpp

```
#include "MyDataStructure.h"
void Swap(MyType *r, MyType *s) {
    MyType tmp = *r;
    *r = *s;
    *s = tmp;
}
MyDataStructure::MyDataStructure() {}
MyDataStructure::~MyDataStructure() {}
```

3.3 Factory class

This part corresponds to the class DataStructureFactory, which is how we create data structures, children of the class MyDataStructure.

Algorithm 3.4: DataStructureFactory.h

```
#pragma once
#include "MyDataStructure.h"
const int N = 8; // number of data structures
class DataStructureFactory {
public:
    static MyDataStructure* makeDataStructure(int choice);
private:
    static MyDataStructure* mDataStructureTypes[N];
};
```

Algorithm 3.5: DataStructureFactory.cpp

```
#include "MySCList.h"
#include "MyDCList.h"
MyDataStructure* DataStructureFactory::mDataStructureTypes[] =
{
    0, new MyArray, new MySCList, new MyDCList,
    new MyCList, new MySkipList, new MyHashTable, new MyBST
};
MyDataStructure* DataStructureFactory::makeDataStructure(int choice) {
    return mDataStructureTypes[choice]->clone();
}
```

Chapter 4

Array and sorting

Contents

4.1 Array

4.1.1 MyArray interface

The class MyArray includes state of the art methods that are typically used with arrays. It includes insert, delete, search, sorting algorithms, and other useful algorithms.

Algorithm 4.1: MyArray.h

```
#pragma once
#include "MyDataStructure.h"
#include <chrono>
#include <string>

class MyArray : public MyDataStructure {
public:
    // constructor
    MyArray();
    MyArray(int n);
    // destructor
    ~MyArray();
    MyDataStructure* clone() { return new MyArray(); }
    // Accessor + Modifiers
    int GetSize() const;
    MyType GetElement(int i) const;
    void SetElement(int i, MyType x);
    bool Find(MyType x);
    pair<bool, int> BinarySearch1(MyType x);
    pair<bool, int> BinarySearch2(MyType x);
    void Delete(MyType x);
    void Insert(MyType x);
    bool Search(MyType x);
    void Display();
    void Display(int low, int high);
    void DisplayFile();
    void DisplayFileC();
    void Invert();
    MyArray* FindOdd();
    MyType GetMax();
    pair<MyType, int> GetMaxArg();
    MyType GetMin();
    pair<MyType, int> GetMinArg();
    double GetAverage();
    double GetStandardDeviation();
    MyType& operator[] (unsigned i);
    MyArray* operator+(const MyArray* a);
    tuple<int, int, int> FindMaxCrossingSubarray(int low, int mid, int high);
    tuple<int, int, int> FindMaximumSubarray(int low, int high);
    // Sorting functions
    bool IsSorted();
    void SwapIndex(int i, int j);
    void DisplayStep(string f);
    // Init functions
    void InitRandom(int v);
    void InitSortedAscending(int v);
```

```
    void InitSortedDescending(int v);
    // Sorting algorithms
    void SelectionSort();
    void InsertionSort();
    void BubbleSort();
    void BubbleOptSort();
    void MergeSort();
    void QuickSort();
private:
    MyType* a; // array
    int n; // size of the array
};
```

4.1.2 Constructors and destructor

Algorithm 4.2: Constructors and destructor (MyArray.cpp)

```
// Number of steps
// to count how many main steps are done in an algorithm.
int step;

// Default constructor
MyArray::MyArray() {
    a =  nullptr;
    n = 0;
}

// Basic constructor
// Create an array of size n1
MyArray::MyArray(int n1) {
    n = n1;
    a = new MyType[n];
    for (int i = 0; i < n; i++)
        a[i] = 0;
}

// Destructor
MyArray::~MyArray() {
    delete[] a;
}
```

4.1.3 Access elements

Algorithm 4.3: Access elements (MyArray.cpp).

```
// Return the number of elements in the array
int MyArray::GetSize() const { return n; }

MyType MyArray::GetElement(int i) const {
    if (i < 0) {
        cout << "Too small index";
        exit(EXIT_FAILURE);
    } else if (i >= n) {
        cout << "Too large index";
        exit(EXIT_FAILURE);
    } else
        return a[i];
}

// a is private
// we have a function to set the value x at the position i
void MyArray::SetElement(int i, MyType x) {
    a[i] = x;
}

MyType& MyArray::operator[] (unsigned i) {
    try {
        return a[i];
    } catch (exception& e) {
        cout << "Problem with index" << e.what() << endl;
    }
}

// Concatenate two arrays with the + operator
MyArray* MyArray::operator+(const MyArray* a)  {
    MyArray* out = new MyArray(this->n + a->GetSize());
    for (int i = 0; i < this->n; i++)
        out->SetElement(i, this->GetElement(i));
    for (int i = this->n; i < out->GetSize(); i++)
        out->SetElement(i+ this->n, a->GetElement(i));
    return out;
}
```

4.1.4 Find, Delete, Insert

// Determine if the value x is in the array

Algorithm 4.4: Find, Delete, Insert.

```
bool MyArray::Find(MyType x) {
    for (int i = 0; i < n; i++) {
        if (a[i] == x) {
            return true;
        }
    }
    return false;
}

void MyArray::Delete(MyType x) {
    if (Find(x)) {
        MyType* a1 = new MyType[n - 1];
        int j = 0;
        for (int i = 0; i < n; i++) {
            if (a[i] != x) {
                a1[j] = a[i];
                j++;
            }
        }
        delete[] a;
        a = a1;
    }
}

void MyArray::Insert(MyType x) {
    MyType* a1 = new MyType[n + 1];
    for (int i = 0; i < n; i++) {
        a1[i] = a[i];
    }
    a1[n] = x;
    delete[] a;
    a = a1;
    n++;
}
```

4.1.5 Binary Search

The binary search is presented in the iterative and recursive versions.

Algorithm 4.5: Binary search.

```
// Return if the value x is in the array or not, and the index of the value.
// Iterative version
pair<bool, int> MyArray::BinarySearch1(MyType x) {
    bool found = false;
    int mid,low = 0, high = n-1;
    while ((low <= high) && (!found)) {
        mid = (low + high) / 2;
        if (x==a[mid])
            return make_pair(true,mid);
        else {
            if (x < a[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
    }
    return make_pair(false, -1);
}
// Recursive version
pair<bool, int> BinarySearchRec(MyType* a, int x, int low, int high) {
    int mid;
    if (low>high) // not found
        return make_pair(false,-1);
    else {
        mid= (low + high) / 2;
        if (x == a[mid])
            return make_pair(true, mid);
        else {
            if (x < a[mid])
                return BinarySearchRec(a, x, low, mid - 1);
            else
                return BinarySearchRec(a, x, mid + 1, high);
        }
    }
}

pair<bool, int> MyArray::BinarySearch2(MyType x) {
    return BinarySearchRec(a, x, 0, n - 1);
}
```

4.1.6 Useful functions

Algorithm 4.6: Invert (MyArray.cpp).

```
void MyArray::Invert() {
    MyType tmp;
    for (int i = 0; i < n/2; i++) {
        tmp = a[i];
        a[i] = a[n - 1 - i];
        a[n - 1 - i] = tmp;
    }
}
```

Algorithm 4.7: Array of odd numbers (MyArray.cpp).

```
MyArray* MyArray::FindOdd() {
    int nodd = 0;
    for (int i = 0; i < n; i++) {
        if (((int)a[i] % 2) == 1)
            nodd++;
    }
    nodd = 0;
    MyArray* a1 = new MyArray(nodd);
    for (int i = 0; i < n; i++) {
        if (((int)a[i] % 2) == 1) {
            a1->SetElement(nodd, a[i]);
            nodd++;
        }
    }
    return a1;
}
```

Algorithm 4.8: Search and return index (MyArray.cpp).

```
bool MyArray::Search(MyType x) {
    for (int i = 0; i < n; i++) {
        if (a[i] == x) {
            return true;
        }
    }
    return false;
}
```

4.1.7 Maximum and Argmax

Algorithm 4.9: Maximum (MyArray.cpp).

```
MyType MyArray::GetMax() {
    if (n > 0) {
        MyType max = a[0];
        for (int i = 1; i < n; i++) {
            if (a[i] > max)
                max = a[i];
        }
        return max;
    }
    else
        return 0;
}
```

Algorithm 4.10: Maximum and argmax (MyArray.cpp).

```
pair<MyType, int> MyArray::GetMaxArg() {
    if (n > 0) {
        MyType max = a[0];
        int argmax = 0;
        for (int i = 1; i < n; i++) {
            if (a[i] > max) {
                max = a[i];
                argmax = i;
            }
        }
        return std::make_pair(max, argmax);
    }
    else
        return std::make_pair(0,-1);
}
```

4.1.8 Minimum and Argmin

Algorithm 4.11: Minimum (MyArray.cpp).

```
MyType MyArray::GetMin() {
    if (n > 0) {
        MyType min = a[0];
        for (int i = 1; i < n; i++) {
            if (a[i] < min)
                min = a[i];
        }
        return min;
    }
    else
        return 0;
}
```

Algorithm 4.12: Minimum and argmin (MyArray.cpp).

```
pair<MyType, int> MyArray::GetMinArg() {
    if (n > 0) {
        MyType min = a[0];
        int argmin = 0;
        for (int i = 1; i < n; i++) {
            if (a[i] < min) {
                min = a[i];
                argmin = i;
            }
        }
        return std::make_pair(min, argmin);
    }
    else
        return std::make_pair(0,-1);
}
```

4.1.9 Mean and Standard Deviation

Algorithm 4.13: Basic statistic (MyArray.cpp).

```
double MyArray::GetAverage() {
    double result = 0;
    if (n > 0) {
        for (int i = 0; i < n; i++)
            result += (double)a[i];
        result /= n;
    }
    return result;
}

double MyArray::GetStandardDeviation() {
    double result = 0;
    if (n > 0) {
        double mean = GetAverage();
        for (int i = 0; i < n; i++) {
            double tmp = (double)a[i] - mean;
            result += tmp*tmp;
        }
        result /= n;
        result = sqrt(result);
    }
    return result;
}
```

4.1.10 FindMaximumSubarray

The FindMaximumSubarray algorithm is a state of the art example of the divide and conquer approach.

Algorithm 4.14: FindMaxCrossingSubarray (MyArray.cpp).

```
tuple<int,int,int> MyArray::FindMaxCrossingSubarray(int low,int mid,int high) {
    int left_sum = -INT_MAX, right_sum= -INT_MAX; // -infinity
    int max_left = 0, max_right = 0, sum = 0;
    for (int i=mid;i>=low;i--) {
        sum=sum+(int)a[i];
        if (sum>left_sum) {
            left_sum = sum;
            max_left = i;
        }
    }
    sum = 0;
    for (int j = mid + 1; j <= high; j++) {
        sum=sum+ (int)a[j];
        if (sum>right_sum) {
            right_sum = sum;
            max_right = j;
        }
    }
    return make_tuple(max_left, max_right, left_sum + right_sum);
}

tuple<int, int, int> MyArray::FindMaximumSubarray(int low, int high) {
    int mid;
    int left_low, left_high, left_sum;
    int right_low, right_high, right_sum;
    int cross_low, cross_high, cross_sum;
    if (high == low)
            return make_tuple(low,high,a[low]);
            // base case: only one element
        else {
            mid=(low+high)/2;
            tie(left_low,left_high,left_sum)=
            FindMaximumSubarray(low, mid);
            tie(right_low,right_high,right_sum)=
            FindMaximumSubarray(mid + 1, high);
            tie(cross_low,cross_high,cross_sum)=
            FindMaxCrossingSubarray(low, mid, high);
            if ((left_sum >= right_sum) && (left_sum >= cross_sum))
                return make_tuple(left_low, left_high, left_sum);
            else if ((right_sum >= left_sum) && (right_sum >=cross_sum))
                return make_tuple(right_low, right_high, right_sum);
            else return make_tuple(cross_low, cross_high, cross_sum);
        }
}
```

Algorithm 4.15: Display (C++) (MyArray.cpp).

```
void MyArray::Display() {
    for (int i = 0; i < n; i++) {
        cout << "Element " << i << " with value " << a[i] << endl;
    }
}

void Display(MyType *a, int start, int end) {
    for (int i = start; i <=end; i++) {
        cout << "Element " << i << " with value " << a[i] << endl;
    }
}

void MyArray::Display(int low,int high) {
    if (low<0 || high>(n - 1))
        exit(EXIT_FAILURE);
    else
        for (int i = low; i <= high; i++)
            cout << "Element " << i << " with value " << a[i] << endl;
}
```

Algorithm 4.16: Print in a file (C++).

```
void MyArray::DisplayFile() {
    ofstream myfile;
    myfile.open("log.txt");
    myfile << "Array of size " << n << endl;
    for (int i = 0; i < n; i++) {
        myfile << "Element " << i << " with value " << a[i] << endl;
    }
    myfile.close();
}
```

Algorithm 4.17: Print in a file (C) (MyArray.cpp).

```
void MyArray::DisplayFileC() {
    FILE* f;
    f=fopen("log.txt", "wt");
    fprintf(f,"Array of size %d\n",n);
    for (int i = 0; i < n; i++) {
        fprintf(f,"Element %d with value %d\n", i,(int)a[i]);
    }
    fclose(f);
}
```

Algorithm 4.18: Array initialization (MyArray.cpp).

```
void MyArray::InitRandom(int v) {
    for (int i = 0; i < n; i++)
        a[i] = rand() % v;
}

void MyArray::InitSortedAscending(int v) {
    if (n > 0) {
```

```
        a[0] = rand() % v;
        for (int i = 1; i < n; i++)
            a[i] = a[i - 1] + rand() % v;
    }
}

void MyArray::InitSortedDescending(int v) {
    if (n > 0) {
        a[0] = rand() % v;
        for (int i = 1; i < n; i++)
            a[i] = a[i - 1] - rand() % v;
    }
}
```

Algorithm 4.19: Display steps (MyArray.cpp).

```
void MyArray::DisplayStep(string f) {
    cout << "Array of size " << n << endl;
    cout << "Number of steps for " << f << " is " << step << endl;
}
```

Algorithm 4.20: Swapping.

```
void MyArray::SwapIndex(int i, int j) {
    MyType tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}

void SwapIndex(MyType* a, int i, int j) {
    MyType tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

4.2 Sorting algorithms

What does it mean for an array to be sorted?
We consider an array a_u of size n. We assume that this array is unsorted. The sorted array a_s represents the sorted elements of the array a_u. We consider the names a_u to denote the fact that a_u is unsorted, and a_s to show that a is sorted but for C++ variables, we may consider a_unsorted and a_sorted to denote the unsorted and sorted variables.

- $x \in a_u$ if and only if $x \in a_s$. If an element is in the initial array, then it must be in the sorted array. The output array may be sorted but if it misses elements it is not correct. In other words, $\exists(i,j)|a[i] = b[j]$.
- If all the elements of a_u are in a_s, then there are as many elements in a_s than in a_u. It means the size of a_s is also n.
- **Minimum**: We defined the minimum x_{min} by:

$$\forall i \in \{0,..,n-1\} \mid a_u[i] \geq x_{min} \ \ and \ \ \exists i \in \{0,..,n-1\}|a_u[i] = x_{min} \tag{4.1}$$

- **Maximum**: We defined the maximum x_{max} by:

$$\forall i \in \{0,..,n-1\} \mid a_u[i] \leq x_{max} \ \ and \ \ \exists i \in \{0,..,n-1\}|a_u[i] = x_{max} \tag{4.2}$$

- a_u is sorted then $\forall i \in \{1,..,n-1\}$, $a_u[i-1] \leq a_u[i]$.
- a_u is sorted then $\forall i_1 \in \{0,..,k\}$, $\forall i_2 \in \{k+1,..,n-1\}$, $a_u[i_1] \leq a_u[i_2]$.

The base case: if there is only one element, then the array is sorted.

Algorithm 4.21: Is the array sorted?.

```
bool MyArray::IsSorted() {
    bool output = true;
    if ((n == 0) || (n==1))
        return true;
    else {
        int i = 1;
        while ((a[i - 1] < a[i]) && (i<n))
            i++;
        return (i==n);
    }
}
```

4.2.1 Selection Sort: $O(n^2)$

Algorithm 4.22: Selection sort (MyArray.cpp).

```
void MyArray::SelectionSort() {
    step = 0;
    // Base case: the array with 1 element is already sorted
    // Invariant: the whole array is sorted between position 0 and i
    for (int i = 0; i <n; ++i) {
        int min = i;
        // min = index of the minimum value for the array between
        for (int j = i + 1; j < n; ++j) {
        // search for the minimum index j between i and n-1
            if (a[j] < a[min]) {
                min = j;
            }
            step++;
        }
        SwapIndex(i, min);
    }
    DisplayStep(__func__); // __func__ : name of the function
}
```

4.2.2 Insertion Sort: $O(n^2)$

Algorithm 4.23: Insertion sort (MyArray.cpp).

```
void MyArray::InsertionSort() {
    step = 0;
    // Base case: the array with 1 element is already sorted
    // Invariant: the array defined between position 0 and i is sorted
    bool done;
    for (int i = 0; i < n-1; ++i) {
        int j = i + 1;
        done = false;
        while ((j > 0) && (!done)) {
            step++;
            if (a[j] < a[j - 1]) {
                SwapIndex(j, j - 1);
            }
            else
                done=true;
            j--;
        }
    }
    DisplayStep(__func__);
}
```

4.2.3 Bubble Sort: $O(n^2)$

Algorithm 4.24: Bubble sort (MyArray.cpp).

```
void MyArray::BubbleSort() {
    step = 0;
    // Invariant: the whole array is sorted between n-1-i and n-1
    for (int i = 0; i < n-1; ++i) {
        for (int j = 0; j < n-i-1; ++j) {
            step++;
            if (a[j] > a[j + 1]) {
                SwapIndex(j, j + 1);
            }
        }
    }
    DisplayStep(__func__);
}

void MyArray::BubbleOptSort() {
    step = 0;
    // Invariant: the whole array is sorted between n-1-i and n-1
    bool sorted=false;
    int i = 0;
    while ((i<n-1) && (!sorted)) {
        sorted = true; // we assume the array is sorted between 0 and n-i-1
        for (int j =0 ; j < n-i-1; j++) {
            step++;
            if (a[j]>a[j+1]) {
                SwapIndex(j,j+1);
                sorted = false;
            }
        }
        i++;
    }
    DisplayStep(__func__);
}
```

4.2.4 Merge sort: $O(n \cdot log(n))$

Algorithm 4.25: Merge (MyArray.cpp).

```
void merge(MyType* a, int start, int mid, int end) {
    // Init a1
    int n1 = mid - start +1;
    MyType* a1 = new MyType[n1];
    for (int i = 0; i < n1; i++)
        a1[i] = a[i+start];
    // Init a2
    int n2 = end - mid;
    MyType* a2 = new MyType[n2];
    for (int i = 0; i < n2; i++)
        a2[i] = a[i+mid+1];
    int i1 = 0, i2 = 0, i3 = start;
    while ((i1 < n1) && (i2 < n2)) {
        if (a1[i1] < a2[i2]) {
            a[i3] = a1[i1];
            i1++;
        }
        else {
            a[i3] = a2[i2];
            i2++;
        }
        i3++;
        step++;
    }
    if (i1 < n1) // -> we left the while loop because i2>=n2
        for (int i = i1; i < n1; i++) {
            a[i3] = a1[i];
            i3++;
            step++;
        }
    else // -> we left the while loop because i1>=n1
        for (int i = i2; i < n2; i++) {
            a[i3] = a2[i];
            i3++;
            step++;
        }
    delete[] a1;
    delete[] a2;
}
```

Algorithm 4.26: Mergesort (MyArray.cpp).

```
void mergesort(MyType* a, int start, int end) {
    if (start < end) {
        int mid = (start + end) / 2;
        mergesort(a, start, mid);
        // a is sorted between start and mid
        mergesort(a, mid + 1, end);
        // a is sorted between mid+1 and end
        merge(a, start, mid, end);
        // a is sorted
    }
    // else there is 1 element and it is already sorted (base case)
}

// 1st call of the function with
// start=0 and end=n-1
void MyArray::MergeSort() {
    step = 0;
    mergesort(a, 0, n - 1);
    DisplayStep(__func__);
}
```

4.2.5 Quicksort: $O(n \cdot log(n))$

Algorithm 4.27: Quicksort (MyArray.cpp).

```cpp
// Hoare partition
int partition(MyType* a, int start, int end) {
    int i = start;
    int j = end;
    MyType pivot_value = a[(start+end)/2]; // in the middle
    bool finished = false;
    while (!finished) {
        while ((i<end) && (a[i] <= pivot_value)) {
            i++; // move to the right
            step++;
        }
        while ((j>start) && (a[j] > pivot_value)) {
            j--; // move to the left
            step++;
        }
        if (i < j)
            SwapIndex(a,i,j);
        else
            finished = true;
    }
    cout << "Pivot: " << pivot_value << " at position " << j << endl;
    Display(a, start, end);
    return j; // index of the pivot, which can move !
}

void quicksort(MyType* a, int start, int end) {
    if (start < end) {
        int pivot_index = partition(a, start, end);
        // All the elements between start and pivot_index-1 are
        // inferior to a[pivot_index].
        // All the elements between pivot_index+1 and end are
        // superior to a[pivot_index].
        quicksort(a,start, pivot_index -1);
        quicksort(a, pivot_index +1,end);
    }
    // else there is 1 element and it is already sorted (base case)
}

// 1st call of the function with
// start=0 and end=n-1
void MyArray::QuickSort() {
    step = 0;
    quicksort(a, 0, n - 1);
    DisplayStep(__func__);
}
```

4.2.6 Some main examples

Algorithm 4.28: Example with Pairs.

```
void main() {
    int n=10;
    MyArray* B = new MyArray(n);
    B->InitRandom(n);
    pair<double, int> r=B->GetMaxArg();
    cout << "vmax: " << r.first << endl;
    cout << "argmax: " << r.second << endl;
}
```

Algorithm 4.29: Example with FindMaxSubArray.

```
void main() {
    int nsize = 8;
    int x1[] = { 8,7,6,5,4,3,2,1 };
    MyArray* A = new MyArray(nsize);
    for (int i = 0; i<nsize; i++) {
        (*A)[i] = x1[i];
    }
    int low = 0;
    int high = nsize - 1;
    int sublow, subhigh, subsum;
    tie(sublow, subhigh, subsum) = A->FindMaximumSubarray(low, high);
    cout << "Sum:" << subsum << endl;
}
```

Algorithm 4.30: Application of Quicksort.

```
void main() {
    int nsize = 12;
    MyArray* A = new MyArray(nsize);
    A->InitRandom(2);
    cout << "Quicksort" << endl;
    A->QuickSort();
    A->Display();
}
```

Algorithm 4.31: Application of Binary Search.

```
void main() {
  MyType x1[] = {2,5,9,13,18,23,27,33};
  // number of elements = total space / space of 1 element
    int nsize = sizeof(x1) / sizeof(x1[0]);
    MyArray* A = new MyArray(nsize);
    for (int i = 0; i<nsize; i++) {
        (*A)[i] = x1[i];
    }
    pair<double, int> r = A->BinarySearch2(13);
    cout << "Found: " << r.first << " at position " << r.second << endl;
    r = A->BinarySearch2(2);
    cout << "Found: " << r.first << " at position " << r.second << endl;
    r = A->BinarySearch2(33);
    cout << "Found: " << r.first << " at position " << r.second << endl;
    r = A->BinarySearch2(12);
    cout << "Found: " << r.first << " at position " << r.second << endl;
}
```

4.2.7 Time measurement

This function measures the time it takes to sort an array.

Algorithm 4.32: Example of benchmarks.

```
void main() {
    int nsize = 10;
    MyArray* A = new MyArray(nsize);
    int type_array = 0;
    for (int type_sort = 1; type_sort <=6; type_sort++) {
        // Create a new array at each iteration!
        if (type_array == 0) {
            A->InitRandom(2);
            cout << "Random array" << endl;
        }
        else if (type_array == 1) {
            A->InitSortedAscending(nsize*nsize);
            cout << "Sorted array" << endl;
        }
        if (nsize < 15) {
            cout << "Input array:" << endl;
            A->Display();
        }
        auto t1 = chrono::high_resolution_clock::now();
        switch (type_sort) {
        case 1: // Selection sort
            A->SelectionSort(); break;
        case 2: // Insertion sort
            A->InsertionSort(); break;
        case 3: // Bubble sort
            A->BubbleSort(); break;
        case 4: // Bubble Opt sort
            A->BubbleOptSort(); break;
        case 5: // Merge sort
            A->MergeSort(); break;
        case 6: // Quick sort
            A->QuickSort(); break;
        }
        if (nsize<15)
            A->Display();
        auto t2 = chrono::high_resolution_clock::now();
        chrono::duration<double,milli> duration_ms = t2 - t1;
        cout << "It took " << duration_ms.count() << " ms" << endl;
    }
}
```

Chapter 5

Lists

Contents

5.1 Simple chained list

5.1.1 Node definition

Algorithm 5.1: Node simple chained list.

```
class NodeSC {
public:
    NodeSC(): data(0), next(nullptr)  {}
    NodeSC(MyType d) : data(d), next(nullptr) {}
    NodeSC(MyType d, NodeSC* nxt) : data(d), next(nxt) {}
    ~NodeSC() {}
    MyType data;
    NodeSC *next;
};
```

5.1.2 Class interface

Algorithm 5.2: Class simple chained list (MySCList.h).

```
class MySCList : public MyDataStructure {
public:
    MySCList();
    ~MySCList();
    int GetSize() const { return n; }
    MyDataStructure* clone() { return new MySCList(); }
    void CreateNode(MyType value);
    bool SearchITE(MyType value);
    bool SearchREC(MyType value);
    void Insert(MyType value);
    void InsertFirst(MyType value);
    void InsertLast(MyType value);
    void InsertPosition(int pos, MyType value);
    void DeleteFirst();
    void DeleteLast();
    void DeletePosition(int pos);
    void DeleteMiddle(MyType value);
    void InitRandom(int n, int v);
    void InitSortedAscending(int n, int v);
    void InitSortedDescending(int n, int v);
    void Reverse();
    void Display();
    void DisplayFile();
private:
    NodeSC *head; // pointer on the head
    NodeSC *tail; // pointer on the tail
    int n;
};
```

5.1.3 Constructor and destructor

Algorithm 5.3: Class simple chained list (MySCList.cpp).

```
MySCList::MySCList(): head(nullptr), tail(nullptr), n(0) { }

MySCList::~MySCList() {
    NodeSC *current = head;
    while (current) {
        NodeSC *next = current->next;
        delete current;
        current = next;
    }
}
```

Algorithm 5.4: Reverse (MySCList.cpp).

```
void MySCList::Reverse() {
    NodeSC *current = head;
    NodeSC *prev = nullptr, *next = nullptr;
    while (current) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    head = prev;
}
```

5.1.4 Search

Algorithm 5.5: Search - Iterative.

```
bool MySCList::SearchITE(MyType value)
{
    NodeSC* current = head;
    bool found = false;
    while (current && !found)
    {
        if (current->data == value)
            found = true;
        else
            current = current->next; // go to the next element
    }
    return found;
}
```

Algorithm 5.6: Search - Recursive.

```
bool SearchREC1(NodeSC* current,MyType value)
{
    if (current)
    {
        if (current->data == value)
            return true; // Base case: found in the head
        else
            return SearchREC1(current->next, value);
    }
    else
        return false; // Base case: empty list
}

bool MySCList::SearchREC(MyType value)
{
  // First call with the head of the list
    return SearchREC1(head, value);
}
```

5.1.5 Insert

Algorithm 5.7: Insert.

```
void MySCList::CreateNode(MyType value) {
    NodeSC *tmp = new NodeSC(value);
    if (head==nullptr) {
        head = tmp;
        tail = tmp;
    } else {
        tail->next = tmp;
        tail = tmp;
    }
    n++;
}

void MySCList::InsertFirst(MyType value) {
    NodeSC *tmp = new NodeSC(value,head);
    head = tmp;
    n++;
}

void MySCList::InsertLast(MyType value) {
    CreateNode(value);
}

void MySCList::Insert(MyType value) {
    InsertMiddle(value);
}
```

In this function, we insert an element in a sorted list. We consider 3 cases: 1) if the list is empty, 2) if we insert at the beginning (we replace the head), or 3) if the element to insert is later in the list.

Algorithm 5.8: Insert (iterative version).

```
void MySCList::InsertMiddle(MyType value) {
    NodeSC *tmp = new NodeSC(value);
    NodeSC *cursor = head;
    if (head == nullptr) { // insert to the head (empty list)
        cout << "Insert to head/tail: " << value << endl;
        head = tmp;
        tail = tmp;
    }
    else if (head->data > value) { // insert to the head
        cout << "Insert to head: " << value << endl;
        tmp->next = head;
        head = tmp;
    }
    else {
        NodeSC *prev = head;
        NodeSC *cursor = head->next;
        while ((cursor != nullptr) && (cursor->data < value)) {
            prev = prev->next;
            cursor = cursor->next;
        }
        // Case 1: we arrive at the end of the list
        if (cursor == nullptr) { // insert to the tail
            cout << "Insert to tail: " << value << endl;
            prev->next = tmp;
            tail = tmp;
        }
        // Case 2: we arrive between 2 elements
        else {
            cout << "Insert middle: " << value << endl;
            prev->next = tmp;
            tmp->next = cursor;
        }
    }
    n++;
}
```

In this function, we insert an element in a sorted list. It is the same as before, however we are considering a recursive function. In such a case, we need 2 functions: one function that is used for the main work, and another function that is used for calling the first function with the proper arguments, so we can hide the head within this function, i.e. there is no need to use the head as an input of the function. It is worth noting that cursor is as input/output, and it is passed by reference. This function is similar to the iterative function, but using a recursive function, the code is cleaner, there is just a call to the function, and no while.

Algorithm 5.9: Insert (recursive version) - Procedure.

```
void MySCList::InsertMiddleRec(NodeSC* &cursor, MyType value) {
    if (cursor == nullptr) { // insert to the head (empty list)
        cout << "Insert to head/tail: " << value << endl;
        NodeSC* tmp = new NodeSC(value);
        cursor = tmp;
        tail = tmp;
        n++; // increment the number of elements
    }
    else if (cursor->data > value) { // insert to the head
        cout << "Insert to head: " << value << endl;
        NodeSC* tmp = new NodeSC(value);
        tmp->next =cursor;
        cursor = tmp; // the "previous->next" is modified
        n++; // increment the number of elements
    }
    else  // cursor->next input/output
        InsertMiddleRec(cursor->next, value);
}

void MySCList::InsertMiddleMainRec(MyType value) {
    InsertMiddleRec(head, value);
}
```

In this function, we insert an element in a sorted list. It is similar as the previous function but we are using a function instead of a procedure (i.e. a function returning void).

Algorithm 5.10: Insert (recursive version) - Function.

```
NodeSC* MySCList::InsertMiddleRecf(NodeSC* cursor, MyType value) {
    if (cursor == nullptr) { // insert to the head (empty list)
        cout << "Insert to head/tail: " << value << endl;
        NodeSC* tmp = new NodeSC(value);
        tail = tmp;
        n++;
        return tmp;
    }
    else if (cursor->data > value) { // insert to the head
        cout << "Insert to head: " << value << endl;
        NodeSC* tmp = new NodeSC(value);
        tmp->next = cursor;
        n++;
        return tmp;
    }
    else {
        cursor->next=InsertMiddleRecf(cursor->next,value);
        return cursor;
    }
}

void MySCList::InsertMiddleMainRecf(MyType value) {
    head=InsertMiddleRecf(head,value);
}
```

Algorithm 5.11: Insert.

```cpp
void MySCList::InsertPosition(int pos, MyType value) {
    if (pos == 0)
        InsertFirst(value);
    else if (pos == n - 1)
        InsertLast(value);
    else {
        NodeSC *previous = nullptr;
        NodeSC *current = head;
        NodeSC *tmp = new NodeSC(value);
        for (int i = 0; i < pos; i++) {
            previous = current;
            current = current->next;
        }
        previous->next = tmp;
        tmp->next = current;
        n++;
    }
}
```

Algorithm 5.12: Initialization.

```cpp
void MySCList::InitRandom(int n,int v) {
    for (int i = 0; i < n; i++)
        InsertLast(rand() % v);
}

void MySCList::InitSortedAscending(int n, int v) {
    MyType tmp = rand() % v;
    for (int i = 0; i < n; i++) {
        MyType tmp1 = tmp + rand() % v;
        InsertLast(tmp1);
        tmp = tmp1;
    }
}

void MySCList::InitSortedDescending(int n, int v) {
    MyType tmp = 10e4+rand() % v;
    for (int i = 0; i < n; i++) {
        MyType tmp1 = tmp - rand() % v;
        InsertLast(tmp1);
        tmp = tmp1;
    }
}
```

5.1.6 Delete

Algorithm 5.13: Delete.

```
void MySCList::DeleteFirst() {
    NodeSC *tmp;
    tmp = head;
    head = head->next;
    delete tmp;
    n--;
}

void MySCList::DeleteLast() {
    NodeSC *current=nullptr;
    NodeSC *previous = nullptr;
    current = head;
    while (current->next!=nullptr) {
        previous=current;
        current=current->next;
    }
    tail=previous;
    previous->next = nullptr;
    delete current;
    n--;
}

void MySCList::DeletePosition(int pos) {
    NodeSC *current=head;
    NodeSC *previous=nullptr;
    for (int i = 1; i<pos; i++) {
        previous = current;
        current = current->next;
    }
    previous->next = current->next;
    delete current;
    n--;
}
```

Algorithm 5.14: Delete in a sorted list.

```
void MySCList::DeleteMiddle(MyType value) {
    NodeSC* current = head;
    if (current != nullptr) {
        if (current->data == value) {
            head = current->next;
            if (tail == current)
                tail = head;
            delete current;
        }
        else {
            if (current->next != nullptr) {
                NodeSC* prev = current;
                current = current->next;
                bool found = false;
                while ((current != nullptr) && (!found)) {
                    if (current->data == value) {
                        prev->next = current->next;
                        delete current;
                        found = true;
                        if (tail == current)
                            tail = prev;
                    }
                    else {
                        prev = current;
                        current = current->next;
                    }
                }
            }
        }
    }
}
```

5.1.7 Display

Algorithm 5.15: Display the list.

```
void MySCList::Display() {
    NodeSC *tmp;
    tmp = head;
    cout << "List of size :" << n << endl;
    cout << "Head:";
    (head != nullptr) ? cout << head->data << endl : cout << "NULL" << endl;
    cout << "Tail:";
    (tail != nullptr) ? cout << tail->data << endl : cout << "NULL" << endl;
    int i = 0;
    while (tmp) {
        cout << "(" << i << ") ";
        if (tmp->next != nullptr)
            cout << tmp->data << "->" << tmp->next->data << endl;
        else
            cout << tmp->data << "-> NULL" << endl;
        tmp = tmp->next;
        i++;
    }
}

void MySCList::DisplayFile() {
    ofstream myfile;
    myfile.open("log.txt");
    NodeSC *tmp;
    tmp = head;
    while (tmp) {
        myfile << tmp->data << endl;
        tmp = tmp->next;
    }
    myfile.close();
}
```

Algorithm 5.16: Example simple chained list.

```
void main() {
    MySCList* L = new MySCList();
    L->InitSortedAscending(12, 100);
    L->Display();
    cout << "Reverse the list" << endl;
    L->Reverse();
    L->Display();
    delete L;
    }
```

5.2 Double chained list

5.2.1 Node definition

Algorithm 5.17: Node simple chained list.

```
class NodeDC {
public:
    NodeDC():
        data(0), next(nullptr), previous(nullptr) {}
    NodeDC(MyType d):
        data(d), next(nullptr), previous(nullptr) {}
    NodeDC(MyType d,NodeDC* nxt,NodeDC* prv):
        data(d), next(nxt), previous(prv) {}
    ~NodeDC() {}
    MyType data;
    NodeDC *next;
    NodeDC *previous;
};
```

Algorithm 5.18: Class interface.

```
class MyDCList : public MyDataStructure {
public:
    MyDCList();
    ~MyDCList();
    MyDataStructure* clone() { return new MyDCList(); }
    void Insert(MyType value);
    void InsertMiddle(MyType value);
    void InsertHead(MyType value);
    void InsertTail(MyType value);
    void InsertPosition(int pos, MyType value);
    void DeleteHead();
    void DeleteTail();
    void DeletePosition(int pos);
    void DeleteMiddle(MyType value);
    void Display();
    void DisplayDC();
    void DisplayFile();
    int GetSize() { return n; }
private:
    NodeDC *head, *tail;
    NodeDC *iterator_start, *iterator_end;
    int n;
};
```

Algorithm 5.19: Constructor and destructor.

```
MyDCList::MyDCList() {
    head = nullptr;
    tail = nullptr;
    n = 0;
}

MyDCList::~MyDCList() {
    NodeDC *current = head;
    while (current) {
        NodeDC *next = current->next;
        delete current;
        current = next;
    }
}
```

5.2.2 Insert

Algorithm 5.20: Insert tail.

```
void MyDCList::InsertTail(MyType value) {
    NodeDC *tmp = new NodeDC(value);
    if (head == nullptr) {
        head = tmp;
        tail = tmp;
    } else {
        tail->next = tmp;
        tmp->previous = tail;
        tail = tmp;
    }
    n++;
}
```

Algorithm 5.21: Insert head.

```
void MyDCList::InsertHead(MyType value) {
    NodeDC *tmp = new NodeDC(value,head,nullptr);
    if (head == nullptr) {
        head = tmp;
        tail = tmp;
    }
    else {
        head->previous = tmp;
        head = tmp;
    }
    n++;
}

// default insert
void MyDCList::Insert(MyType value) {
    InsertHead(value);
}
```

Algorithm 5.22: Insert at position.

```
void MyDCList::InsertPosition(int pos, MyType value) {
    NodeDC *pre = new NodeDC;
    NodeDC *cur = new NodeDC;
    NodeDC *tmp = new NodeDC;
    cur = head;
    for (int i = 1; i<pos; i++) {
        pre = cur;
        cur = cur->next;
    }
    tmp->data = value;
    pre->next = tmp;
    tmp->next = cur;
    n++;
}
```

Algorithm 5.23: Insert anywhere.

```
// Iterative version
void MyDCList::InsertMiddle(MyType value) {
    NodeDC *tmp = new NodeDC(value);
    NodeDC *cursor = head;
    if (head == nullptr) { // insert to the head
        cout << "Insert to head/tail: " << value << endl;
        head = tmp;
        tail = tmp;
    }
    else if (head->data > value) { // insert to the head
        cout << "Insert to head: " << value << endl;
        head->previous = tmp;
        tmp->next = head;
        head = tmp;
    }
    else {
        NodeDC *prev = head;
        NodeDC *cursor = head->next;
        while ((cursor != nullptr) && (cursor->data < value)) {
            prev = prev->next;
            cursor = cursor->next;
        }
        if (cursor == nullptr) { // insert to the tail
            cout << "Insert to tail: " << value << endl;
            prev->next = tmp;
            tmp->previous = prev;
            tail = tmp;
        }
        else {
            cout << "Insert middle: " << value << endl;
            prev->next = tmp;
            tmp->previous = prev;
            tmp->next = cursor;
            cursor->previous = tmp;
        }
    }
    n++;
}
```

Algorithm 5.24: Insert anywhere.

```
// Recursive version (with pointers)
void InsertMiddleDC1(NodeDC **head, NodeDC *prev, NodeDC **tail, MyType value) {
    if ((*head) == nullptr) {
        cout << "Insert to head/tail: " << value << endl;
        NodeDC *tmp = new NodeDC(value,nullptr,prev);
        *head = tmp;
        *tail = tmp;
    }
    else if ((*head)->data > value) {
        cout << "Insert to head: " << value << endl;
        NodeDC *tmp = new NodeDC(value);
        (*head)->previous = tmp;
        tmp->next = (*head);
        (*head) = tmp;
        if (prev != nullptr) {
            cout << "Insert to middle: " << value << endl;
            tmp->previous = prev;
            prev->next = tmp;
        }
    }
    else
        InsertMiddleDC1(&(*head)->next,(*head),tail, value);
}

void MyDCList::InsertMiddle1(MyType value) {
    InsertMiddleDC1(&head,nullptr,&tail, value);
    n++;
}
```

Algorithm 5.25: Insert anywhere.

```
// Recursive version (with references)
void InsertMiddleDC2(NodeDC* &head, NodeDC *prev, NodeDC* &tail, MyType value) {
    if (head==nullptr) {
        cout << "Insert to head/tail: " << value << endl;
        NodeDC *tmp = new NodeDC(value,nullptr,prev);
        head = tmp;
        tail = tmp;
    }
    else if (head->data > value) {
        cout << "Insert to head: " << value << endl;
        NodeDC *tmp = new NodeDC(value);
        head->previous = tmp;
        tmp->next = (*head);
        head = tmp;
        if (prev != nullptr) {
            cout << "Insert to middle: " << value << endl;
            tmp->previous = prev;
            prev->next = tmp;
        }
    }
    else
        InsertMiddleDC1(head->next,head,tail, value);
}

void MyDCList::InsertMiddle2(MyType value) {
    InsertMiddleDC2(head,nullptr,tail, value);
    n++;
}
```

5.2.3 Delete

Algorithm 5.26: Delete head and tail.

```
void MyDCList::DeleteHead() {
    NodeDC *tmp = new NodeDC;
    tmp = head;
    head = head->next;
    head->previous = nullptr;
    delete tmp;
    n--;
}

void MyDCList::DeleteTail() {
    NodeDC *current=nullptr;
    NodeDC *previous = nullptr;
    current = head;
    while (current->next != nullptr) {
        previous = current;
        current = current->next;
    }
    tail = previous;
    previous->next = nullptr;
    delete current;
    n--;
}
```

Algorithm 5.27: Delete position.

```
void MyDCList::DeletePosition(int pos) {
    if (pos == 0)
        DeleteHead();
    else if (pos==n-1)
        DeleteTail();
    else {
        NodeDC *current = head;
        NodeDC *previous = nullptr;
        for (int i = 1; i <= pos; i++) {
            previous = current;
            current = current->next;
        }
        previous->next = current->next;
        current->next->previous = previous;
        delete current;
        n--;
    }
}
```

Algorithm 5.28: Delete anywhere.

```
// Iterative version
void MyDCList::DeleteMiddle(MyType value) {
    if (head->data == value) { // remove to the head
        cout << "Remove to head: " << value << endl;
        NodeDC *tmp = head;
        head = head->next;
        head->previous = nullptr;
        delete tmp;
    }
    else {
        NodeDC *cursor = head;
        NodeDC *prev = cursor->previous;
        while ((cursor != nullptr) && (cursor->data!=value)) {
            prev = cursor;
            cursor = cursor->next;
        }
        if (cursor!= nullptr) { // remove
            if (cursor->next == nullptr) { // it is the tail
                cout << "Remove tail: " << value << endl;
                prev->next = nullptr;
                tail = prev;
            }
            else
            {
                cout << "Remove middle: " << value << endl;
                cursor->next->previous = prev;
                prev->next = cursor->next;
            }
            delete cursor;
        }
    }
    n--;
}
```

Algorithm 5.29: Delete anywhere.

```
// Recursive version
void DeleteMiddleDC(NodeDC **head, NodeDC **tail, MyType value) {
    if ((*head) != nullptr) {
        if ((*head)->data == value) { // remove to the head
            NodeDC *tmp = *head;
            *head = (*head)->next;
            if ((*head)==nullptr)
                *tail = tmp->previous;
            else
                (*head)->previous = tmp->previous;
            delete tmp;
        }
        else
            DeleteMiddleDC(&(*head)->next, tail, value);
    }
}

void MyDCList::DeleteMiddle1(MyType value) {
    DeleteMiddleDC(&head, &tail, value);
    n--;
}
```

5.2.4 Display

Algorithm 5.30: Display details.

```
void MyDCList::DisplayDC() {
    NodeDC *tmp;
    tmp = head;
    MyType data_prev, data_next, data_head, data_tail;
    cout << "List of size :" << n << endl;
    cout << "Head:";
    (head != nullptr) ? cout << head->data << endl : cout << "NULL" << endl;
    cout << "Tail:";
    (tail != nullptr) ? cout << tail->data << endl : cout << "NULL" << endl;
    while (tmp != nullptr) {
        if (tmp->previous != nullptr)
            data_prev = tmp->previous->data;
        else
            data_prev = -1;
        if (tmp->next != nullptr)
            data_next = tmp->next->data;
        else
            data_next = -1;
        cout << "(" << data_prev << ","
                    << tmp->data << ","
                                << data_next << ")" << endl;
        tmp = tmp->next;
    }
}
```

Algorithm 5.31: Display.

```
void MyDCList::Display() {
    NodeDC *tmp = new NodeDC;
    tmp = head;
    while (tmp != nullptr) {
        cout << tmp->data << "\n";
        tmp = tmp->next;
    }
}

void MyDCList::DisplayFile() {
    ofstream myfile;
    myfile.open("log.txt");
    NodeDC *tmp = new NodeDC;
    tmp = head;
    while (tmp != nullptr) {
        myfile << tmp->data << "\n";
        tmp = tmp->next;
    }
    myfile.close();
}
```

Algorithm 5.32: Example.

```
void main() {
    cout << "Test the Double Chained List" << endl;
    MyDCList* LD = new MyDCList();
    LD->DisplayDC();
    LD->InsertMiddle(5);
    LD->InsertMiddle(10);
    LD->InsertMiddle(15);
    LD->InsertMiddle(20);
    LD->InsertMiddle(3);
    LD->InsertMiddle(2);
    LD->InsertMiddle(1);
    LD->InsertMiddle(11);
    LD->InsertMiddle(50);
    LD->InsertMiddle(17);
    LD->InsertMiddle(-50);
    LD->DeleteMiddle(10);
    LD->DeleteMiddle(50);
    LD->DeleteMiddle(-50);
    LD->DeleteMiddle(5);
    LD->DeleteMiddle(20);
    LD->DisplayDC();
    delete LD;
}
```

5.2.5 The Sieve of Eratosthenes

Iterator

We want to browse the elements of the list from a particular element to the end. We add iterator_start and iterator_end as properties. We add the following functions to the class:

Algorithm 5.33: Iterator.

```
void GetNext() {
    iterator_start = iterator_start->next;
}
MyType GetIterator() {
    return iterator_start->data;
}
void SetStartIterator(int start) {
    iterator_start = Search(start);
}
void SetEndIterator(int end) {
    iterator_end = Search(end);
}
void SetIterator(int start, int end) {
    iterator_start = Search(start);
    iterator_end = Search(end);
}
bool IsFinishedIterator() {
    return (iterator_start == iterator_end);
}
```

Algorithm 5.34: Example.

```
void main();
    MyDCList* L = new MyDCList();
    L->InsertHead(4);
    L->InsertHead(5);
    L->InsertHead(6);
    L->InsertHead(8);
    for (L->SetIterator(0, 4); !L->IsFinishedIterator(); L->GetNext())
        cout << L->GetIterator() << endl;
    delete L;
```

Find Prime numbers

The sieve of Eratosthenes is a simple and ancient algorithm to determine all the prime numbers up to any given number.

Algorithm 5.35: Prime numbers until n.

```
void FindPrime(int n) {
    MyDCList* L=new MyDCList();
    // L contains the current list of prime numbers from 2 to i
    L->InsertTail(2);
    for (int i = 3; i < n; i++) {
        // determine if i is prime
        bool prime = true;
        // Start at the position 0 of the list
        // Finish at the end of the list
        L->SetIterator(0,L->GetSize());
        cout << "For: " << i << " Check: " << L->GetSize() << " numbers ";
        while ((!L->IsFinishedIterator()) &&
                        (pow(L->GetIterator(),2)<=i) &&
                        prime) {
            if (i % (int)L->GetIterator() == 0)
                prime = false;
            // Go to the next element in the current list of prime numbers
            L->GetNext();
        }
        if (prime) {
            L->InsertTail(i);
            cout << i << " is prime." << endl;
        }
        else
            cout << i << " is not prime." << endl;
    }
    delete L;
}
```

5.3 Circular list

5.3.1 Class definition

Algorithm 5.36: Circular list - Class definition.

```
class Cnode {
public:
    Cnode() : data(0), next(nullptr) {}
    Cnode(MyType x, Cnode* next1) : data(x), next(next1) {}
    MyType data;
    Cnode *next;
};

class MyCList : public MyDataStructure {
public:
    MyCList() : head(nullptr) {}
    ~MyCList();
    MyDataStructure* clone() { return new MyCList(); }
    void Insert(MyType value);
    void InsertBegin(MyType value);
    void InsertAfter(MyType value, int position);
    void Delete(MyType value);
    bool Search(MyType value);
    void Update(MyType value, int position);
    void Display();
private:
    Cnode* head;
};
```

Algorithm 5.37: Destructor.

```
MyCList::~MyCList() {
    if (head != nullptr) {
        Cnode *current = head->next;
        while (current != head) {
            Cnode *next = current->next;
            delete current;
            current = next;
        }
        delete head;
    }
}
```

5.3.2 Insert

Algorithm 5.38: Insert.

```
void MyCList::Insert(MyType value) {
    Cnode *temp= new Cnode(value,nullptr);
    if (head == nullptr) {
        head = temp;
        temp->next = head;
    }
    else {
        temp->next = head->next;
        head->next = temp;
        head = temp;
    }
}
// Insertion of element at beginning
void MyCList::InsertBegin(MyType value) {
    if (head == nullptr)
        cout << "First Create the list." << endl;
    else {
        Cnode *temp = new Cnode(value, head->next);
        head->next = temp;
    }
}
// Insertion of element at a particular place
void MyCList::InsertAfter(MyType value, int position) {
    if (head == nullptr) {
        cout << "First Create the list." << endl;
        return;
    }
    Cnode *temp, *s=head->next;
    for (int i = 0; i < position - 1; i++) {
        s = s->next;
        if (s == head->next) {
            cout << "There are less than "
                      << position << " in the list" << endl;
            return;
        }
    }
    temp = new Cnode(value, s->next);
    s->next = temp;
    // Element inserted at the end
    if (s == head) {
        head = temp;
    }
}
```

5.3.3 Delete

Algorithm 5.39: Delete.

```
// Deletion of element from the list
void MyCList::Delete(MyType value) {
    Cnode *temp, *s;
    s = head->next;
    // If List has only one element
    if (head->next == head && head->data == value) {
        temp = head;
        head = nullptr;
        delete temp;
        cout << "Element " << value << " deleted" << endl;
    }
    else if (s->data == value) {
        temp = s;
        head->next = s->next;
        delete temp;
        cout << "Element " << value << " deleted" << endl;
    }
    else {
        bool found = false;
        while ((s->next != head) && (!found)) {
            if (s->next->data == value) {
                temp = s->next;
                s->next = temp->next;
                delete temp;
                cout << "Element " << value << " deleted" << endl;
                found = true;
            }
            s = s->next;
        }
        if (!found) {
            if (s->next->data == value) {
                temp = s->next;
                s->next = head->next;
                delete temp;
                cout << "Element " << value << " deleted" << endl;
                head = s;
            }
            else
                cout << "Element " << value
                         << " is not found in the list" << endl;
        }
    }
}
```

5.3.4 Search and display

Algorithm 5.40: Search, Display.

```
bool MyCList::Search(MyType value) {
    Cnode *s;
    bool found = false;
    int counter = 0;
    s = head->next;
    while ((s != head) && (!found)) {
        counter++;
        if (s->data == value) {
            cout << "Element " << value
                      << " found at position " << counter << endl;
            found = true;
        }
        s = s->next;
    }
    if (!found) {
        if (s->data == value) {
            counter++;
            cout << "Element " << value
                 << " found at position " << counter << endl;
        }
        else
            cout << "Element " << value
                 << " not found in the list" << endl;
    }
    return found;
}
void MyCList::Display() {
    Cnode *s;
    if (head == nullptr) {
        cout << "List is empty" << endl;
    }
    else {
        s = head->next;
        cout << "Circular Linked List: " << endl;
        while (s != head) {
            cout << s->data << "->";
            s = s->next;
        }
        cout << s->data << endl;
    }
}
```

Algorithm 5.41: Update.

```
void MyCList::Update(MyType value, int position) {
    if (head == nullptr)
        cout << "The list is empty." << endl;
    else {
        Cnode *s;
        s = head->next;
        int i = 0;
        while ((i<position-1) && (s!=head)) {
            s = s->next;
            i++;
        }
        if (s != head)
            s->data = value;
    }
}
```

5.4 Skip list

5.4.1 Class definition

Algorithm 5.42: Skip list - Class definition.

```
class SkipNode {
public:
    SkipNode() : next(nullptr), data(0) {};
    SkipNode(MyType key, int level);
    // Array of pointers to nodes of different levels
    SkipNode **next;
    MyType data;
    int level;
};

class MySkipList : public MyDataStructure {
public:
    MySkipList();
    MySkipList(int MAXLVL, float P);
    ~MySkipList();
    MyDataStructure* clone() { return new MySkipList(); }
    void Insert(MyType x);
    void Delete(MyType x);
    bool Search(MyType x);
    void Display();
    void DisplayFile();
private:
    int RandomLevel();
    int MaxLvl; // Maximum level for this skip list
                // P is the fraction of the nodes with level
                // i pointers also having level i+1 pointers
    float P;
    // current level of skip list
    int level;
    SkipNode *head; // pointer to header node
};
```

5.4.2 Main functions

Algorithm 5.43: Skip list - functions.

```
SkipNode::SkipNode(MyType x, int level1) {
    data = x;
    level = level1;
    next = new SkipNode*[level + 1];
    // Fill the next array with NULL
    memset(next, 0, sizeof(SkipNode*)*(level + 1));
}

MySkipList::MySkipList() {
    MaxLvl = 4;
    P = 0.5;
    level = 0;
    head = new SkipNode(-1, MaxLvl);
}

MySkipList::MySkipList(int MAXLVL1, float P1) {
    MaxLvl = MAXLVL1;
    P = P1;
    level = 0;
    // -1 for smallest value
    head = new SkipNode(-1, MaxLvl);
}

MySkipList::~MySkipList() {
    SkipNode *current = head;
    while (current) {
        SkipNode *next = current->next[0];
        delete current;
        current=next;
    }
}

int MySkipList::RandomLevel() {
    float r = (float)rand()/RAND_MAX;
    int lvl = 0;
    while (r < P && lvl < MaxLvl) {
        lvl++;
        r = (float)rand()/RAND_MAX;
    }
    return lvl;
}
```

5.4.3 Insert and delete

Algorithm 5.44: Insert and delete.

```
void MySkipList::Insert(MyType x) {
    SkipNode *current = head;
    SkipNode **update=new SkipNode*[MaxLvl + 1];
    memset(update, 0, sizeof(SkipNode*)*(MaxLvl + 1));
    for (int i = level; i >= 0; i--) {
        while (current->next[i] != nullptr &&
               current->next[i]->data < x)
            current = current->next[i];
        update[i] = current;
    }
    current = current->next[0];
    if (current == nullptr || current->data != x) {
        int rlevel = RandomLevel();
        if (rlevel > level) {
            for (int i = level + 1; i<rlevel + 1; i++)
                update[i] = head;
            level = rlevel;
        }
        SkipNode* n = new SkipNode(x, rlevel);
        for (int i = 0; i <= rlevel; i++) {
            n->next[i] = update[i]->next[i];
            update[i]->next[i] = n;
        }
    }
}
void MySkipList::Delete(MyType x) {
    SkipNode *current = head;
    SkipNode **update = new SkipNode*[MaxLvl + 1];
    memset(update, 0, sizeof(SkipNode*)*(MaxLvl + 1));
    for (int i = level; i >= 0; i--) {
        while (current->next[i] != nullptr &&
            current->next[i]->data < x)
            current = current->next[i];
        update[i] = current;
    }
    if (current->next[0] != nullptr) {
        current = current->next[0];
        if (current->data == x) {
            for (int i = 0; i <= current->level; i++) {
                update[i]->next[i] = current->next[i];
            }
            delete current;
        }
    }
}
```

5.4.4 Search and display

Algorithm 5.45: Search and display.

```
bool MySkipList::Search(MyType x) {
    SkipNode *current = head;
    SkipNode **update = new SkipNode*[MaxLvl + 1];
    memset(update, 0, sizeof(SkipNode*)*(MaxLvl + 1));
    for (int i = level; i >= 0; i--) {
        while (current->next[i] != nullptr &&
            current->next[i]->data < x)
            current = current->next[i];
        update[i] = current;
    }
    if (current->next[0] == nullptr)
        return false; // not found
    if (current->next[0]->data == x)
        return true; // found
    return false; // not found
}

void MySkipList::Display() {
    for (int i = 0; i <= level; i++) {
        SkipNode *node = head->next[i];
        cout << "Level: " << i << endl;
        while (node != nullptr) {
            cout << node->data << " ";
            cout << "(" << node->level << ") ";
            node = node->next[i];
        }
        cout << endl;
    }
}

void MySkipList::DisplayFile() {
    ofstream myfile;
    myfile.open("log_skiplist.txt");
    for (int i = 0; i <= level; i++) {
        SkipNode *node = head->next[i];
        myfile << "Level: " << i << endl;
        while (node != nullptr) {
            myfile << node->data << " ";
            node = node->next[i];
        }
        myfile << endl;
    }
    myfile.close();
}
```

5.4.5 Example

Algorithm 5.46: Example.

```
void main() {
    MySkipList* s = new MySkipList(2,0.5);
    s->Insert(10);
    s->Insert(20);
    s->Insert(5);
    s->Insert(7);
    s->Insert(9);
    s->Insert(8);
    s->Insert(5);
    s->Insert(15);
    s->Insert(25);
    s->Insert(16);
    s->Insert(26);

    cout << s->Search(25) << endl;
    cout << s->Search(14) << endl;
    cout << s->Search(26) << endl;
    cout << s->Search(27) << endl;
    cout << s->Search(3) << endl;
    cout << s->Search(12) << endl;
    cout << s->Search(5) << endl;
    s->Display();

    s->Delete(15);
    s->Display();

    delete s;
}
```

Chapter 6

Hash tables

Contents

6.1 Hash tables

6.1.1 Class definition

Algorithm 6.1: Hash Table - Class definition.

```
class MyHashTable : public MyDataStructure {
public:
    MyHashTable();
    MyHashTable(int n, int m, int type);
    ~MyHashTable();
    MyDataStructure* clone() { return new MyHashTable(); }
    void Insert(MyType x);
    void Delete(MyType x);
    bool Search(MyType x);
    pair<bool, int> SearchKey(MyType x);
    void Display();
    void DisplayFile();
private:
    int HashFunction(MyType x);
    int n; // size
    int m; // modulus value
    MyType* ht; // hash table
    bool* htd; // present or not
    int type; // collision management: 0: linear, 1: quadratic probing
};
```

6.1.2 Main functions

Algorithm 6.2: Hash Table - functions.

```
MyHashTable::MyHashTable() : n(0), m(0), type(0) {}

MyHashTable::MyHashTable(int n1, int m1, int type1) {
    n = n1;
    m = m1;
    type = type1;
    ht = new MyType[n];
    htd = new bool[n];
    for (int i = 0; i < n; i++) {
        ht[i] = -1;
        htd[i] = false;
    }
}

MyHashTable::~MyHashTable() {
    delete[] ht;
    delete[] htd;
}

void MyHashTable::Delete(MyType x) {
    pair<bool, int> r = SearchKey(x);
    if (r.second!=-1)
        htd[r.second] = false;
}

int MyHashTable::HashFunction(MyType x) {
    return (int)x % m;
}
```

6.1.3 Insert

Algorithm 6.3: Hash Table - Insert.

```
void MyHashTable::Insert(MyType x) {
    int key = HashFunction(x);
    if (htd[key]) { // collision
            int probe = key + 1;
            int step = 1;
            int k = 1;
            bool place = false;
            while ((k < n) && (!place)) {
                if (!htd[probe]) {
                    place = true;
                    ht[probe] = x;
                    htd[probe] = true;
                }
                else {
                    step++;
                    if (type == 0) // linear probing
                        probe = key + step;
                    else // quadratic probing
                        probe = key + step*step;
                    probe = probe % m;
                }
                k++;
            }
    }
    else {
        ht[key] = x;
        htd[key] = true;
    }
}
```

6.1.4 Search

Algorithm 6.4: Hash Table - Search.

```
bool MyHashTable::Search(MyType x) {
    int key = HashFunction(x);
    if (ht[key] == x) {
        return true;
    }
    else {
        int probe = key + 1;
        int step = 1;
        int k = 1;
        while (k < n) {
            if ((ht[probe] == x) && (htd[probe])) { // found
                return true;
            }
            else if (!htd[probe]) // it is a hole
                return false;
            else {
                step++;
                if (type == 0) // linear probing
                    probe = key + step;
                else // quadratic probing
                    probe = key + step*step;
                probe = probe % m;
            }
            k++;
        }
        return false;
    }
}
```

Algorithm 6.5: Hash Table - SearchKey.

```
pair<bool,int> MyHashTable::SearchKey(MyType x) {
    int key = HashFunction(x);
    if (ht[key] == x) {
        return make_pair(true, key);
    }
    else {
        int probe = key + 1;
        int step = 1;
        int k = 1;
        while (k < n) {
            if ((ht[probe] == x) && (htd[probe])) { // found
                return make_pair(true,probe);
            }
            else if (!htd[probe]) // it is a hole
                return make_pair(false, -1);
            else {
                step++;
                if (type == 0) // linear probing
                    probe = key + step;
                else // quadratic probing
                    probe = key + step*step;
                probe = probe % m;
            }
            k++;
        }
        return make_pair(false, -1);
    }
}
```

6.1.5 Display

Algorithm 6.6: Hash Table - Display.

```
void MyHashTable::Display() {
    cout << "Hash table of size " << n << endl;
    for (int i = 0; i < n; i++) {
        cout << "Key: " << i << " with value: " << ht[i]
            << "(" << htd[i] << ")" << endl;
    }
    cout << endl;
}

void MyHashTable::DisplayFile() {
    ofstream myfile;
    myfile.open("log_hashtable.txt");
    myfile << "Hash table of size " << n << endl;
    for (int i = 0; i < n; i++) {
        myfile << "Key: " << i << " with value: " << ht[i]
                << "(" << htd[i] << ")" << endl;
    }
    myfile.close();
}
```

6.1.6 Examples

Algorithm 6.7: Example.

```
void main() {
    MyHashTable* H = new MyHashTable(10, 10, 1);
    H->Insert(39);
    H->Insert(13);
    H->Insert(23);
    H->Insert(63);
    H->Insert(30);
    H->Insert(31);
    H->Insert(49);
    H->Delete(49);
    H->Insert(59);
    H->Display();
    cout << "Search 39: " << H->Search(39) << endl;
    cout << "Search 49: " << H->Search(49) << endl;
    cout << "Search 59: " << H->Search(59) << endl;
    cout << "Search 23: " << H->Search(23) << endl;
    delete H;
}
```

Algorithm 6.8: Distribution of the characters in a string.

```
void main() {
  string str = "ao_abc ! zozo za"; // input string
    int n = 26; // number of letters in the alphabet
    // out: output array containing the number of letters, for each letter
    int* out = new int[n];
    // initialization
    for (int i = 0; i < n; i++)
        out[i] = 0;
    for (int i = 0; i < str.size(); i++) {

        int codechar = str[i] - 'a';
        if ((codechar >= 0) && (codechar < 26))
            out[codechar]++;
    }
    // Display output array
    for (int i = 0; i < n; i++) {
        cout << (char)('a' + i) << "  " << out[i] << endl;
    }
    delete[] out;
}
```

Chapter 7

Trees

Contents

7.1 Binary Search Trees

7.1.1 Definitions

Notation:

- **Depth**: The depth of a node: the number of edges from the root to the node.
- **Height**: The height of a node corresponds to the number of edges from the node to the deepest leaf. The height of a tree is the height of the root.
- **Levels**: The level of a particular node represents how many generations the node is from the root. The root node is at Level 0 (start at 0), the root node's children are at Level 1, the root node's grandchildren are at Level 2, etc.
- **Keys**: One data field in an object is usually designated a key value. It is used to search for the item.
- **Traversing**: To traverse a tree means to visit all the nodes in a specified order.
- **Size**: the total number of nodes in that tree

Special types of binary trees:

- **Binary Search Tree (BST)**: It is a binary tree in which a node's left child has a key less than its parent, and a node's right child has a key greater than or equal to its parent
- **Complete binary tree**: It is a binary tree in which all the nodes at one level must have values before starting the next level, and all the nodes in the last level must be completed from left to right.
- **Full binary tree**: It is a binary tree in which every node has either 0 or 2 children
- **Perfect binary trees**: It is a binary tree in which all interior nodes have 2 children and all leaves have the same depth or same level. Hence, it is a full binary tree and all leaf nodes are at the same level.

7.1.2 TreeNode

Algorithm 7.1: TreeNode definition.

```
class TreeNode {
public:
    TreeNode() : data(0), left(nullptr), right(nullptr) { }
    TreeNode(int d) : data(d), left(nullptr), right(nullptr) { }
    ~TreeNode() {}
    int data;          // data in this node
    TreeNode *left;    // pointer to the left subtree
    TreeNode *right;   // pointer to the right subtree
};
```

Algorithm 7.2: TreeNode functions.

```
void PrintNode(TreeNode* root);
int CountNodes(TreeNode* root);

// Tree traversal
void PreorderNode(TreeNode* root, void(*fct)(TreeNode* root));
void InorderNode(TreeNode* root, void(*fct)(TreeNode* root));
void PostorderNode(TreeNode* root, void(*fct)(TreeNode* root));
void PrintPreorderNode(TreeNode* root, int lvl);
void PrintInorderNode(TreeNode* root, int lvl);
void PrintPostorderNode(TreeNode* root, int lvl);
void PrintLevelOrder(TreeNode*  root);

void GetNumberNodesLevel(TreeNode*  root);
TreeNode* InvertTreeNode(TreeNode* root);
// Search
bool SearchNode(TreeNode* root, MyType data);
bool SearchNode1(TreeNode* root, MyType data);
// Insert
void InsertNode(TreeNode** root, MyType data);
void InsertNode1(TreeNode* &root, MyType data);
TreeNode* InsertNode2(TreeNode* root, MyType data);
void InsertNode3(TreeNode** root, MyType data);
void InsertNode4(TreeNode* &root, MyType data);
// Delete
TreeNode* DeleteNode(TreeNode *root, MyType data);
MyType FindMinTree(TreeNode *root);
MyType FindMaxTree(TreeNode *root);
TreeNode* FindMinNode(TreeNode *root);
TreeNode* FindMaxNode(TreeNode *root);
int MaxDepthTree2(TreeNode *root);
int MaxDepthTree(TreeNode* root);
int MinDepthTree(TreeNode *root);

void DestroyTree(TreeNode *root);

// Comparisons
bool SameTree(TreeNode* t1, TreeNode* t2);
bool IsBST(TreeNode* node, int min, int max);
```

```cpp
bool IsCompleteTree(TreeNode* root, int index, int nnodes);
bool IsFullTree(TreeNode*root);
bool IsPerfectTree(TreeNode* root);
```

Algorithm 7.3: Destroy the nodes in the tree.

```cpp
void DestroyTree(TreeNode *root) {
    if (root != nullptr) {
        DestroyTree(root->left);
        DestroyTree(root->right);
        delete root;
    }
}
```

Algorithm 7.4: Count the nodes in the tree.

```cpp
// Count the nodes in the binary tree to which root points.
int CountNodes(TreeNode* root) {
    if (!root)
        return 0;  // The tree is empty.
    else {
        int count = 1;   // Start by counting the root.
        // Add the number of nodes in the left subtree
        count += CountNodes(root->left);
        // Add the number of nodes  in the right subtree
        count += CountNodes(root->right);
        return count;
    }
}
```

7.1.3 Tree Traversal

Algorithm 7.5: Tree traversal: Pre-In-Post.

```
void PrintNode(TreeNode* root) {
    cout << root->data << " ";
}

void PreorderNode(TreeNode* root, void(*fct)(TreeNode* root)) {
    if (root!=nullptr) {
        (*fct)(root);
        PreorderNode(root->left,fct);
        PreorderNode(root->right,fct);
    }
}
void InorderNode(TreeNode* root, void(*fct)(TreeNode* root)) {
    if (root != nullptr) {
        InorderNode(root->left, fct);
        (*fct)(root);
        InorderNode(root->right, fct);
    }
}
void PostorderNode(TreeNode* root, void(*fct)(TreeNode* root)) {
    if (root != nullptr) {
        PostorderNode(root->left, fct);
        PostorderNode(root->right, fct);
        (*fct)(root);
    }
}
```

Algorithm 7.6: Tree traversal example.

```
void main() {
  TreeNode* n = nullptr;
    InsertNode1(&n, 50);
    InsertNode1(&n, 25);
    InsertNode(&n, 75);
    InsertNode(&n, 5);
    InsertNode(&n, 15);
    InsertNode(&n, 65);
    InsertNode(&n, 85);
    void(*fct)(TreeNode*)=PrintNode;
    PreorderNode(n,fct);
}
```

Algorithm 7.7: Tree traversal: Pre-In-Post.

```cpp
void PrintPreorderNode(TreeNode* root,int lvl) {
    if (root != nullptr) {
        cout << root->data << " (" << lvl << ")" << endl;
        PrintPreorderNode(root->left, lvl + 1);
        PrintPreorderNode(root->right, lvl + 1);
    }
}

void PrintInorderNode(TreeNode* root,int lvl) {
    if (root != nullptr) {
        PrintInorderNode(root->left, lvl + 1);
        cout << root->data << " (" << lvl << ")" << endl;
        PrintInorderNode(root->right, lvl + 1);
    }
}

void PrintPostorderNode(TreeNode* root, int lvl) {
    if (root != nullptr) {
        PrintPostorderNode(root->left, lvl + 1);
        PrintPostorderNode(root->right, lvl + 1);
        cout << root->data << " (" << lvl << ")" << endl;
    }
}
```

Algorithm 7.8: Tree traversal: level order.

```
// Print nodes at a given level
void PrintGivenLevel(TreeNode* root, int level) {
    if (root != nullptr) {
        if (level == 0)
            cout << " " << root->data << " " ;
        else {
            PrintGivenLevel(root->left, level - 1);
            PrintGivenLevel(root->right, level - 1);
        }
    }
}

void PrintLevelOrder(TreeNode*  root) {
    int h = MaxDepthTree(root);
    int i;
    for (i = 0; i <= h; i++) {
        PrintGivenLevel(root, i);
        // at each line we have max 2^h
        cout << endl;
    }
}
```

Algorithm 7.9: Tree traversal: Nodes per level.

```
// Nodes per level
int GetNumberNodesGivenLevel(TreeNode* root, int level) {
    if (root != nullptr) {
        if (level == 0)
            return 1;
        else
            return GetNumberNodesGivenLevel(root->left, level - 1) +
                   GetNumberNodesGivenLevel(root->right, level - 1);
    }
    else
        return 0;
}

void GetNumberNodesLevel(TreeNode*  root) {
    int h = MaxDepthTree(root);
    int i;
    for (i = 0; i <= h; i++)
        cout << "level:" << i << " with "
             << GetNumberNodesGivenLevel(root, i) << " nodes" << endl;
}
```

Algorithm 7.10: Invert Tree.

```
TreeNode* InvertTreeNode(TreeNode* root) {
    if (root==nullptr) {
        return nullptr; // terminal condition
    }
    auto left = InvertTreeNode(root->left);   // invert left sub-tree
    auto right = InvertTreeNode(root->right); // invert right sub-tree
    root->left = right; // put right on left
    root->right = left; // put left on right
    return root;
}
```

7.1.4 Search

Algorithm 7.11: Search.

```
// Recursive version
bool SearchNode(TreeNode* root, MyType data) {
    if (root == nullptr)
        return false;
    else if (root->data == data)
        return true;
    else if (data <root->data)
        SearchNode(root->left, data);
    else
        SearchNode(root->right, data);
}

// Iterative version
bool SearchNode1(TreeNode* root, MyType data) {
    if (root == nullptr)
        return false;
    else {
        TreeNode* current = root;
        bool found = false;
        while ((!found) && (current != nullptr)) {
            if (current->data == data)
                found == true;
            else if (data < current->data)
                current = current->left;
            else
                current = current->right;
        }
        return found;
    }
}
```

7.1.5 Insert

There are multiple solutions to create the Insert method. It can be set as a function that returns a new root node, or a function that take the root as input/output. The solution can be recursive (V1, V2, V3), or iterative. In both cases, it is possible to use pointers or references. The following examples show some possible implementations of the insert method.

Algorithm 7.12: Insert (recursive version).

```
// Recursive version V1
// root as input/output (pointer)
void InsertNode(TreeNode** root, MyType data) {
    if ((*root) == nullptr)
        (*root)= new TreeNode(data);
    else if ((*root)->data == data)
        cout << "Value already in the tree." << endl;
    else if (data < (*root)->data )
        InsertNode(&((*root)->left), data);
    else
        InsertNode(&((*root)->right), data);
}

// Recursive version V2
// root as input/output (reference)
void InsertNode1(TreeNode* &root, MyType data) {
    if (root == nullptr)
        root = new TreeNode(data);
    else if (root->data == data)
        cout << "Value already in the tree." << endl;
    else if (data < root->data)
        InsertNode1(root->left, data);
    else
        InsertNode1(root->right, data);
}
```

Algorithm 7.13: Insert (recursive version).

```
// Recursive version V3
// root as input only
TreeNode* InsertNode2(TreeNode* root, MyType data) {
    if (root == nullptr) {
        return  new TreeNode(data);
    }
    else if (root->data == data) {
        cout << "Value already in the tree." << endl;
        return root;
    }
    else if (data < root->data) {
        root->left = InsertNode2(root->left, data);
        return root;
    }
    else {
        root->right = InsertNode2(root->right, data);
        return root;
    }
}
```

Algorithm 7.14: Insert (iterative version).

```
// With pointer
void InsertNode3(TreeNode** root, MyType data) {
    if ((*root) == nullptr) {
        (*root) = new TreeNode(data);
    }
    else {
        TreeNode* current= (*root);
        TreeNode* parent = nullptr;
        int direction = 0;
        while (current != nullptr) {
            parent = current;
            if (current->data == data) {
                cout << "Value already in the tree." << endl;
                current = nullptr;
            }
            else if (data < current->data)
                current = current->left;
            else
                current = current->right;
        }
        current= new TreeNode(data);
        if (data < parent->data)
            parent->left = current;
        else if (data > parent->data)
            parent->right = current;
    }
}
```

Algorithm 7.15: Insert (iterative version).

```
// With reference
void InsertNode4(TreeNode* &root, MyType data) {
    if (root == nullptr) {
        root = new TreeNode(data);
    }
    else {
        TreeNode* current = root;
        TreeNode* parent = nullptr;
        int direction = 0;
        while (current != nullptr) {
            parent = current;
            if (current->data == data) {
                cout << "Value already in the tree." << endl;
                current = nullptr;
            }
            else if (data < current->data)
                current = current->left;
            else
                current = current->right;
        }
        current = new TreeNode(data);
        if (data < parent->data)
            parent->left = current;
        else if (data > parent->data)
            parent->right = current;
    }
}
```

7.1.6 Delete

Algorithm 7.16: Delete.

```
TreeNode* DeleteNode(TreeNode *root, MyType data) {
    if (root == nullptr)
        return nullptr;
    if (data < root->data)  // Data is in the left sub tree.
        root->left = DeleteNode(root->left, data);
    else if (data > root->data) // Data is in the right sub tree.
        root->right = DeleteNode(root->right, data);
    else {
        // case 1: no children
        if (root->left == nullptr && root->right == nullptr) {
            delete root;
            root = nullptr;
        }
        // case 2: 1 child (right)
        else if (root->left == nullptr) {
            TreeNode *temp = root; // Save current node as a backup
            root = root->right;
            delete temp;
        }
        // case 3: 1 child (left)
        else if (root->right == nullptr) {
            TreeNode *temp = root; // Save current node as a backup
            root = root->left;
            delete temp;
        }
        // case 4: 2 children
        else {
          // Find minimal value of right sub tree
            TreeNode *temp = FindMinNode(root->right);
            root->data = temp->data; // Duplicate the node
            // Delete the duplicate node
            root->right = DeleteNode(root->right, temp->data);
        }
    }
    return root; // parent node can update reference
}
```

7.1.7 Min, Max

Algorithm 7.17: Find minimum and maximum values.

```
MyType FindMinTree(TreeNode *root) {
    if (root==nullptr) {
        return INT_MAX; // or undefined.
    }
    if (root->left!=nullptr) {
        return FindMinTree(root->left); // left tree is smaller
    }
    return root->data;
}

MyType FindMaxTree(TreeNode *root) {
    if (root == nullptr) {
        return INT_MAX; // or undefined.
    }
    if (root->right!=nullptr) {
        return FindMaxTree(root->right); // right tree is bigger
    }
    return root->data;
}
```

Algorithm 7.18: Find minimum and maximum nodes.

```
TreeNode* FindMinNode(TreeNode *root) {
    if (root == nullptr) {
        return nullptr;
    }
    if (root->left != nullptr) {
        return FindMinNode(root->left); // left tree is smaller
    }
    return root;
}

TreeNode* FindMaxNode(TreeNode *root) {
    if (root == nullptr) {
        return nullptr;
    }
    if (root->right != nullptr) {
        return FindMaxNode(root->right); // left tree is smaller
    }
    return root;
}
```

7.1.8 Max and Min Depth

Algorithm 7.19: MaxDepthTree.

```
int MaxDepthTree2(TreeNode *root) {
    if (root == nullptr)
        return 0;
    else if ((root->left == nullptr) && (root->right == nullptr))
        return 0;
    else
        return 1 + max(MaxDepthTree2(root->left),
                       MaxDepthTree2(root->right));
}
int MaxDepthTree(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    else if ((root->left == nullptr) && (root->right == nullptr))
        return 0;
    else {
        // compute the depth of each subtree
        int leftDepth = MaxDepthTree(root->left);
        int rightDepth = MaxDepthTree(root->right);
        // use the larger subtree
        if (leftDepth > rightDepth)
            return leftDepth + 1;
        else
            return rightDepth + 1;
    }
}
```

Algorithm 7.20: MinDepthTree.

```
int MinDepthTree(TreeNode *root) {
    if (root == nullptr)
        return 0;
    // Base case : Leaf Node. This accounts for height = 1.
    if (root->left == nullptr && root->right == nullptr)
        return 1;
    // If left subtree is NULL, recur for right subtree
    if (!root->left)
        return MinDepthTree(root->right) + 1;
    // If right subtree is NULL, recur for right subtree
    if (!root->right)
        return MinDepthTree(root->left) + 1;
    return min(MinDepthTree(root->left), MinDepthTree(root->right)) + 1;
}
```

7.1.9 Comparisons

Algorithm 7.21: Same tree?

```
bool SameTree(TreeNode* t1, TreeNode* t2) {
    if (t1 == nullptr && t2 == nullptr) // both empty
        return true;
    if (t1 != nullptr && t2 != nullptr) { // both non-empty
        return  ((t1->data == t2->data) &&
                 (SameTree(t1->left, t2->left)) &&
                         (SameTree(t1->right, t2->right))
                        );
    }
    return false; // one empty, one not -> false
}
```

Algorithm 7.22: Is it a BST?

```
// True if the tree is a BST and its values are >= min and <= max.
bool IsBST(TreeNode* node, int min, int max) {
    if (!node)
        return true;
    if (node->data<min || node->data>max)
        return false;
    return (IsBST(node->left, min, node->data) &&
            IsBST(node->right, node->data + 1, max)
                    );
}
```

Algorithm 7.23: Is it a complete tree?

```
// Array representation of a binary tree
// True if the tree is complete
bool IsCompleteTree(TreeNode* root, int index, int nnodes) {
    if (root == nullptr)
        return true; // An empty tree is complete
    if (index >= nnodes)
        return false;
    return (IsCompleteTree(root->left, 2 * index + 1, nnodes) &&
            IsCompleteTree(root->right, 2 * index + 2, nnodes));
}
```

Algorithm 7.24: Is it a full tree?

```
// True if the tree is full
bool IsFullTree(TreeNode*root) {
    if ((root == nullptr) || ((root->left==nullptr) && (root->right==nullptr)))
        return true; // An empty tree is full
    else if ((root->left != nullptr) && (root->right != nullptr))
        return (IsFullTree(root->left) && IsFullTree(root->right));
    else
        return false;
}
```

Algorithm 7.25: Is it a perfect tree?

```
// True if the tree is perfect
bool IsPerfectTree(TreeNode* root) {
    if (root == nullptr)
        return true; //An empty tree is perfect
    else {
        int h = MaxDepthTree(root);
        int n = CountNodes(root);
        return (n == pow(2, h + 1) - 1);
    }
}
```

7.1.10 MyBST

Algorithm 7.26: BST interface.

```
class MyBST : public MyDataStructure {
public:
    MyBST() { root = nullptr; }
    ~MyBST() { DestroyTree(root); }

    MyDataStructure* clone() { return new MyBST(); }

    void Insert(int data) { InsertNode(&root, data); }
    void Insert1(int data) { InsertNode1(root, data); }
    void Insert2(int data) { root=InsertNode2(root, data); }
    void Insert3(int data) { InsertNode3(&root, data); }
    void Insert4(int data) { InsertNode4(root, data); }

    void Delete(int data) { root=DeleteNode(root, data); }

    void Preorder(void(*fct)(TreeNode* root)) { PreorderNode(root,fct); }
    void Inorder(void(*fct)(TreeNode* root)) { InorderNode(root,fct); }
    void Postorder(void(*fct)(TreeNode* root)) { PostorderNode(root,fct); }

    void PrintPreorder() { PrintPreorderNode(root,0); }
    void PrintInorder() { PrintInorderNode(root,0); }
    void PrintPostorder() { PrintPostorderNode(root,0); }
    void PrintLevelorder() { PrintLevelOrder(root); }
    void PrintGetNumberNodesLevel() { GetNumberNodesLevel(root); }

    void InvertTree() { InvertTreeNode(root); }
    int Height() { return MaxDepthTree(root);}
    int Size() { return CountNodes(root); }

    bool Search(MyType data) { return  SearchNode(root,data); }
    bool IsComplete() { return IsCompleteTree(root,0, CountNodes(root)); }
    bool IsFull() { return IsFullTree(root); }
    bool IsPerfect() { return IsPerfectTree(root); }

    int IsBSTv2() { return(IsBST(root, INT_MIN, INT_MAX)); }

private:
    TreeNode* root; // pointer to the root
};
```

Algorithm 7.27: BST example 1.

```
main () {
    MyBST* t = new MyBST();
    t->Insert(5);
    t->Insert(3);
    t->Insert(8);
    t->Insert(2);
    t->Insert(4);
    t->Insert(6);
    t->Insert(9);
    void(*fct)(TreeNode*) = PrintNode;
    cout << "Print Pre-order: " << endl;
    t->Preorder(fct);
    cout << "MaxDepth: " << t->Height() << endl;
    cout << "Print Pre-order: " << endl;
    t->PrintPreorder();
    cout << "Print In-order: " << endl;
    t->PrintInorder();
    cout << "Print Post-order: " << endl;
    t->PrintPostorder();
    cout << "Print Level-order: " << endl;
    t->PrintLevelorder();
    cout << "Number of nodes per level: " << endl;
    t->PrintGetNumberNodesLevel();
    cout << "IsComplete: " << t->IsComplete() << endl;
    cout << "IsFull: " << t->IsFull() << endl;
    cout << "IsPerfect: " << t->IsPerfect() << endl;
    delete t;
}
```

Algorithm 7.28: BST example 2.

```
void main() {
    MyBST* t1 = new MyBST();
    t1->Insert(5);
    t1->Insert(3);
    t1->Insert(8);
    t1->Insert(2);
    t1->Insert(4);
    cout << "IsComplete: " << t1->IsComplete() << endl;
    cout << "IsFull: " << t1->IsFull() << endl;
    cout << "IsPerfect: " << t1->IsPerfect() << endl;
    delete t1;
}
```

7.2 AVL Trees

7.2.1 Class interface

Algorithm 7.29: AVL interface.

```
class AVLnode {
public:
    MyType data;
    int balance;
    AVLnode *left, *right, *parent;
    AVLnode(MyType k, AVLnode *p) : data(k), balance(0), parent(p),
        left(nullptr), right(nullptr) {}
};

class MyAVL : public MyDataStructure {
public:
    MyAVL();
    ~MyAVL();
    MyDataStructure* clone() { return new MyAVL(); }
    void Insert(MyType x);
    void Delete(MyType x);
    bool Search(MyType x);
    void Display();
    void Display1();
private:
    void InsertNode(AVLnode** root, AVLnode* parent, MyType data);
    AVLnode* DeleteNode(AVLnode *root, MyType data);
    void rebalance(AVLnode *n);
    AVLnode *root;
};

AVLnode* rotateLeft(AVLnode *a);
AVLnode* rotateRight(AVLnode *a);
AVLnode* rotateLeftThenRight(AVLnode *n);
AVLnode* rotateRightThenLeft(AVLnode *n);

int height(AVLnode *n);
void setBalance(AVLnode *n);
void printBalance(AVLnode *n);
```

7.2.2 Class main functions

Algorithm 7.30: Constructor and destructor.

```
void DestroyTree(AVLnode *root) {
    if (root != nullptr) {
        DestroyTree(root->left);
        DestroyTree(root->right);
        delete root;
    }
}
MyAVL::MyAVL() : root(nullptr) {}
MyAVL::~MyAVL() { DestroyTree(root); }
```

Algorithm 7.31: Rebalance.

```
AVLnode* rebalance_node(AVLnode* n) {
    setBalance(n);
    if (n->balance == -2) {
        if (height(n->left->left) >= height(n->left->right))
            n = rotateRight(n);
        else
            n = rotateLeftThenRight(n);
    }
    else if (n->balance == 2) {
        if (height(n->right->right) >= height(n->right->left))
            n = rotateLeft(n);
        else
            n = rotateRightThenLeft(n);
    }
    return n;
}

void MyAVL::rebalance(AVLnode* n) {
    if (n != nullptr) {
        n=rebalance_node(n);
        if (n->parent != nullptr)
            rebalance(n->parent);
        else
            root = n; // very important to maintain the root
    }
}
```

7.2.3 Rotations and balance

Algorithm 7.32: Rotate Left.

```
AVLnode* rotateLeft(AVLnode *a) {
    if (a != nullptr) {
        AVLnode *b = a->right;
        b->parent = a->parent;
        a->right = b->left;
        if (a->right != nullptr)
            a->right->parent = a;
        b->left = a;
        a->parent = b;
        if (b->parent != nullptr) {
            if (b->parent->right == a) {
                b->parent->right = b;
            }
            else {
                b->parent->left = b;
            }
        }
        setBalance(a);
        setBalance(b);
        return b;
    }
    else
        return nullptr;
}
```

Algorithm 7.33: Rotate Right.

```
AVLnode* rotateRight(AVLnode *a) {
    if (a != nullptr) {
        AVLnode *b = a->left;
        b->parent = a->parent;
        a->left = b->right;
        if (a->left != nullptr)
            a->left->parent = a;
        b->right = a;
        a->parent = b;
        if (b->parent != nullptr) {
            if (b->parent->right == a) {
                b->parent->right = b;
            }
            else {
                b->parent->left = b;
            }
        }
        setBalance(a);
        setBalance(b);
        return b;
    }
    else
        return nullptr;
}
```

Algorithm 7.34: Rotate Left then Right.

```
AVLnode* rotateLeftThenRight(AVLnode *n) {
    n->left = rotateLeft(n->left);
    return rotateRight(n);
}
```

Algorithm 7.35: Rotate Right then Left.

```
AVLnode* rotateRightThenLeft(AVLnode *n) {
    n->right = rotateRight(n->right);
    return rotateLeft(n);
}
```

Algorithm 7.36: Height.

```
int height(AVLnode *n) {
    if (n == nullptr)
        return -1;
    return 1 + max(height(n->left), height(n->right));
}
```

Algorithm 7.37: Set Balance.

```
void setBalance(AVLnode *n) {
    if (n!=nullptr)
        n->balance = height(n->right) - height(n->left);
}
```

Algorithm 7.38: Print balance.

```
void printBalance(AVLnode *n) {
    if (n != nullptr) {
        printBalance(n->left);
        cout << n->balance << " ";
        printBalance(n->right);
    }
}
```

7.2.4 Display

Algorithm 7.39: Display.

```
void PrintGivenLevel(AVLnode* root, int level) {
    if (root != nullptr) {
        if (level == 0) {
            if (root->parent != nullptr)
                cout << " " << root->data << "(" << root->parent->data << ") ";
            else
                cout << " " << root->data << "(null) ";
        }
        else {
            PrintGivenLevel(root->left, level - 1);
            PrintGivenLevel(root->right, level - 1);
        }
    }
}

void MyAVL::Display() {
    int h = height(root);
    int i;
    for (i = 0; i <= h; i++) {
        PrintGivenLevel(root, i);
        // at each line we have max 2^h
        cout << endl;
    }
}
```

7.2.5 Search, Insert, Delete

Algorithm 7.40: Search.

```
bool SearchNode(AVLnode* root, MyType data) {
    if (root == nullptr)
        return false;
    else
    {
        if (root->data == data)
            return true;
        else if (data < root->data)
            SearchNode(root, data);
        else
            SearchNode(root, data);
    }
}

bool MyAVL::Search(MyType x) {
    return SearchNode(root, x);
}
```

Algorithm 7.41: Insert.

```
void MyAVL::InsertNode(AVLnode** root, AVLnode* parent, MyType data) {
    if ((*root) == nullptr) {
        (*root) = new AVLnode(data,parent);
        rebalance(parent);
    }
    else if ((*root)->data == data) {
        cout << "Value already in the tree." << endl;
    }
    else if (data < (*root)->data)
        InsertNode(&((*root)->left), (*root),data);
    else
        InsertNode(&((*root)->right), (*root),data);
}

void MyAVL::Insert(MyType x) {
    InsertNode(&root,nullptr,x);
}
```

Algorithm 7.42: Find minimum.

```
AVLnode* FindMinNode(AVLnode *root) {
    if (root == nullptr) {
        return nullptr;
    }
    if (root->left != nullptr) {
        return FindMinNode(root->left); // left tree is smaller
    }
    return root;
}
```

Algorithm 7.43: Delete.

```
void MyAVL::Delete(MyType x) {
    root = DeleteNode(root, x);
}
```

Algorithm 7.44: Delete.

```
AVLnode* MyAVL::DeleteNode(AVLnode *n, MyType data) {
    if (n == nullptr)
        return nullptr;
    if (data < n->data)  // data is in the left sub tree.
        n->left = DeleteNode(n->left,data);
    else if (data > n->data) // data is in the right sub tree.
        n->right = DeleteNode(n->right,data);
    else { // data == n->data
        // case 1: no children
        if (n->left == nullptr && n->right == nullptr) {
            delete n;
            n = nullptr;
        }
        // case 2: 1 child (right)
        else if (n->left == nullptr) {
            AVLnode *temp = n; // save current node as a backup
            n->right->parent = n->parent;
            n = n->right;
            delete temp;
        }
        // case 3: 1 child (left)
        else if (n->right == nullptr) {
            AVLnode *temp = n; // save current node as a backup
            n->left->parent = n->parent;
            n = n->left;
            delete temp;
        }
        // case 4: 2 children
        else {
          // find minimal value of right sub tree
            AVLnode *temp = FindMinNode(n->right);
            // duplicate the node
            n->data = temp->data;
            // delete the duplicate node
            n->right = DeleteNode(n->right,temp->data);
        }
    }
    if (n == nullptr)
        return nullptr;
    else
      return rebalance_node(n);
}
```

7.3 23-Trees

7.3.1 Class definition

Algorithm 7.45: Class interface.

```
class My23Node {
public:
    My23Node() : key1(-1),key2(-1), nkeys(0),
        left(nullptr), middle(nullptr), right(nullptr) { }
    My23Node(int k1) : key1(k1), key2(-1), nkeys(1),
        left(nullptr), middle(nullptr), right(nullptr) { }
    My23Node(int k1,int k2) : key1(k1), key2(k2), nkeys(2),
            left(nullptr), middle(nullptr), right(nullptr) { }
    ~My23Node() {}
    MyType key1,key2;   // data in this node
    My23Node *left;     // pointer to the left subtree
    My23Node *middle;   // pointer to the middle subtree
    My23Node *right;    // pointer to the right subtree
    int nkeys;
};

class My23Tree {
public:
    My23Tree() { root = nullptr; }
    ~My23Tree();
    tuple<bool, int> Search(MyType data);
    void Insert(MyType data);
    void Delete(MyType data);
    void Display();
    void DisplayLevel();
private:
    My23Node* root;
};
```

Algorithm 7.46: Methods.

```
void destroy(My23Node* root) {
    if (root != nullptr) {
        if (root->nkeys == 1) {
            destroy(root->left);
            destroy(root->middle);
        }
        else {
            destroy(root->left);
            destroy(root->middle);
            destroy(root->right);
        }
        delete root;
    }
}

My23Tree::~My23Tree() {
    if (root != nullptr)
        destroy(root);
}

tuple<bool, int> My23Tree::Search(MyType data) {
    return Search23(root, data, 0);
}

void My23Tree::Insert(MyType data) {
    Insert23(root, data);
}

void My23Tree::Delete(MyType data) {
    Delete23(root, data);
}

void My23Tree::Display() {
    Display23(root,0);
}

void My23Tree::DisplayLevel() {
    PrintLevelOrder23(root);
}
```

Algorithm 7.47: Useful functions.

```
bool IsLeaf(My23Node* root) {
    return ((root->left == nullptr)
        && (root->middle == nullptr)
        && (root->right == nullptr));
}

My23Node* minValue(My23Node* root) {
    if (root != nullptr) {
        if (root->left == nullptr)
            return root;
        else
            return minValue(root->left);
    }
    else
        return nullptr;
}

My23Node* FindInOrderSuccessor(My23Node* root,MyType data) {
    if (root != nullptr) {
        if (root->nkeys == 1) // 2 node
            return minValue(root->middle);
        else if ((root->nkeys == 2) && (root->key1 == data)) // 3 node
            return minValue(root->middle);
        else if ((root->nkeys == 2) && (root->key2 == data)) // 3 node
            return minValue(root->right);
    }
}
```

Algorithm 7.48: Order keys.

```
tuple<MyType, MyType, MyType> OrderKeys(My23Node* current, MyType k) {
    MyType a, b, c; // with a < b
    a = current->key1;
    b = current->key2;
    c = k;
    if (c < a)
        return make_tuple(c, a, b);
    else if (c>b)
        return make_tuple(a, b, c);
    else
        return make_tuple(a, c, b);
}
```

7.3.2 Search

Algorithm 7.49: Search.

```
tuple<bool,int> Search23(My23Node* root,MyType data,int level) {
    if (root != nullptr) {
        if (root->nkeys == 1) {          //2 Node
            if (data==root->key1)
                return make_tuple(true,level);
            else if (data<root->key1)
                return Search23(root->left,data,level+1);
            else
                return Search23(root->middle, data,level+1);
        }
        else { // 3 Node
            if (data==root->key1)
                return make_tuple(true,level);
            else if (data==root->key2)
                return make_tuple(true,level);
            else if (data<root->key1)
                return Search23(root->left,data,level+1);
            else if (data<root->key2)
                return Search23(root->middle,data,level+1);
            else
                return Search23(root->right,data,level+1);
        }
    }
    else
        return make_tuple(false,-1);
}
```

7.3.3 Insert

Algorithm 7.50: Insert.

```
// Always inserting into a leaf node!!!
int Insert23(My23Node* &root,MyType &data) {
    int r;
    //Case 1: Empty tree
    if (root == nullptr) {
        My23Node* tmp = new My23Node(data);
        root = tmp;
        return 1;
    }
    //Case 2: Leaf Nodes
    else if (root->left == nullptr) {
        if (root->nkeys == 1) {  //root = 2 node
            if (data < root->key1) {
                root->key2 = root->key1;
                root->key1 = data;
            }
            else
                root->key2 = data;
            root->nkeys = 2;
            return 1;
        }
        else  { //root = 3 node
            r=FixInsert(root,data,nullptr);
            return r;
        }
    }
    //Case 3: Non-Leaf Nodes
    else  {  // root = non-leaf node
        if (root->nkeys == 1) {  // root = 2 node
            if (data < root->key1) {
                r = Insert23(root->left, data);
                if (r == 2)
                    return  FixInsert(root, data, root->left);
            }
            else if (data > root->key1) {
                r=Insert23(root->middle, data);
                if (r == 2)
                    return FixInsert(root, data, root->middle);
            }
        }
        else { //root is a 3 node
            if (data < root->key1) {
                r = Insert23(root->left, data);
                if (r == 2)
                    return FixInsert(root, data, root->left);
            }
            else if (data < root->key2) {
                r = Insert23(root->middle, data);
                if (r == 2)
                    return FixInsert(root, data, root->middle);
```

```
            }
            else {
                r = Insert23(root->right, data);
                if (r == 2)
                    return FixInsert(root, data, root->right);
            }
        }
    }
}
```

Algorithm 7.51: Insert Upward.

```
int FixInsert(My23Node* &current, MyType &k, My23Node* child) {
        if (current->nkeys == 1) { // 2 node parent
            if (current->left==child) { // left child of parent
                current->key2 = current->key1;
                current->key1 = k;
                current->nkeys = 2;
                current->right = current->middle;
                current->middle = child->middle;
                current->left = child->left;
                delete child;
                return 1;
            }
            else { // middle child of parent
                current->key2 = k;
                current->nkeys = 2;
                current->middle = child->left;
                current->right = child->middle;
                delete child;
                return 1;
            }
        }
        else { // 3 parent node
            if (current->left == nullptr) {
                MyType S, M, L;
                tie(S, M, L) = OrderKeys(current, k);
                My23Node* tmp1 = new My23Node(S);
                My23Node* tmp2 = new My23Node(L);
                current->key1 = M;
                current->nkeys = 1;
                current->left = tmp1;
                current->middle = tmp2;
                current->right = nullptr;
                k = M; // update the value to insert for the upper level
                return 2;
            }
            else {
                if (current->left == child) {
                    My23Node* tmp1 = new My23Node(current->key2);
                    tmp1->left = current->middle;
                    tmp1->middle = current->right;
                    current->nkeys = 1;
                    current->left = child;
                }
                else if (current->middle == child) {
                    My23Node* tmp1 = new My23Node(current->key2);
                    tmp1->left = child->middle;
                    tmp1->middle = current->right;
                    current->nkeys = 1;
                    current->middle = child->left;
                    child->left = current;
                    child->middle = tmp1;
                    current = child;
                }
```

```
                else { // right child
                    My23Node* tmp1 = new My23Node(current->key1);
                    current->key1 = current->key2;
                    current->nkeys = 1;
                    tmp1->middle = current->middle;
                    tmp1->left = current->left;
                    current->middle = child;
                    current->left = tmp1;
                }
            }
    }
    return 0;
}
```

7.3.4 Delete

Algorithm 7.52: Delete.

```
My23Node* Delete23(My23Node* &root, MyType data) {
    My23Node* child=nullptr;
    //Case 1: Empty tree
    if (root == nullptr) {
        cout << "Empty tree." << endl;
        return nullptr;
    }
    //Case 2: Leaf Nodes
    else if (root->left == nullptr) {
        if (root->nkeys == 2) {  //root = 3 node
            if (data == root->key1) {
                root->key1 = root->key2;
                root->nkeys = 1;
            }
            else if (data == root->key2) {
                root->nkeys = 1;
            }
            else
                cout << "Key:" << data << " unfound." << endl;
        }
        else { //root = 2 node : only 1 element !!
            if (data == root->key1) {
                return root;
            }
            else {
                cout << "Key:" << data << " unfound." << endl;
                return nullptr;
            }
        }
    }
    //Case 3: Non-Leaf Nodes
    else {  // root = non-leaf node
        if (root->nkeys == 1) {  // root = 2 node
            if (data < root->key1)
                child=Delete23(root->left, data);
            else if (data > root->key1) {
                child=Delete23(root->middle, data);
            }
            else {// found
                My23Node* successor = FindInOrderSuccessor(root, data);
                MyType tmp = successor->key1;
                child = Delete23(root->middle, tmp);
                root->key1 = tmp;
                return nullptr;
            }
        }
        else { //root is a 3 node
            if (data == root->key1) {
                My23Node* successor = FindInOrderSuccessor(root, data);
                MyType tmp = successor->key1;
```

```
                    child = Delete23(root->middle, tmp);
                    root->key1 = tmp;
                    return nullptr;
                }
                else if (data == root->key2) {
                    My23Node* successor = FindInOrderSuccessor(root, data);
                    MyType tmp = successor->key1;
                    child = Delete23(root->right, tmp);
                    root->key2 = tmp;
                    return nullptr;
                }
                else if (data < root->key1)
                    child=Delete23(root->left, data);
                else if (data < root->key2)
                    child=Delete23(root->middle, data);
                else
                    child=Delete23(root->right, data);
            }
        }
        if (child != nullptr) FixDelete(root, child);
        return child;
}
```

Algorithm 7.53: Fix delete.

```
void FixDelete(My23Node* &current, My23Node* &child) {
    if (current->nkeys == 1) {
        // Redistribute right
        if ((current->middle == child) &&
           (current->left->nkeys==2)) {
            child->key1 = current->key1;
            current->key1 = current->left->key2;
            current->left->nkeys = 1;
        }
        // Redistribute left
        else if ((current->left == child) &&
            (current->middle->nkeys == 2)) {
            child->key1 = current->key1;
            current->key1 = current->middle->key1;
            current->middle->key1 = current->middle->key2;
            current->middle->nkeys = 1;
        }
        // Merge right
        else if ((current->middle == child) &&
            (current->left->nkeys == 1)) {
            current->key2 = current->key1;
            current->key1 = current->left->key1;
            current->nkeys = 2;
            current->left = nullptr;
            current->middle = nullptr;
            delete child;
            delete current->left;
        }
        // Merge left
        else if ((current->left == child) &&
            (current->middle->nkeys == 1)) {
            current->key2 = current->middle->key1;
            current->nkeys = 2;
            current->left = nullptr;
            current->middle = nullptr;
            delete child;
            delete current->middle;
        }
    }
    else { // (parent->nkeys == 2)
        // child to delete is on the left
        if ((current->left == child) &&
            (current->middle->nkeys == 1)) {
            child->key1 = current->key1;
            current->middle->key2 = current->right->key1;
            current->key1 = current->middle->key1;
            current->middle->key1 = current->key2;
            current->middle->nkeys = 2;
            current->nkeys = 1;
            delete current->right;
        }
        else if ((current->left == child) &&
            (current->middle->nkeys == 2)) {
```

```
                child->key1 = current->key1;
                current->key1 = current->middle->key1;
                current->middle->key1 = current->middle->key2;
                current->middle->nkeys = 1;
            }
            // child to delete is on the right
            if ((current->right == child) &&
                (current->middle->nkeys == 1)) {
                current->left->key2 = current->key1;
                current->left->nkeys = 2;
                current->key1 = current->middle->key1;
                current->middle->key1 = current->key2;
                current->nkeys= 1;
                delete current->right;
            }
            else if ((current->right == child) &&
                (current->middle->nkeys == 2)) {
                current->right->key1 = current->key2;
                current->key2 = current->middle->key2;
                current->middle->nkeys = 1;
            }
            // child to delete is in the middle
            else if ((current->middle == child) &&
                (current->left->nkeys == 1) &&
                (current->right->nkeys == 1)) {
                current->left->key2 = current->key1;
                current->left->nkeys = 2;
                delete current->middle;
                current->middle = current->right;
                current->key1 = current->key2;
                current->nkeys = 1;
            }
            else if ((current->middle == child) &&
                (current->left->nkeys == 2) &&
                (current->right->nkeys == 2)) {
                child->key1 = current->key1;
                child->key2 = current->key2;
                current->key1 = current->left->key2;
                current->key2 = current->right->key1;
                current->right->key1 = current->right->key2;
                current->left->nkeys = 1;
                current->right->nkeys = 1;
            }
            else if ((current->middle == child) &&
                (current->left->nkeys == 2) &&
                (current->right->nkeys == 1)) {
                current->middle->key1 = current->key1;
                current->key1 = current->left->key2;
                current->left->nkeys = 1;
            }
            else if ((current->middle == child) &&
                (current->left->nkeys == 1) &&
                (current->right->nkeys == 2)) {
                current->middle->key1 = current->key2;
```

```
            current->key2 = current->right->key1;
            current->right->key1 = current->right->key2;
            current->right->nkeys = 1;
        }
    }
}
```

7.3.5 Display

Algorithm 7.54: Print nodes at a given level.

```
void PrintGivenLevel23(My23Node* root, int level) {
    if (root != nullptr) {
        if (level == 0) {
            if (root->nkeys==1)
                cout << " (" << root->key1 << ") ";
            else
                cout << " (" << root->key1 << ","
               << root->key2 << ") ";
        }
        else {
            if (root->nkeys == 1) {
                PrintGivenLevel23(root->left, level - 1);
                PrintGivenLevel23(root->middle, level - 1);
            }
            else {
                PrintGivenLevel23(root->left, level - 1);
                PrintGivenLevel23(root->middle, level - 1);
                PrintGivenLevel23(root->right, level - 1);
            }
        }
    }
}
```

Algorithm 7.55: MaxDepthTree.

```
int MaxDepthTree23(My23Node*  root) {
    if (root == nullptr) {
        return 0;
    }
    else if ((root->left == nullptr) && (root->middle == nullptr)
           && (root->right==nullptr))
        return 0;
    else {
        if (root->nkeys == 1) {
            int leftDepth = MaxDepthTree23(root->left);
            int middleDepth = MaxDepthTree23(root->middle);
            if (leftDepth > middleDepth)
                return leftDepth + 1;
            else
                return middleDepth + 1;
        }
        else {
            int leftDepth = MaxDepthTree23(root->left);
            int middleDepth = MaxDepthTree23(root->middle);
            int rightDepth = MaxDepthTree23(root->right);
            if ((leftDepth >= middleDepth) && (leftDepth >= rightDepth))
                return leftDepth + 1;
            else if ((middleDepth >= leftDepth) && (middleDepth >= rightDepth))
                return middleDepth + 1;
            else
                return rightDepth + 1;
        }
    }
}
```

Algorithm 7.56: PrintLevelOrder.

```
void PrintLevelOrder23(My23Node*  root) {
    int h = MaxDepthTree23(root);
    cout << "Height:" << h << endl;
    int i;
    for (i = 0; i <= h; i++) {
        PrintGivenLevel23(root, i);
        // at each line we have max 2^h
        cout << endl;
    }
}
```

Algorithm 7.57: Display.

```
void Display23(My23Node* root,int level) {
    if (root != nullptr) {
        if (root->nkeys == 1) { // 2 node
            Display23(root->left, level+1);
            cout << " (" << root->key1 << "," << level << ")";
            Display23(root->middle,level+1);
        }
        else { // 3 Node
            Display23(root->left,level+1);
            cout << " (" << root->key1 << "," << level << ")";
            Display23(root->middle, level + 1);
            cout << " (" << root->key2 << "," << level << ")";
            Display23(root->right, level + 1);
        }
    }
    else
        cout << " NULL ";
}
```

7.4 Red & Black Trees

7.4.1 Definitions

Red-Black Trees (RBTs) are a type of Binary Search Tree (BST) with one additional bit of storage per node, its color that can be defined as RED or BLACK. There is a constraint of the node colors on any simple path from the root to a leaf, to ensure that no such path is more than two times as long as any other, so the tree is more or less balanced. Therefore, each node of an RBT has the following attributes: Color, Key, a pointer to the Left subtree, a pointer to the Right subtree, and a pointer to the Parent node. If a child or the parent of a node does not exist, then the corresponding pointer attribute of the node contains the value NULL. The black-height of a node x is defined by $bh(x)$. It corresponds to the number of black nodes on any simple path from x (excluded) down to a leaf in the tree.

There are 5 key properties in an RBT:

1. All the nodes are Red or Black.
2. The color of the root is Black.
3. All the leaves are Black.
4. If the color of a node is Red then both children of this node are Black.
5. For each node of an RBT, all simple paths from a node to descendant leaves has the same number of black nodes.

7.4.2 Family

If we consider a node x, then the parent is $x.p$, the grand-parent is $x.p.p$. The uncle of x is $x.p.p.left$ if $x.p.p.right == x.p$, otherwise if it is $x.p.p.right$.

7.4.3 Class definition

Algorithm 7.58: Class interface.

```
enum class Color {
    RED,
    BLACK
};

class MyRBNode {
public:
    MyRBNode(): left(nullptr), right(nullptr), parent(nullptr),
        color(Color::RED), key(-1) {}
    // Default color is RED for insert
    MyRBNode(MyType k): left(nullptr), right(nullptr), parent(nullptr),
        color(Color::RED), key(k) {}
    MyRBNode* left;
    MyRBNode* right;
    MyRBNode* parent;
    Color color;
    MyType key;
};

class MyRBTree
{
public:
    MyRBTree() { root = nullptr; }
    ~MyRBTree();
    tuple<bool, Color, MyType> Search(MyType key);
    void Insert(MyType key);
    void Delete(MyType key);
    void Display();
private:
    void TransplantRB(MyRBNode* u, MyRBNode* v);
    void InsertFixup(MyRBNode* z);
    void DeleteFixup(MyRBNode* x);
    void LeftRotation(MyRBNode* x);
    void RightRotation(MyRBNode* x);
    MyRBNode* root;
};
```

Algorithm 7.59: Family business.

```
bool IsRed(MyRBNode* z) {
    if (z == nullptr)
        return false;
    else
        return (z->color == Color::RED);
}
bool IsBlack(MyRBNode* z) {
    return z->color == Color::BLACK;
}
void SetRed(MyRBNode* z) {
    z->color = Color::RED;
}
void SetBlack(MyRBNode* z) {
    z->color = Color::BLACK;
}
MyRBNode* GrandParent(MyRBNode* z) {
    return z->parent->parent;
}
bool IsLeftFamily(MyRBNode* z) {
    return (z->parent == z->parent->parent->left);
}
bool IsLeftChild(MyRBNode* x) {
    return (x == x->parent->left);
}
MyRBNode* GetRightSibling(MyRBNode* x) {
    return x->parent->right;
}
MyRBNode* GetLeftSibling(MyRBNode* x) {
    return x->parent->left;
}
MyRBNode* GetRightUncle(MyRBNode* z) {
    return z->parent->parent->right;
}
MyRBNode* GetLeftUncle(MyRBNode* z) {
    return z->parent->parent->left;
}
```

Algorithm 7.60: Destructor.

```
void destroyRB(MyRBNode* root) {
    if (root != nullptr) {
        destroyRB(root->left);
        destroyRB(root->right);
        delete root;
    }
}
MyRBTree::~MyRBTree() {
    destroyRB(root);
};
```

Algorithm 7.61: Transplant and TreeMinimum.

```
void MyRBTree::TransplantRB(MyRBNode* u, MyRBNode* v) {
    if (u->parent == nullptr)
        root = v;
    else if (IsLeftChild(u))
        u->parent->left = v;
    else
        u->parent->right = v;
    if (v!=nullptr)
        v->parent = u->parent;
}

MyRBNode* TreeMinimum(MyRBNode* x) {
    MyRBNode* y = x;
    while (y->left != nullptr)
        y = y->left;
    return y;
}
```

Algorithm 7.62: Rotations again.

```
void MyRBTree::LeftRotation(MyRBNode* x) {
    if (x->right != nullptr) {
        MyRBNode* y = x->right;
        x->right = y->left;
        if (y->left != nullptr)
            y->left->parent = x;
        y->parent = x->parent;
        if (x->parent == nullptr)
            root = y;
        else if (IsLeftChild(x))
            x->parent->left = y;
        else
            x->parent->right = y;
        y->left = x;
        x->parent = y;
    }
}

void MyRBTree::RightRotation(MyRBNode* x) {
    if (x->left != nullptr) {
        MyRBNode* y = x->left;
        x->left = y->right;
        if (y->right != nullptr)
            y->right->parent = x;
        y->parent = x->parent;
        if (x->parent == nullptr)
            root = y;
        else if (IsLeftChild(x))
            x->parent->left = y;
        else
            x->parent->right = y;
        y->right = x;
        x->parent = y;
    }
}
```

7.4.4 Search

Algorithm 7.63: Search.

```
tuple<bool,Color,MyType> SearchRB(MyRBNode* root, MyType data) {
    if (root == nullptr)
        return make_tuple(false,Color::RED,0);
    else if (root->key == data)
        return make_tuple(true,root->color,root->key);
    else if (data <root->key)
        SearchRB(root->left, data);
    else
        SearchRB(root->right, data);
}

tuple<bool, Color, MyType> MyRBTree::Search(MyType key)
{ return SearchRB(root, key); }
```

7.4.5 Insert

Algorithm 7.64: Insert.

```
void MyRBTree::Insert(MyType v) {
    MyRBNode* z = new MyRBNode(v);
    MyRBNode* y = nullptr;
    MyRBNode* x = root;
    while (x != nullptr) {
        y = x;
        if (z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;
    if (y == nullptr)
        root = z;
    else if (z->key < y->key)
        y->left = z;
    else
        y->right = z;
    InsertFixup(z);
}
```

Algorithm 7.65: InsertFixup.

```
void MyRBTree::InsertFixup(MyRBNode* z) {
    while ((z->parent!=nullptr) && IsRed(z->parent)) {
        if (IsLeftFamily(z)) {
            MyRBNode* y = GetRightUncle(z);
            if (IsRed(y)) { // Case 1
                SetBlack(z->parent);
                SetBlack(y);
                SetRed(z->parent->parent);
                z = z->parent->parent;
            }
            else {
                if (z == z->parent->right) {
                    z = z->parent;
                    LeftRotation(z);
                }
                SetBlack(z->parent);
                SetRed(z->parent->parent);
                RightRotation(z->parent->parent);
            }
        }
        else {
            MyRBNode* y = GetLeftUncle(z);
            if (IsRed(y)) { // Case 1
                SetBlack(z->parent);
                SetBlack(y);
                SetRed(z->parent->parent);
                z = z->parent->parent;
            }
            else {
                if (IsLeftChild(z)) { // Case 2
                    z = z->parent;
                    RightRotation(z);
                } // Case 3
                SetBlack(z->parent);
                SetRed(z->parent->parent);
                LeftRotation(z->parent->parent);
            }
        }
    }
    SetBlack(root);
}
```

7.4.6 Delete

Algorithm 7.66: Delete.

```
void MyRBTree::Delete(MyType v) {
    if (root == nullptr)
        cout << "Empty Tree." << endl;
    else {
        MyRBNode *z=root;
        MyRBNode* y = nullptr;
        MyRBNode* x = nullptr;
        bool found = false;
        while (z!= nullptr && !found) {
            if (z->key == v)
                found = true;
            if (!found) {
                if (z->key < v)
                    z = z->right;
                else
                    z = z->left;
            }
        }
        if (!found)
            cout << "Element not found" << endl;
        else {
            y = z;
            Color yoriginal = y->color;
            if (z->left == nullptr) {
                x = z->right;
                TransplantRB(z, z->right);
            }
            else if (z->right == nullptr) {
                x = z->left;
                TransplantRB(z, z->left);
            }
            else { // 2 children
                y = TreeMinimum(z->right);
                yoriginal = y->color;
                x = y->right;
                if (y->parent == z)
                    x->parent = y;
                else {
                    TransplantRB(y,y->right);
                    y->right = z->right;
                    y->right->parent = y;
                }
                TransplantRB(z,y);
                y->left = z->left;
                y->left->parent = y;
                y->color = z->color;
            }
            if ((yoriginal == Color::BLACK) && (x!=nullptr))
                DeleteFixup(x);
        }
```

```
        }
}
```

Algorithm 7.67: DeleteFixup.

```
void MyRBTree::DeleteFixup(MyRBNode* x) {
    while ((x != root) && IsBlack(x)) {
        if (IsLeftChild(x)) {
            MyRBNode* w = GetRightSibling(x);
            if (IsRed(w)) {
                SetBlack(w);
                SetRed(x->parent);
                LeftRotation(x->parent);
                w = GetRightSibling(x);
            }
            if (IsBlack(w->left) && IsBlack(w->right)) {
                SetRed(w);
                x = x->parent;
            }
            else {
                if (IsBlack(w->right)) {
                    SetBlack(w->left);
                    SetRed(w);
                    RightRotation(w);
                    w = GetRightSibling(x);
                }
                else {
                    w->color = x->parent->color;
                    SetBlack(x->parent);
                    SetBlack(w->right);
                    LeftRotation(x->parent);
                    x = root;
                }
            }
        }
        else {
            MyRBNode* w = GetLeftSibling(x);
            if (IsRed(w)) {
                SetBlack(w);
                SetRed(x->parent);
                RightRotation(x->parent);
                w = GetLeftSibling(x);
            }
            if (IsBlack(w->left) && IsBlack(w->right)) {
                SetRed(w);
                x = x->parent;
            }
            else {
                if (IsBlack(w->left)) {
                    SetBlack(w->right);
                    SetRed(w);
                    LeftRotation(w);
                    w = GetLeftSibling(x);
                }
                else {
                    w->color = x->parent->color;
                    SetBlack(x->parent);
                    SetBlack(w->left);
```

```
                    RightRotation(x->parent);
                    x = root;
                }
            }
        }
    }
    SetBlack(x);
}
```

Chapter 8

Heaps

Contents

8.1 Binary Heaps

A binary heap is a binary tree with two constraints:

- **Shape property**: A binary heap is a complete binary tree. All the levels of the tree (except possibly the last one) are fully filled. If the last level is not complete, then the level is filled from left to right.
- **Heap property**: The key stored in each node is either $\geq$ or $\leq$ to the keys in the node's children - according to some total order.

- **Sift-Down**: Swaps a node that is too small with its largest child (moving it down) until it is at least as large as both nodes below it.
- **Sift-Up**: Swaps a node that is too large with its parent (moving it up) until it is no larger than the node above it
- **BuildHeap**: Array of unsorted items and moves them until it they all satisfy the heap property

The members of the class MyMaxHeap are:

- a: A pointer to a MyType, corresponding to an array of elements of type MyType.
- n: The size of the array that corresponds to the number of elements in the associated binary heap that is equivalent to a complete tree.
- heap: A boolean that indicates if the array a is a heap or not. It is useful after sorting the array to know if we have a sorted array or if it remains a max heap.

Algorithm 8.1: Class interface.

```
class MyMaxHeap {
public:
    MyMaxHeap();
    ~MyMaxHeap();
    void InitArray(int n1);
    void InitHeap(int n1);
    bool IsMaxHeap();
    bool IsSorted();
    void MaxHeapify(int i, int size);
    void BuildMaxHeap();
    void HeapSort();
    void InsertMaxHeap(int x);
    void DeleteMaxHeap(int x);
    void Display();
private:
    MyType* a;
    int n;
    bool heap;
};
```

Algorithm 8.2: Heap constructor and destructor.

```
MyMaxHeap::MyMaxHeap() {
    n = 0;
    a = nullptr;
    heap = true;
}

MyMaxHeap::~MyMaxHeap() {
    delete[] a;
}
```

Algorithm 8.3: Heap methods.

```
void MyMaxHeap::MaxHeapify(int i, int size) {
    int left, right, largest;
    left = 2 * i + 1;  // left child of a[i]
    right = 2 * i + 2; // right child of a[i]
    largest = i; // original largest element
    if ((left<size) && a[left]>a[i])
        largest = left;
    if ((right<size) && (a[right]>a[largest]))
        largest = right;
    if (largest != i) { // there was a swap
        swap(a[i], a[largest]);
        MaxHeapify(largest, size);
    }
}

// Determine if the array corresponds to a max heap
bool MyMaxHeap::IsMaxHeap() {
    bool isheap = true;
    int left, right;
    int i = 0;
    for (int i = 0; (i < n) && isheap; i++) {
        left = 2 * i + 1;
        right = 2 * i + 2;
        if ((left < n) && a[left] > a[i])
            isheap = false;
        if ((right < n) && (a[right] > a[i]))
            isheap = false;
    }
    heap = isheap;
    return isheap;
}

// Determine if the elements in the array are sorted
bool MyMaxHeap::IsSorted() {
    if (n <= 1)
        return true;
    else {
        int i = 1;
        while ((a[i - 1] <= a[i]) && (i < n))
            i++;
        return (i == n);
```

```
    }
}

void MyMaxHeap::BuildMaxHeap() {
    // Traverse the complete tree backward
    // (right to left, bottom to top)
    for (int i = n / 2 - 1; i >= 0; i--) {
        MaxHeapify(i, n);
    }
    heap = true;
}
```

Algorithm 8.4: Init heap or array.

```
// Create a array of size n1 with random values
void MyMaxHeap::InitArray(int n1) {
    if (n1 > 0) {
        delete[] a;
        n = n1;
        a = new MyType[n];
        for (int i = 0; i < n; i++)
            a[i] = rand() % 200;
    }
    else
        PositiveNumberCheck(__func__);
}


// Create a heap with n1 random elements
void MyMaxHeap::InitHeap(int n1) {
    InitArray(n1);
    BuildMaxHeap();
}
```

The first step is to build a Max Heap with the elements that are in the array. The maximum element will be therefore at the first position. Then, it will be moved at its final place in the array. The element that was at the end of the array will then come at the beginning. This swap is likely to break the max heap property of the data structure. Hence, we have to rebuild the heap in relation to this change.

Algorithm 8.5: Heapsort.

```
void MyMaxHeap::HeapSort() {
    // First build the MaxHeap
    BuildMaxHeap();
    // Go backward as the Max is placed at the end
    // Invariant: the elements between i to n-1 are at their correct places
    // i is the size of the heap
    for (int i = n - 1; i>0; i--) {
        // a[0] always contains the max value
        swap(a[0], a[i]);
        // the heap constraint must be maintained
        MaxHeapify(0, i);
    }
    heap = false;
}
```

Algorithm 8.6: Heap insert, delete, display.

```
void MyMaxHeap::InsertMaxHeap(int x) {
    MyType* a1 = new MyType[n + 1];
    a1[0] = x;
    for (int i = 0; i < n; i++) {
        a1[1 + i] = a[i];
    }
    delete[] a;
    a = a1;
    MaxHeapify(0, n + 1);
    n++;
}

void MyMaxHeap::DeleteMaxHeap(int x) {
    int i = 0, indx = 0;
    bool found = false;
    while ((i < n) && (!found)) {
        if (a[i] == x) {
            found = true;
            indx = i;
        }
        i++;
    }
    if (found) {
        MyType* a1 = new MyType[n - 1];
        a[indx] = a[n - 1];
        for (int i = 0; i < n - 1; i++) {
            a1[i] = a[i];
        }
        delete[] a;
        a = a1;
        MaxHeapify(indx, n - 1);
        n--;
    }
}

void MyMaxHeap::Display() {
    for (int i = 0; i < n; i++) {
        cout << a[i] << " ";
    }
    cout << endl;
}
```

Algorithm 8.7: Heapsort example.

```
void main() {
    int n = 10;
    MyMaxHeap* H = new MyMaxHeap();
    H->InitArray(n);
    cout << "Sorted: " << H->IsSorted() << endl;
    H->HeapSort();
    cout << "Sorted: " << H->IsSorted() << endl;
    H->BuildMaxHeap(); // Need to build the heap first
    cout << "IsHeap:" << H->IsMaxHeap() << endl;
    H->Display();
    cout << "Insert.." << endl;
    H->InsertMaxHeap(99); // Insert 99 in the heap
    H->Display();
    cout << "IsHeap:" << H->IsMaxHeap() << endl;
    cout << "Insert.." << endl;
    H->DeleteMaxHeap(99); // Delete 99 from the heap
    H->Display();
    cout << "IsHeap:" << H->IsMaxHeap() << endl;
    delete H;
}
```

8.2 Priority Queue

The Priority Queue is implemented as a Min Heap.

Algorithm 8.8: MyPriorityQueue class interface.

```
struct MyData {
    int index;
    double value;
};

class MyPriorityQueue
{
public:
    MyPriorityQueue();
    MyPriorityQueue(int c);
    ~MyPriorityQueue();
    int GetNElements() { return nelements; }
    bool IsEmpty();
    bool IsFull();
    int GetParent(int child);
    int GetLeftChild(int parent);
    int GetRightChild(int parent);
    void Push(int index, double value);
    MyData Pop();
    void DecreaseKey(int index, double value);
    void BuildMinHeap();
    void Display();
private:
    void MinHeapify(int i, int size);
    int capacity;
    int nelements;
    MyData* queue;
};
```

Algorithm 8.9: Constructors and destructor.

```
MyPriorityQueue::MyPriorityQueue() {
    capacity = 0;
    nelements = 0;
    queue = nullptr;
}

MyPriorityQueue::MyPriorityQueue(int c) {
    capacity = c;
    nelements = 0;
    queue = new MyData[capacity];
}

MyPriorityQueue::~MyPriorityQueue() {
    delete[] queue;
}
```

Algorithm 8.10: Methods.

```
bool MyPriorityQueue::IsEmpty() {
    return (nelements == 0);
}

bool MyPriorityQueue::IsFull() {
    return (nelements == capacity);
}

int MyPriorityQueue::GetParent(int child) {
    if (child % 2 == 0)
        return (child / 2) - 1;
    else
        return child / 2;
}

int MyPriorityQueue::GetLeftChild(int parent) {
    return (2 * parent + 1);
}

int MyPriorityQueue::GetRightChild(int parent) {
    return (2 * parent + 2);
}
```

Algorithm 8.11: Heap methods.

```
void MyPriorityQueue::MinHeapify(int i, int size) {
    int left, right, smallest;
    left = 2 * i + 1;  // left child of a[i]
    right = 2 * i + 2; // right child of a[i]
    smallest = i; // original smallest element
    if ((left<size) && queue[left].value<queue[i].value)
        smallest = left;
    if ((right<size) && (queue[right].value<queue[smallest].value))
        smallest = right;
    if (smallest != i) { // there was a swap
        swap(queue[i], queue[smallest]);
        MinHeapify(smallest, size);
    }
}

void MyPriorityQueue::BuildMinHeap() {
    // Traverse the complete tree backward
    // (right to left, bottom to top)
    for (int i = nelements / 2 - 1; i >= 0; i--) {
        MinHeapify(i, nelements);
    }
}
```

Algorithm 8.12: Push/Enqueue.

```
void MyPriorityQueue::Push(int index,double value) {
    if (nelements < capacity) {
        MyData x;
        x.index = index;
        x.value = value;

        int i = nelements;
        while ((i != 0) && (x.value < queue[i / 2].value)) {
            queue[i] = queue[i / 2]; // move down
            i /= 2;
        } // move to parent
        queue[i] = x;
        nelements++;
        cout << "Added (" << index
                  << "," << value << ") size="
                  << nelements << endl;
    }
    else
        cout << "Out of capacity" << endl;
}
```

Algorithm 8.13: Pop/Dequeue.

```
MyData MyPriorityQueue::Pop() {
    if (nelements > 0) {
        MyData tmp = queue[0];
        queue[0] = queue[nelements - 1];
        MinHeapify(0, nelements);
        nelements--;
        return tmp;
    }
    else
    {
        MyData tmp;
        tmp.value = 0;
        tmp.index = -1;
        return tmp;
    }
}
```

Algorithm 8.14: Display.

```
void MyPriorityQueue::Display() {
    cout << "Priority Queue" << endl;
    cout << "\t capacity: " << capacity << endl;
    cout << "\t nelements: " << nelements << endl;
    for (int i = 0; i < nelements; i++) {
        cout << i << ": (" << queue[i].index << ","
                           << queue[i].value << ")" << endl;
    }
    cout << endl;
}
```

Algorithm 8.15: Decrease key.

```
void MyPriorityQueue::DecreaseKey(int index, double value) {
    int i = 0;
    bool found = false;
    while ((!found) && (i < nelements)) {
        if (queue[i].index == index) {
            queue[i].value = value;
            found = true;
        }
        i++;
    }
    if (found) {
        int child = i-1;
        int parent = GetParent(child);
        while ((queue[child].value < queue[parent].value) &&
               (child >= 0 && parent >= 0)) {
            swap(queue[child],queue[parent]);
            child = parent;
            parent = GetParent(child);
        }
    }
}
```

8.3 Fibonacci heaps

Algorithm 8.16: Node definition.

```
class MyFibNode {
public:
    MyFibNode(MyType k)
        :key(k), mark(false), p(nullptr), left(nullptr),
        right(nullptr), child(nullptr), degree(-1) {}
    ~MyFibNode() {}
    MyType key;
    bool mark;
    MyFibNode *p,*left,*right,*child;
    int degree;
};
```

Algorithm 8.17: Class interface.

```
class MyFibonacciHeap {
public:
    MyFibonacciHeap();
    ~MyFibonacciHeap();
    MyFibNode* Minimum();
    int GetSize() const;
    void Insert(MyFibNode *x);
    void Insert(MyType k);
    void DecreaseKey(MyFibNode* x, int k);
    MyFibonacciHeap* Union(MyFibonacciHeap *H1, MyFibonacciHeap *H2);
    void Link(MyFibNode* y, MyFibNode* x);
    MyFibNode* ExtractMin();
    void Consolidate();
    void Cut(MyFibNode* x, MyFibNode* y);
    void CascadingCut(MyFibNode* y);
    void Delete(MyFibNode* x);
    bool Empty() const;
    void DisplayChild(MyFibNode* p);
    void Display();
private:
    int n;
    MyFibNode *min;
};
```

Algorithm 8.18: Constructor and destructor.

```
MyFibonacciHeap::MyFibonacciHeap() : n(0), min(nullptr) {}

void DeleteFibnodes(MyFibNode *x) {
    if (x) {
        MyFibNode *current = x;
        bool finished = false;
        while (!finished) {
            if (current->left && current->left != x) {
                MyFibNode *tmp = current;
                current = current->left;
                if (tmp->child)
                    DeleteFibnodes(tmp->child);
                delete tmp;
            }
            else {
                if (current->child)
                    DeleteFibnodes(current->child);
                delete current;
                finished=true;
            }
        }
    }
}

MyFibonacciHeap::~MyFibonacciHeap() {
    DeleteFibnodes(min);
}
```

Algorithm 8.19: Accessors.

```
// The minimum node of the heap.
MyFibNode* MyFibonacciHeap::Minimum() {
    return min;
}

int MyFibonacciHeap::GetSize() const {
    return n;
}

bool MyFibonacciHeap::Empty() const {
    return n == 0;
}
```

Algorithm 8.20: Insert.

```
void MyFibonacciHeap::Insert(MyFibNode *x) {
    x->degree = 0;
    x->p = nullptr;
    x->child = nullptr;
    x->mark = false;
    if (min == nullptr)
        min = x->left = x->right = x;
    else {
        min->left->right = x;
        x->left = min->left;
        min->left = x;
        x->right = min;
        if (x->key < min->key) {
            min = x;
        }
    }
    n++;
}

void MyFibonacciHeap::Insert(MyType k) {
    MyFibNode *x = new MyFibNode(k);
    Insert(x);
}
```

Algorithm 8.21: Delete.

```
void MyFibonacciHeap::Delete(MyFibNode* x) {
    DecreaseKey(x,0); // set to a minimum value
    MyFibNode *m = ExtractMin();
    delete m;
}
```

Algorithm 8.22: Union.

```
MyFibonacciHeap* MyFibonacciHeap::Union(MyFibonacciHeap *H1, MyFibonacciHeap *H2) {
    MyFibonacciHeap* H = new MyFibonacciHeap();
    H->min = H1->min;
    if (H->min != nullptr && H2->min != nullptr) {
        H->min->right->left = H2->min->left;
        H2->min->left->right = H->min->right;
        H->min->right = H2->min;
        H2->min->left = H->min;
    }
    if (H1->min == nullptr ||
          (H2->min != nullptr && H2->min->key < H1->min->key))
        H->min = H2->min;
    H->n = H1->n + H2->n;
    return H;
}
```

Algorithm 8.23: Extract Min.

```
MyFibNode* MyFibonacciHeap::ExtractMin() {
    MyFibNode *z, *x, *next;
    MyFibNode ** childList;
    z = min;
    if (z != nullptr) {
        x = z->child;
        if (x != nullptr) {
            childList = new MyFibNode*[z->degree];
            next = x;
            for (int i = 0; i < (int)z->degree; i++) {
                childList[i] = next;
                next = next->right;
            }
            for (int i = 0; i < (int)z->degree; i++) {
                x = childList[i];
                min->left->right = x;
                x->left = min->left;
                min->left = x;
                x->right = min;
                x->p = nullptr;
            }
            delete[] childList;
        }
        z->left->right = z->right;
        z->right->left = z->left;
        if (z == z->right)
            min = nullptr;
        else {
            min = z->right;
            Consolidate();
        }
        n--;
    }
    return z;
}
```

Algorithm 8.24: Consolidate.

```
void MyFibonacciHeap::Consolidate() {
    MyFibNode* w, *next, *x, *y;
    MyFibNode** A, ** rootList;
    // Max degree <= log base golden ratio of n
    int d, rootSize;
    double goldenratio = (1.0 + sqrt(5.0))/2.0;
    int max_degree = static_cast<int>(floor(log(static_cast<double>(n)) /
                                            log(goldenratio)));

    A = new MyFibNode*[max_degree + 2];
    // + 2 : max degree and A[max_degree+1] == NULL
    for (int i = 0; i < max_degree + 2; i++)
        A[i] = nullptr;
    w = min;
    rootSize = 0;
    next = w;
    do {
        rootSize++;
        next = next->right;
    } while (next != w);
    rootList = new MyFibNode*[rootSize];
    for (int i = 0; i < rootSize; i++) {
        rootList[i] = next;
        next = next->right;
    }
    for (int i = 0; i < rootSize; i++) {
        w = rootList[i];
        x = w;
        d = x->degree;
        while (A[d] != nullptr) {
            y = A[d];
            if (x->key > y->key)
                swap(x, y);
            Link(y, x);
            A[d] = nullptr;
            d++;
        }
        A[d] = x;
    }
    delete[] rootList;
    min = nullptr;
    for (int i = 0; i < max_degree + 2; i++) {
        if (A[i] != nullptr) {
            if (min == nullptr) {
                min = A[i]->left = A[i]->right = A[i];
            }
            else {
                min->left->right = A[i];
                A[i]->left = min->left;
                min->left = A[i];
                A[i]->right = min;
                if (A[i]->key < min->key) {
                    min = A[i];
```

```
                        }
                }
            }
        }
        delete[] A;
}
```

Algorithm 8.25: Link.

```
void MyFibonacciHeap::Link(MyFibNode* y, MyFibNode* x) {
    // Remove y from the root list of heap
    y->left->right = y->right;
    y->right->left = y->left;
    // Make y a child of x, incrementing x.degree
    if (x->child != nullptr) {
        x->child->left->right = y;
        y->left = x->child->left;
        x->child->left = y;
        y->right = x->child;
    }
    else {
        x->child = y;
        y->right = y;
        y->left = y;
    }
    y->p = x;
    x->degree++;
    y->mark = false;
}
```

Algorithm 8.26: Decrease Key.

```
void MyFibonacciHeap::DecreaseKey(MyFibNode* x, int k) {
    MyFibNode* y;
    if (k > x->key) {
        cout << "The new key is greater than current key" << endl;
    }
    else {
        x->key = k;
        y = x->p;
        if (y != nullptr && x->key < y->key) {
            Cut(x, y);
            CascadingCut(y);
        }
        if (x->key < min->key) {
            min = x;
        }
    }
}
```

Algorithm 8.27: Cut.

```
void MyFibonacciHeap::Cut(MyFibNode* x, MyFibNode* y) {
    if (x->right == x) {
        y->child = nullptr;
    }
    else {
        x->right->left = x->left;
        x->left->right = x->right;
        if (y->child == x) {
            y->child = x->right;
        }
    }
    y->degree--;
    min->right->left = x;
    x->right = min->right;
    min->right = x;
    x->left = min;
    x->p = nullptr;
    x->mark = false;
}
```

Algorithm 8.28: CascadingCut.

```
void MyFibonacciHeap::CascadingCut(MyFibNode* y) {
    MyFibNode* z;
    z = y->p;
    if (z != nullptr) {
        if (y->mark == false)
            y->mark = true;
        else {
            Cut(y, z);
            CascadingCut(z);
        }
    }
}
```

Algorithm 8.29: Display.

```
void MyFibonacciHeap::DisplayChild(MyFibNode* p) {
    MyFibNode* copy = p;
    do {
        cout << p->key;
        p = p->right;
        if (p != copy) {
            cout << "->";
        }
    } while (p != copy);
    cout << endl;
    p = copy;
    do {
        if (p->child != nullptr) {
            cout << "Display child of :" << p->key << endl;
            DisplayChild(p->child);
        }
        p = p->right;
    } while (p != copy);
    cout << endl;
}

void MyFibonacciHeap::Display() {
    cout << "Minimum value : " << min->key << endl;
    MyFibNode* p = min;
    if (p == nullptr)
        cout << "The Heap is Empty" << endl;
    else {
        cout << "Root nodes: " << endl;
        do {
            cout << p->key;
            p = p->right;
            if (p != min)
                cout << "->";
        } while (p != min);
        cout << endl;
        p = min;
        do {
            if (p->child != nullptr) {
                cout << "Display child of :" << p->key << endl;
                DisplayChild(p->child);
            }
            p = p->right;
        } while (p != min);
    }
}
```

8.3.1 Proof by induction

What has to be done:

- To prove the default case (basic case). The first case is typically obvious. Yet, you need to tell which rule, or which definition has been used, if it is not a simple arithmetic operation.
- To prove the case for the next step (n+1) by using **only** the set of rules given by original definition, and the hypothesis.

Remark:
The way you organize the sequence on your draft is up to you, you can derive the expression from both sides. It is more a pattern matching game where you have to decompose the terms in order to retrieve the hypothesis and use it. When you have an equality (Prove: A=B), you may go with the development of A to get B, or to get A from B. Therefore, on your working sheet you may work on both sides to search where it is easier for you to extract the hypothesis. Keep the expected target in mind, so you don't derive one side in such a way that you go too far from the target. Some additional rules (from the definitions) may be needed to reach the target, hence it must be kept in mind in order to consider the right rules. You may go with A-B to arrive to 0.

There are 2 parts:

- The part on your working sheet, to find the sequence that leads to the proof. You may attack the problem in the side of the equality that is the most complex (with a $\sum$ sign) to extract the hypothesis. In Lemma 2, we use directly the definition and the inductive step is hidden in the sum. In Lemma 3, the inductive step is direct, but the definition of Φ must be used.
- The clean part you should write on the final or midterm, that contains the proper sequence that leads to the proof. In the sequence, each line may be justified by the application of a rule (for example, commutativity of the addition, decomposition of the sum,...). Depending on the problem, some rules are totally implied. Once the induction hypothesis is used, then you can present the sequence of the development of the proof in the proper order, going from A to B, or from B to A.

8.3.2 Lemma 2

With the sequence of Fibonacci, we have:

$$F_0 = 0 \tag{8.1}$$
$$F_1 = 1 \tag{8.2}$$
$$F_k = F_{k-1} + F_{k-2} \forall k \geq 2 \tag{8.3}$$

We want to prove that:

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \forall k \geq 0 \tag{8.4}$$

Step 1: We verify it is true for k=0.

$$F_{0+2} = F_2 = F_1 + F_0 = 1 + 0 = 1 \tag{8.5}$$
$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 + 1 \tag{8.6}$$

Step 2: We want to show that the statement holds for (k+1) when we use the hypothesis for k. (You can start from any side of the equality, you can also keep the target you want to reach on the side, so you know where you need to go).

In the first line, if we develop F_{k+1+2}, we just use the definition. In the second line, we use the inductive step.

$$\begin{aligned} F_{(k+1)+2} &= F_{k+2} + F_{k+1} && (8.7) \\ &= 1 + \sum_{i=0}^{k} F_i + F_{k+1} && (8.8) \\ &= 1 + \sum_{i=0}^{k+1} F_i && (8.9) \end{aligned}$$

We can start the other way:

$$\begin{aligned} 1 + \sum_{i=0}^{k+1} F_i &= 1 + \sum_{i=0}^{k} F_i + F_{k+1} && (8.10) \\ &= F_{k+2} + F_{k+1} && (8.11) \\ &= F_{(k+1)+2} && (8.12) \end{aligned}$$

In the last line, we change the variables to match the definition.

8.3.3 Lemma 3

We want to prove that $\forall k \geq 0$, the (k+2)nd Fibonacci number satisfies $F_{k+2} \geq \Phi^k$. The most difficult part in to proof of this lemma is to think about going back to the definition of the golden ratio, which is a solution to:

$$x^2 - x - 1 = 0 \qquad (8.13)$$

So we have: $\Phi^2 = \Phi + 1$.

Step 1: We verify it is true for k=0. If we have $k = 0$, $F_2 = 1 = \Phi^0$. If we have $k = 1$, $F_3 = 2 > 1.62 > \Phi^1$.

Step 2: We want to show that the statement holds for (k+1) when we use the hypothesis for k. In the first line, we will just consider the definition. In the second line, we use the inductive step. In fourth line, we use the expression of $\Phi^2 = \Phi + 1$.

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k && (8.14) \\ &\geq \Phi^{k-1} + \Phi^{k-2} && (8.15) \\ &= \Phi^{k-2} * (\Phi + 1) && (8.16) \\ &= \Phi^{k-2} * (\Phi^2) && (8.17) \\ &= \Phi^k && (8.18) \end{aligned}$$

Chapter 9

Graphs

Contents

9.1 Definitions

A graph $G = (V, E)$ is a couple of two sets defined by V corresponding to the set of vertices, and E corresponding to the set f edges.

Main types of graphs:

- **Undirected**: edge $(u, v) = (v, u)$ $\forall v, (v, v) \notin E$ (there are no self loops.)
- **Directed**: (u, v) is a edge from u to v, denoted as u $\rightarrow$ v. Self loops are allowed.
- **Weighted**: Each edge has an associated weight, given by a weight function w : E $\in$ R.
- **Mixed**: some edges may be directed and some may be undirected
- **Multigraph**: multiple edges are two or more edges that connect the same two vertices.
- We say a graph is **dense** if $|E| \approx |V|^2$.
- We say a graph is **sparse**: $|E| << |V|^2$.

where $|V|$ represents the number of vertices and $|E|$ represents the number of edges. If $(u, v) \in E$, then vertex v is adjacent to vertex u. It is worth noting that the adjacency relationship is symmetric if G is undirected. However, it is not necessarily if G is directed. If G is connected then there is a path between every pair of vertices. $|E| \geq |V| - 1$. Furthermore, if $|E| = |V| - 1$, then G is a tree. The **degree** of a vertex v is the number of edges attached to the vertex v. **Simple** graph is undirected; both multiple edges and loops are disallowed. Each edge is an unordered pair of distinct vertices. Hence the degree of every vertex is at most $n - 1$ with n vertices.

Types of graphs:

- **Connected** graphs: Every unordered pair of vertices in the graph is connected, there is a path from any point to any other point in the graph.
- **Bipartite** graphs: Vertices can be divided into 2 disjoint and independent sets U and V. Every edge connects a vertex in U to one in V
- **Planar** graph: Vertices and edges can be drawn in a plane such that no two of the edges intersect
- **Cycle** graphs: Connected graphs in which the degree of all vertices is 2.
- **Tree**: A connected graph with no cycles.
- **Regular** graphs: Each vertex has the same number of neighbors every vertex has the same degree.
- **Complete** graphs: A simple undirected graph, where every pair of distinct vertices is connected by a unique edge.
- **Finite** graphs: The vertex set and the edge set are finite sets.
- **Eulerian** graphs: If the graph is both connected and has a closed trail containing all edges of the graph. It corresponds to a walk with no repeated edges.

More definitions:

- **Path**: It is a sequence of vertices $v_1, \ldots, v_k$ where each (v_i, v_{i+1}) is an edge.

- **Simple path**: A path that does not repeat vertices.
- **Path Length**: Number of edges in the path.
- **Circuit**: It is a path that begins and ends at the same vertex.
- **Cycle**: It is a circuit that doesn't repeat vertices.
- **Euler path**: It is a path that travels through all edges of a connected graph.
- **Euler circuit**: It is a circuit that visits all edges of a connected graph.

Hamiltonian graphs

- Hamiltonian **graph**: it is a graph possessing a Hamiltonian cycle. A graph that is not Hamiltonian is said to be non-hamiltonian.
- Hamiltonian **cycle**: Hamiltonian path that is a cycle.
- Hamiltonian **path**: A path in an undirected or directed graph that visits each vertex exactly once

Euler's theorem

1. **Circuit**: If a graph has any vertex of odd degree, then it cannot have an Euler circuit. If a graph is connected and every vertex is of even degree, it has at least has 1 Euler circuit.
2. **Path**: If a graph has more than 2 vertices of odd degree, then it cannot have an Euler path. If a graph is connected and has just 2 vertices of odd degree, then it has at least has one Euler path. Any such path must start at one of the odd-vertices and end at the other odd vertex.

Fleury's algorithm

With the Fleury's algorithm, we need to check first that the graph is connected and all vertices are of even degree. It starts at any vertex, travels through an edge, if it is not a bridge for the untraveled part or there is no other alternative (a bridge is an edge that if removed, it produces a disconnected graph). It then labels the edges in the order in which they have been visited. When it is not possible to travel further then it stops.

Main idea: expand a trail C_i while avoiding bridges in $G - C_i$ until no other choices are possible.

Hierholzer's algorithm

The Hierholzer's algorithm produces circuits in a graph G that are pairwise edge disjoint. When these circuits are combined properly, they form an Eulerian circuit of G. This patching together of circuits hinges on the circuits having a common vertex following from the connectivity of the graph. Once a circuit is formed, if all edges have not been used, then there must be an edge that is incident to a vertex of the circuit, and we use this edge to begin the next circuit. These circuits then share a common vertex.

Main idea: Combining circuits.

Algorithm 1: Fleury's algorithm.

Input: A connected graph $G = (V, E)$
Output: An eulerian circuit C of G
Pick any $v_0 \in V$
Let $C_0 = v_0$
Let $i \leftarrow 0$
while $(i! = |E|)$ **do**
  Assume the trail $C_i = v_0, e_1, v_1, \ldots, e_i, v_i$ has already been chosen.
  At v_i pick any edge e_{i+1} that is not on C_i and
  that is not a bridge of the graph $G_i = G - E(C_i)$ unless there is no other choice.
  Define $C_{i+1} = C_i, e_{i+1}, v_{i+1}$.
  $i \leftarrow i + 1$
end while

Algorithm 2: Hierholzer's algorithm.

Input: A connected graph $G = (V, E)$
Output: An eulerian circuit C of G
Pick any $v \in V$
Create a circuit C_0 starting with v by traversing at each step,
any edge not yet included in the circuit.
Let $i \leftarrow 0$
while $(E(C_i)! = E(G))$ **do**
  Pick a vertex v_i on C_i that is incident to an edge not on C_i
  Create a circuit C_i^* starting with v_i in the graph $G - E(C_i)$
  Create a circuit C_{i+1} that contains C_i and C_i^* by starting at v_{i-1}
  traversing C_i until reaching v_i, then traversing completely C_i^*
  and then completing the traversal of C_i
  $i \leftarrow i + 1$
end while

9.2 Adjacency lists

Algorithm 9.1: Class interface.

```
class NodeGAL {
public:
    NodeGAL() :
        v(0), weight(0), next(nullptr) {}
    NodeGAL(int v1,double weight1) :
        v(v1), weight(weight1), next(nullptr) {}
    ~NodeGAL() {}
    int v; // vertex index
    double weight; // weight of the edge to reach v
    NodeGAL *next;
};

class MyGraphAL {
public:
    MyGraphAL();
    MyGraphAL(int n1);
    ~MyGraphAL();
    int GetNumberVertices() { return n; }
    int GetDegree(int u) { return degree[u]; }
    bool ExistEdge(int u, int v);
    double GetEdgeWeight(int u, int v);
    void SetDirectedEdge(int u, int v, double w);
    void SetDirectedEdge(int u, int v);
    void SetUndirectedEdge(int u, int v, double w);
    void SetUndirectedEdge(int u, int v);
    void RemoveDirectedEdge(int u, int v);
    void RemoveUndirectedEdge(int u, int v);
    bool HasSelfLoops();
    bool IsUndirected();

    bool ExistAdjacent(int u);
    void SetCurrentVertex(int u);
    int GetNextAdjacent(int u);
    void Display();
    void BFS(int s);
    void DFS();
private:
    int n; // number of vertices
    NodeGAL** l; // array of lists
    NodeGAL** current;  // iterators
    int* degree; // degree for each vertex
};
```

Algorithm 9.2: Constructor and Destructor.

```
MyGraphAL::MyGraphAL() {
    int n = 0;
    l = nullptr;
    current=nullptr;
    degree=nullptr;
}

MyGraphAL::MyGraphAL(int n1) {
    n = n1;
    l = new NodeGAL*[n];
    degree = new int[n];
    for (int u = 0; u < n; u++) {
        l[u] = nullptr;
        current[u]=nullptr;
        degree[u] = 0;
    }
}

MyGraphAL::~MyGraphAL() {
    for (int i = 0; i < n; i++) {
        NodeGAL *current = l[i];
        while (current) {
            NodeGAL* next = current->next;
            delete current;
            current = next;
        }
    }
    delete[] l;
    delete[] current;
    delete[] degree;
}
```

Algorithm 9.3: Exist and get edges.

```
bool MyGraphAL::ExistEdge(int u,int v) {
    if (u < n) {
        NodeGAL* cursor = l[u];
        while (cursor != nullptr) {
            if (cursor->v == v)
                return true;
            else
                cursor = cursor->next;
        }
        return false;
    }
    else
        return false;
}

double MyGraphAL::GetEdgeWeight(int u,int v) {
    if (u < n) {
        NodeGAL* cursor = l[u];
        while (cursor != nullptr) {
            if (cursor->v == v)
                return cursor->weight;
            else
                cursor = cursor->next;
        }
        return 0;
    }
    else
        return 0;
}
```

Algorithm 9.4: Set edges.

```
void MyGraphAL::SetDirectedEdge(int u, int v, double w) {
    if (!ExistEdge(u, v)) {
        NodeGAL* tmp = new NodeGAL(v, w);
        tmp->next = l[u];
        l[u] = tmp;
        degree[u]++;
    }
}

void MyGraphAL::SetDirectedEdge(int u, int v) {
    SetDirectedEdge(u, v, 1);
}

void MyGraphAL::SetUndirectedEdge(int u, int v) {
    SetDirectedEdge(u, v, 1);
    SetDirectedEdge(v, u, 1);
}

void MyGraphAL::SetUndirectedEdge(int u, int v, double w) {
    SetDirectedEdge(u, v, w);
    SetDirectedEdge(v, u, w);
}
```

Algorithm 9.5: Remove edges.

```
void MyGraphAL::RemoveDirectedEdge(int u, int v) {
    NodeGAL* cursor = l[u];
    if (cursor!=nullptr) {
        if (cursor->v == v) {
            l[u] = cursor->next;
            delete cursor;
        }
        else {
            if (cursor->next != nullptr) {
                NodeGAL* prev = cursor;
                cursor = cursor->next;
                bool found = false;
                while ((cursor != nullptr) && (!found) ){
                    if (cursor->v == v) {
                        prev->next = cursor->next;
                        delete cursor;
                        found = true;
                    }
                    else {
                        prev = cursor;
                        cursor = cursor->next;
                    }
                }
            }
        }
    }
    degree[u]--;
}

void MyGraphAL::RemoveUndirectedEdge(int u, int v) {
    RemoveDirectedEdge(u,v);
    RemoveDirectedEdge(v,u);
}
```

Algorithm 9.6: Comparisons.

```
bool MyGraphAL::HasSelfLoops() {
    int u = 0;
    while (u < n) {
        if (GetEdgeWeight(u,u)!= 0)
            return true;
        u++;
    }
    return false;
}

bool MyGraphAL::IsUndirected() {
    int v, u = 0;
    while (u < n) {
        v = u;
        while (v < n) {
            if (GetEdgeWeight(u,v)!= GetEdgeWeight(v,u))
                return false;
            v++;
        }
        u++;
    }
    return !HasSelfLoops();
}
```

Algorithm 9.7: Display.

```
void MyGraphAL::Display() {
    for (int u = 0; u < n; u++) {
        cout << u << ":";
        NodeGAL* cursor = l[u];
        while (cursor != nullptr) {
            cout << cursor->v << "(" << cursor->weight << ") ";
            cursor = cursor->next;
        }
        cout << endl;

    }
}
```

Algorithm 9.8: Iterator functions.

```
bool MyGraphAL::ExistAdjacent(int u) {
    return current[u] != nullptr;
}

void MyGraphAL::SetCurrentVertex(int u) {
    current[u] = l[u];
}

int MyGraphAL::GetNextAdjacent(int u) {
    int v = current[u]->v;
    current[u] = current[u]->next;
    return v;
}
```

9.2.1 Breadth First Search

Algorithm 9.9: Color of Nodes.

```
enum class ColorNode {
    Visited,
    Unvisited,
    Finished
};
```

Algorithm 9.10: BFS main function.

```
void BFS1(int s,MyGraphAL* G, ColorNode* &color, int* &distance,int* &pi) {
    int n = G->GetNumberVertices();
    if (!((s >= 0) && (s < n)))
        cout << "Bad source !" << endl;
    else {
        MyQueue* Q = new MyQueue(n);
        color = new ColorNode[n];
        distance = new int[n];
        pi = new int[n];
        for (int u = 0; u < n; u++) {
            color[u] = ColorNode::Unvisited;
            distance[u] = 0;
            pi[u] = 0;
        }
        color[s] = ColorNode::Visited;
        distance[s] = 0;
        pi[s] = -1;
        Q->Enqueue(s);
        while (!Q->IsEmpty()) {
            int u = Q->Dequeue();
            for (int v = 0; v < n; v++) {
                if (G->ExistEdge(u,v)) { // Adjancent to u
                    if (color[v] == ColorNode::Unvisited) {
                        color[v] = ColorNode::Visited;
                        distance[v] = distance[u] + 1;
                        pi[v] = u;
                        Q->Enqueue(v);
                    }
                }
            }
            color[u] = ColorNode::Finished;
        }
        delete Q;
    }
}
```

Algorithm 9.11: Breadth First Search Display.

```
void MyGraphAL::BFS(int s) {
    cout << "BFS" << endl;
    ColorNode *color;
    int *distance, *pi;
    BFS1(s,this,color,distance,pi);
    for (int u = 0; u < n; u++)
        cout << "from " << s << " to " << u
             << " Distance=" << distance[u]
             << " Pi=" << pi[u] << endl;
    delete[] color;
    delete[] distance;
    delete[] pi;
}
```

9.2.2 Depth First Search

Algorithm 9.12: DFS functions.

```
void DFSVisit(int u,MyGraphAL* G, ColorNode* color,
              int* discovery,int* finished, int* pi, int t) {
    color[u] = ColorNode::Visited;
    t++;
    discovery[u] = t;
    G->SetCurrentVertex(u);
    while (G->ExistAdjacent(u)) {
        int v = G->GetNextAdjacent(u);
        if (color[v] == ColorNode::Unvisited) { // not already visited
            pi[v] = u;
            DFSVisit(v, G, color, discovery, finished, pi, t);
        }
    }
    color[u] = ColorNode::Finished;
    t++;
    finished[u] = t;
}

void DFS1(MyGraphAL* G, ColorNode* &color,
              int* &discovery, int* &finished, int* &pi) {
    int n = G->GetNumberVertices();
    color = new ColorNode[n];
    discovery = new int[n];
    finished = new int[n];
    pi = new int[n];
    for (int u = 0; u < n; u++) {
        color[u] = ColorNode::Unvisited;
        discovery[u] = 0;
        finished[u] = 0;
        pi[u] = 0;
    }
    int t = 0;
    for (int u = 0; u < n; u++) {
        if (color[u] == ColorNode::Unvisited)
            DFSVisit(u, G, color, discovery,finished, pi,t);
    }
}
```

Algorithm 9.13: Depth First Search Display.

```
void MyGraphAL::DFS() {
    cout << "DFS" << endl;
    ColorNode *color;
    int *discovery,*finished,*pi;
    DFS1(this, color, discovery, finished, pi);
    for (int u = 0; u < n; u++)
        cout << u << ": (" << discovery[u] << ","
                           << finished[u] << ")  Pi="
                           << pi[u]<< endl;
    delete[] color;
    delete[] discovery;
    delete[] finished;
    delete[] pi;
}
```

Algorithm 9.14: Example.

```
void main() {
    int n = 5;
    MyGraphAL* G = new MyGraphAL(n);
    G->SetUndirectedEdge(0, 1);
    G->SetUndirectedEdge(1, 2);
    G->SetUndirectedEdge(2, 3);
    G->SetUndirectedEdge(3, 0);
    G->SetUndirectedEdge(0, 4);
    G->SetUndirectedEdge(1, 4);
    G->Display();
    delete G;
}
```

9.3 Shortest paths

This section contains the algorithm of Bellman Ford and Dijkstra. For Dijkstra, we consider a Queue in which we add at each iteration the completed vertex, and a Priority Queue (see Heap section) that provides both the index of the vertex containing the minimum value and its minimum value. In the relaxation part, note that it is possible to decrease the key or it is possible to not update the value of the element in the priority queue and just add a new element in it. In the latter case, the Priority Queue should be able to get all these new elements.

Algorithm 9.15: Display path information.

```
void DisplayShortestPath(double* d, int* pi, int n) {
    for (int i = 0; i < n; i++) {
        cout << "Vertex:" << i
            << " : d=" << d[i]
            << " : pi:" << pi[i]
            << endl;
    }
    cout << endl;
}
```

9.3.1 Bellman Ford algorithm

Algorithm 9.16: Bellman Ford.

```
bool MyGraphAL::BellmanFord(int s,double* &d, int* &pi) {
    // Initialize Single Source
    d = new double[n];
    pi = new int[n];
    for (int u = 0; u < n ; u++) {
        d[u] = DBL_MAX;
        pi[u] = -1;
    }
    d[s] = 0;
    for (int i = 0; i < n - 1; i++) {
        for (int u = 0; u < n; u++) { // For each edge ...
            SetCurrentVertex(u);
            while (ExistAdjacent(u)) {
                int v = GetNextAdjacent(u);
                // Relaxation
                double w = GetEdgeWeight(u, v);
                if (d[v] > d[u] + w) {
                    d[v] = d[u] + w;
                    pi[v] = u;
                }
            }
        }
        DisplayShortestPath(d, pi, n);
    }
    for (int u = 0; u < n; u++) { // For each edge
        SetCurrentVertex(u);
        while (ExistAdjacent(u)) {
            int v = GetNextAdjacent(u);
            double w = GetEdgeWeight(u, v);
            if (d[v] > d[u] + w)
                return false;
        }
    }
    return true;
}
```

Algorithm 9.17: Bellman Ford example.

```
void main() {
    int n = 5;
    MyGraphAL G(n);
    G.SetDirectedEdge(0, 1, 6);
    G.SetDirectedEdge(0, 4, 7);
    G.SetDirectedEdge(1, 2, 5);
    G.SetDirectedEdge(1, 3, -4);
    G.SetDirectedEdge(1, 4, 8);
    G.SetDirectedEdge(2, 1, -2);
    G.SetDirectedEdge(3, 2, 7);
    G.SetDirectedEdge(3, 0, 2);
    G.SetDirectedEdge(4, 2, -3);
    G.SetDirectedEdge(4, 3, 9);
    G.Display();
    cout << "BellmanFord" << endl;
    int s = 0;
    double* d;
    int* pi;
    bool out=G.BellmanFord(s,d,pi);
    DisplayShortestPath(d, pi, n);
    delete d;
    delete pi;
}
```

9.3.2 Dijkstra algorithm

Algorithm 9.18: Dijkstra.

```
void MyGraphAL::Dijkstra(int s,double* &d, int* &pi) {
    // Initialize Single Source
    d = new double[n];
    pi = new int[n];
    MyPriorityQueue* PQ = new MyPriorityQueue(n);
    MyQueue* Q = new MyQueue(n);
    for (int u = 0; u < n; u++) {
        d[u] = DBL_MAX;
        pi[u] = -1;
    }
    d[s] = 0;
    for (int u = 0; u < n ; u++)
        PQ->Push(u, d[u]);
    PQ->Display();
    while (!Q->IsFull()) {
        MyData out = PQ->Pop();
        PQ->Display();
        Q->Enqueue(out.index);
        int u = out.index;
        cout << "Use: " << out.index << " with d:" << out.value << endl;
        SetCurrentVertex(u);
        while (ExistAdjacent(u)) {
            int v = GetNextAdjacent(u);
            // Relaxation
            double w = GetEdgeWeight(u, v);
            if (d[v] > d[u] + w) {
                d[v] = d[u] + w;
                pi[v] = u;
                PQ->DecreaseKey(v, d[v]);
            }
        }
        DisplayShortestPath(d, pi, n);
        PQ->Display();
    }
    delete PQ;
    delete Q;
}
```

Algorithm 9.19: Dijkstra example.

```
void main() {
    int n = 5;
    MyGraphAL G(n);
    G.SetDirectedEdge(0, 1, 10);
    G.SetDirectedEdge(0, 4, 5);
    G.SetDirectedEdge(1, 2, 1);
    G.SetDirectedEdge(1, 4, 2);
    G.SetDirectedEdge(2, 3, 4);
    G.SetDirectedEdge(3, 0, 7);
    G.SetDirectedEdge(3, 2, 6);
    G.SetDirectedEdge(4, 1, 3);
    G.SetDirectedEdge(4, 2, 9);
    G.SetDirectedEdge(4, 3, 2);
    G.Display();
    cout << "Dijkstra" << endl;
    int s = 0;
    double* d;
    int* pi;
    G.Dijkstra(s, d, pi);
    DisplayShortestPath(d, pi, n);
    delete d;
    delete pi;
}
```

9.4 Adjacency matrix

Algorithm 9.20: Class interface.

```
class MyGraphAM {
public:
    MyGraphAM();
    MyGraphAM(int n);
    ~MyGraphAM();
    int GetNumberVertices() { return n; }
    int GetDegree(int u);
    int GetIndex(int u,int v);
    bool ExistEdge(int u, int v);
    void SetDirectedEdge(int u, int v, double w);
    void SetDirectedEdge(int u, int v);
    void SetUndirectedEdge(int u, int v, double w);
    void SetUndirectedEdge(int u, int v);
    void RemoveDirctedEdge(int u, int v);
    void RemoveUndirectedEdge(int u, int v);
    bool HasSelfLoops();
    bool IsUndirected();
    void Display();
    void DisplayDirectedEdge();
    void DisplayUndirectedEdge();
    void SetCurrentVertex(int u);
    bool GetNextAdjacent(int u, int &vout);
    void BFS(int s);
    void DFS();
    void PrintAllPairs(MyMatrix* pi, int i, int j);
    void FloydWarshall(MyMatrix* &dout, MyMatrix* &piout);
    void TransitiveClosure(MyMatrix* &TCout);
    void MSTKruskal();
    void MSTPrim();
    int minKey(int* key, bool* mstSet);
    void DisplayMST(int* parent);
private:
    int n; // number of vertices
    double* M;
    int* current;
};
```

Algorithm 9.21: Constructors and Destructor.

```
MyGraphAM::MyGraphAM() {
    n = 0;
    M=nullptr;
    current = nullptr;
}

MyGraphAM::MyGraphAM(int n1) {
    n = n1;
    M = new double[n*n];
    for (int i = 0; i < n*n; i++)
        M[i] = 0;
    current = new int[n];
    for (int i = 0; i < n; i++)
        current[i] = 0;
}

MyGraphAM::~MyGraphAM() {
    delete[] M;
    delete[] current;
}
```

Algorithm 9.22: Get, Exist, and Set edges.

```
int MyGraphAM::GetIndex(int u,int v) {
    // u: source, v: destination
    return u*n + v;
}

bool MyGraphAM::ExistEdge(int u, int v) {
    return (M[GetIndex(u, v)] != 0);
}

int MyGraphAM::GetDegree(int u) {
    int degree = 0;
    for (int v = 0; v < n; v++)
        if (ExistEdge(u, v))
            degree++;
    return degree;
}

void MyGraphAM::SetDirectedEdge(int u, int v, double w) {
    M[GetIndex(u, v)] = w;
}

void MyGraphAM::SetDirectedEdge(int u, int v) {
    M[GetIndex(u, v)] = 1;
}

void MyGraphAM::SetUndirectedEdge(int u, int v, double w) {
    M[GetIndex(u, v)] = w;
    M[GetIndex(v, u)] = w;
}

void MyGraphAM::SetUndirectedEdge(int u, int v) {
    M[GetIndex(u, v)] = 1;
    M[GetIndex(v, u)] = 1;
}

void MyGraphAM::RemoveDirctedEdge(int u, int v) {
    M[GetIndex(u, v)] = 0;
}

void MyGraphAM::RemoveUndirectedEdge(int u, int v) {
    M[GetIndex(u, v)] = 0;
    M[GetIndex(v, u)] = 0;
}
```

9.4.1 Comparisons

Algorithm 9.23: Comparisons.

```
bool MyGraphAM::HasSelfLoops() {
    int u = 0;
    while (u < n) {
        if (M[GetIndex(u, u)] != 0)
            return true;
        u++;
    }
    return false;
}

bool MyGraphAM::IsUndirected() {
    int v,u = 0;
    while (u < n) {
        v = u;
        while (v < n) {
            if (M[GetIndex(u, v)] != M[GetIndex(v, u)])
                return false;
            v++;
        }
        u++;
    }
    return !HasSelfLoops();
}
```

9.4.2 Display

Algorithm 9.24: Display.

```
void MyGraphAM::Display() {
    int k = 0;
    cout << " :";
    for (int v = 0; v < n; v++)
        cout << v << "\t";
    cout << endl;
    for (int u = 0; u < n; u++) {
        cout << u << ":";
        for (int v = 0; v < n; v++) {
            cout << M[k] << "\t";
            k++;
        }
        cout << endl;
    }
}

void MyGraphAM::DisplayDirectedEdge() {
    int k = 0;
    cout << "List of edges :" << endl;
    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            if (ExistEdge(u, v))
                cout << "(" << u << ","
                << v << ") w:" << GetEdgeWeight(u, v)
                << endl;
        }
    }
}

void MyGraphAM::DisplayUndirectedEdge() {
    int k = 0;
    cout << "List of edges :" << endl;
    for (int u = 0; u < n; u++) {
        for (int v = u+1; v < n; v++) {
            if (ExistEdge(u, v))
                cout << "(" << u << ","
                << v << ") w:" << GetEdgeWeight(u, v)
                << endl;
        }
    }
}
```

9.4.3 Breadth First Search and Depth First Search

The functions are similar to the the graph using the adjacency lists implementation. However, for DFS, there are slight changes in the way we access the different adjacent vertices to a given vertex.

Algorithm 9.25: Adjacent iterator.

```
void MyGraphAM::SetCurrentVertex(int u) {
    current[u] = -1;
}

bool MyGraphAM::GetNextAdjacent(int u,int &vout) {
    int v= current[u] + 1;
    vout = -1;
    bool found = false;
    while ((!found) && (v < n)) {
        if (ExistEdge(u, v)) {
            found = true;
            vout=v;
        }
        v++;
    }
    current[u] = vout;
    return found;
}

void DFSVisit(int u, MyGraphAM* G, ColorNode* color,
            int* discovery, int* finished, int* pi, int t) {
    color[u] = ColorNode::Visited;
    t++;
    discovery[u] = t;
    G->SetCurrentVertex(u);
    int v;
    while (G->GetNextAdjacent(u,v)) {
        if (v!=-1)
        if (color[v] == ColorNode::Unvisited) {
            pi[v] = u;
            DFSVisit(v, G, color, discovery, finished, pi, t);
        }
    }
    color[u] = ColorNode::Finished;
    t++;
    finished[u] = t;
}
```

Algorithm 9.26: Example.

```
void main() {
    int n = 5;
    MyGraphAM* G = new MyGraphAM(n);
    G->SetUndirectedEdge(0, 1);
    G->SetUndirectedEdge(1, 2);
    G->SetUndirectedEdge(2, 3);
    G->SetUndirectedEdge(3, 0);
    G->SetUndirectedEdge(0, 4);
    G->SetUndirectedEdge(1, 4);
    G->Display();
    delete G;
}
```

9.5 Dynamic programming

9.5.1 Definitions

Dynamic programming (DP) is about solving problems by combining the solution to subproblems. Contrary to the Divide & Conquer approach, the partition of the problem is into disjoint subproblems. Dynamic programming is when the subproblems overlap. In such a case, we solves each subproblem one time and then saves its answer in a table. The goal is to avoid the work of recomputing the answer every time it solves each sub-sub-problem.

DP solves problems by combining solutions to subproblems. To apply DP: subproblems are not independent and subproblems may share subsubproblems. However, a solution to one subproblem may not affect the solutions to other subproblems of the same problem. DP reduces computation by:

- Solving subproblems in a bottom-up fashion.
- Storing solution to a subproblem the first time it is solved.
- Looking up the solution when subproblem is encountered again.

DP involves the following 4 main steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
   - Bottom-up with a table
   - Top-down with caching
4. Construct an optimal solution from computed information.

Memoization is an optimization technique used primarily to speed up computer programs. It is used by storing the results of expensive function calls, and by returning the cached result when the same inputs occur again. It ensures that a method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs. It is a common strategy for DP problems.

9.5.2 Back to Fibonacci

The variable step is used to count the number of calls to the Fibonacci function.

Algorithm 9.27: Default Fibonacci.

```
int Fibonacci1(int x, int &step) {
    if (x == 0) {
        step++;
        return 0;
    }
    else if (x == 1) {
        step++;
        return 1;
    }
    else {
        step++;
        return Fibonacci1(x - 1,step) + Fibonacci1(x - 2, step);
    }
}
```

Algorithm 9.28: Fibonacci with DP.

```
int Fibonacci2(int x,int* t,int &step) {
    if (x == 0) {
        t[x] = 0;
        step++;
        return 0;
    }
    else if (x == 1) {
        t[x] = 1;
        step++;
        return 1;
    }
    else {
        int a,b;
        a = (t[x-1]!=-1) ? t[x - 1] : Fibonacci2(x-1,t,step);
        b = (t[x-2]!=-1) ? t[x - 2] : Fibonacci2(x-2,t,step);
        t[x] = a+b;
        step++;
        return t[x];
    }
}
```

Algorithm 9.29: Comparisons.

```
void main() {
    int step,n=12;
    for (int v = 0; v < n; v++) {
        step = 0;
        cout << "F1:" << v << ":"
            << Fibonacci1(v, step);
        cout << " with " << step << " steps " << endl;
        // F1:11:89 with 287 steps
        int* table = new int[n];
        for (int i = 0; i < n; i++)
            table[i] = -1;
        step = 0;
        cout << "F2:" << v << ":"
             << Fibonacci2(v, table, step);
        cout << " with " << step  << " steps " << endl;
        // F2:11:89 with 12 steps
        delete[] table;
    }
}
```

9.6 Shortest path - all pairs

This section is about the Floyd Warshall algorithm, which is based on dynamic programming. The method is added to the Graph with the Adjacency matrix implementation. In the presented code, it computes the matrix of distances, the matrix of predecessors (Π), and the transitive closure (TC), i.e., to determine whether the graph G contains a path from i to j for all vertex pairs (i,j). The algorithm is based on 3 nested loops. However, it is critical to pay attention to the indices. In C++, we start the array at 0, and the original algorithm goes from 1 to n. Therefore, we allocate the space for n+1, where n is the number of vertices in G. In the algorithm, we consider the values of k from 1 to n, and as the indices in the matrix go from 0 to n-1, we access the values of k-1 in the main loop.

Algorithm 9.30: Print all pairs.

```
void MyGraphAM::PrintAllPairs(MyMatrix* pi,int i, int j) {
    if (i == j)
        cout << i ;
    else if (pi->GetCell(i, j) == -1)
        cout << "No path from " << i << " to " << j << endl;
    else {
        PrintAllPairs(pi, i, pi->GetCell(i, j));
        cout << "," << j;
    }
}
```

Algorithm 9.31: Floyd Warshall.

```
void MyGraphAM::FloydWarshall(MyMatrix* &dout, MyMatrix* &piout) {
    MyMatrix** d = new MyMatrix*[n+1]; // 0 + n steps
    MyMatrix** pi = new MyMatrix*[n+1];
    d[0] = new MyMatrix(n, n); // matrix of distances
    pi[0] = new MyMatrix(n, n); // matrix predecessors
    // Initialization
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            double w = GetEdgeWeight(i, j);
            if (i==j)
                d[0]->SetCell(i, j, 0);
            else {
                if (w != 0)
                    d[0]->SetCell(i, j, w);
                else
                    d[0]->SetCell(i, j, DBL_MAX);
            }
            if ((i == j) || (w == 0))
                pi[0]->SetCell(i, j, -1); // No Predecessor
            else
                pi[0]->SetCell(i, j, i);
        }
    // Dynamic programming
    for (int k = 1; k <= n; k++) {
        d[k]=new MyMatrix(n, n);
        pi[k]=new MyMatrix(n, n);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) {
                d[k]->SetCell(i, j,min(d[k - 1]->GetCell(i,j),
                        d[k-1]->GetCell(i,k-1) + d[k-1]->GetCell(k-1,j)));
                if (d[k]->GetCell(i,j) == d[k-1]->GetCell(i,j))
                    pi[k]->SetCell(i,j,pi[k-1]->GetCell(i,j));
                else
                    pi[k]->SetCell(i,j,pi[k-1]->GetCell(k-1,j));
            }
    }
    dout = d[n]; piout = pi[n];
    for (int k = 0; k < n; k++) {
        delete d[k]; delete pi[k];
    }
    delete[] d;  delete[] pi;
}
```

Algorithm 9.32: Transitive closure.

```
void MyGraphAM::TransitiveClosure(MyMatrix* &TCout) {
    MyMatrix** TC = new MyMatrix*[n + 1];
    TC[0] = new MyMatrix(n, n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            double w = GetEdgeWeight(i, j);
            if ((i == j) || (w != 0))
                TC[0]->SetCell(i, j, 1);
            else
                TC[0]->SetCell(i, j, 0);
        }
    for (int k = 1; k <= n; k++) {
        TC[k] = new MyMatrix(n, n);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                TC[k]->SetCell(i, j, min(TC[k - 1]->GetCell(i, j),
                    TC[k - 1]->GetCell(i, k - 1) +
                    TC[k - 1]->GetCell(k - 1, j)));
    }
    TCout = TC[n];
    for (int k = 0; k < n; k++)
        delete TC[k];
    delete[] TC;
}
```

Algorithm 9.33: Floyd Warshall Example.

```
void main() {
    cout << "Floyd Warshall test" << endl;
    int n = 5;
    MyGraphAM* G = new MyGraphAM(n);
    G->SetDirectedEdge(0, 1, 3);
    G->SetDirectedEdge(0, 2, 8);
    G->SetDirectedEdge(0, 4, -4);
    G->SetDirectedEdge(1, 3, 1);
    G->SetDirectedEdge(1, 4, 7);
    G->SetDirectedEdge(2, 1, 4);
    G->SetDirectedEdge(3, 0, 2);
    G->SetDirectedEdge(3, 2, -5);
    G->SetDirectedEdge(4, 3, 6);
    cout << "Display graph:" << endl;
    G->Display();
    MyMatrix *d,*pi,*TC;
    G->FloydWarshall(d, pi);
    G->TransitiveClosure(TC);
    cout << "Matrix with distances:" << endl;
    d->Display();
    cout << "Matrix with predecessor:" << endl;
    pi->Display();
    cout << "Matrix with Transitive Closure:" << endl;
    TC->Display();
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            cout << "Path from " << i << " to " << j << endl;
            G->PrintAllPairs(pi, i, j);
            cout << endl;
        }
}
```

9.7 Minimum Spanning Tree

Algorithm 9.34: minKey (Prim).

```
int MyGraphAM::minKey(int* key, bool* mstSet) {
    int min = INT_MAX;
    int min_index = INT_MAX;
    for (int v = 0; v<n; v++)
        if ((mstSet[v] == false) && (key[v] < min)) {
            min = key[v];
            min_index = v;
        }
    return min_index;
}
```

Algorithm 9.35: Display MST.

```
void MyGraphAM::DisplayMST(int* parent) {
    cout << "MST Edge Weight" << endl;
    for (int i = 1; i<n; i++)
        cout << "(" << parent[i] << "-> " << i << ") w="
       << GetEdgeWeight(i,parent[i]) << endl;
}
```

Algorithm 9.36: Acyclic.

```
void Acyclic(int u, MyGraphAM &G, int* color, int* pi, int* back, int parent) {
    color[u] = 1; // visited;
    int v,n = G.GetNumberVertices();
    G.SetCurrentVertex(u);
    while (G.GetNextAdjacent(u, v)) {
        if ( G.ExistEdge(u,v) && (v!=u) && (v!=parent)) {
            if (color[v] == 0) { // not already visited
                pi[v] = u;
                Acyclic(v, G, color, pi, back, u);
            }
            else
                back[v] = u;
        }
    }
    color[u] = 2;
}
```

Algorithm 9.37: IsAcyclic.

```
bool IsAcyclic(int u, MyGraphAM &G) {
    int n = G.GetNumberVertices();
    int* color = new int[n];
    int* back = new int[n];
    int* pi = new int[n];
    for (int i = 0; i < n; i++) {
        color[i] = 0;
        back[i] = -1;
        pi[i] = -1;
    }
    Acyclic(u,G,color,pi,back,-1);
    int i = 0;
    bool acyclic=true;
    bool found = false;
    while ((i < n) && (!found)) {
        if ((back[i] == u) && (pi[i]!=u)) {
            found = true;
            acyclic = false;
        }
        i++;
    }
    delete[] color;     delete[] back;
    return acyclic;
}
```

9.7.1 Kruskal

Algorithm 9.38: Kruskal.

```
void MyGraphAM::MSTKruskal1() {
    MyGraphAM A(n); // edges we will keep
    MyMatrix S(n, n); // copy edges
    S.Init(0);
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            S(i, j) = GetEdgeWeight(i, j);
    int total = 0; // number of edges
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            if (S(i, j) != 0)
                total++;
    int argi, argj;
    double value;
    tie(value, argi, argj) = S.GetMin();
    S(argi, argj) = 0;
    A.SetUndirectedEdge(argi, argj, value);
    for (int i = 0; i<total; i++) { // for each edge
        tie(value, argi, argj) = S.GetMin();
        S(argi, argj) = 0;
        A.SetUndirectedEdge(argi, argj, value);
        if (!IsAcyclic(argi, A))
            A.RemoveUndirectedEdge(argi, argj);
    }
    cout << "Display graph" << endl;
    A.DisplayUndirectedEdge();
}
```

9.7.2 Prim

Algorithm 9.39: Prim.

```
void MyGraphAM::MSTPrim() {
    int* key = new int[n];
    int* pi = new int[n];
    bool* mstSet = new bool[n];
    for (int u = 0; u < n; u++) {
        key[u] = INT_MAX; // infinite value
        mstSet[u] = false;
    }
    key[0] = 0;
    pi[0] = -1;
    for (int i = 0; i < n - 1; i++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        //Consider only those vertices which are not yet included in MST
        SetCurrentVertex(u);
        int v;
        while (GetNextAdjacent(u, v)) {
            double w = GetEdgeWeight(u, v);
            if ((mstSet[v] == false) && (w < key[v])) {
                pi[v] = u;
                key[v] = w;
            }
        }
    }
    DisplayMST(pi);
    delete[] pi;
    delete[] key;
    delete[] mstSet;
}
```

Algorithm 9.40: Example MST.

```
void main() {
  int n = 9;
    MyGraphAM* G = new MyGraphAM(n);
    G->SetUndirectedEdge(0, 1, 4);
    G->SetUndirectedEdge(1, 2, 8);
    G->SetUndirectedEdge(2, 3, 7);
    G->SetUndirectedEdge(3, 4, 9);
    G->SetUndirectedEdge(4, 5, 10);
    G->SetUndirectedEdge(5, 6, 2);
    G->SetUndirectedEdge(6, 7, 1);
    G->SetUndirectedEdge(7, 0, 8);
    G->SetUndirectedEdge(7, 8, 7);
    G->SetUndirectedEdge(8, 2, 2);
    G->SetUndirectedEdge(8, 6, 6);
    G->SetUndirectedEdge(2, 5, 4);
    G->SetUndirectedEdge(3, 5, 14);
    cout << "Kruskal" << endl;
    G->MSTKruskal();
    cout << "Prim" << endl;
    G->MSTPrim();
```

Chapter 10

Matrix

Contents

10.1 Class interface

The class MyMatrix is a useful class with a large number of different implementations. The proposed implementation highlight how to overload operators and it provides a state of the art algorithm related to dynamic programming with the multiplication of a sequence of matrices. In this implementation, we consider arrays of arrays, contrary to the implementation of the adjacency matrix that was used with the graphs.

Algorithm 10.1: Class interface.

```
class MyMatrix {
public:
    MyMatrix();
    MyMatrix(int n, int m);
    MyMatrix(const MyMatrix&);
    ~MyMatrix();
    void Init(double x);
    void Identity();
    void Display();
    int GetRows() const { return n; }
    int GetCols() const { return m; }
    double GetCell(int i, int j);
    void SetCell(int i, int j, double x);
    double operator()(int i, int j) const;
    double& operator()(int i, int j);
    void operator=(MyMatrix &M1);
    MyMatrix Add(MyMatrix &M1);
    MyMatrix Sub(MyMatrix &M1);
    MyMatrix operator+(MyMatrix& a);
    MyMatrix operator+(double x);
    MyMatrix operator-(MyMatrix& a);
    MyMatrix operator-(double x);
    MyMatrix operator*(MyMatrix& a);
    MyMatrix operator*(double x);
    tuple<double, int, int> GetMin();
private:
    int n, m; // size of the matrix
    double** A;
    void Clean();
};
```

10.2 Main methods

Algorithm 10.2: Constructors and destructor

```
MyMatrix::MyMatrix() {
    n = 0;
    m = 0;
    A = nullptr;
}
MyMatrix::MyMatrix(int n1, int m1) {
    n = n1; m = m1;
    A = new double*[n];
    for (int i = 0; i < n; i++) {
        A[i] = new double[m];
        for (int j = 0; j < m; j++)
            A[i][j] = 0;
    }
}
MyMatrix::MyMatrix(const MyMatrix& M) {
    n = M.n; m = M.m;
    A = new double*[n];
    for (int i = 0; i < n; i++) {
        A[i] = new double[m];
        for (int j = 0; j < m; j++)
            A[i][j] = M.A[i][j];
    }
}
void MyMatrix::Clean() {
    for (int i = 0; i < n; i++)
        delete[] A[i];
    delete[] A;
}
MyMatrix::~MyMatrix() {
    Clean();
}
```

Algorithm 10.3: Accessors and modifiers

```
double MyMatrix::GetCell(int i,int j) {
    try {
        return A[i][j];
    }
    catch (exception& e) {
        cout << "Problem with index" << e.what() << endl;
    }
}

// Get value
double MyMatrix::operator()(int i, int j) const {
    return A[i][j];
}
// Set value
double& MyMatrix::operator()(int i, int j) {
    return A[i][j];
}

void MyMatrix::SetCell(int i, int j,double x) {
    try {
       A[i][j]=x;
    }
    catch (exception& e) {
        cout << "Problem with index" << e.what() << endl;
    }
}
```

Algorithm 10.4: Display

```
void MyMatrix::Display() {
    cout << "size: " << n << "x" << m << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            cout << A[i][j] << " ";
        cout << endl;
    }

}
```

10.3 Matrix operations

Algorithm 10.5: Initialization

```
void MyMatrix::Init(double x) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            A[i][j] = x;
}
```

Algorithm 10.6: Identity

```
void MyMatrix::Identity() {
    if (n == m) {
        Init(0);
        for (int i = 0; i < n; i++)
                A[i][i] = 1;
    }
}
```

Algorithm 10.7: Assignment

```
void MyMatrix::operator=(MyMatrix &M1) {
    Clean();
    n=M1.n;
    m=M1.m;
    A = new double*[n];
    for (int i = 0; i < n; i++) {
        A[i] = new double[m];
        for (int j = 0; j < m; j++)
            A[i][j] = M1.A[i][j];
    }
}
```

Algorithm 10.8: Addition

```
MyMatrix MyMatrix::Add(MyMatrix &M1) {
    MyMatrix out(n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            out(i,j)=this->A[i][j] + M1.A[i][j];
    return out;
}

MyMatrix MyMatrix::operator+(MyMatrix &M1) {
    MyMatrix out(n,m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            out(i,j)=A[i][j] + M1.A[i][j];
    return out;
}

MyMatrix MyMatrix::operator+(double x) {
    MyMatrix out(n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            out(i,j)=A[i][j] + x;
    return out;
}
```

Algorithm 10.9: Subtraction

```
MyMatrix MyMatrix::Add(MyMatrix &M1) {
    MyMatrix out(n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            out(i,j)=A[i][j] - M1.A[i][j];
    return out;
}
MyMatrix MyMatrix::operator-(MyMatrix &M1) {
    MyMatrix out(n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            out.SetCell(i, j, A[i][j] - M1.A[i][j]);
    return out;
}
MyMatrix MyMatrix::operator-(double x) {
    MyMatrix out(n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            out.SetCell(i, j, A[i][j] - x);
    return out;
}
```

Algorithm 10.10: Multiplication

```
MyMatrix MyMatrix::operator*(MyMatrix &M1) {
    if (this->m != M1.n) {
        cout << "Problem with the dimensions." << endl;
        return *this;
    }
    else
    {
        double tmp;
        MyMatrix out(n,M1.m);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < M1.m; j++) {
                tmp = 0;
                for (int k = 0; k < m; k++)
                    tmp += A[i][k] * M1(k,j);
                out.A[i][j] = tmp;
            }
        return out;
    }
}

MyMatrix MyMatrix::operator*(double x) {
    MyMatrix out(n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            out.SetCell(i, j,A[i][j] * x);
    return out;
}
```

The goal of the following function is to return the minimum value of the matrix with its corresponding indices. The element is then set to 0.

Algorithm 10.11: GetMin

```
tuple<double, int, int> MyMatrix::GetMin() {
    double minvalue = DBL_MAX;
    int argi = 0;
    int argj = 0;
    for (int i = 0; i<n; i++)
        for (int j = 0; j < m; j++) {
            if ((A[i][j] < minvalue) && (A[i][j] > 0)) {
                minvalue = A[i][j];
                argi = i;
                argj = j;
            }
        }
    A[argi][argj] = 0; // set the min to 0
    return make_tuple(minvalue, argi, argj);
}
```

10.4 Multiplication of multiple matrices

This algorithm uses the concept of dynamic programming that was seen in Chapter 9.

Algorithm 10.12: Print brackets

```
void PrintBrackets(MyMatrix &s,int i,int j) {
    if (i == j)
        cout << "A" << i;
    else {
        cout << "(";
        PrintBrackets(s, i, s(i, j));
        PrintBrackets(s,s(i,j)+1,j);
        cout << ")";
    }
}
```

Algorithm 10.13: Multiplication

```
MyMatrix Multiply(MyMatrix** sequence,MyMatrix &s, int i, int j) {
    if (i == j)
        return *(sequence[i]);
    else
        return Multiply(sequence,s, i, s(i, j))*
               Multiply(sequence,s, s(i, j) + 1, j);
}
```

Algorithm 10.14: MatrixChainOrder

```
void MatrixChainOrder(int* p, int n, MyMatrix &m, MyMatrix &s) {
    m = MyMatrix(n,n);
    s = MyMatrix(n-1, n);
    for (int i = 0; i < n; i++)
        m(i,i) = 0; // base case
    for (int l = 2; l <= n; l++) { // chain length
        for (int i = 0; i < n - l + 1; i++) {
            int j = i + l - 1;
            m(i,j) = INT_MAX;
            for (int k = i; k <= j - 1; k++) {
                int q = m(i,k) + m(k+1,j) + p[i] * p[k+1] * p[j+1];
                if (q < m(i, j)) {
                    m(i, j) = q; // cost Ai*...*Aj
                    s(i, j) = k;
                }
            }
        }
    }
}
```

Algorithm 10.15: Example 1.

```
void MatrixChainOrderMult(MyMatrix** sequence, int n) {
    int* p = new int[n + 1];
    p[0] = sequence[0]->GetRows();
    for (int i = 1; i <= n; i++) {
```

```
        p[i] = sequence[i-1]->GetCols();
    }
    for (int i = 0; i <= n; i++)
        cout << p[i] << ", ";
    cout << endl;
    MyMatrix M, S;
    MatrixChainOrder(p, n, M, S);
    cout << "M:" << endl;
    M.Display();
    cout << "S:" << endl;
    S.Display();
    PrintBrackets(S, 0, n-1);
    MyMatrix result=Multiply(sequence,S, 0, n - 1);
    result.Display();
    delete[] p;
}
```

List of Algorithms and Examples

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The design and analysis of computer algorithms." Addison-Wesley, 1974, pp. 145–147.

[2] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[3] D. A. Bell, "The principles of sorting," *The Computer Journal*, vol. 1, pp. 71–77, 1958.

[4] L. Bonacci, *The Book of Calculation*, 1202.

[5] A. D. Booth and A. J. T. Colin, "On the efficiency of a new method of dictionary construction," *Information and control*, vol. 3, pp. 327–334, 1960.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms - 3rd Edition*. MIT Press and McGraw-Hill, 2009.

[7] A. S. Douglas, "Techniques for the recording of, and reference to data in a computer," *The Computer Journal*, vol. 2, pp. 1–9, 1959.

[8] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[9] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the Association for Computing Machinery*, vol. 34, no. 3, pp. 596–615, 1987.

[10] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. of the 19th Annual Symposium on Foundations of Computer Science*, 1978, pp. 8–21.

[11] T. Hibbard, "Some combinatorial properties of certain trees," *Assoc. Comp. Mach.*, vol. 9, p. 13, 1962.

[12] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," in *Proc. of the American Mathematical Society*, vol. 7, 1956, pp. 48–50.

[13] E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*. Teubner, 1909, 2 volumes. Reprinted by Chelsea, New York, 1953.

[14] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[15] B. Stroustrup, *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, Aug. 2013.

[16] J. Vuillemin, "A data structure for manipulating priority queues," *Communications of the ACM*, vol. 21, no. 4, pp. 309–315, 1978.

[17] P. F. Windley, "Trees, forests, and rearranging," *The Computer Journal*, vol. 3, p. 84, 1960.