Data Structures & Algorithms for Students in Computer Science

Hubert Cecotti

Spring 2019 (v0.005)

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. 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++.
You should keep this document to get prepared for job interviews.

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...
- $\forall$ For all...
- $\in$ belongs (example: $x \in X$, x belongs to X).
- ∞ infinity
- $\emptyset$ empty set

2 Definitions

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

$$\Theta(g(n)) = \{f(n) : \exists\{c_1, c_2, n_0\} | 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0\} \tag{1}$$

- O: Big O, asymptotic upper bound.

$$\text{Ø}(g(n)) = \{f(n) : \exists\{c, n_0\} | 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\} \tag{2}$$

- Ω: Big omega, asymptotic lower bound

$$\Omega(g(n)) = \{f(n) : \exists\{c, n_0\} | 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\} \tag{3}$$

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

$$\text{ø}(g(n)) = \{f(n) : \forall c > 0, \exists n_0 | 0 \leq f(n) < c \cdot g(n) \forall n \geq n_0\} \tag{4}$$

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

$$\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 | 0 \leq c \cdot g(n) < f(n) \forall n \geq n_0\} \tag{5}$$

Table 1: Definitions summary.

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

2.1.1 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: $n^{c+1} > n^c$, $n > log(n)$, $n \cdot log(n) > log(n) \cdot log(n)$.

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

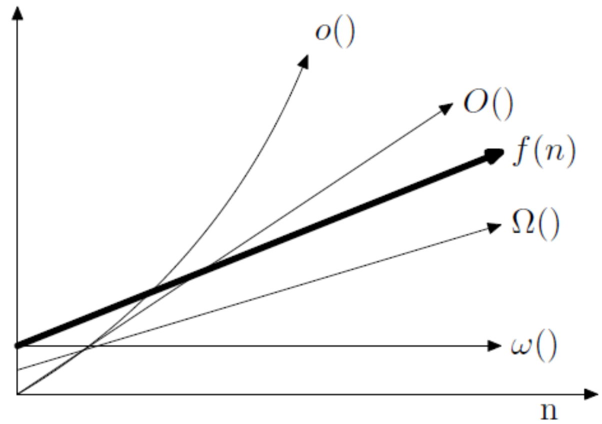

Figure 1: Relationships between the notations.

2.2 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.

2.3 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.

2.4 Master theorem

$$T(n) \quad = \quad aT(n/b) + f(n) \tag{6}$$

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:

$$c_{crit} \quad = \quad log_b(a) \tag{7}$$

2.4.1 Case 1

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

2.4.2 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 $T(n) = \Omega(n^{c_{crit}} \cdot log^{k+1}(n))$

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

2.4.4 Examples

Table 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	$\Theta(n \cdot log(n))$
$3T(n/3)+n/2$	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	$\Theta(n^3)$
$16T(n/4)+n!$	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

2.5 Time complexity

Table 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 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 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 6: Heap (average case)

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

3 C++ examples

3.1 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: Includes.

```cpp
void main() {
  const int i = 12;
    int j,l = 12;
    // i = 13; // cannot change i because it was defined as a const
    j = i; // j can change its value as it s not a const

    // a literal such as 56 is a prvalue
    // 56 = i; // it does not work
    j = 56; // it works

    // left side is an expression BUT it returns an lvalue ( j or l)
    ((j < 30) ? j : l) = 7;

    // It does not work as it  returns an evaluated expression j*2 or l*2
    // ((j < 30) ? j*2 : l*2) = 7;
}
```

3.2 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 2: Includes.

```
using namespace std;
#include <chrono>     // to use functions related to the time
#include <string>     // to use strings
#include <tuple>      // to use tuple
#include <iostream>   // input/output
#include <fstream>    // files
```

3.3 Types

Listing 3: Type conversion and promotion.

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

3.4 Functions

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 intput and the function returns a as an output.

Listing 4: C++ examples.

```cpp
void f1(int* a,int n) {
    a = new int[n];
    for (int i = 0; i < n; i++)
        a[i] = i;
}
void f2(int** a, int n) {
    *a = new int[n];
    for (int i = 0; i < n; i++)
        (*a)[i] = i;
}
int* f3(int n) {
    int *a = new int[n];
    for (int i = 0; i < n; i++)
        a[i] = i;
    return a;
}
void DisplayArray(int* a, int n) {
    for (int i = 0; i < n; i++)
        cout << a[i] << ",";
    cout << endl;
}
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);
}
```

Listing 5: C++ examples.

```cpp
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) {
    (*x)++;
}

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

3.5 Sum and product

Listing 6: Variables declaration.

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

Sum 1:

$$x = \sum_{i=1}^{n} i \tag{8}$$

Sum 2:

$$x = \sum_{i=1}^{n} 1/i \tag{9}$$

Sum 3:

$$x = \sum_{i=0}^{n-1} a(i) \cdot b(i) \tag{10}$$

Product 1:

$$x = \prod_{i=1}^{n} i \tag{11}$$

Product 1:

$$x = \prod_{i=0}^{n-1} a(i) \cdot b(i) \tag{12}$$

Listing 7: 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 8: C++ examples.

```cpp
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;
}
```

3.6 No break, no continue!

This examples illustrates different version ways to find an element in an array, with a for loop using a break, a while loop, and a do while loop.

Listing 9: For loop with break.

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

Listing 10: While loop.

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

Listing 11: Do While loop.

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

}
```

Listing 12: Main.

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

4 Recursivity

4.1 Factorial

Definition:

$$Factorial(x) = x \cdot Factorial(x-1) \tag{13}$$

Listing 13: Factorial functions.

```
int Factorial1(int x) {
    int result = 1;
    for (int i = 2; i <= x; i++)
        result *= i;
    return result;
}

int Factorial2(int x) {
    int result = 1;
    int i = 2;
    while (i <= x) {
        result *= i;
        i++;
    }
    return result;
}

int Factorial3(int x) {
    if (x <= 1)
        return 1;
    else
        return x*Factorial3(x - 1);
}
```

Listing 14: Factorial examples.

```
void main() {
    cout << Factorial1(6) << endl; // 720
    cout << Factorial2(6) << endl; // 720
    cout << Factorial3(6) << endl; // 720
}
```

4.2 Fibonacci

Definition:

$$Fibonacci(0) = 0 \tag{14}$$

$$Fibonacci(1) = 1 \tag{15}$$

$$Fibonacci(x) = Fibonacci(x-1) + Fibonacci(x-2) \tag{16}$$

Listing 15: Fibonacci function.

```
int Fibonacci(int x) {
    if (x == 0)
        return 0;
    else if (x == 1)
        return 1;
    else
        return Fibonacci(x - 1) + Fibonacci(x - 2);
}
```

Listing 16: Fibonacci example.

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

4.3 Anagram

Listing 17: Rotate.

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

Listing 18: Do anagram.

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

Listing 19: Anagram example.

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

4.4 Hanoi Tower

Listing 20: 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 21: 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.
}
```

5 Stacks

Listing 22: Stack - class definition

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

Listing 23: Stack - functions

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

Listing 24: Example

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

6 Queues

Listing 25: Circular queue - class definition.

```
typedef double MyType;
class MyQueue {
public:
    MyQueue();
    MyQueue(int capacity1);
    ~MyQueue();
    bool isFull();
    bool isEmpty();
    void Enqueue(MyType x);
    MyType Dequeue();
    MyType Front();
    MyType Rear();
    void Display();
public:
    int front, rear, size;
    int capacity;
    MyType* q;
};
```

Listing 26: Circular queue - functions.

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

```
}
void MyQueue::Display() {
    cout << "Max capacity: " << capacity << endl;
    cout << "Size: " << size << endl;
    for (int i = 0; i < size; i++)
        cout << "Element: " << q[i] << " at position " << i << endl;
    cout << endl;
}
```

Listing 27: Example

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

7 Prototype pattern

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.

7.1 Main program

Listing 28: 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;
}
```

7.2 Parent class

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

Listing 29: 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 30: MyDataStructure.cpp

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

7.3 Factory class

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

Listing 31: 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 32: 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;
    }
};
```

8 Array

8.1 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 33: MyArray.h

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

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

```
    // Sorting functions
    bool IsSorted();
    void SwapIndex(int i, int j);
    void DisplayStep(string f);
    // Init functions
    void InitRandom(int v);
    void InitSortedAscending(int v);
    void InitSortedDescending(int v);
    // Sorting algorithms
    void SelectionSort();
    void InsertionSort();
    void BubbleSort();
    void BubbleOptSort();
    void MergeSort();
    void QuickSort();
private:
    MyType* a; // array
    int n; // size of the array
};
```

8.2 Constructors and destructor

Listing 34: Constructors and destructor (MyArray.cpp)

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

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

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

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

8.3 Access elements

Listing 35: Access elements (MyArray.cpp).

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

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

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

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

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

8.4 Find, Delete, Insert

// Determine if the value x is in the array

Listing 36: Find, Delete, Insert (MyArray.cpp).

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

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

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

8.5 Binary Search

Listing 37: Binary search (MyArray.cpp).

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

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

8.6 Useful functions

Listing 38: Invert (MyArray.cpp).

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

Listing 39: Array of odd numbers (MyArray.cpp).

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

Listing 40: Search and return index (MyArray.cpp).

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

8.7 Maximum and Argmax

Listing 41: 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 42: 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);
}
```

8.8 Minimum and Argmin

Listing 43: 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 44: 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);
}
```

8.9 Mean and Standard Deviation

Listing 45: Basic statistic (MyArray.cpp).

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

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

8.10 FindMaximumSubarray

Listing 46: 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 47: 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 48: 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 49: 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 50: 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-1));
    }
}
```

Listing 51: 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 52: 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 53: Swapping.

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

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

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

Listing 54: 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
}
```

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

Listing 55: 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__);
}
```

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

Listing 56: Bubble sort (MyArray.cpp).

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

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

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

Listing 57: Merge (MyArray.cpp).

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

Listing 58: Mergesort (MyArray.cpp).

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

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

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

Listing 59: Quicksort (MyArray.cpp).

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

void quicksort(MyType* a, int start, int end) {
    if (start < end) {
        int pivot_index = partition(a, start, end);
        // All the elements between start and pivot_index-1 are
        // inferior to a[pivot_index].
        // All the elements between pivot_index+1 and end are
        // superior to a[pivot_index].
        quicksort(a,start, pivot_index -1);
        quicksort(a, pivot_index +1,end);
    }
}

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

8.16 Some main examples

Listing 60: 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 61: 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 62: Application of Quicksort.

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

Listing 63: 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;
}
```

8.17 Time measurement

Listing 64: 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;
    }
}
```

9 Simple chained list

9.1 Node definition

Listing 65: Node simple chained list.

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

9.2 Class interface

Listing 66: Class simple chained list (MySCList.h).

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

9.3 Constructor and destructor

Listing 67: Class simple chained list (MySCList.cpp).

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

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

Listing 68: Creation and insertion (MySCList.cpp).

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

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

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

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

Listing 69: Insert (MySCList.cpp).

```
void MySCList::InsertPosition(int pos, MyType value) {
    NodeSC *pre=NULL;
```

```
    NodeSC *cur = NULL;
    NodeSC *tmp = NULL;
    cur = head;
    for (int i=1;i<pos;i++) {
        pre = cur;
        cur = cur->next;
    }
    tmp->data = value;
    pre->next = tmp;
    tmp->next = cur;
    n++;
}
```

Listing 70: Initialization (MySCList.cpp).

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

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

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

Listing 71: Delete (MySCList.cpp).

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

```cpp

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

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

Listing 72: Reverse (MySCList.cpp).

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

Listing 73: Display (MySCList.cpp).

```cpp
void MySCList::Display() {
    NodeSC *tmp;
    tmp = head;
    while (tmp) {
```

```
            cout << tmp->data << endl;
            tmp = tmp->next;
        }
}

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

Listing 74: Example simple chained list.

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

10 Double chained list

10.1 Node definition

Listing 75: Node simple chained list.

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

Listing 76: 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:
    DCnode *head, *tail;
    DCnode *iterator_start, *iterator_end;
    int n;
};
```

Listing 77: Constructor and destructor.

```
MyDCList::MyDCList() {
    head = NULL;
    tail = NULL;
    n = 0;
}
MyDCList::~MyDCList() {
    DCnode *cursor = head;
    DCnode *cursor1 = cursor;
    for (int i = 0; i < n; i++) {
        cursor1 = cursor;
        delete cursor1;
        cursor = cursor->next;
    }
}
```

Listing 78: Insert tail.

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

Listing 79: Insert head.

```
void MyDCList::InsertHead(MyType value) {
    DCnode *tmp = new DCnode(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 80: Insert at position.

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

Listing 81: Insert anywhere.

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

Listing 82: Delete anywhere.

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

Listing 83: Delete head and tail.

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

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

Listing 84: Delete position.

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

Listing 85: Display details.

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

Listing 86: Display.

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

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

Listing 87: Example.

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

10.2 The Sieve of Eratosthenes

10.2.1 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 88: 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 89: 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;
```

10.2.2 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 90: 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;
}
```

11 Circular list

Listing 91: 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 92: Destructor.

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

Listing 93: Insert.

```
void MyCList::Insert(MyType value) {
    Cnode *temp= new Cnode(value,NULL);
    if (head == NULL) {
        head = temp;
        temp->next = head;
    }
    else {
        temp->next = head->next;
        head->next = temp;
        head = temp;
    }
}

// Insertion of element at beginning
void MyCList::InsertBegin(MyType value) {
    if (head == NULL)
        cout << "First Create the list." << endl;
    else {
        Cnode *temp = new Cnode(value, head->next);
        head->next = temp;
    }
}

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

```
}
```

Listing 94: 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;
        }
    }
}
```

Listing 95: 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 96: Update.

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

12 Skip list

Listing 97: Skip list - Class definition.

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

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

Listing 98: 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;
}
```

Listing 99: Insert and delete.

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

Listing 100: Search and display.

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

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

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

Listing 101: Example.

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

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

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

    delete s;
}
```

13 Hash tables

Listing 102: Hash Table - Class definition.

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

Listing 103: Hash Table - functions.

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

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

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

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

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

Listing 104: Hash Table - Insert.

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

}
```

Listing 105: Hash Table - Search.

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

Listing 106: 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);
    }
}
```

Listing 107: Hash Table - Display.

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

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

Listing 108: Example.

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

14 Binary Search Trees

14.1 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.

14.2 TreeNode

Listing 109: 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 110: TreeNode functions.

```
void PrintNode(TreeNode* root);
int CountNodes(TreeNode* root);
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);
bool SearchNode(TreeNode* root, MyType data);
bool SearchNode1(TreeNode* root, MyType data);
void InsertNode(TreeNode** root, MyType data);
void InsertNode1(TreeNode** root, MyType data);
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);
bool SameTree(TreeNode* t1, TreeNode* t2);
bool IsBST(TreeNode* node, int min, int max);
bool IsCompleteTree(TreeNode* root, int index, int nnodes);
bool IsFullTree(TreeNode*root);
bool IsPerfectTree(TreeNode* root);
```

Listing 111: Destroy the nodes in the tree.

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

Listing 112: Count the nodes in the tree.

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

14.3 Tree Traversal

Listing 113: Tree traversal: Pre-In-Post.

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

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

Listing 114: Tree traversal example.

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

Listing 115: Tree traversal: Pre-In-Post.

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

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

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

Listing 116: 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 117: 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 118: 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;
}
```

Listing 119: Search.

```
// Recursive version
bool SearchNode(TreeNode* 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);
}

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

Listing 120: Insert.

```
// Recursive version
void InsertNode(TreeNode** root, MyType data) {
    TreeNode* T = new TreeNode(data);
    if ((*root) == NULL) {
        (*root)=T;
    } 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);
}

// Iterative version
void InsertNode1(TreeNode** root, MyType data) {
    TreeNode* T = new TreeNode(data);
    if ((*root) == NULL) {
        (*root) = T;
    }
    else {
        TreeNode* current= (*root);
        while (current != NULL) {
            if (current->data == data) {
                cout << "Value already in the tree." << endl;
                current = NULL;
            }
            else if (data < current->data)
                current = current->left;
            else
                current = current->right;
        }
    }
}
```

Listing 121: Delete.

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

Listing 122: 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 123: 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;
}
```

Listing 124: MaxDepthTree.

```c
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 125: MinDepthTree.

```c
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;
}
```

14.4 Comparisons

Listing 126: 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 127: 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 128: 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 129: 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 130: 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);
    }
}
```

14.5 MyBST

Listing 131: BST interface.

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

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

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

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

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

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

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

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

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

Listing 132: 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 133: 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;
}
```

15 AVL Trees

16 B-Trees

17 Red & Black Trees

18 Heaps

19 Fibonacci heaps

19.1 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.

19.2 Lemma 2

With the sequence of Fibonacci, we have:

$$F_0 = 0 \tag{17}$$
$$F_1 = 1 \tag{18}$$
$$F_k = F_{k-1} + F_{k-2} \forall k \geq 2 \tag{19}$$

We want to prove that:

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \forall k \geq 0 \tag{20}$$

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

$$F_{0+2} = F_2 = F_1 + F_0 = 1 + 0 = 1 \tag{21}$$
$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 + 1 \tag{22}$$

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.

$$
\begin{aligned}
F_{(k+1)+2} &= F_{k+2} + F_{k+1} && (23)\\
&= 1 + \sum_{i=0}^{k} F_i + F_{k+1} && (24)\\
&= 1 + \sum_{i=0}^{k+1} F_i && (25)
\end{aligned}
$$

We can start the other way:

$$
\begin{aligned}
1 + \sum_{i=0}^{k+1} F_i &= 1 + \sum_{i=0}^{k} F_i + F_{k+1} && (26)\\
&= F_{k+2} + F_{k+1} && (27)\\
&= F_{(k+1)+2} && (28)
\end{aligned}
$$

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

19.3 Lemma 3

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

$$
x^2 - x - 1 = 0 \qquad (29)
$$

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 $\Phi^2 = \Phi + 1$.

$$
\begin{aligned}
F_{k+2} &= F_{k+1} + F_k && (30)\\
&\geq \Phi^{k-1} + \Phi^{k-2} && (31)\\
&= \Phi^{k-2} * (\Phi + 1) && (32)\\
&= \Phi^{k-2} * (\Phi^2) && (33)\\
&= \Phi^k && (34)
\end{aligned}
$$

20 Graphs