

# Rapport devoir 1

8INF840 - Structures de données avancées et leurs algorithmes  
Antoine de Saint-Martin, Pierre Honoré

## Exercice 1:

### Classes:

Pour mener à bien son fonctionnement, à partir de l'énoncé, nous avons dégagé la nécessité de créer plusieurs classes distinctes.

La classe Carte :

```
class Carte
{
public:
    //le numero allant de
    int numero;
    //la couleur étant re
    bool couleur;
    //le bonus est la pu
    int bonus;
```

La classe carte comporte trois attributs, un numéro, une couleur et un bonus, comme énoncé dans le sujet.

La classe PileDM :

```
template <typename T>
class PileDM
{
public:
    // constructeurs et destructeurs

    //constructeur
    PileDM(int max = MAX_PILE) throw(std::bad_alloc)
    {
        tab = new T[max];
        sommet = -1; //valeur fausse pour montrer que l
        tailleMax = max;
        taille = 0;
    }

    //constructeur copie
    PileDM(const PileDM<T>& p)
    {
        tab = new T[p.tailleMax];
        tailleMax = p.tailleMax;
        for (int i = 0; i < tailleMax; i++)
        {
            tab[i] = p.tab[i];
        }
        sommet = p.sommet;
    }
}
```

La classe générique PileDM est complètement disponible dans le PDF de notre cours.

La classe CompteurPoints :

```
class CompteurPoints
{
public :
    //gains du joueur 1.
    float gainsJ1;
    //gains du joueur 2
    float gainsJ2;
```

La classe compteur de points va récupérer dans son constructeur les paquets (Piles de cartes) de gains de chaque joueur et va procéder à l'addition de l'ensemble des scores de chaque carte. Pour chaque paquet, gainsJ1 et gainsJ2 sont attribués, nous pourrions déterminer le vainqueur.

La classe TourDeJeu :

```
class TourDeJeu
{
public :
    //carte courante du joueur 1
    carte::Carte carteJoueur1;
    //carte courante du joueur 2
    carte::Carte carteJoueur2;
    //si gagnantTour = 1, J1 sinon
    int gagnantTour;
```

TourDeJeu va nous permettre de confronter deux cartes, nous aurons donc autant de TourDeJeu que de cartes distribuées à chacun des joueurs. En fonction de la couleur, du bonus et du numéro de la carte, TourDeJeu détermine la carte gagnante et nous pourrions ajouter cette carte au paquet de gains du joueur respectif.

La classe GameMaster :

```
public :
    //Constructeur
    GameMaster(int nombreCarteVoulu, std::string nomJoueur1, std::string nomJoueur2) : nombreCarteTirage(nombreCarteVoulu)
    {
        paquetGlobal = pile::PileDM<carte::Carte>(100);
        joueurs = std::vector<joueur::Joueur>(2);
        generationPaquet();
        creationJoueur(nomJoueur1, nomJoueur2);
        tirageCartes(nombreCarteVoulu);
        executionPartie();
        determinerGagnant();
    }
```

GameMaster est la classe gérant l'ensemble des autres éléments, initialiser les paquets de chaque joueur, créer nos joueurs, exécuter l'ensemble de la partie et enfin déterminer le gagnant. GameMaster est instancié dans le main associé.

## Initialisation:

```
void demandeRejouer()
{
    std::string autrePartie = GameMaster::demandeAutrePartie();
    if (autrePartie == "oui" || autrePartie == "Oui")
    {
        return;
    }
    if (autrePartie == "non" || autrePartie == "Non")
    {
        std::exit(1);
    }
}

void deroulementPartie()
{
    std::vector<std::string> noms = GameMaster::demandeNomsJoueurs();
    int nombreCarte = GameMaster::demanderNombreCarteTirage();
    GameMaster GM = GameMaster::GameMaster(nombreCarte, noms[0], noms[1]);
    demandeRejouer();
}

//-----

//-----

int
main()
{
    while (1)
    {
        deroulementPartie();
    }
    return 0;
}
```

Vu que nous avons la possibilité de recommencer une partie, notre programme tourne dans une boucle infinie.

## Exécution:

```
Veillez inserer le nom du Joueur 1 :
Antoine
Joueur 1 > Antoine

Veillez inserer le nom du Joueur 2 :
Pierre
Joueur 2 > Pierre

Veillez donner un nombre de cartes par Joueur (2-50) :
38

Pierre a gagne la partie avec 408.5 points contre 389 points pour Antoine
Voulez vous refaire une partie ? Oui ou Non
Non

Sortie de C:\Users\anto\Documents\UQAC\Algorithmie\Devoir1\devoir1_algortihmie\Devoir1\Debug\Devoir1.exe (processus 32
84). Code : 1.
Pour fermer automatiquement la console quand le débogage s'arrête, activez Outils->Options->Débogage->Fermer automatique
ment la console à l'arrêt du débogage.
Appuyez sur une touche pour fermer cette fenêtre. . .
```

## Exercice 2:

### Classes:

Machine:

```
typedef File<double> flux;
class Machine
{
public:
    Machine(double tempsTraitement, int ID);
    double panne();
    double traitement();

protected:
    int m_ID;
    double m_tempsTraitement;
};
```

La classe Machine sert à instancier nos différentes machines.

Elle a un attribut qui indique le temps qu'elle prend à traiter une pièce (m\_tempsTraitement). Une méthode permettant de simuler une panne (panne()), elle retourne le temps que rajoute la panne si elle a bien lieu (une chance sur quatre).

Une méthode traitement, appelant la méthode panne et rajoutant au possible temps de panne le temps de traitement afin d'accéder à l'information via un appel externe à la classe.

File:

La classe File est détaillée dans le cours, nous ne revenons pas dessus.

### Initialisation:

À l'aide des deux classes définies précédemment, nous allons simuler la chaîne d'opérations. Tous d'abord, on instancie et initialise les différents objets permettant de simuler la chaîne.

```
typedef File<double> flux;
```

```
int NB_PISTON = 100;
int NB_PIECE = 500;

flux s(NB_PIECE);
flux st(NB_PIECE);
flux sj(NB_PIECE);
flux sa(NB_PIECE);
flux tp(NB_PIECE);
flux jp(NB_PIECE);
flux ap(NB_PIECE);
flux p(NB_PIECE);
```

On crée des flux, permettant de simuler les chaînes reliant la machine. Cela correspond à une file de double. Le double représentant les pièces et transportant l'information du timestamp de quand la pièce a été traitée la dernière fois.

Il y a 8 flux:

- 1 pour l'entrée de la machine de tri.

- 3 pour les sorties de la machine de tri, les reliant aux 3 machines de d'usinage.
- 3 pour les sorties des machines d'usinage, elles sont reliées à la machine d'assemblage.
- 1 pour la sorte de la machine d'assemblage.

Ils sont initialisés à une taille de 500 pièces afin d'avoir un flux assez grand pour contenir 500 pièces ou plus.

Les machines sont instanciées, avec une durée de traitement.

Des timescodes sont initialisés à zéro pour chaque machine pour calculer le temps de production.

## Exécution:

```
for (int i = 0; i < NB_PIECE; i++)  
{  
    s.enfiler(0.0);  
}
```

On commence l'exécution en créant notre boîte de pièces à l'entrée de la chaîne. Il y a 500 pièces dont le timecode est initialisé à 0, car la production n'a pas commencé.

```

// La machine de tri traite une piece
if (s.taille() != 0)
{
    tcs += ms.traitemement();

    mt19937_64 gen{ random_device() };
    uniform_real_distribution<float> dis{ 0.0, 3.0 };
    for (int i = 0; i < NB_PIECE; i++)
    {
        s.defiler();
        float nb = dis(gen);
        if (nb < 1)
            st.enfiler(tcs);
        else if (nb >= 2)
            sj.enfiler(tcs);
        else
            sa.enfiler(tcs);
    }
}

```

On distribue avec la machine de tri aux machines d'usinage. Il y a une chance sur trois que la pièce est, soit une jupe, soit un axe, soit une tête. Elle est ensuite enlevée du flux d'entrée de la chaîne et distribuée respectivement aux flux d'entrées des machines d'usinages.

```

// La machine d'usinage de tete traite une piece
if (st.taille() > 0)
{
    double pi = st.defiler();
    if (pi > tct)
        tct = pi + mp.traitemement();
    else
        tct += mp.traitemement();
    tp.enfiler(tct);
}

```

Pour chaque machine d'usinage, la pièce est enlevée du flux d'entrée, traitée, potentiellement en panne. Le timecode de la pièce est modifiée en conséquence et l'attente si la machine traite déjà une pièce.

```

// La machine d'assemblage traite une piece
if ((tp.taille() > 0) && (jp.taille() > 0) && (ap.taille() > 0))
{
    double pit = tp.defiler();
    double pij = jp.defiler();
    double pia = ap.defiler();

    double pi;
    if (pit > pij)
        pi = pit;
    else
        pi = pij;
    if (pia > pi)
        pi = pia;

    if (pi > tcp)
        tcp = pi + mp.traitemement();
    else
        tcp += mp.traitemement();

    p.enfiler(tcp);
}

```

Enfin, dès que la machine d'assemblage reçoit trois pièces différentes en entrée. Elle enlève les pièces des flux d'entrées, traite les pièces, et dépose les pistons en sortie.

```

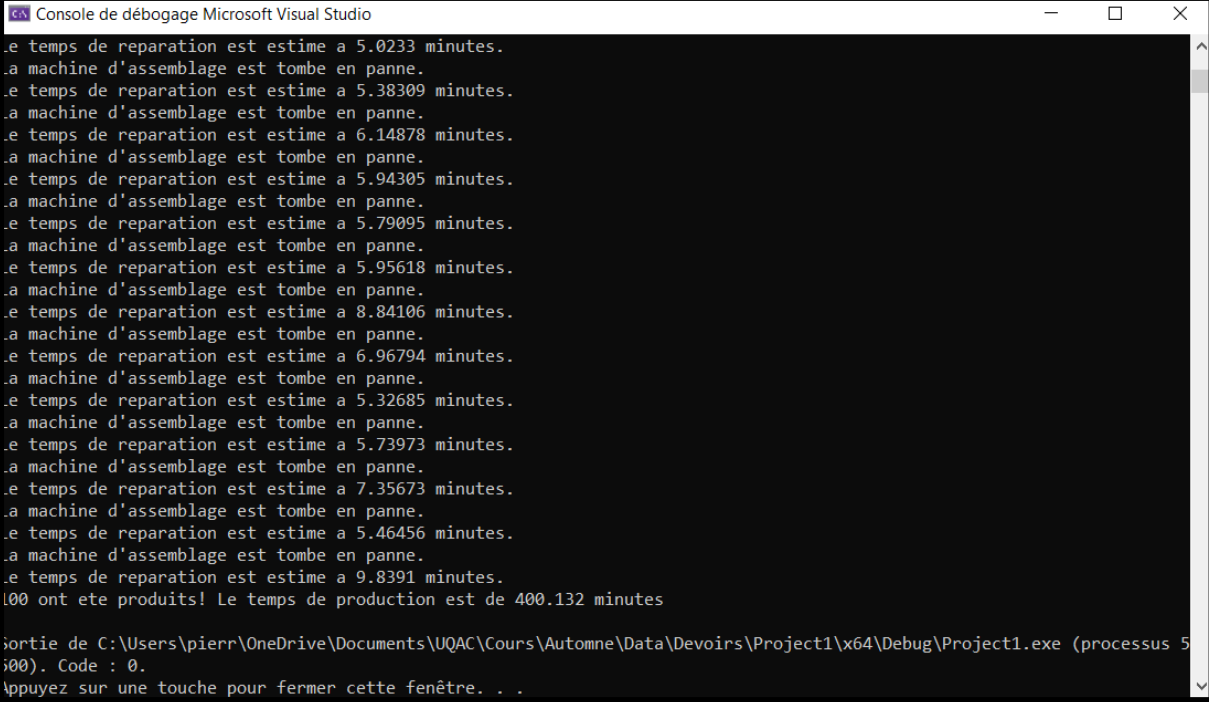
// La machine d'assemblage a traite le nombre de pistons demandes
if (p.taille() == NB_PISTON)
{
    std::cout << p.taille() << " ont ete produits! Le temps de production est de " << p.dernier() << "\n";
    exit(EXIT_SUCCESS);
}

```

Dès que la consigne de pistons est atteinte, le programme s'arrête.

## Résultat:

Lors de l'exécution, le terminal annonce chaque panne pour chaque machine et quand les 100 pistons sont produits. Les autres informations ne sont pas affichées, car ce sont les réactions normales de la chaîne.



```
Console de débogage Microsoft Visual Studio
Le temps de reparation est estime a 5.0233 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 5.38309 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 6.14878 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 5.94305 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 5.79095 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 5.95618 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 8.84106 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 6.96794 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 5.32685 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 5.73973 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 7.35673 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 5.46456 minutes.
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 9.8391 minutes.
100 ont ete produits! Le temps de production est de 400.132 minutes

Sortie de C:\Users\pierr\OneDrive\Documents\UQAC\Cours\Automne\Data\Devoirs\Project1\x64\Debug\Project1.exe (processus 5100). Code : 0.
Appuyez sur une touche pour fermer cette fenêtre. . .
La machine d'assemblage est tombe en panne.
Le temps de reparation est estime a 5.32133 minutes.
100 ont ete produits! Le temps de production est de 351.655 minutes
100 ont ete produits! Le temps de production est de 356.399 minutes
```

Plusieurs simulations permettent d'observer que le temps de production de 100 pistons est globalement compris entre 300 et 400 minutes.

Ce qui est logique, parce que la machine la plus longue à produire ses pièces est la machine de jupes qui prend 3 minutes. Pour 100 pièces à produire, nécessaire à la production de 100 pistons, elle va prendre minimum 300 minutes (3 min x 100 pièces). À cela s'ajoute des minutes supplémentaires de panne. Les autres machines ainsi que leurs temps de traitement et de panne peuvent être ignorés pour l'estimation théorique du temps de production, car elles sont négligeables par rapport à la machine de jupes (vu qu'elles produisent leurs pièces en parallèles).

Un temps de production compris entre 300 minutes et 400 minutes et donc cohérent si on prend en compte les pannes.

## Exercice 3:

### Classes:

La classe dictionnaire :

```
class Dictionnaire
{
public :
    //-----

    Dictionnaire()
    {
        //Notre racine possède le caractère "0"
        char c = '0';
        arbre = new arbreBinaire::ArbreBinaire<char>(c);
    }
}
```

Notre classe dictionnaire initialise notre arbre binaire de type char. Chaque noeud aura une donnée T de type char. Nous créons la racine dans le constructeur et lui affectons la valeur arbitraire 0.

La classe Noeud :

```
template <typename T>
class Noeud {
public:
    T data; //donnée portée par notre noeud
    Noeud* gauche; //pointe vers le fils de gauche
    Noeud* droite; //pointe vers le fils de droite
    int hauteur; //hauteur du noeud dans l'arbre
}
```

La classe noeud permet de contenir les données nécessaires au bon fonctionnement de notre arbre pour chaque point d'intersection. Nous avons un arbre binaire donc un noeud possède au maximum deux enfants, un gauche et un droit.

La classe ArbreBinaire :

```
template <typename T>
class ArbreBinaire
{
public :
    //-----

    //Constructeur par défaut
    ArbreBinaire(T data)
    {
        const T & _data = data;
        racine = new Noeud<T>(_data, 0);
        cpt = 0;
    }
}
```

La classe arbre binaire comporte l'ensemble des méthodes permettant l'utilisation de la structure. Les ajouts, suppressions, recherche et parcours. Le compteur cpt stocke le nombre total de noeuds présents dans notre arbre. La racine est initialisée dès la création de l'arbre.



## Initialisation:

```
int
main()
{
    Dictionnaire* dict = new Dictionnaire();
    std::cout << "\nOn ajoute les mots abas, arbre, arbuste, bas puis afficher le dictionnaire :\n\n";
    dict->AjouterMot("abas");
    dict->AjouterMot("arbre");
    dict->AjouterMot("arbuste");
    dict->AjouterMot("bas");
    dict->AfficherDictionnaire();
    std::cout << "\nOn recherche les mots abas, pol, navire et arbre :\n\n";
    dict->ChercherMot("abas");
    dict->ChercherMot("pol");
    dict->ChercherMot("navire");
    dict->ChercherMot("arbre");
    std::cout << "\nEnlever le mot abas et afficher le dictionnaire :\n\n";
    dict->EnleverMot("abas");
    dict->AfficherDictionnaire();
    std::cout << "\n";
    std::cout << "\nEnlever le mot voiture alors qu'il n'y est pas, l'ajouter et afficher le dictionnaire :\n\n";
    dict->EnleverMot("voiture");
    dict->AjouterMot("voiture");
    dict->AfficherDictionnaire();
    std::cout << "\nEssayer d'enlever le mot a nouveau, afficher le dictionnaire :\n\n";
    dict->EnleverMot("voiture");
    dict->AfficherDictionnaire();

    return 0;
}
```

Voici le contenu de notre main servant à tester notre structure de dictionnaire via l'utilisation d'un arbre binaire. Les opérations sont testées et chacun des modules met en avant une fonctionnalité testée.

## Exécution:

```
On ajoute les mots abas, arbre, arbuste, bas puis afficher le dictionnaire :  
  
abas  
arbre  
arbuste  
bas  
  
On recherche les mots abas, pol, navire et arbre :  
  
le mot est dans le dictionnaire  
le mot n'est pas dans le dictionnaire  
le mot n'est pas dans le dictionnaire  
le mot est dans le dictionnaire  
  
Enlever le mot abas et afficher le dictionnaire :  
  
arbre  
arbuste  
bas  
  
Enlever le mot voiture alors qu'il n'y est pas, l'ajouter et afficher le dictionnaire :  
  
le mot n'est pas dans le dictionnaire  
arbre  
arbuste  
bas  
voiture  
  
Essayer d'enlever le mot a nouveau, afficher le dictionnaire :  
  
arbre  
arbuste  
bas
```

## Exercice 4:

### Classes:

La classe personne :

```
class Personne {  
    public :  
  
        std::string nom;  
  
        std::string prenom;  
  
        int anneeNaissance;  
  
        std::string couleurYeux;
```

La classe personne définit le type de donnée qui sera stockée dans notre arbre généalogique.

La classe NoeudArbreGenealogique :

```
template <typename T>  
class NoeudArbreGenealogique {  
    public:  
  
        T data; // donnée portée par notre noeud  
        std::vector<NoeudArbreGenealogique<T>*> enfants; // pointe vers le/les enfants  
        std::vector<NoeudArbreGenealogique<T>*> parents; // pointe vers le/les parents, avec deux parents maximum  
        int hauteur; // hauteur du noeud dans l'arbre
```

La classe NoeudArbreGenealogique permet de contenir les données nécessaires au bon fonctionnement de notre arbre pour chaque point d'intersection. Nous avons un arbre qui autorise à un élément la possession de multiples enfants et de deux parents. Ainsi les vecteurs enfants et parents permettent de garder les références des éléments liés à notre noeud.

La classe ArbreGenealogique :

```
class ArbreGenealogique  
{  
    public :  
  
        ArbreGenealogique(int anneeCourante) : anneeCourante(anneeCourante)  
        {  
            arbre = new Arbre<Personne>();  
        }
```

La classe ArbreGenealogique correspond à notre classe Dictionnaire de l'exercice précédent. Elle initialise l'arbre avec T de type personne puis implémente les méthodes demandées dans le sujet.

La classe Arbre :

```
template <typename T>
class Arbre
{
public:

    //Compteur éléments de l'arbre
    int cpt = 0;

    //correspond au premier element de l'arbre
    NoeudArbreGenealogique<T>* racine = (NoeudArbreGenealogique<T>*) malloc(sizeof(NoeudArbreGenealogique<T>));

    //liste d'éléments sans parents.
    std::vector<NoeudArbreGenealogique<T>*> elemSansParent;
```

La classe arbre permet d'offrir l'ensemble des méthodes permettant l'utilisation complète de notre arbre, ajout, recherche, parcours... Vu que nous sommes face à un arbre généalogique, certains éléments peuvent initialement n'être liés à personne (ajout d'une personne par alliance sans développer son arbre par exemple), pour ne pas perdre la référence vers cet élément nous le gardons dans une liste d'éléments sans parents. Il pourra être retrouvé via celle-ci lors du parcours de l'arbre.

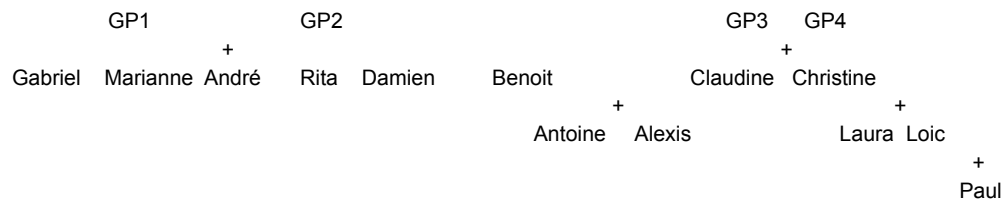
Initialisation:

```
ArbreGenealogique* arbreG = new ArbreGenealogique(2022);

//On ajoute une liste de perzonnes à notre arbre en créant des liens de parenté.
Personne GP1 = arbreG->ajouterMembre("GP1", "Inconnu", "Noir", 1948);
Personne GP2 = arbreG->ajouterMembre("GP2", "Inconnu", "Marron", 1944);
Personne Gabriel = arbreG->ajouterMembre("Gabriel", "DSM", "Vert", 1965, GP1, GP2);
Personne Marianne = arbreG->ajouterMembre("Marianne", "DSM", "Marron", 1965, GP1, GP2);
Personne Andre = arbreG->ajouterMembre("Andre", "DSM", "Vert", 1965, GP1, GP2);
Personne Rita = arbreG->ajouterMembre("Rita", "DSM", "Bleu", 1965, GP1, GP2);
Personne Damien = arbreG->ajouterMembre("Damien", "DSM", "Noir", 1965, GP1, GP2);
Personne Benoit = arbreG->ajouterMembre("Benoit", "DSM", "Bleu", 1965, GP1, GP2);
Personne GP3 = arbreG->ajouterMembre("GP3", "Inconnu", "Bleu", 1941);
Personne GP4 = arbreG->ajouterMembre("GP4", "Inconnu", "Bleu", 1942);
Personne Claudine = arbreG->ajouterMembre("Claudine", "Fayard", "Vert", 1959, GP3, GP4);
Personne Christine = arbreG->ajouterMembre("Christine", "Fayard", "Noir", 1962, GP3, GP4);
Personne Antoine = arbreG->ajouterMembre("Antoine", "DSM", "Bleu", 2001, Benoit, Claudine);
Personne Alexis = arbreG->ajouterMembre("Alexis", "DSM", "Vert", 2003, Benoit, Claudine);
Personne Marine = arbreG->ajouterMembre("Marine", "Veillon", "Vert", 2000);
Personne Laura = arbreG->ajouterMembre("Laura", "Saque", "Vert", 1996, Christine);
Personne Loic = arbreG->ajouterMembre("Loic", "Saque", "Marron", 1990, Christine);
Personne Paul = arbreG->ajouterMembre("Paul", "Saque", "Bleu", 2018, Loic);

//Test des différentes méthodes implémentées.
arbreG->calculerMoyenneAge();
arbreG->listerPersonnesCouleurYeux("Vert");
arbreG->listerPersonnesCouleurYeux("Marron");
arbreG->listerPersonnesCouleurYeux("Bleu");
arbreG->listerPersonnesCouleurYeuxAvant(Antoine, "Marron");
arbreG->listerPersonnesCouleurYeuxAvant(Rita, "Noir");
arbreG->nombreDeGeneration();
//Parcours dans l'arbre, par défaut In Order si aucun paramètre n'est rentrée.
arbreG->lireDescendancePersonne(GP1,0); //In order
arbreG->lireDescendancePersonne(GP1,2); //Pré order
//arbreG->lireDescendancePersonne(Benoit,0); //In order
arbreG->lireDescendancePersonne(GP1, 1); //Post order
return 0;
```

Nous créons dans notre main l'ensemble de notre arbre généalogique, voici sa structure finale :



Elements sans parents : GP, GP2, GP3, GP4, Marine.

## Exécution:

Post order :

Prenom : Antoine, nom : DSM, annee de naissance : 2001, couleur des yeux : Bleu  
Prenom : Alexis, nom : DSM, annee de naissance : 2003, couleur des yeux : Vert  
Prenom : Benoit, nom : DSM, annee de naissance : 1965, couleur des yeux : Bleu  
Prenom : Damien, nom : DSM, annee de naissance : 1965, couleur des yeux : Noir  
Prenom : Rita, nom : DSM, annee de naissance : 1965, couleur des yeux : Bleu  
Prenom : Andre, nom : DSM, annee de naissance : 1965, couleur des yeux : Vert  
Prenom : Marianne, nom : DSM, annee de naissance : 1965, couleur des yeux : Marron  
Prenom : GP1, nom : Inconnu, annee de naissance : 1948, couleur des yeux : Noir

In order :

Prenom : Alexis, nom : DSM, annee de naissance : 2003, couleur des yeux : Vert  
Prenom : Benoit, nom : DSM, annee de naissance : 1965, couleur des yeux : Bleu  
Prenom : Antoine, nom : DSM, annee de naissance : 2001, couleur des yeux : Bleu  
Prenom : Damien, nom : DSM, annee de naissance : 1965, couleur des yeux : Noir  
Prenom : Rita, nom : DSM, annee de naissance : 1965, couleur des yeux : Bleu  
Prenom : Andre, nom : DSM, annee de naissance : 1965, couleur des yeux : Vert  
Prenom : Marianne, nom : DSM, annee de naissance : 1965, couleur des yeux : Marron  
Prenom : Gabriel, nom : DSM, annee de naissance : 1965, couleur des yeux : Vert  
Prenom : GP1, nom : Inconnu, annee de naissance : 1948, couleur des yeux : Noir

Pre order :

Prenom : GP1, nom : Inconnu, annee de naissance : 1948, couleur des yeux : Noir  
Prenom : Benoit, nom : DSM, annee de naissance : 1965, couleur des yeux : Bleu  
Prenom : Alexis, nom : DSM, annee de naissance : 2003, couleur des yeux : Vert  
Prenom : Antoine, nom : DSM, annee de naissance : 2001, couleur des yeux : Bleu  
Prenom : Damien, nom : DSM, annee de naissance : 1965, couleur des yeux : Noir  
Prenom : Rita, nom : DSM, annee de naissance : 1965, couleur des yeux : Bleu  
Prenom : Andre, nom : DSM, annee de naissance : 1965, couleur des yeux : Vert  
Prenom : Marianne, nom : DSM, annee de naissance : 1965, couleur des yeux : Marron  
Prenom : Gabriel, nom : DSM, annee de naissance : 1965, couleur des yeux : Vert

La moyenne d'age est de : 50.1111ans.

Les personnes ayant les yeux Vert sont :

Prenom : Andre, nom : DSM, annee de naissance : 1965, couleur des yeux : Vert

Prenom : Gabriel, nom : DSM, annee de naissance : 1965, couleur des yeux : Vert

Prenom : Alexis, nom : DSM, annee de naissance : 2003, couleur des yeux : Vert

Prenom : Claudine, nom : Fayard, annee de naissance : 1959, couleur des yeux : Vert

Prenom : Laura, nom : Saque, annee de naissance : 1996, couleur des yeux : Vert

Prenom : Marine, nom : Veillon, annee de naissance : 2000, couleur des yeux : Vert

Les personnes ayant les yeux Marron sont :

Prenom : GP2, nom : Inconnu, annee de naissance : 1944, couleur des yeux : Marron

Prenom : Marianne, nom : DSM, annee de naissance : 1965, couleur des yeux : Marron

Prenom : Loic, nom : Saque, annee de naissance : 1990, couleur des yeux : Marron

Les personnes ayant les yeux Bleu sont :

Prenom : Benoit, nom : DSM, annee de naissance : 1965, couleur des yeux : Bleu

Prenom : Rita, nom : DSM, annee de naissance : 1965, couleur des yeux : Bleu

Prenom : GP4, nom : Inconnu, annee de naissance : 1942, couleur des yeux : Bleu

Prenom : GP3, nom : Inconnu, annee de naissance : 1941, couleur des yeux : Bleu

Prenom : Paul, nom : Saque, annee de naissance : 2018, couleur des yeux : Bleu

Prenom : Antoine, nom : DSM, annee de naissance : 2001, couleur des yeux : Bleu

Les ancetres de Antoine DSM ayant les yeux Marron sont :

Prenom : GP2, nom : Inconnu, annee de naissance : 1944, couleur des yeux : Marron

Les ancetres de Rita DSM ayant les yeux Noir sont :

Prenom : GP1, nom : Inconnu, annee de naissance : 1948, couleur des yeux : Noir

Le nombre de generations de l'arbre est de : 4.

## Exercice 5:

### Classes:

Nœud:

```
template <typename T>
class Noeud
{
public:
    Noeud(const T&);
    ~Noeud();

    void setPrecedent(Noeud<T> *nPrecedent);
    void setSuivant(Noeud<T> *nSuivant);
    void setId(int nId);
    void setIdPrecedent(int nId);
    void setIdSuivant(int nId);

    Noeud<T>* getPrecedent() const;
    Noeud<T>* getSuivant() const;
    int getId() const;
    T* getData();

    int somme();
    int sommePrecedent(int somme);
    int sommeSuivant(int somme);

private:
    Noeud* precedent;
    Noeud* suivant;
    T data;
    int id;
};
```

La classe Nœud permet de référencer deux autres nœuds (un précédent et un suivant), elle contient une donnée et un identifiant.

Seuls les méthodes somme() seront détaillés, car les autres méthodes sont des getters et setters.

La méthode somme() appelle sommePrécedent() et sommeSuivant(), afin de calculer, à partir d'un nœud, la somme des ID de la chaîne. Pour cela, elle va sommer les ID des nœuds se situant avant lui et ensuite sommer les ID des nœuds. La somme totale correspond alors à l'addition des deux sommes.



Liste:

```
namespace liste
{
    template<typename T>
    class Liste {
    public:
        Noeud<T>* tete;
        Noeud<T>* queue;
    public:
        Liste();
        void ajouterTete(const T& e);
        void ajouterQueue(const T& e);
        void ajouterAvant(Noeud<T> *sNoeud, const T& e);
        void ajouterApres(Noeud<T> *pNoeud, const T& e);
        void supprimer(Noeud<T> *pNoeud);
        Noeud<T>* getTete();
        Noeud<T>* getQueue();
        void setTete(Noeud<T> *teteNoeud);
        void setQueue(Noeud<T>* queueNoeud);
        void displayList();
    };
};
```

La classe Liste est une classe référençant un pointeur sur la tête de la liste et la queue de la liste, on ajoute des nœuds via cette classe afin de modifier ces pointeurs en conséquence. C'est cette classe également qui va modifier les pointeurs des nœuds.

Exécution:

```
Liste<int> l = Liste<int>();
l.ajouterTete(5);
l.ajouterTete(3);
l.ajouterTete(2);
l.ajouterTete(9);
l.ajouterTete(4);
l.ajouterTete(6);
l.ajouterTete(1);
l.ajouterTete(8);
l.ajouterTete(0);
l.ajouterTete(7);
l.ajouterQueue(2);
```

Dans la fonction main(), une liste est créée (ici une liste d'entier), on y ajoute plusieurs nœuds, soit par la tête, soit par la queue de la liste.

```
Noeud<int> *head = l.getTete();
Noeud<int> *tail = l.getQueue();
Noeud<int> *middle = head->getSuivant();

int total1 = head->somme();
int total2 = tail->somme();
int total3 = middle->somme();
```

Ensuite, on fait la somme des ID de la chaîne. Nous avons décidé de la faire depuis trois de la même chaîne pour vérifier si le résultat est le même.

Les nœuds choisis sont la tête de la liste, la queue de la liste et un entre les deux. Car cela correspond à trois cas particuliers de nœud.

## Résultat:

```
Console de débogage Microsoft Visual Studio
Ajout d'un element a la tete: 5
Ajout d'un element a la tete: 3 <==> 5
Ajout d'un element a la tete: 2 <==> 3 <==> 5
Ajout d'un element a la tete: 9 <==> 2 <==> 3 <==> 5
Ajout d'un element a la tete: 4 <==> 9 <==> 2 <==> 3 <==> 5
Ajout d'un element a la tete: 6 <==> 4 <==> 9 <==> 2 <==> 3 <==> 5
Ajout d'un element a la tete: 1 <==> 6 <==> 4 <==> 9 <==> 2 <==> 3 <==> 5
Ajout d'un element a la tete: 8 <==> 1 <==> 6 <==> 4 <==> 9 <==> 2 <==> 3 <==> 5
Ajout d'un element a la tete: 0 <==> 8 <==> 1 <==> 6 <==> 4 <==> 9 <==> 2 <==> 3 <==> 5
Ajout d'un element a la tete: 7 <==> 0 <==> 8 <==> 1 <==> 6 <==> 4 <==> 9 <==> 2 <==> 3 <==> 5
Ajout d'un element a la queue: 7 <==> 0 <==> 8 <==> 1 <==> 6 <==> 4 <==> 9 <==> 2 <==> 3 <==> 5 <==> 2
Somme des ID en partant de la tete de liste: 55
Somme des ID en partant de la queue de liste: 55
Somme des ID en partant d'une element quelconque de la liste: 55

Sortie de C:\Users\pierr\OneDrive\Documents\UQAC\Cours\Automne\Data\Devoirs\Exercice5\x64\Debug\Exercice5.exe (processus
22040). Code : 0.
Appuyez sur une touche pour fermer cette fenêtre. . .
```

On observe que les insertions d'éléments dans la liste doublement chaînés s'effectuent bien. Les sommes sur les trois nœuds définis fonctionnent bien, chaque somme a le même résultat malgré leurs positions différentes de nœuds. Ici le résultat est 55, parce que les ID des nœuds dans la liste sont codés comme étant uniques et commençant par 0. La liste ayant 11 éléments, la somme des ID correspond  $\sum_0^{10} n$  soit 55.