Data Structures & Algorithms for Students in Computer Science

Hubert Cecotti

Spring 2019 (v0.013)

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 are covered during the class. 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. 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++.
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.

Chapter 1

Introduction

Contents

1.1 Symbols

Symbols are used in mathematical and logical expressions. They can also used pseudo-code to represent variables. For a better understanding, it is critical to know how to pronounce these different symbols.

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

1.2 Sets

To define the elements in a set, we use curly brackets: { }. Example: $A = \{1, 2, 3\}$. The set A contains the elements 1, 2, and 3.
It should not be confused with the { } that are used in C++ to defined the beginning and the end of a block of instructions, or a static array.

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: $B \not\supseteq A$

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

- 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: $C = A'$

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

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

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

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

$$C = \{c|c \in A \wedge c \in B\} \tag{1.12}$$

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

$$C = \{a|a \in A \wedge a \notin B\} \tag{1.13}$$

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

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

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

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 functions that map positive integers to positive real numbers.

- 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 $f(n) \in \Omega(g(n))$.

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

- O: Big O, asymptotic upper bound.

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

- Ω: Big omega, asymptotic lower bound

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

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

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

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

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

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

- $O(1)$: constant.
- $O(log(n))$: logarithmic (examples: finding an item in a sorted array with a binary search or a balanced search tree).
- $O(n)$: linear (examples: finding an item in an unsorted list or in an unsorted array).
- $O(n \cdot log(n))$: loglinear (example: mergesort).
- $O(n^2)$: quadratic (examples: selection sort and insertion sort).

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

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

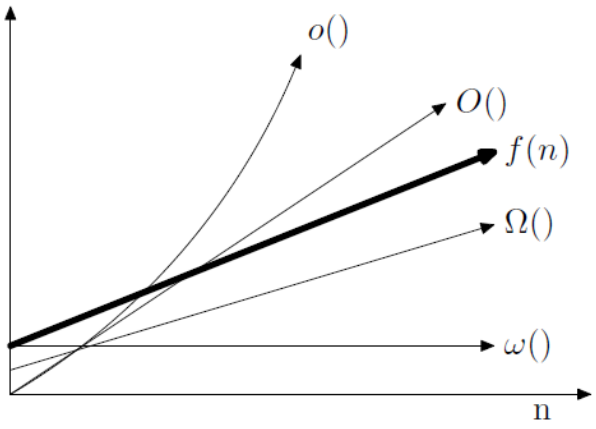

Figure 1.1: Relationships between the notations.

1.3.2 Divide and conquer

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

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

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

with

- n: size of an input problem.
- 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.

The critical exponent is defined by:

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

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 $T(n) = \Omega(f(n))$.

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 growing
$0.5T(n/2)+1/n$	0.5	2	NO	$a<1$
$64T(n/8)-n^2logn$	64	8	NO	$f(n)$ not positive

1.3.5 Time complexity

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

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.

Listing 1.1: Expression values.

```
1  void main() {
2    const int i = 12;
3      int j,l = 12;
4      // i = 13; // cannot change i because it was defined as a const
5      j = i; // j can change its value as it s not a const
6
7      // a literal such as 56 is a prvalue
8      // 56 = i; // it does not work
9      j = 56; // it works
10
11     // left side is an expression BUT it returns an lvalue ( j or l)
12     ((j < 30) ? j : l) = 7;
13
14     // It does not work as it  returns an evaluated expression j*2 or l*2
15     // ((j < 30) ? j*2 : l*2) = 7;
16 }
```

1.4.2 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.

Listing 1.2: Includes.

```
1 using namespace std;
2 #include <chrono>     // to use functions related to the time
3 #include <string>     // to use strings
4 #include <tuple>      // to use tuple
5 #include <iostream>   // to use cout, cin
6 #include <fstream>    // to use files
7 #include <algorithm> // to use max, min
8 #include <stdlib.h>   // to use rand
```

1.4.3 Types

Listing 1.3: Type conversion and promotion.

```
1  void main() {
2      int i = 5;
3      int j = 2;
4      double d = 2.1;
5      cout << i + d << endl; // 7.1 (double)
6      cout << i* d << endl;  // 10.5 (double)
7      cout << d* i << endl;  // 10.5 (double)
8      cout << i / d << endl; // 2.38095 (double)
9      cout << d / i << endl; // 0.42 (double)
10     cout << i / j << endl; // 2 (int)
11 }
```

1.4.4 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.

Listing 1.4: Static and dynamic arrays.

```
1  void main() {
2    // Static array
3      int s[5]; // 5 elements only, cannot be changed
4      // primes: 7 elements only
5      int primes[] = { 1, 2, 3, 5, 7, 11, 13 };
6      // primes1: 10 elements, 5 first elements are initialized
7      int primes1[10] = { 1, 2, 3, 5, 7 };
8      // Dynamic array
9      int n = 10;
10     int *x = new int[n]; // an array of n int
11     cout << "Number of elements in primes: ";
12     cout << sizeof(primes) / sizeof(int) << endl; // 7
13     cout << "Number of elements in primes1: ";
14     cout << sizeof(primes1) / sizeof(int) << endl; // 10
15     cout << "Number of elements in x: ";
16     cout << sizeof(*x)/sizeof(double) << endl; // 1
17     delete[] x;  // delete the array x
18 }
```

1.4.5 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.

Listing 1.5: C++ examples.

```
1  // void: no return keyword
2  void f1(int* a,int n) {
3      a = new int[n];
4      for (int i = 0; i < n; i++)
5          a[i] = i;
6  }
7
8  // void: no return keyword
9  void f2(int** a, int n) {
10     *a = new int[n];
11     for (int i = 0; i < n; i++)
12         (*a)[i] = i;
13 }
14
15 // f3 must return an int*
16 int* f3(int n) {
17     int *a = new int[n];
18     for (int i = 0; i < n; i++)
19         a[i] = i;
20     return a;
21 }
22 void DisplayArray(int* a, int n) {
23     for (int i = 0; i < n; i++)
24         cout << a[i] << ",";
25     cout << endl;
26 }
```

Listing 1.6: Main test functions.

```
void main() {
    int n = 10;
    int* a = NULL;
    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 = NULL;
    a=f3(n);
    DisplayArray(a, n);
}
```

Arrays

Reference versus Pointers for arrays.

Listing 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++;
}
```

Listing 1.8: C++ function calls.

```
1  int main() {
2      int x = 6;
3      cout << f0(x) << endl; // x=6, f0(x) returns 6
4      cout << f1(x) << endl; // x=6, f1(x) returns 7
5      f2(&x); // x=6, after f2, x=6
6      cout << x << endl; // x=6
7      f3(&x); // x=6, after f3, x=7
8      cout << x << endl; // x=7
9      f4(x); // x=7, after f4, x=8
10     cout << x << endl; // x=8
11 }
```

It is not because you have a * that you pass the variable as a pointer, you need to consider the type of the variable you are manipulating.

Listing 1.9: C++ functions.

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

Listing 1.10: C++ function calls.

```
void main() {
    int *v1=NULL,*v2=NULL,*v3=NULL;
    int v4;
    f5(&v2, 5);
    f6(v1,5);
    f7(v3,5);
    f8(v4,5);
    cout << v1[3] << endl; // 3
    cout << v2[3] << endl; // 3
    cout << v3[3] << endl; // read access violation
    cout << v4;

}
```

1.4.6 Sum and product

Listing 1.11: Variables declaration.

```
1   int n = 10;
2     double x;
3     double* a = new double[n];
4     double* b = new double[n];
5     for (int i = 0; i < n; i++) {
6         a[i] = (2*(double)i+1)/n;
7         b[i] = (5* (double)i+2)/n;
8     }
```

Sum 1:

$$x = \sum_{i=1}^{n} i \tag{1.22}$$

Sum 2:

$$x = \sum_{i=1}^{n} 1/i \tag{1.23}$$

Sum 3:

$$x = \sum_{i=0}^{n-1} a(i) \cdot b(i) \tag{1.24}$$

Product 1:

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

Product 1:

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

Listing 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.

Listing 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;
}
```

1.4.7 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.

Listing 1.14: For loop with break.

```
1  void Search_v01(int* a, int n,int x) {
2      bool find = false;
3      int argx = -1;
4      for (int i = 0; i < n; i++) {
5          if (a[i] > x) {
6              find = true;
7              argx = i;
8              break;
9          }
10     }
11     if (find)
12         cout << "Found " << a[argx] << endl;
13     else
14         cout << "Not found" << endl;
15 }
```

Listing 1.15: While loop.

```
1  void Search_v02(int* a, int n, int x) {
2      bool find = false;
3      int argx = -1, i = 0;
4      while ((!find) && (i < n)) {
5          if (a[i] > x) {
6              find = true;
7              argx = i;
8          }
9          i++;
10     }
11     if (find)
12         cout << "Found " << a[argx] << endl;
13     else
14         cout << "Not found" << endl;
15 }
```

Listing 1.16: Do While loop.

```
1  void Search_v03(int* a, int n, int x) {
2      bool find = false;
3      int argx = -1, i = 0;
4      do {
5          if (a[i] > x) {
6              find = true;
7              argx = i;
8          }
9          i++;
10     } while ((!find) && (i < n));
11     if (find)
12         cout << "Found " << a[argx] << endl;
13     else
14         cout << "Not found" << endl;
15
16 }
```

Listing 1.17: Main.

```
1  void main() {
2      int n = 10, x = 100;
3      int* a = new int[n];
4      for (int i = 0; i < n; i++)
5          a[i] = rand() % 200;
6      Search_v01(a, n, x);
7      Search_v02(a, n, x);
8      Search_v03(a, n, x);
9  }
```

1.5 Recursivity

1.5.1 Factorial

Definition:

$$Factorial(x) \quad = \quad x \cdot Factorial(x-1) \tag{1.27}$$

Listing 1.18: Factorial functions.

```
1  int Factorial1(int x) {
2      int result = 1;
3      for (int i = 2; i <= x; i++)
4          result *= i;
5      return result;
6  }
7
8  int Factorial2(int x) {
9      int result = 1;
10     int i = 2;
11     while (i <= x) {
12         result *= i;
13         i++;
14     }
15     return result;
16 }
17
18 int Factorial3(int x) {
19     if (x <= 1)
20         return 1;
21     else
22         return x*Factorial3(x - 1);
23 }
```

Listing 1.19: Factorial examples.

```
1  void main() {
2      cout << Factorial1(6) << endl; // 720
3      cout << Factorial2(6) << endl; // 720
4      cout << Factorial3(6) << endl; // 720
5  }
```

1.5.2 Fibonacci

Definition:

$$Fibonacci(0) = 0 \tag{1.28}$$
$$Fibonacci(1) = 1 \tag{1.29}$$
$$Fibonacci(x) = Fibonacci(x-1) + Fibonacci(x-2) \tag{1.30}$$

Listing 1.20: Fibonacci function.

```
1 int Fibonacci(int x) {
2     if (x == 0)
3         return 0;
4     else if (x == 1)
5         return 1;
6     else
7         return Fibonacci(x - 1) + Fibonacci(x - 2);
8 }
```

Listing 1.21: Fibonacci example.

```
1 void main() {
2     cout << Fibonacci(6) << endl; // 8
3 }
```

1.5.3 Anagram

Listing 1.22: Rotate.

```
1  //  Method to rotate left all characters from position to end
2  void Rotate(char* str, int size, int newsize) {
3      int position = size - newsize;
4      char temp = str[position];
5      int i;
6      for (i = position + 1; i < size; i++) {
7          str[i-1] = str[i];
8      }
9      str[i-1] = temp;
10 }
```

Listing 1.23: Do anagram.

```
1  void DoAnagram(char* str, int size, int newsize) {
2      if (newsize > 1) {
3          for (int loop = 0; loop < newsize; loop++) {
4              DoAnagram(str, size, (newsize - 1));
5              if (newsize == 2) {
6                  for (int i = 0; i < size; i++) {
7                      cout << str[i];
8                  }
9                  cout << endl;
10             }
11             Rotate(str, size, newsize);
12         }
13     }
14 }
```

Listing 1.24: Anagram example.

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

1.5.4 Hanoi Tower

Listing 1.25: 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);
    }
}
```

Listing 1.26: 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

Listing 2.1: Stack - class definition

```
1  typedef double MyType;
2  class MyStack {
3  public:
4      MyStack();
5      MyStack(int capacity1);
6      ~MyStack();
7      bool IsFull();
8      bool IsEmpty();
9      MyType Pop();
10     MyType Top();
11     void Push(MyType x);
12     void Display();
13 public:
14     MyType* s;
15     int capacity;
16     int size;
17 };
```

Listing 2.2: Stack - functions

```
1  MyStack::MyStack() {
2      s = NULL;
3      capacity = 0;
4      size = 0;
5  }
6  MyStack::MyStack(int capacity1) {
7      capacity = capacity1;
8      s = new MyType[capacity];
9      size = 0;
10 }
11 MyStack::~MyStack() {
12     delete[] s;
13 }
14 bool MyStack::IsFull() {
15     return (size == capacity);
16 }
17 bool MyStack::IsEmpty() {
18     return (size == 0);
19 }
20 MyType MyStack::Pop() {
21     size--;
22     return s[size];
23 }
24 MyType MyStack::Top() {
25     return s[size - 1];
26 }
27 void MyStack::Push(MyType x) {
28     if (size < capacity) {
29         s[size] = x;
30         size++;
31     }
32 }
33 void MyStack::Display() {
34     cout << "Max capacity: " << capacity << endl;
35     cout << "Size: " << size << endl;
36     for (int i = 0; i < size; i++)
37         cout << "Element: " << s[i] << " at position " << i << endl;
38     cout << endl;
39 }
```

Listing 2.3: Example

```
1  void main() {
2          MyStack* S = new MyStack(5);
3          S->Push(4);
4          S->Push(6);
5          S->Push(8);
6          S->Push(10);
7          S->Push(12);
8          S->Push(16);
9          S->Push(18);
10         S->Display();
11         cout << "Pop: " << S->Pop() << endl;
12         cout << "Pop: " << S->Pop() << endl;
13         cout << "Top: " << S->Top() << endl;
14         S->Push(20);
15         S->Display();
16         delete s;
17 }
```

2.2 Queues

Listing 2.4: Circular queue - class definition.

```
1  typedef double MyType;
2  class MyQueue {
3  public:
4      MyQueue();
5      MyQueue(int capacity1);
6      ~MyQueue();
7      bool IsFull();
8      bool IsEmpty();
9      void Enqueue(MyType x);
10     MyType Dequeue();
11     MyType Front();
12     MyType Rear();
13     void Display();
14 public:
15     int front, rear, size;
16     int capacity;
17     MyType* q;
18 };
```

Listing 2.5: Circular queue - functions.

```
1  MyQueue::MyQueue() {
2      capacity = 0;
3      front=size=0;
4      rear=capacity-1;
5      q=NULL;
6  }
7  MyQueue::~MyQueue() {
8      delete[] q;
9  }
10 MyQueue::MyQueue(int capacity1) {
11     capacity=capacity1;
12     front=size=0;
13     rear=capacity-1;  // important, see the enqueue
14     q=new MyType[capacity];
15 }
16 bool MyQueue::IsFull() {
17     return (size==capacity);
18 }
19 bool MyQueue::IsEmpty() {
20     return (size==0);
21 }
22 void MyQueue::Enqueue(MyType x) {
23     if (!IsFull()) {
24         rear = (rear + 1) % capacity;
25         q[rear] = x;
26         size = size + 1;
27     }
28 }
29 MyType MyQueue::Dequeue() {
30     if (IsEmpty())
31         return INT_MIN;
32     MyType item = q[front];
33     front=(front+1)%capacity;
34     size=size-1;
35     return item;
36 }
37 MyType MyQueue::Front() {
38     if (IsEmpty())
39         return INT_MIN;
40     return q[front];
41 }
42 MyType MyQueue::Rear() {
43     if (IsEmpty())
44         return INT_MIN;
45     return q[rear];
```

```
46 }
47 void MyQueue::Display() {
48     cout << "Max capacity: " << capacity << endl;
49     cout << "Size: " << size << endl;
50     for (int i = 0; i < size; i++)
51         cout << "Element: " << q[i] << " at position " << i << endl;
52     cout << endl;
53 }
```

Listing 2.6: Example

```
1  void main() {
2          MyQueue* Q = new MyQueue(4);
3          Q->Enqueue(4);
4          Q->Enqueue(6);
5          Q->Enqueue(8);
6          Q->Enqueue(10);
7          Q->Enqueue(12);
8          Q->Enqueue(16);
9          Q->Enqueue(18);
10         Q->Display();
11         cout << "Dequeue: " << Q->Dequeue() << endl;
12         cout << "Dequeue: " << Q->Dequeue() << endl;
13         Q->Enqueue(20);
14         Q->Enqueue(22);
15         Q->Enqueue(24);
16         Q->Enqueue(26);
17         Q->Display();
18         cout << "Dequeue: " << Q->Dequeue() << endl;
19         Q->Enqueue(28);
20         Q->Display();
21         delete Q;
22 }
```

Chapter 3

Prototype pattern

Contents

The goal of this section is to show how you can create the prototype pattern. 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

Listing 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.

Listing 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() {}
};
```

Listing 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.

Listing 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];
};
```

Listing 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();
}
struct Destruct {
    void operator()(MyDataStructure *a) const {
        delete a;
    }
};
```

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.

Listing 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);
```

```
40      // Sorting functions
41      bool IsSorted();
42      void SwapIndex(int i, int j);
43      void DisplayStep(string f);
44      // Init functions
45      void InitRandom(int v);
46      void InitSortedAscending(int v);
47      void InitSortedDescending(int v);
48      // Sorting algorithms
49      void SelectionSort();
50      void InsertionSort();
51      void BubbleSort();
52      void BubbleOptSort();
53      void MergeSort();
54      void QuickSort();
55  private:
56      MyType* a; // array
57      int n; // size of the array
58  };
```

4.1.2 Constructors and destructor

Listing 4.2: Constructors and destructor (MyArray.cpp)

```
1  // Number of steps
2  // to count how many main steps are done in an algorithm.
3  int step;
4
5  // Default constructor
6  MyArray::MyArray() {
7      a = NULL;
8      n = 0;
9  }
10
11 // Basic constructor
12 // Create an array of size n1
13 MyArray::MyArray(int n1) {
14     n = n1;
15     a = new MyType[n];
16     for (int i = 0; i < n; i++)
17         a[i] = 0;
18 }
19
20 // Destructor
21 MyArray::~MyArray() {
22     delete[] a;
23 }
```

4.1.3 Access elements

Listing 4.3: Access elements (MyArray.cpp).

```
1  // Return the number of elements in the array
2  int MyArray::GetSize() const { return n; }
3
4  MyType MyArray::GetElement(int i) const {
5      if (i < 0) {
6          cout << "Too small index";
7          exit(EXIT_FAILURE);
8      } else if (i >= n) {
9          cout << "Too large index";
10         exit(EXIT_FAILURE);
11     } else
12         return a[i];
13 }
14
15 // a is private
16 // we have a function to set the value x at the position i
17 void MyArray::SetElement(int i, MyType x) {
18     a[i] = x;
19 }
20
21 MyType& MyArray::operator[] (unsigned i) {
22     try {
23         return a[i];
24     } catch (exception& e) {
25         cout << "Problem with index" << e.what() << endl;
26     }
27 }
28
29 // Concatenate two arrays with the + operator
30 MyArray* MyArray::operator+(const MyArray* a)  {
31     MyArray* out = new MyArray(this->n + a->GetSize());
32     for (int i = 0; i < this->n; i++)
33         out->SetElement(i, this->GetElement(i));
34     for (int i = this->n; i < out->GetSize(); i++)
35         out->SetElement(i+ this->n, a->GetElement(i));
36     return out;
37 }
```

4.1.4 Find, Delete, Insert

// Determine if the value x is in the array

Listing 4.4: Find, Delete, Insert (MyArray.cpp).

```
1  bool MyArray::Find(MyType x) {
2      for (int i = 0; i < n; i++) {
3          if (a[i] == x) {
4              return true;
5          }
6      }
7      return false;
8  }
9
10 void MyArray::Delete(MyType x) {
11     if (Find(x)) {
12         MyType* a1 = new MyType[n - 1];
13         int j = 0;
14         for (int i = 0; i < n; i++) {
15             if (a[i] != x) {
16                 a1[j] = a[i];
17                 j++;
18             }
19         }
20         delete[] a;
21         a = a1;
22     }
23 }
24
25 void MyArray::Insert(MyType x) {
26     MyType* a1 = new MyType[n + 1];
27     for (int i = 0; i < n; i++) {
28         a1[i] = a[i];
29     }
30     a1[n] = x;
31     delete[] a;
32     a = a1;
33     n++;
34 }
```

4.1.5 Binary Search

Listing 4.5: Binary search (MyArray.cpp).

```
1  // Return if the value x is in the array or not, and the index of the value.
2  // Iterative version
3  pair<bool, int> MyArray::BinarySearch1(MyType x) {
4      bool found = false;
5      int mid,low = 0, high = n-1;
6      while ((low <= high) && (!found)) {
7          mid = (low + high) / 2;
8          if (x==a[mid])
9              return make_pair(true,mid);
10         else {
11             if (x < a[mid])
12                 high = mid - 1;
13             else
14                 low = mid + 1;
15         }
16     }
17     return make_pair(false, -1);
18 }
19 // Recursive version
20 pair<bool, int> BinarySearchRec(MyType* a, int x, int low, int high) {
21     int mid;
22     if (low>high) // not found
23         return make_pair(false,-1);
24     else {
25         mid= (low + high) / 2;
26         if (x == a[mid])
27             return make_pair(true, mid);
28         else {
29             if (x < a[mid])
30                 return BinarySearchRec(a, x, low, mid - 1);
31             else
32                 return BinarySearchRec(a, x, mid + 1, high);
33         }
34     }
35 }
36
37 pair<bool, int> MyArray::BinarySearch2(MyType x) {
38     return BinarySearchRec(a, x, 0, n - 1);
39 }
```

4.1.6 Useful functions

Listing 4.6: Invert (MyArray.cpp).

```
1  void MyArray::Invert() {
2      MyType tmp;
3      for (int i = 0; i < n/2; i++) {
4          tmp = a[i];
5          a[i] = a[n - 1 - i];
6          a[n - 1 - i] = tmp;
7      }
8  }
```

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

```
1  MyArray* MyArray::FindOdd() {
2      int nodd = 0;
3      for (int i = 0; i < n; i++) {
4          if (((int)a[i] % 2) == 1)
5              nodd++;
6      }
7      nodd = 0;
8      MyArray* a1 = new MyArray(nodd);
9      for (int i = 0; i < n; i++) {
10         if (((int)a[i] % 2) == 1) {
11             a1->SetElement(nodd, a[i]);
12             nodd++;
13         }
14     }
15     return a1;
16 }
```

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

```
1  bool MyArray::Search(MyType x) {
2      for (int i = 0; i < n; i++) {
3          if (a[i] == x) {
4              return true;
5          }
6      }
7      return false;
8  }
```

4.1.7 Maximum and Argmax

Listing 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;
}
```

Listing 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

Listing 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;
}
```

Listing 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

Listing 4.13: Basic statistic (MyArray.cpp).

```
1  double MyArray::GetAverage() {
2      double result = 0;
3      if (n > 0) {
4          for (int i = 0; i < n; i++)
5              result += (double)a[i];
6          result /= n;
7      }
8      return result;
9  }
10
11 double MyArray::GetStandardDeviation() {
12     double result = 0;
13     if (n > 0) {
14         double mean = GetAverage();
15         for (int i = 0; i < n; i++) {
16             double tmp = (double)a[i] - mean;
17             result += tmp*tmp;
18         }
19         result /= n;
20         result = sqrt(result);
21     }
22     return result;
23 }
```

4.1.10 FindMaximumSubarray

Listing 4.14: FindMaxCrossingSubarray (MyArray.cpp).

```
tuple<int,int,int> MyArray::FindMaxCrossingSubarray(int low,int mid,int high) {
    int left_sum = -10000, right_sum= -10000; // -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);
        }
}
```

Listing 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;
}
```

Listing 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();
}
```

Listing 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);
}
```

Listing 4.18: 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);
    }
}
```

Listing 4.19: 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;
    }
}
```

Listing 4.20: Display steps (MyArray.cpp).

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

Listing 4.21: Swapping.

```
1  void MyArray::SwapIndex(int i, int j) {
2      MyType tmp = a[i];
3      a[i] = a[j];
4      a[j] = tmp;
5  }
6
7  void SwapIndex(MyType* a, int i, int j) {
8      MyType tmp = a[i];
9      a[i] = a[j];
10     a[j] = tmp;
11 }
```

4.2 Sorting algorithms

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

Listing 4.22: Selection sort (MyArray.cpp).

```
void MyArray::SelectionSort() {
    step = 0;
    // 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: $O(n^2)$

Listing 4.23: Insertion sort (MyArray.cpp).

```
void MyArray::InsertionSort() {
    step = 0;
    // Invariant: the array defined between position 0 and i is sorted
    bool done;
    for (int i = 0; i < n; ++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: $O(n^2)$

Listing 4.24: Bubble sort (MyArray.cpp).

```
1  void MyArray::BubbleSort() {
2      step = 0;
3      // Invariant: the whole array is sorted between n-1-i and n-1
4      for (int i = 0; i < n-1; ++i) {
5          for (int j = 0; j < n-i-1; ++j) {
6              step++;
7              if (a[j] > a[j + 1]) {
8                  SwapIndex(j, j + 1);
9              }
10         }
11     }
12     DisplayStep(__func__);
13 }
14
15 void MyArray::BubbleOptSort() {
16     step = 0;
17     // Invariant: the whole array is sorted between n-1-i and n-1
18     bool sorted=false;
19     int i = 0;
20     while ((i<n-1) && (!sorted)) {
21         sorted = true; // we assume the array is sorted between 0 and n-i-1
22         for (int j =0 ; j < n-i-1; j++) {
23             step++;
24             if (a[j]>a[j+1]) {
25                 SwapIndex(j,j+1);
26                 sorted = false;
27             }
28         }
29         i++;
30     }
31     DisplayStep(__func__);
32 }
```

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

Listing 4.25: Merge (MyArray.cpp).

```
1  void merge(MyType* a, int start, int mid, int end) {
2      // Init a1
3      int n1 = mid - start +1;
4      MyType* a1 = new MyType[n1];
5      for (int i = 0; i < n1; i++)
6          a1[i] = a[i+start];
7      // Init a2
8      int n2 = end - mid;
9      MyType* a2 = new MyType[n2];
10     for (int i = 0; i < n2; i++)
11         a2[i] = a[i+mid+1];
12     int i1 = 0, i2 = 0, i3 = start;
13     while ((i1 < n1) && (i2 < n2)) {
14         if (a1[i1] < a2[i2]) {
15             a[i3] = a1[i1];
16             i1++;
17         }
18         else {
19             a[i3] = a2[i2];
20             i2++;
21         }
22         i3++;
23         step++;
24     }
25     if (i1 < n1) // -> we left the while loop because i2>=n2
26         for (int i = i1; i < n1; i++) {
27             a[i3] = a1[i];
28             i3++;
29             step++;
30         }
31     else // -> we left the while loop because i1>=n1
32         for (int i = i2; i < n2; i++) {
33             a[i3] = a2[i];
34             i3++;
35             step++;
36         }
37     delete[] a1;
38     delete[] a2;
39 }
```

Listing 4.26: Mergesort (MyArray.cpp).

```
1  void mergesort(MyType* a, int start, int end) {
2      if (start < end) {
3          int mid = (start + end) / 2;
4          mergesort(a, start, mid);
5          // a is sorted between start and mid
6          mergesort(a, mid + 1, end);
7          // a is sorted between mid+1 and end
8          merge(a, start, mid, end);
9          // a is sorted
10     }
11 }
12
13 // 1st call of the function with
14 // start=0 and end=n-1
15 void MyArray::MergeSort() {
16     step = 0;
17     mergesort(a, 0, n - 1);
18     DisplayStep(__func__);
19 }
```

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

Listing 4.27: Quicksort (MyArray.cpp).

```
1  // Hoare partition
2  int partition(MyType* a, int start, int end) {
3      int i = start;
4      int j = end;
5      MyType pivot_value = a[(start+end)/2]; // in the middle
6      bool finished = false;
7      while (!finished) {
8          while ((i<end) && (a[i] <= pivot_value)) {
9              i++; // move to the right
10             step++;
11         }
12         while ((j>start) && (a[j] > pivot_value)) {
13             j--; // move to the left
14             step++;
15         }
16         if (i < j)
17             SwapIndex(a,i,j);
18         else
19             finished = true;
20     }
21     cout << "Pivot: " << pivot_value << " at position " << j << endl;
22     Display(a, start, end);
23     return j; // index of the pivot, which can move !
24 }
25
26 void quicksort(MyType* a, int start, int end) {
27     if (start < end) {
28         int pivot_index = partition(a, start, end);
29         // All the elements between start and pivot_index-1 are
30         // inferior to a[pivot_index].
31         // All the elements between pivot_index+1 and end are
32         // superior to a[pivot_index].
33         quicksort(a,start, pivot_index -1);
34         quicksort(a, pivot_index +1,end);
35     }
36 }
37
38 // 1st call of the function with
39 // start=0 and end=n-1
40 void MyArray::QuickSort() {
41     step = 0;
42     quicksort(a, 0, n - 1);
43     DisplayStep(__func__);
44 }
```

4.2.6 Some main examples

Listing 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;
}
```

Listing 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;
}
```

Listing 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();
}
```

Listing 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

Listing 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

Listing 5.1: Node simple chained list.

```
1  class NodeSC {
2  public:
3      NodeSC(): data(0), next(NULL)  {}
4      NodeSC(MyType d) : data(d), next(NULL) {}
5      NodeSC(MyType d, NodeSC* nxt) : data(d), next(nxt) {}
6      ~NodeSC() {}
7      MyType data;
8      NodeSC *next;
9  };
```

5.1.2 Class interface

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

```
1  class MySCList : public MyDataStructure {
2  public:
3      MySCList();
4      ~MySCList();
5      int GetSize() const { return n; }
6      MyDataStructure* clone() { return new MySCList(); }
7      void CreateNode(MyType value);
8      void Insert(MyType value);
9      void InsertFirst(MyType value);
10     void InsertLast(MyType value);
11     void InsertPosition(int pos, MyType value);
12     void DeleteFirst();
13     void DeleteLast();
14     void DeletePosition(int pos);
15     void DeleteMiddle(MyType value);
16     void InitRandom(int n, int v);
17     void InitSortedAscending(int n, int v);
18     void InitSortedDescending(int n, int v);
19     void Reverse();
20     void Display();
21     void DisplayFile();
22 private:
23     NodeSC *head; // pointer on the head
24     NodeSC *tail; // pointer on the tail
25     int n;
26 };
```

5.1.3 Constructor and destructor

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

```
1  MySCList::MySCList(): head(NULL), tail(NULL), n(0) { }
2
3  MySCList::~MySCList() {
4      NodeSC *current = head;
5      while (current) {
6          NodeSC *next = current->next;
7          delete current;
8          current = next;
9      }
10 }
```

Listing 5.4: Reverse (MySCList.cpp).

```
1  void MySCList::Reverse() {
2      NodeSC *current = head;
3      NodeSC *prev = NULL, *next = NULL;
4      while (current) {
5          next = current->next;
6          current->next = prev;
7          prev = current;
8          current = next;
9      }
10     head = prev;
11 }
```

5.1.4 Insert

Listing 5.5: Insert (MySCList.cpp).

```
1  void MySCList::CreateNode(MyType value) {
2      NodeSC *tmp = new NodeSC(value);
3      if (head==NULL) {
4          head = tmp;
5          tail = tmp;
6      } else {
7          tail->next = tmp;
8          tail = tmp;
9      }
10     n++;
11 }
12
13 void MySCList::InsertFirst(MyType value) {
14     NodeSC *tmp = new NodeSC(value,head);
15     head = tmp;
16     n++;
17 }
18
19 void MySCList::InsertLast(MyType value) {
20     CreateNode(value);
21 }
22
23 void MySCList::Insert(MyType value) {
24     InsertMiddle(value);
25 }
```

Listing 5.6: Insert (MySCList.cpp).

```
1  void MySCList::InsertMiddle(MyType value) {
2      NodeSC *tmp = new NodeSC(value);
3      NodeSC *cursor = head;
4      if (head == NULL) { // insert to the head
5          cout << "Insert to head/tail: " << value << endl;
6          head = tmp;
7          tail = tmp;
8      }
9      else if (head->data > value) { // insert to the head
10         cout << "Insert to head: " << value << endl;
11         tmp->next = head;
12         head = tmp;
13     }
14     else {
15         NodeSC *prev = head;
16         NodeSC *cursor = head->next;
17         while ((cursor != NULL) && (cursor->data < value)) {
18             prev = prev->next;
19             cursor = cursor->next;
20         }
21         if (cursor == NULL) { // insert to the tail
22             cout << "Insert to tail: " << value << endl;
23             prev->next = tmp;
24             tail = tmp;
25         }
26         else {
27             cout << "Insert middle: " << value << endl;
28             prev->next = tmp;
29             tmp->next = cursor;
30         }
31     }
32     n++;
33 }
```

Listing 5.7: Insert (MySCList.cpp).

```
1  void MySCList::InsertPosition(int pos, MyType value) {
2      if (pos == 0)
3          InsertFirst(value);
4      else if (pos == n - 1)
5          InsertLast(value);
6      else {
7          NodeSC *previous = NULL;
8          NodeSC *current = head;
9          NodeSC *tmp = new NodeSC(value);
10         for (int i = 0; i < pos; i++) {
11             previous = current;
12             current = current->next;
13         }
14         previous->next = tmp;
15         tmp->next = current;
16         n++;
17     }
18 }
```

Listing 5.8: Initialization (MySCList.cpp).

```
1  void MySCList::InitRandom(int n,int v) {
2      for (int i = 0; i < n; i++)
3          InsertLast(rand() % v);
4  }
5
6  void MySCList::InitSortedAscending(int n, int v) {
7      MyType tmp = rand() % v;
8      for (int i = 0; i < n; i++) {
9          MyType tmp1 = tmp + rand() % v;
10         InsertLast(tmp1);
11         tmp = tmp1;
12     }
13 }
14
15 void MySCList::InitSortedDescending(int n, int v) {
16     MyType tmp = 10e4+rand() % v;
17     for (int i = 0; i < n; i++) {
18         MyType tmp1 = tmp - rand() % v;
19         InsertLast(tmp1);
20         tmp = tmp1;
21     }
22 }
```

5.1.5 Delete

Listing 5.9: Delete (MySCList.cpp).

```
1  void MySCList::DeleteFirst() {
2      NodeSC *tmp;
3      tmp = head;
4      head = head->next;
5      delete tmp;
6      n--;
7  }
8
9  void MySCList::DeleteLast() {
10     NodeSC *current=NULL;
11     NodeSC *previous = NULL;
12     current = head;
13     while (current->next!=NULL) {
14         previous=current;
15         current=current->next;
16     }
17     tail=previous;
18     previous->next = NULL;
19     delete current;
20     n--;
21 }
22
23 void MySCList::DeletePosition(int pos) {
24     NodeSC *current=head;
25     NodeSC *previous=NULL;
26     for (int i = 1; i<pos; i++) {
27         previous = current;
28         current = current->next;
29     }
30     previous->next = current->next;
31     delete current;
32     n--;
33 }
```

Listing 5.10: Delete in a sorted list. (MySCList.cpp).

```
1  void MySCList::DeleteMiddle(MyType value) {
2      NodeSC* current = head;
3      if (current != NULL) {
4          if (current->data == value) {
5              head = current->next;
6              if (tail == current)
7                  tail = head;
8              delete current;
9          }
10         else {
11             if (current->next != NULL) {
12                 NodeSC* prev = current;
13                 current = current->next;
14                 bool found = false;
15                 while ((current != NULL) && (!found)) {
16                     if (current->data == value) {
17                         prev->next = current->next;
18                         delete current;
19                         found = true;
20                         if (tail == current)
21                             tail = prev;
22                     }
23                     else {
24                         prev = current;
25                         current = current->next;
26                     }
27                 }
28             }
29         }
30     }
31 }
```

5.1.6 Display

Listing 5.11: Display (MySCList.cpp).

```
1  void MySCList::Display() {
2      NodeSC *tmp;
3      tmp = head;
4      cout << "List of size :" << n << endl;
5      cout << "Head:";
6      (head != NULL) ? cout << head->data << endl : cout << "NULL" << endl;
7      cout << "Tail:";
8      (tail != NULL) ? cout << tail->data << endl : cout << "NULL" << endl;
9      int i = 0;
10     while (tmp) {
11         cout << "(" << i << ") ";
12         if (tmp->next != NULL)
13             cout << tmp->data << "->" << tmp->next->data << endl;
14         else
15             cout << tmp->data << "-> NULL" << endl;
16         tmp = tmp->next;
17         i++;
18     }
19 }
20
21 void MySCList::DisplayFile() {
22     ofstream myfile;
23     myfile.open("log.txt");
24     NodeSC *tmp;
25     tmp = head;
26     while (tmp) {
27         myfile << tmp->data << endl;
28         tmp = tmp->next;
29     }
30     myfile.close();
31 }
```

Listing 5.12: Example simple chained list.

```
1  void main() {
2      MySCList* L = new MySCList();
3      L->InitSortedAscending(12, 100);
4      L->Display();
5      cout << "Reverse the list" << endl;
6      L->Reverse();
7      L->Display();
8      delete L;
9      }
```

5.2 Double chained list

5.2.1 Node definition

Listing 5.13: Node simple chained list.

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

Listing 5.14: 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;
};
```

Listing 5.15: Constructor and destructor.

```
1  MyDCList::MyDCList() {
2      head = NULL;
3      tail = NULL;
4      n = 0;
5  }
6
7  MyDCList::~MyDCList() {
8      NodeDC *current = head;
9      while (current) {
10         NodeDC *next = current->next;
11         delete current;
12         current = next;
13     }
14 }
```

5.2.2 Insert

Listing 5.16: Insert tail.

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

Listing 5.17: Insert head.

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

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

Listing 5.18: 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;
```

```
 9        }
10        tmp->data = value;
11        pre->next = tmp;
12        tmp->next = cur;
13        n++;
14    }
```

Listing 5.19: Insert anywhere.

```
1  // Iterative version
2  void MyDCList::InsertMiddle(MyType value) {
3      NodeDC *tmp = new NodeDC(value);
4      NodeDC *cursor = head;
5      if (head == NULL) { // insert to the head
6          cout << "Insert to head/tail: " << value << endl;
7          head = tmp;
8          tail = tmp;
9      }
10     else if (head->data > value) { // insert to the head
11         cout << "Insert to head: " << value << endl;
12         head->previous = tmp;
13         tmp->next = head;
14         head = tmp;
15     }
16     else {
17         NodeDC *prev = head;
18         NodeDC *cursor = head->next;
19         while ((cursor != NULL) && (cursor->data < value)) {
20             prev = prev->next;
21             cursor = cursor->next;
22         }
23         if (cursor == NULL) { // insert to the tail
24             cout << "Insert to tail: " << value << endl;
25             prev->next = tmp;
26             tmp->previous = prev;
27             tail = tmp;
28         }
29         else {
30             cout << "Insert middle: " << value << endl;
31             prev->next = tmp;
32             tmp->previous = prev;
33             tmp->next = cursor;
34             cursor->previous = tmp;
35         }
36     }
37     n++;
38 }
```

Listing 5.20: Insert anywhere.

```
1  // Recursive version
2  void InsertMiddleDC(NodeDC **head, NodeDC *prev, NodeDC **tail, MyType value) {
3      if ((*head) == NULL) {
4          cout << "Insert to head/tail: " << value << endl;
5          NodeDC *tmp = new NodeDC(value);
6          *head = tmp;
7          *tail = tmp;
8      }
9      else if ((*head)->data > value) {
10         cout << "Insert to head: " << value << endl;
11         NodeDC *tmp = new NodeDC(value);
12         (*head)->previous = tmp;
13         tmp->next = (*head);
14         (*head) = tmp;
15         if (prev != NULL) {
16             cout << "Insert to middle: " << value << endl;
17             tmp->previous = prev;
18             prev->next = tmp;
19         }
20     }
21     else
22         InsertMiddleDC(&(*head)->next,(*head),tail, value);
23
24 }
25
26 void MyDCList::InsertMiddle1(MyType value) {
27     InsertMiddleDC(&head,NULL,&tail, value);
28     n++;
29 }
```

5.2.3 Delete

Listing 5.21: Delete head and tail.

```
1  void MyDCList::DeleteHead() {
2      NodeDC *tmp = new NodeDC;
3      tmp = head;
4      head = head->next;
5      head->previous = NULL;
6      delete tmp;
7      n--;
8  }
9
10 void MyDCList::DeleteTail() {
11     NodeDC *current=NULL;
12     NodeDC *previous = NULL;
13     current = head;
14     while (current->next != NULL) {
15         previous = current;
16         current = current->next;
17     }
18     tail = previous;
19     previous->next = NULL;
20     delete current;
21     n--;
22 }
```

Listing 5.22: Delete position.

```
1  void MyDCList::DeletePosition(int pos) {
2      if (pos == 0)
3          DeleteHead();
4      else if (pos==n-1)
5          DeleteTail();
6      else {
7          NodeDC *current = head;
8          NodeDC *previous = NULL;
9          for (int i = 1; i <= pos; i++) {
10             previous = current;
11             current = current->next;
12         }
13         previous->next = current->next;
14         current->next->previous = previous;
15         delete current;
16         n--;
17     }
18 }
```

Listing 5.23: Delete anywhere.

```
1  // Iterative version
2  void MyDCList::DeleteMiddle(MyType value) {
3      if (head->data == value) { // remove to the head
4          cout << "Remove to head: " << value << endl;
5          NodeDC *tmp = head;
6          head = head->next;
7          head->previous = NULL;
8          delete tmp;
9      }
10     else {
11         NodeDC *cursor = head;
12         NodeDC *prev = cursor->previous;
13         while ((cursor != NULL) && (cursor->data!=value)) {
14             prev = cursor;
15             cursor = cursor->next;
16         }
17         if (cursor!= NULL) { // remove
18             if (cursor->next == NULL) { // it is the tail
19                 cout << "Remove tail: " << value << endl;
20                 prev->next = NULL;
21                 tail = prev;
22             }
23             else
24             {
25                 cout << "Remove middle: " << value << endl;
26                 cursor->next->previous = prev;
27                 prev->next = cursor->next;
28             }
29             delete cursor;
30         }
31     }
32     n--;
33 }
```

Listing 5.24: Delete anywhere.

```
1  // Recursive version
2  void DeleteMiddleDC(NodeDC **head, NodeDC **tail, MyType value) {
3      if ((*head) != NULL) {
4          if ((*head)->data == value) { // remove to the head
5              NodeDC *tmp = *head;
6              *head = (*head)->next;
7              if ((*head)==NULL)
8                  *tail = tmp->previous;
9              else
10                 (*head)->previous = tmp->previous;
11             delete tmp;
12         }
13         else
14             DeleteMiddleDC(&(*head)->next, tail, value);
15     }
16 }
17
18 void MyDCList::DeleteMiddle1(MyType value) {
19     DeleteMiddleDC(&head, &tail, value);
20     n--;
21 }
```

5.2.4 Display

Listing 5.25: Display details.

```
1  void MyDCList::DisplayDC() {
2      NodeDC *tmp;
3      tmp = head;
4      MyType data_prev, data_next, data_head, data_tail;
5      cout << "List of size :" << n << endl;
6      cout << "Head:";
7      (head != NULL) ? cout << head->data << endl : cout << "NULL" << endl;
8      cout << "Tail:";
9      (tail != NULL) ? cout << tail->data << endl : cout << "NULL" << endl;
10     while (tmp != NULL) {
11         if (tmp->previous != NULL)
12             data_prev = tmp->previous->data;
13         else
14             data_prev = -1;
15         if (tmp->next != NULL)
16             data_next = tmp->next->data;
17         else
18             data_next = -1;
19         cout << "(" << data_prev << ","
20                     << tmp->data << ","
21                                  << data_next << ")" << endl;
22         tmp = tmp->next;
23     }
24 }
```

Listing 5.26: Display.

```
1  void MyDCList::Display() {
2      NodeDC *tmp = new NodeDC;
3      tmp = head;
4      while (tmp != NULL) {
5          cout << tmp->data << "\n";
6          tmp = tmp->next;
7      }
8  }
9
10 void MyDCList::DisplayFile() {
11     ofstream myfile;
12     myfile.open("log.txt");
13     NodeDC *tmp = new NodeDC;
14     tmp = head;
15     while (tmp != NULL) {
16         myfile << tmp->data << "\n";
17         tmp = tmp->next;
18     }
19     myfile.close();
20 }
```

Listing 5.27: Example.

```
1  void main() {
2      cout << "Test the Double Chained List" << endl;
3      MyDCList* LD = new MyDCList();
4      LD->DisplayDC();
5      LD->InsertMiddle(5);
6      LD->InsertMiddle(10);
7      LD->InsertMiddle(15);
8      LD->InsertMiddle(20);
9      LD->InsertMiddle(3);
10     LD->InsertMiddle(2);
11     LD->InsertMiddle(1);
12     LD->InsertMiddle(11);
13     LD->InsertMiddle(50);
14     LD->InsertMiddle(17);
15     LD->InsertMiddle(-50);
16     LD->DeleteMiddle(10);
17     LD->DeleteMiddle(50);
18     LD->DeleteMiddle(-50);
19     LD->DeleteMiddle(5);
20     LD->DeleteMiddle(20);
21     LD->DisplayDC();
22     delete LD;
23 }
```

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:

Listing 5.28: 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);
}
```

Listing 5.29: 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.

Listing 5.30: 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

Listing 5.31: Circular list - Class definition.

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

class MyCList : public MyDataStructure {
public:
    MyCList() : head(NULL) {}
    ~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;
};
```

Listing 5.32: Destructor.

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

5.3.2 Insert

Listing 5.33: Insert.

```
1  void MyCList::Insert(MyType value) {
2      Cnode *temp= new Cnode(value,NULL);
3      if (head == NULL) {
4          head = temp;
5          temp->next = head;
6      }
7      else {
8          temp->next = head->next;
9          head->next = temp;
10         head = temp;
11     }
12 }
13 // Insertion of element at beginning
14 void MyCList::InsertBegin(MyType value) {
15     if (head == NULL)
16         cout << "First Create the list." << endl;
17     else {
18         Cnode *temp = new Cnode(value, head->next);
19         head->next = temp;
20     }
21 }
22 // Insertion of element at a particular place
23 void MyCList::InsertAfter(MyType value, int position) {
24     if (head == NULL) {
25         cout << "First Create the list." << endl;
26         return;
27     }
28     Cnode *temp, *s=head->next;
29     for (int i = 0; i < position - 1; i++) {
30         s = s->next;
31         if (s == head->next) {
32             cout << "There are less than "
33                     << position << " in the list" << endl;
34             return;
35         }
36     }
37     temp = new Cnode(value, s->next);
38     s->next = temp;
39     // Element inserted at the end
40     if (s == head) {
41         head = temp;
42     }
43 }
```

5.3.3 Delete

Listing 5.34: 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 = NULL;
        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

Listing 5.35: 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 == NULL) {
        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;
    }
}
```

Listing 5.36: Update.

```
1  void MyCList::Update(MyType value, int position) {
2      if (head == NULL)
3          cout << "The list is empty." << endl;
4      else {
5          Cnode *s;
6          s = head->next;
7          int i = 0;
8          while ((i<position-1) && (s!=head)) {
9              s = s->next;
10             i++;
11         }
12         if (s != head)
13             s->data = value;
14     }
15 }
```

5.4 Skip list

5.4.1 Class definition

Listing 5.37: Skip list - Class definition.

```
1  class SkipNode {
2  public:
3      SkipNode() : next(NULL), data(0) {};
4      SkipNode(MyType key, int level);
5      // Array of pointers to nodes of different levels
6      SkipNode **next;
7      MyType data;
8      int level;
9  };
10
11 class MySkipList : public MyDataStructure {
12 public:
13     MySkipList();
14     MySkipList(int MAXLVL, float P);
15     ~MySkipList();
16     MyDataStructure* clone() { return new MySkipList(); }
17     void Insert(MyType x);
18     void Delete(MyType x);
19     bool Search(MyType x);
20     void Display();
21     void DisplayFile();
22 private:
23     int RandomLevel();
24     int MaxLvl; // Maximum level for this skip list
25                 // P is the fraction of the nodes with level
26                 // i pointers also having level i+1 pointers
27     float P;
28     // current level of skip list
29     int level;
30     SkipNode *head; // pointer to header node
31 };
```

5.4.2 Main functions

Listing 5.38: 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

Listing 5.39: Insert and delete.

```
1  void MySkipList::Insert(MyType x) {
2      SkipNode *current = head;
3      SkipNode **update=new SkipNode*[MaxLvl + 1];
4      memset(update, 0, sizeof(SkipNode*)*(MaxLvl + 1));
5      for (int i = level; i >= 0; i--) {
6          while (current->next[i] != NULL &&
7                  current->next[i]->data < x)
8              current = current->next[i];
9          update[i] = current;
10     }
11     current = current->next[0];
12     if (current == NULL || current->data != x) {
13         int rlevel = RandomLevel();
14         if (rlevel > level) {
15             for (int i = level + 1; i<rlevel + 1; i++)
16                 update[i] = head;
17             level = rlevel;
18         }
19         SkipNode* n = new SkipNode(x, rlevel);
20         for (int i = 0; i <= rlevel; i++) {
21             n->next[i] = update[i]->next[i];
22             update[i]->next[i] = n;
23         }
24     }
25 }
26 void MySkipList::Delete(MyType x) {
27     SkipNode *current = head;
28     SkipNode **update = new SkipNode*[MaxLvl + 1];
29     memset(update, 0, sizeof(SkipNode*)*(MaxLvl + 1));
30     for (int i = level; i >= 0; i--) {
31         while (current->next[i] != NULL &&
32             current->next[i]->data < x)
33             current = current->next[i];
34         update[i] = current;
35     }
36     if (current->next[0] != NULL) {
37         current = current->next[0];
38         if (current->data == x) {
39             for (int i = 0; i <= current->level; i++) {
40                 update[i]->next[i] = current->next[i];
41             }
42             delete current;
43         }
44     }
```

```
45  }
```

5.4.4 Search and display

Listing 5.40: Search and display.

```
1  bool MySkipList::Search(MyType x) {
2      SkipNode *current = head;
3      SkipNode **update = new SkipNode*[MaxLvl + 1];
4      memset(update, 0, sizeof(SkipNode*)*(MaxLvl + 1));
5      for (int i = level; i >= 0; i--) {
6          while (current->next[i] != NULL &&
7              current->next[i]->data < x)
8              current = current->next[i];
9          update[i] = current;
10     }
11     if (current->next[0] == NULL)
12         return false; // not found
13     if (current->next[0]->data == x)
14         return true; // found
15     return false; // not found
16 }
17
18 void MySkipList::Display() {
19     for (int i = 0; i <= level; i++) {
20         SkipNode *node = head->next[i];
21         cout << "Level: " << i << endl;
22         while (node != NULL) {
23             cout << node->data << " ";
24             cout << "(" << node->level << ") ";
25             node = node->next[i];
26         }
27         cout << endl;
28     }
29 }
30
31 void MySkipList::DisplayFile() {
32     ofstream myfile;
33     myfile.open("log_skiplist.txt");
34     for (int i = 0; i <= level; i++) {
35         SkipNode *node = head->next[i];
36         myfile << "Level: " << i << endl;
37         while (node != NULL) {
38             myfile << node->data << " ";
39             node = node->next[i];
40         }
41         myfile << endl;
42     }
43     myfile.close();
44 }
```

5.4.5 Example

Listing 5.41: Example.

```
1  void main() {
2      MySkipList* s = new MySkipList(2,0.5);
3      s->Insert(10);
4      s->Insert(20);
5      s->Insert(5);
6      s->Insert(7);
7      s->Insert(9);
8      s->Insert(8);
9      s->Insert(5);
10     s->Insert(15);
11     s->Insert(25);
12     s->Insert(16);
13     s->Insert(26);
14
15     cout << s->Search(25) << endl;
16     cout << s->Search(14) << endl;
17     cout << s->Search(26) << endl;
18     cout << s->Search(27) << endl;
19     cout << s->Search(3) << endl;
20     cout << s->Search(12) << endl;
21     cout << s->Search(5) << endl;
22     s->Display();
23
24     s->Delete(15);
25     s->Display();
26
27     delete s;
28 }
```

Chapter 6

Hash tables

Contents

6.1 Hash tables

6.1.1 Class definition

Listing 6.1: Hash Table - Class definition.

```
1  class MyHashTable : public MyDataStructure {
2  public:
3      MyHashTable();
4      MyHashTable(int n, int m, int type);
5      ~MyHashTable();
6      MyDataStructure* clone() { return new MyHashTable(); }
7      void Insert(MyType x);
8      void Delete(MyType x);
9      bool Search(MyType x);
10     pair<bool, int> SearchKey(MyType x);
11     void Display();
12     void DisplayFile();
13 private:
14     int HashFunction(MyType x);
15     int n; // size
16     int m; // modulus value
17     MyType*  ht; // hash table
18     bool* htd; // present or not
19     int type; // collision management: 0: linear, 1: quadratic probing
20 };
```

6.1.2 Main functions

Listing 6.2: Hash Table - functions.

```
1  MyHashTable::MyHashTable() : n(0), m(0), type(0) {}
2
3  MyHashTable::MyHashTable(int n1, int m1, int type1) {
4      n = n1;
5      m = m1;
6      type = type1;
7      ht = new MyType[n];
8      htd = new bool[n];
9      for (int i = 0; i < n; i++) {
10         ht[i] = -1;
11         htd[i] = false;
12     }
13 }
14
15 MyHashTable::~MyHashTable() {
16     delete[] ht;
17     delete[] htd;
18 }
19
20 void MyHashTable::Delete(MyType x) {
21     pair<bool, int> r = SearchKey(x);
22     if (r.second!=-1)
23         htd[r.second] = false;
24 }
25
26 int MyHashTable::HashFunction(MyType x) {
27     return (int)x % m;
28 }
```

6.1.3 Insert

Listing 6.3: Hash Table - Insert.

```
1  void MyHashTable::Insert(MyType x) {
2      int key = HashFunction(x);
3      if (htd[key]) { // collision
4              int probe = key + 1;
5              int step = 1;
6              int k = 1;
7              bool place = false;
8              while ((k < n) && (!place)) {
9                  if (!htd[probe]) {
10                     place = true;
11                     ht[probe] = x;
12                     htd[probe] = true;
13                 }
14                 else {
15                     step++;
16                     if (type == 0) // linear probing
17                         probe = key + step;
18                     else // quadratic probing
19                         probe = key + step*step;
20                     probe = probe % m;
21                 }
22                 k++;
23             }
24     }
25     else {
26         ht[key] = x;
27         htd[key] = true;
28     }
29
30 }
```

6.1.4 Search

Listing 6.4: Hash Table - Search.

```
1  bool MyHashTable::Search(MyType x) {
2      int key = HashFunction(x);
3      if (ht[key] == x) {
4          return key;
5      }
6      else {
7          int probe = key + 1;
8          int step = 1;
9          int k = 1;
10         bool place = false;
11         while ((k < n) && (!place)) {
12             if (ht[probe]==x) {
13                 return true;
14             }
15             else {
16                 step++;
17                 if (type == 0) // linear probing
18                     probe = key + step;
19                 else // quadratic probing
20                     probe = key + step*step;
21                 probe = probe % m;
22             }
23             k++;
24         }
25         return false;
26     }
27 }
```

Listing 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;
        bool place = false;
        while ((k < n) && (!place)) {
            if (ht[probe] == x) {
                return make_pair(true,probe);
            }
            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

Listing 6.6: Hash Table - Display.

```
1  void MyHashTable::Display() {
2      cout << "Hash table of size " << n << endl;
3      for (int i = 0; i < n; i++) {
4          cout << "Key: " << i << " with value: " << ht[i]
5              << "(" << htd[i] << ")" << endl;
6      }
7      cout << endl;
8  }
9
10 void MyHashTable::DisplayFile() {
11     ofstream myfile;
12     myfile.open("log_hashtable.txt");
13     myfile << "Hash table of size " << n << endl;
14     for (int i = 0; i < n; i++) {
15         myfile << "Key: " << i << " with value: " << ht[i]
16                << "(" << htd[i] << ")" << endl;
17     }
18     myfile.close();
19 }
```

Listing 6.7: Example.

```
1  void main() {
2      MyHashTable* H = new MyHashTable(10, 10, 1);
3      H->Insert(39);
4      H->Insert(13);
5      H->Insert(23);
6      H->Insert(63);
7      H->Insert(30);
8      H->Insert(31);
9      H->Insert(49);
10     H->Delete(49);
11     H->Insert(59);
12     H->Display();
13     cout << "Search 39: " << H->Search(39) << endl;
14     cout << "Search 49: " << H->Search(49) << endl;
15     cout << "Search 59: " << H->Search(59) << endl;
16     cout << "Search 23: " << H->Search(23) << endl;
17     delete H;
18 }
```

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 childrenandall leaves have the samedepthor samelevel. Hence, it is a full binary tree and all leaf nodes are at the same level.

7.1.2 TreeNode

Listing 7.1: TreeNode definition.

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

Listing 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);
TreeNode* InsertNode1(TreeNode* root, MyType data);
void InsertNode2(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);
```

```
33
34 // Comparisons
35 bool SameTree(TreeNode* t1, TreeNode* t2);
36 bool IsBST(TreeNode* node, int min, int max);
37 bool IsCompleteTree(TreeNode* root, int index, int nnodes);
38 bool IsFullTree(TreeNode*root);
39 bool IsPerfectTree(TreeNode* root);
```

Listing 7.3: Destroy the nodes in the tree.

```
1  void DestroyTree(TreeNode *root) {
2      if (root != NULL) {
3          DestroyTree(root->left);
4          DestroyTree(root->right);
5          delete root;
6      }
7  }
```

Listing 7.4: Count the nodes in the tree.

```
1  // Count the nodes in the binary tree to which root points.
2  int CountNodes(TreeNode* root) {
3      if (!root)
4          return 0;  // The tree is empty.
5      else {
6          int count = 1;   // Start by counting the root.
7          // Add the number of nodes in the left subtree
8          count += CountNodes(root->left);
9          // Add the number of nodes  in the right subtree
10         count += CountNodes(root->right);
11         return count;
12     }
13 }
```

7.1.3 Tree Traversal

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

```
1  void PrintNode(TreeNode* root) {
2      cout << root->data << " ";
3  }
4
5  void PreorderNode(TreeNode* root, void(*fct)(TreeNode* root)) {
6      if (root!=NULL) {
7          (*fct)(root);
8          PreorderNode(root->left,fct);
9          PreorderNode(root->right,fct);
10     }
11 }
12 void InorderNode(TreeNode* root, void(*fct)(TreeNode* root)) {
13     if (root != NULL) {
14         InorderNode(root->left, fct);
15         (*fct)(root);
16         InorderNode(root->right, fct);
17     }
18 }
19 void PostorderNode(TreeNode* root, void(*fct)(TreeNode* root)) {
20     if (root != NULL) {
21         PostorderNode(root->left, fct);
22         PostorderNode(root->right, fct);
23         (*fct)(root);
24     }
25 }
```

Listing 7.6: Tree traversal example.

```
1  void main() {
2    TreeNode* n = NULL;
3      InsertNode1(&n, 50);
4      InsertNode1(&n, 25);
5      InsertNode(&n, 75);
6      InsertNode(&n, 5);
7      InsertNode(&n, 15);
8      InsertNode(&n, 65);
9      InsertNode(&n, 85);
10     void(*fct)(TreeNode*)=PrintNode;
11     PreorderNode(n,fct);
12 }
```

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

```
1  void PrintPreorderNode(TreeNode* root,int lvl) {
2      if (root != NULL) {
3          cout << root->data << " (" << lvl << ")" << endl;
4          PrintPreorderNode(root->left, lvl + 1);
5          PrintPreorderNode(root->right, lvl + 1);
6      }
7  }
8
9  void PrintInorderNode(TreeNode* root,int lvl) {
10     if (root != NULL) {
11         PrintInorderNode(root->left, lvl + 1);
12         cout << root->data << " (" << lvl << ")" << endl;
13         PrintInorderNode(root->right, lvl + 1);
14     }
15 }
16
17 void PrintPostorderNode(TreeNode* root, int lvl) {
18     if (root != NULL) {
19         PrintPostorderNode(root->left, lvl + 1);
20         PrintPostorderNode(root->right, lvl + 1);
21         cout << root->data << " (" << lvl << ")" << endl;
22     }
23 }
```

Listing 7.8: Tree traversal: level order.

```
// Print nodes at a given level
void PrintGivenLevel(TreeNode* root, int level) {
    if (root != NULL) {
        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;
    }
}
```

Listing 7.9: Tree traversal: Nodes per level.

```
// Nodes per level
int GetNumberNodesGivenLevel(TreeNode* root, int level) {
    if (root != NULL) {
        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;
}
```

Listing 7.10: Invert Tree.

```
TreeNode* InvertTreeNode(TreeNode* root) {
    if (root==NULL) {
        return NULL; // 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

Listing 7.11: Search.

```
1  // Recursive version
2  bool SearchNode(TreeNode* root, MyType data) {
3      if (root == NULL)
4          return false;
5      else if (root->data == data)
6          return true;
7      else if (data <root->data)
8          SearchNode(root,data);
9      else
10         SearchNode(root, data);
11 }
12
13 // Iterative version
14 bool SearchNode1(TreeNode* root, MyType data) {
15     if (root == NULL)
16         return false;
17     else {
18         TreeNode* current = root;
19         bool found = false;
20         while ((!found) && (current != NULL)) {
21             if (current->data == data)
22                 found == true;
23             else if (data < current->data)
24                 current = current->left;
25             else
26                 current = current->right;
27         }
28         return found;
29     }
30 }
```

7.1.5 Insert

Listing 7.12: Insert (recursive version).

```
// Recursive version V1
// root as input/output
void InsertNode(TreeNode** root, MyType data) {
    if ((*root) == NULL) {
        (*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 only
TreeNode* InsertNode1(TreeNode* root, MyType data) {
    if (root == NULL) {
        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 = InsertNode1(root->left, data);
        return root;
    }
    else {
        root->right = InsertNode1(root->right, data);
        return root;
    }
}
```

Listing 7.13: Insert (iterative version).

```
1  void InsertNode2(TreeNode** root, MyType data) {
2      if ((*root) == NULL) {
3          (*root) = new TreeNode(data);
4      }
5      else {
6          TreeNode* current= (*root);
7          TreeNode* parent = NULL;
8          int direction = 0;
9          while (current != NULL) {
10             parent = current;
11             if (current->data == data) {
12                 cout << "Value already in the tree." << endl;
13                 current = NULL;
14             }
15             else if (data < current->data)
16                 current = current->left;
17             else
18                 current = current->right;
19         }
20         current= new TreeNode(data);
21         if (data < parent->data)
22             parent->left = current;
23         else if (data > parent->data)
24             parent->right = current;
25     }
26 }
```

7.1.6 Delete

Listing 7.14: Delete.

```
1  TreeNode* DeleteNode(TreeNode *root, MyType data) {
2      if (root == NULL)
3          return NULL;
4      if (data < root->data)  // Data is in the left sub tree.
5          root->left = DeleteNode(root->left, data);
6      else if (data > root->data) // Data is in the right sub tree.
7          root->right = DeleteNode(root->right, data);
8      else {
9          // case 1: no children
10         if (root->left == NULL && root->right == NULL) {
11             delete root;
12             root = NULL;
13         }
14         // case 2: 1 child (right)
15         else if (root->left == NULL) {
16             TreeNode *temp = root; // Save current node as a backup
17             root = root->right;
18             delete temp;
19         }
20         // case 3: 1 child (left)
21         else if (root->right == NULL) {
22             TreeNode *temp = root; // Save current node as a backup
23             root = root->left;
24             delete temp;
25         }
26         // case 4: 2 children
27         else {
28           // Find minimal value of right sub tree
29             TreeNode *temp = FindMinNode(root->right);
30             root->data = temp->data; // Duplicate the node
31             // Delete the duplicate node
32             root->right = DeleteNode(root->right, temp->data);
33         }
34     }
35     return root; // parent node can update reference
36 }
```

7.1.7 Min, Max

Listing 7.15: Find minimum and maximum values.

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

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

Listing 7.16: Find minimum and maximum nodes.

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

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

7.1.8 Max and Min Depth

Listing 7.17: MaxDepthTree.

```
int MaxDepthTree2(TreeNode *root) {
    if (root == NULL)
        return 0;
    else if ((root->left == NULL) && (root->right == NULL))
        return 0;
    else
        return 1 + max(MaxDepthTree2(root->left),
                       MaxDepthTree2(root->right));
}
int MaxDepthTree(TreeNode* root) {
    if (root == NULL) {
        return 0;
    }
    else if ((root->left == NULL) && (root->right == NULL))
        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;
    }
}
```

Listing 7.18: MinDepthTree.

```
int MinDepthTree(TreeNode *root) {
    if (root == NULL)
        return 0;
    // Base case : Leaf Node. This accounts for height = 1.
    if (root->left == NULL && root->right == NULL)
        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

Listing 7.19: Same tree?

```
bool SameTree(TreeNode* t1, TreeNode* t2) {
    if (t1 == NULL && t2 == NULL) // both empty
        return true;
    if (t1 != NULL && t2 != NULL) { // 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
}
```

Listing 7.20: 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)
                    );
}
```

Listing 7.21: 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 == NULL)
        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));
}
```

Listing 7.22: Is it a full tree?

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

Listing 7.23: Is it a perfect tree?

```
// True if the tree is perfect
bool IsPerfectTree(TreeNode* root) {
    if (root == NULL)
        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

Listing 7.24: BST interface.

```
1  class MyBST : public MyDataStructure {
2  public:
3      MyBST() { root = NULL; }
4      ~MyBST() { DestroyTree(root); }
5
6      MyDataStructure* clone() { return new MyBST(); }
7
8      void Insert(int data) { InsertNode(&root, data); }
9      void Insert1(int data) { root=InsertNode1(root, data); }
10     void Insert2(int data) { InsertNode2(&root, data); }
11
12     void Delete(int data) { root=DeleteNode(root, data); }
13
14     void Preorder(void(*fct)(TreeNode* root)) { PreorderNode(root,fct); }
15     void Inorder(void(*fct)(TreeNode* root)) { InorderNode(root,fct); }
16     void Postorder(void(*fct)(TreeNode* root)) { PostorderNode(root,fct); }
17
18     void PrintPreorder() { PrintPreorderNode(root,0); }
19     void PrintInorder() { PrintInorderNode(root,0); }
20     void PrintPostorder() { PrintPostorderNode(root,0); }
21     void PrintLevelorder() { PrintLevelOrder(root); }
22     void PrintGetNumberNodesLevel() { GetNumberNodesLevel(root); }
23
24     void InvertTree() { InvertTreeNode(root); }
25     int Height() { return MaxDepthTree(root);}
26     int Size() { return CountNodes(root); }
27
28     bool Search(MyType data) { return  SearchNode(root,data); }
29     bool IsComplete() { return IsCompleteTree(root,0, CountNodes(root)); }
30     bool IsFull() { return IsFullTree(root); }
31     bool IsPerfect() { return IsPerfectTree(root); }
32
33     int IsBSTv2() { return(IsBST(root, INT_MIN, INT_MAX)); }
34
35 private:
36     TreeNode* root; // pointer to the root
37 };
```

Listing 7.25: 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;
}
```

Listing 7.26: 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

Listing 7.27: 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(NULL), right(NULL) {}
};

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

Listing 7.28: Constructor and destructor.

```
1  void DestroyTree(AVLnode *root) {
2      if (root != NULL) {
3          DestroyTree(root->left);
4          DestroyTree(root->right);
5          delete root;
6      }
7  }
8  MyAVL::MyAVL() : root(NULL) {}
9  MyAVL::~MyAVL() { DestroyTree(root); }
```

Listing 7.29: Rebalance.

```
1  AVLnode* rebalance_node(AVLnode* n) {
2      setBalance(n);
3      if (n->balance == -2) {
4          if (height(n->left->left) >= height(n->left->right))
5              n = rotateRight(n);
6          else
7              n = rotateLeftThenRight(n);
8      }
9      else if (n->balance == 2) {
10         if (height(n->right->right) >= height(n->right->left))
11             n = rotateLeft(n);
12         else
13             n = rotateRightThenLeft(n);
14     }
15     return n;
16 }
17
18 void MyAVL::rebalance(AVLnode* n) {
19     if (n != NULL) {
20         n=rebalance_node(n);
21         if (n->parent != NULL)
22             rebalance(n->parent);
23         else
24             root = n; // very important to maintain the root
25     }
26 }
```

7.2.3 Rotations and balance

Listing 7.30: Rotate Left.

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

Listing 7.31: Rotate Right.

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

Listing 7.32: Rotate Left then Right.

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

Listing 7.33: Rotate Right then Left.

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

Listing 7.34: Height.

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

Listing 7.35: Set Balance.

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

Listing 7.36: Print balance.

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

7.2.4 Display

Listing 7.37: Display.

```
void PrintGivenLevel(AVLnode* root, int level) {
    if (root != NULL) {
        if (level == 0) {
            if (root->parent != NULL)
                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

Listing 7.38: Search.

```
bool SearchNode(AVLnode* root, MyType data) {
    if (root == NULL)
        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);
}
```

Listing 7.39: Insert.

```
void MyAVL::InsertNode(AVLnode** root, AVLnode* parent, MyType data) {
    if ((*root) == NULL) {
        (*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,NULL,x);
}
```

Listing 7.40: Find minimum.

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

Listing 7.41: Delete.

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

Listing 7.42: Delete.

```
AVLnode* MyAVL::DeleteNode(AVLnode *n, MyType data) {
    if (n == NULL)
        return NULL;
    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 == NULL && n->right == NULL) {
            delete n;
            n = NULL;
        }
        // case 2: 1 child (right)
        else if (n->left == NULL) {
            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 == NULL) {
            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 == NULL)
        return NULL;
    else
      return rebalance_node(n);
}

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

```
46 }
```

7.3 B-Trees

7.4 Red & Black Trees

Chapter 8

Heaps

Contents

8.1 Binary Heaps

A binary heap is a binary tree with two constraints:

- **Shape property**: A binary heap is acomplete 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 sometotal 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.

Listing 8.1: Class interface.

```
1  class MyMaxHeap {
2  public:
3      MyMaxHeap();
4      ~MyMaxHeap();
5      void InitArray(int n1);
6      void InitHeap(int n1);
7      bool IsMaxHeap();
8      bool IsSorted();
9      void MaxHeapify(int i, int size);
10     void BuildMaxHeap();
11     void HeapSort();
12     void InsertMaxHeap(int x);
13     void DeleteMaxHeap(int x);
14     void Display();
15 private:
16     MyType* a;
17     int n;
18     bool heap;
19 };
```

Listing 8.2: Heap constructor and destructor.

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

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

Listing 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;
}

```

```
33 // Determine if the elements in the array are sorted
34 bool MyMaxHeap::IsSorted() {
35     if (n <= 1)
36         return true;
37     else {
38         int i = 1;
39         while ((a[i - 1] <= a[i]) && (i < n))
40             i++;
41         return (i == n);
42     }
43 }
44
45 void MyMaxHeap::BuildMaxHeap() {
46     // Traverse the complete tree backward
47     // (right to left, bottom to top)
48     for (int i = n / 2 - 1; i >= 0; i--) {
49         MaxHeapify(i, n);
50     }
51     heap = true;
52 }
```

Listing 8.4: Init heap or array.

```
1  // Create a array of size n1 with random values
2  void MyMaxHeap::InitArray(int n1) {
3      if (n1 > 0) {
4          delete[] a;
5          n = n1;
6          a = new MyType[n];
7          for (int i = 0; i < n; i++)
8              a[i] = rand() % 200;
9      }
10     else
11         PositiveNumberCheck(__func__);
12 }
13
14
15 // Create a heap with n1 random elements
16 void MyMaxHeap::InitHeap(int n1) {
17     InitArray(n1);
18     BuildMaxHeap();
19 }
```

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.

Listing 8.5: Heapsort.

```
1  void MyMaxHeap::HeapSort() {
2      // First build the MaxHeap
3      BuildMaxHeap();
4      int i;
5      // Go backward as the Max is placed at the end
6      // Invariant: the elements between i to n-1 are at their correct places
7      // i is the size of the heap
8      for (i = n - 1; i>0; i--) {
9          // a[0] always contains the max value
10         swap(a[0], a[i]);
11         // the heap constraint must be maintained
12         MaxHeapify(0, i);
13     }
14     heap = false;
15 }
```

Listing 8.6: Heap insert, delete, display.

```
1  void MyMaxHeap::InsertMaxHeap(int x) {
2      MyType* a1 = new MyType[n + 1];
3      a1[0] = x;
4      for (int i = 0; i < n; i++) {
5          a1[1 + i] = a[i];
6      }
7      delete[] a;
8      a = a1;
9      MaxHeapify(0, n + 1);
10     n++;
11 }
12
13 void MyMaxHeap::DeleteMaxHeap(int x) {
14     int i = 0, indx = 0;
15     bool found = false;
16     while ((i < n) && (!found)) {
17         if (a[i] == x) {
18             found = true;
19             indx = i;
20         }
21         i++;
22     }
23     if (found) {
24         MyType* a1 = new MyType[n - 1];
25         a[indx] = a[n - 1];
26         for (int i = 0; i < n - 1; i++) {
27             a1[i] = a[i];
28         }
29         delete[] a;
30         a = a1;
31         MaxHeapify(indx, n - 1);
32         n--;
33     }
34 }
35
36 void MyMaxHeap::Display() {
37     for (int i = 0; i < n; i++) {
38         cout << a[i] << " ";
39     }
40     cout << endl;
41 }
```

Listing 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.

Listing 8.8: MyPriorityQueue class interface.

```
1  struct MyData {
2      int index;
3      double value;
4  };
5
6  class MyPriorityQueue
7  {
8  public:
9      MyPriorityQueue();
10     MyPriorityQueue(int c);
11     ~MyPriorityQueue();
12     int GetNElements() { return nelements; }
13     bool IsEmpty();
14     bool IsFull();
15     int GetParent(int child);
16     int GetLeftChild(int parent);
17     int GetRightChild(int parent);
18     void Push(int index, double value);
19     MyData Pop();
20     void DecreaseKey(int index, double value);
21     void BuildMinHeap();
22     void Display();
23 private:
24     void MinHeapify(int i, int size);
25     int capacity;
26     int nelements;
27     MyData* queue;
28 };
```

Listing 8.9: Constructors and destrucor.

```
1  MyPriorityQueue::MyPriorityQueue() {
2      capacity = 0;
3      nelements = 0;
4      queue = NULL;
5  }
6
7  MyPriorityQueue::MyPriorityQueue(int c) {
8      capacity = c;
9      nelements = 0;
10     queue = new MyData[capacity];
11 }
12
13 MyPriorityQueue::~MyPriorityQueue() {
14     delete[] queue;
15 }
```

Listing 8.10: Methods.

```
1  bool MyPriorityQueue::IsEmpty() {
2      return (nelements == 0);
3  }
4
5  bool MyPriorityQueue::IsFull() {
6      return (nelements == capacity);
7  }
8
9  int MyPriorityQueue::GetParent(int child) {
10     if (child % 2 == 0)
11         return (child / 2) - 1;
12     else
13         return child / 2;
14 }
15
16 int MyPriorityQueue::GetLeftChild(int parent) {
17     return (2 * parent + 1);
18 }
19
20 int MyPriorityQueue::GetRightChild(int parent) {
21     return (2 * parent + 2);
22 }
```

Listing 8.11: Heap methods.

```
1  void MyPriorityQueue::MinHeapify(int i, int size) {
2      int left, right, smallest;
3      left = 2 * i + 1;  // left child of a[i]
4      right = 2 * i + 2; // right child of a[i]
5      smallest = i; // original smallest element
6      if ((left<size) && queue[left].value<queue[i].value)
7          smallest = left;
8      if ((right<size) && (queue[right].value<queue[smallest].value))
9          smallest = right;
10     if (smallest != i) { // there was a swap
11         swap(queue[i], queue[smallest]);
12         MinHeapify(smallest, size);
13     }
14 }
15
16 void MyPriorityQueue::BuildMinHeap() {
17     // Traverse the complete tree backward
18     // (right to left, bottom to top)
19     for (int i = nelements / 2 - 1; i >= 0; i--) {
20         MinHeapify(i, nelements);
21     }
22 }
```

Listing 8.12: Push/Enqueue.

```
1  void MyPriorityQueue::Push(int index,double value) {
2      if (nelements < capacity) {
3          MyData x;
4          x.index = index;
5          x.value = value;
6
7          int i = nelements;
8          while ((i != 0) && (x.value < queue[i / 2].value)) {
9              queue[i] = queue[i / 2]; // move down
10             i /= 2;
11         } // move to parent
12         queue[i] = x;
13         nelements++;
14         cout << "Added (" << index
15                 << "," << value << ") size="
16                 << nelements << endl;
17     }
18     else
19         cout << "Out of capacity" << endl;
20 }
```

Listing 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;
    }
}
```

Listing 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;
}
```

Listing 8.15: Decrease key.

```
1  void MyPriorityQueue::DecreaseKey(int index, double value) {
2      int i = 0;
3      bool found = false;
4      while ((!found) && (i < nelements)) {
5          if (queue[i].index == index) {
6              queue[i].value = value;
7              found = true;
8          }
9          i++;
10     }
11     if (found) {
12         int child = i-1;
13         int parent = GetParent(child);
14         while ((queue[child].value < queue[parent].value) &&
15                (child >= 0 && parent >= 0)) {
16             swap(queue[child],queue[parent]);
17             child = parent;
18             parent = GetParent(child);
19         }
20     }
21 }
```

8.3 Fibonacci heaps

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:

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

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 v. Self loops are allowed.
- **Weighted**: Each edge has an associated weight, given by a weight function w : E R.
- **Mixed**: some edges may be directed and some may be undirected
- **Multigraph**: multiple edgesare 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 avertex 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 ofdistinctvertices. 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 thegraph.
- **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 samedegree.
- **Complete** graphs: Asimpleundirected graph, where every pair of distinctverticesis connected by a uniqueedge.
- **Finite** graphs: The vertex set and the edge set arefinite 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 agraphpossessing aHamiltonian cycle. A graph that is not Hamiltonian is said to benon-hamiltonian.
- Hamiltonian **cycle**: Hamiltonian path that is acycle.
- Hamiltonian **path**: Apathin an undirected or directed graph that visits eachvertexexactly 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.

9.2 Adjacency lists

Listing 9.1: Class interface.

```
class NodeGAL {
public:
    NodeGAL() :
        v(0), weight(0), next(NULL) {}
    NodeGAL(int v1,double weight1) :
        v(v1), weight(weight1), next(NULL) {}
    ~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 IsDirected();

    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
};
```

Listing 9.2: Constructor and Destructor.

```
1  MyGraphAL::MyGraphAL() {
2      int n = 0;
3      l = NULL;
4      current=NULL;
5      degree=NULL;
6  }
7
8  MyGraphAL::MyGraphAL(int n1) {
9      n = n1;
10     l = new NodeGAL*[n];
11     degree = new int[n];
12     for (int u = 0; u < n; u++) {
13         l[u] = NULL;
14         current[u]=NULL;
15         degree[u] = 0;
16     }
17 }
18
19 MyGraphAL::~MyGraphAL() {
20     for (int i = 0; i < n; i++) {
21         NodeGAL *current = l[i];
22         while (current) {
23             NodeGAL* next = current->next;
24             delete current;
25             current = next;
26         }
27     }
28     delete[] l;
29     delete[] current;
30     delete[] degree;
31 }
```

Listing 9.3: Exist and get edges.

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

Listing 9.4: Set edges.

```
1  void MyGraphAL::SetDirectedEdge(int u, int v, double w) {
2      if (!ExistEdge(u, v)) {
3          NodeGAL* tmp = new NodeGAL(v, w);
4          tmp->next = l[u];
5          l[u] = tmp;
6          degree[u]++;
7      }
8  }
9
10 void MyGraphAL::SetDirectedEdge(int u, int v) {
11     SetDirectedEdge(u, v, 1);
12 }
13
14 void MyGraphAL::SetUndirectedEdge(int u, int v) {
15     SetDirectedEdge(u, v, 1);
16     SetDirectedEdge(v, u, 1);
17 }
18
19 void MyGraphAL::SetUndirectedEdge(int u, int v, double w) {
20     SetDirectedEdge(u, v, w);
21     SetDirectedEdge(v, u, w);
22 }
```

Listing 9.5: Remove edges.

```
1  void MyGraphAL::RemoveDirectedEdge(int u, int v) {
2      NodeGAL* cursor = l[u];
3      if (cursor!=NULL) {
4          if (cursor->v == v) {
5              l[u] = cursor->next;
6              delete cursor;
7          }
8          else {
9              if (cursor->next != NULL) {
10                 NodeGAL* prev = cursor;
11                 cursor = cursor->next;
12                 bool found = false;
13                 while ((cursor != NULL) && (!found) ){
14                     if (cursor->v == v) {
15                         prev->next = cursor->next;
16                         delete cursor;
17                         found = true;
18                     }
19                     else {
20                         prev = cursor;
21                         cursor = cursor->next;
22                     }
23                 }
24             }
25         }
26     }
27     degree[u]--;
28 }
29
30 void MyGraphAL::RemoveUndirectedEdge(int u, int v) {
31     RemoveDirectedEdge(u,v);
32     RemoveDirectedEdge(v,u);
33 }
```

Listing 9.6: Comparisons.

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

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

Listing 9.7: Display.

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

    }
}
```

Listing 9.8: Iterator functions.

```
1  bool MyGraphAL::ExistAdjacent(int u) {
2      return current[u] != NULL;
3  }
4
5  void MyGraphAL::SetCurrentVertex(int u) {
6      current[u] = l[u];
7  }
8
9  int MyGraphAL::GetNextAdjacent(int u) {
10     int v = current[u]->v;
11     current[u] = current[u]->next;
12     return v;
13 }
```

9.2.1 Breadth First Search

Listing 9.9: BFS main function.

```
void BFS1(int s,MyGraphAL* G,int* &color,int* &distance,int* &pi) {
    int n = G->GetNumberVertices();
    if (!((s >= 0) && (s < n)))
        cout << "Bad source !" << endl;
    else {
        MyQueue* Q = new MyQueue(n);
        // initialization
        color = new int[n];
        distance = new int[n];
        pi = new int[n];
        for (int u = 0; u < n; u++) {
            color[u] = 0; // unvisited
            distance[u] = 0;
            pi[u] = 0;
        }
        // init s
        color[s] = 1;
        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] == 0) { // white (not visited before)
                        color[v] = 1; // visited
                        distance[v] = distance[u] + 1;
                        pi[v] = u;
                        Q->Enqueue(v);
                    }
                }
            }
            color[u] = 2; // black: finished
        }
        delete Q;
    }
}
```

Listing 9.10: Breadth First Search Display.

```
1  void MyGraphAL::BFS(int s) {
2      cout << "BFS" << endl;
3      int *color, *distance, *pi;
4      BFS1(s,this,color,distance,pi);
5      for (int u = 0; u < n; u++)
6          cout << "from " << s << " to " << u
7               << " Distance=" << distance[u]
8               << " Pi=" << pi[u] << endl;
9      delete[] color;
10     delete[] distance;
11     delete[] pi;
12 }
```

9.2.2 Depth First Search

Listing 9.11: DFS functions.

```
1  void DFSVisit(int u,MyGraphAL* G, int* color, int* discovery,
2                    int* finished, int* pi, int t) {
3      color[u] = 1; // visited;
4      t++;
5      discovery[u] = t;
6      G->SetCurrentVertex(u);
7      while (G->ExistAdjacent(u)) {
8          int v = G->GetNextAdjacent(u);
9          if (color[v] == 0) { // not already visited
10             pi[v] = u;
11             DFSVisit(v, G, color, discovery, finished, pi, t);
12         }
13     }
14     color[u] = 2; // black: finished
15     t++;
16     finished[u] = t;
17 }
18
19 void DFS1(MyGraphAL* G, int* &color, int* &discovery,
20                         int* &finished, int* &pi) {
21     int n = G->GetNumberVertices();
22     color = new int[n];
23     discovery = new int[n];
24     finished = new int[n];
25     pi = new int[n];
26     for (int u = 0; u < n; u++) {
27         color[u] = 0; // unvisited
28         discovery[u] = 0;
29         finished[u] = 0;
30         pi[u] = 0;
31     }
32     int t = 0;
33     for (int u = 0; u < n; u++) {
34         if (color[u] == 0) // not visited already?
35             DFSVisit(u, G, color, discovery,finished, pi,t);
36     }
37 }
```

Listing 9.12: Depth First Search Display.

```
void MyGraphAL::DFS() {
    cout << "DFS" << endl;
    int *color, *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;
}
```

Listing 9.13: 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.

Listing 9.14: Display path information.

```
1 void DisplayShortestPath(double* d, int* pi, int n) {
2     for (int i = 0; i < n; i++) {
3         cout << "Vertex:" << i
4             << " : d=" << d[i]
5             << " : pi:" << pi[i]
6             << endl;
7     }
8     cout << endl;
9 }
```

9.3.1 Bellman Ford

Listing 9.15: 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;
}
```

Listing 9.16: 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

Listing 9.17: Dijkstra.

```
1  void MyGraphAL::Dijkstra(int s,double* &d, int* &pi) {
2      // Initializa Single Source
3      d = new double[n];
4      pi = new int[n];
5      MyPriorityQueue* PQ = new MyPriorityQueue(n);
6      MyQueue* Q = new MyQueue(n);
7      for (int u = 0; u < n; u++) {
8          d[u] = DBL_MAX;
9          pi[u] = -1;
10     }
11     d[s] = 0;
12     for (int u = 0; u < n ; u++)
13         PQ->Push(u, d[u]);
14     PQ->Display();
15     while (!Q->IsFull()) {
16         MyData out = PQ->Pop();
17         PQ->Display();
18         Q->Enqueue(out.index);
19         int u = out.index;
20         cout << "Use: " << out.index << " with d:" << out.value << endl;
21         SetCurrentVertex(u);
22         while (ExistAdjacent(u)) {
23             int v = GetNextAdjacent(u);
24             // Relaxation
25             double w = GetEdgeWeight(u, v);
26             if (d[v] > d[u] + w) {
27                 d[v] = d[u] + w;
28                 pi[v] = u;
29                 PQ->DecreaseKey(v, d[v]);
30             }
31         }
32         DisplayShortestPath(d, pi, n);
33         PQ->Display();
34     }
35     delete PQ;
36     delete Q;
37 }
```

Listing 9.18: 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

Listing 9.19: Class interface.

```
1  class MyGraphAM {
2  public:
3      MyGraphAM();
4      MyGraphAM(int n);
5      ~MyGraphAM();
6      int GetNumberVertices() { return n; }
7      int GetDegree(int u);
8      int GetIndex(int u,int v);
9      bool ExistEdge(int u, int v);
10     void SetDirectedEdge(int u, int v, double w);
11     void SetDirectedEdge(int u, int v);
12     void SetUndirectedEdge(int u, int v, double w);
13     void SetUndirectedEdge(int u, int v);
14     void RemoveDirctedEdge(int u, int v);
15     void RemoveUndirectedEdge(int u, int v);
16     bool HasSelfLoops();
17     bool IsDirected();
18     void Display();
19     void DisplayDirectedEdge();
20     void DisplayUndirectedEdge();
21     void SetCurrentVertex(int u);
22     bool GetNextAdjacent(int u, int &vout);
23     void BFS(int s);
24     void DFS();
25     void PrintAllPairs(MyMatrix* pi, int i, int j);
26     void FloydWarshall(MyMatrix* &dout, MyMatrix* &piout);
27     void TransitiveClosure(MyMatrix* &TCout);
28     void MSTKruskal();
29     void MSTPrim();
30     int minKey(int* key, bool* mstSet);
31     void DisplayMST(int* parent);
32 private:
33     int n; // number of vertices
34     double* M;
35     int* current;
36 };
```

Listing 9.20: Constructors and Destructor.

```
1  MyGraphAM::MyGraphAM() {
2      n = 0;
3      M=NULL;
4      current = NULL;
5  }
6
7  MyGraphAM::MyGraphAM(int n1) {
8      n = n1;
9      M = new double[n*n];
10     for (int i = 0; i < n*n; i++)
11         M[i] = 0;
12     current = new int[n];
13     for (int i = 0; i < n; i++)
14         current[i] = 0;
15 }
16
17 MyGraphAM::~MyGraphAM() {
18     delete[] M;
19     delete[] current;
20 }
```

Listing 9.21: 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

Listing 9.22: Comparisons.

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

bool MyGraphAM::IsDirected() {
    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 true;
}
```

9.4.2 Display

Listing 9.23: Display.

```
1  void MyGraphAM::Display() {
2      int k = 0;
3      cout << " :";
4      for (int v = 0; v < n; v++)
5          cout << v << "\t";
6      cout << endl;
7      for (int u = 0; u < n; u++) {
8          cout << u << ":";
9          for (int v = 0; v < n; v++) {
10             cout << M[k] << "\t";
11             k++;
12         }
13         cout << endl;
14     }
15 }
16
17 void MyGraphAM::DisplayDirectedEdge() {
18     int k = 0;
19     cout << "List of edges :" << endl;
20     for (int u = 0; u < n; u++) {
21         for (int v = 0; v < n; v++) {
22             if (ExistEdge(u, v))
23                 cout << "(" << u << ","
24                 << v << ") w:" << GetEdgeWeight(u, v)
25                 << endl;
26         }
27     }
28 }
29
30 void MyGraphAM::DisplayUndirectedEdge() {
31     int k = 0;
32     cout << "List of edges :" << endl;
33     for (int u = 0; u < n; u++) {
34         for (int v = u+1; v < n; v++) {
35             if (ExistEdge(u, v))
36                 cout << "(" << u << ","
37                 << v << ") w:" << GetEdgeWeight(u, v)
38                 << endl;
39         }
40     }
41 }
```

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.

Listing 9.24: Adjacent iterator.

```
1  void MyGraphAM::SetCurrentVertex(int u) {
2      current[u] = -1;
3  }
4
5  bool MyGraphAM::GetNextAdjacent(int u,int &vout) {
6      int v= current[u] + 1;
7      vout = -1;
8      bool found = false;
9      while ((!found) && (v < n)) {
10         if (ExistEdge(u, v)) {
11             found = true;
12             vout=v;
13         }
14         v++;
15     }
16     current[u] = vout;
17     return found;
18 }
19
20 void DFSVisit(int u, MyGraphAM* G, int* color, int* discovery,
21                     int* finished, int* pi, int t) {
22     color[u] = 1; // visited;
23     t++;
24     discovery[u] = t;
25     G->SetCurrentVertex(u);
26     int v;
27     while (G->GetNextAdjacent(u,v)) {
28         if (v!=-1)
29         if (color[v] == 0) { // not already visited
30             pi[v] = u;
31             DFSVisit(v, G, color, discovery, finished, pi, t);
32         }
33     }
34     color[u] = 2; // black: finished
35     t++;
36     finished[u] = t;
37 }
```

Listing 9.25: 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 anoptimizationtechnique used primarily to speed upcomputer programs. It is used by storing the results of expensivefunction calls, and by returning the cached result when the same inputs occur again. It ensures that amethoddoesn'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 forDPproblems.

9.5.2 Back to Fibonacci

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

Listing 9.26: 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);
    }
}
```

Listing 9.27: 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];
    }
}
```

Listing 9.28: 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.

Listing 9.29: Print all pairs.

```
1  void MyGraphAM::PrintAllPairs(MyMatrix* pi,int i, int j) {
2      if (i == j)
3          cout << i ;
4      else if (pi->GetCell(i, j) == -1)
5          cout << "No path from " << i << " to " << j << endl;
6      else {
7          PrintAllPairs(pi, i, pi->GetCell(i, j));
8          cout << "," << j;
9      }
10 }
```

Listing 9.30: 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;
}
```

Listing 9.31: 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;
}
```

Listing 9.32: 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

Listing 9.33: 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;
}
```

Listing 9.34: 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;
}
```

Listing 9.35: 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;
}
```

Listing 9.36: 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

Listing 9.37: 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

Listing 9.38: 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;
}
```

Listing 9.39: Example MST.

```
1  void main() {
2    int n = 9;
3      MyGraphAM* G = new MyGraphAM(n);
4      G->SetUndirectedEdge(0, 1, 4);
5      G->SetUndirectedEdge(1, 2, 8);
6      G->SetUndirectedEdge(2, 3, 7);
7      G->SetUndirectedEdge(3, 4, 9);
8      G->SetUndirectedEdge(4, 5, 10);
9      G->SetUndirectedEdge(5, 6, 2);
10     G->SetUndirectedEdge(6, 7, 1);
11     G->SetUndirectedEdge(7, 0, 8);
12     G->SetUndirectedEdge(7, 8, 7);
13     G->SetUndirectedEdge(8, 2, 2);
14     G->SetUndirectedEdge(8, 6, 6);
15     G->SetUndirectedEdge(2, 5, 4);
16     G->SetUndirectedEdge(3, 5, 14);
17     cout << "Kruskal" << endl;
18     G->MSTKruskal();
19     cout << "Prim" << endl;
20     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.

Listing 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

Listing 10.2: Constructors and destructor

```
1  MyMatrix::MyMatrix() {
2      n = 0;
3      m = 0;
4      A = NULL;
5  }
6  MyMatrix::MyMatrix(int n1, int m1) {
7      n = n1; m = m1;
8      A = new double*[n];
9      for (int i = 0; i < n; i++) {
10         A[i] = new double[m];
11         for (int j = 0; j < m; j++)
12             A[i][j] = 0;
13     }
14 }
15 MyMatrix::MyMatrix(const MyMatrix& M) {
16     n = M.n; m = M.m;
17     A = new double*[n];
18     for (int i = 0; i < n; i++) {
19         A[i] = new double[m];
20         for (int j = 0; j < m; j++)
21             A[i][j] = M.A[i][j];
22     }
23 }
24 void MyMatrix::Clean() {
25     for (int i = 0; i < n; i++)
26         delete[] A[i];
27     delete[] A;
28 }
29 MyMatrix::~MyMatrix() {
30     Clean();
31 }
```

Listing 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;
    }
}
```

Listing 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

Listing 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;
}
```

Listing 10.6: Identity

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

Listing 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];
    }
}
```

Listing 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;
}
```

Listing 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;
}
```

Listing 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.

Listing 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

Listing 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 << ")";
    }
}
```

Listing 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);
}
```

Listing 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;
                }
            }
        }
    }
}
```

Listing 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;
}
```

Listing 10.16: Example 2

```
void main() {
    MyMatrix* A0=new MyMatrix(30,35);
    MyMatrix* A1=new MyMatrix(35,15);
    MyMatrix* A2=new MyMatrix(15,5);
    MyMatrix* A3=new MyMatrix(5,10);
    MyMatrix* A4=new MyMatrix(10,20);
    MyMatrix* A5=new MyMatrix(20,25);
    MyMatrix* sequence[] = { A0, A1, A2, A3, A4, A5 };
    MatrixChainOrderMult(sequence, 6);
}
```