Devoir I

8INF840 – Structures de données avancées et leurs algorithmes

Hiver 2025

Aymen Sioud

UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
Département d'Informatique et de Mathématique

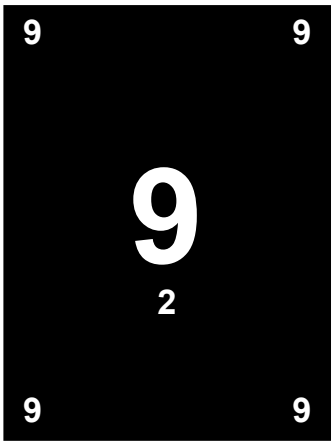
Consignes

- **A remettre au plus tard le 25 février 2025 avant le cours (13:00 EST).**
- **Le travail est en équipe de 3.**
- **Vous devrez utilisez Visual Studio 2019 ou plus.**
- **Le travail doit être effectué en C++.**
- **Il ne devra pas y avoir de dossier Debug.**
- **Les 5 exercices devront être dans un seul projet.**
- **Le travail devra être remis via la plateforme Moodle en un seul répertoire compressé.**
- **Le travail remis devra être nommé : Devoir1_ nom1_nom2_8INF840_H2025.**
- **Vous devrez remettre un rapport contenant vos captures d'écrans et traces d'exécutions pour chacun des exercices.**
- **Chaque exercice a une pondération de 20%.**

Exercice 1 : Jeu de batailles

À l'aide de C++, vous devez écrire un programme qui va simuler un jeu de carte intitulé « Bataille ». Le fonctionnement de ce jeu est décrit ci-dessus.

Le jeu comporte 100 cartes qui comportent des numéros allant de 1 à 10. Chaque carte possède aussi une couleur : rouge ou noir. Finalement chaque carte possède ce qu'on appelle un bonus. Il est évident qu'une même carte peut se répéter plusieurs fois.



La carte de gauche est une carte noire de valeur 9 et de puissance 2 alors que celle de droite est un 5 rouge de puissance 3.

Le jeu se joue avec deux joueurs. Et c'est l'ordinateur qui va représenter les deux joueurs. L'utilisateur devra gérer les conditions de jeu.

Au début de chaque jeu, on demandera à l'utilisateur combien de cartes chaque joueur aura (pas plus de 50 chacun des deux joueurs). Préalablement, 100 cartes devront être créées aléatoirement et mise dans une pile. La création d'une carte aléatoirement consiste à lui assigner une valeur de 1 à 10, une couleur : rouge ou noir et un bonus de 1 à 4.

Ensuite les cartes seront distribuées aux deux joueurs une à une à partir de cette pile de carte. Le premier joueur prendra la première carte, le deuxième la deuxième carte et ainsi de suite.

La distribution s'effectue aussi dans une pile (chaque joueur aura sa pile de carte).

Les cartes étant distribuées, le jeu peut commencer. À chaque tour, chaque joueur fera confronter une carte. Le joueur dont la carte aura une plus grande valeur gagne le tour (ordre croissant des cartes 10 bat toutes les autres cartes). Alors il enverra sa carte, ainsi que celle de l'autre joueur dans sa pile de gain. Si les valeurs des cartes sont égales, on compare les couleurs, le rouge gagne. Si les couleurs sont aussi identiques alors chaque joueur récupère sa carte. Il y aura donc autant de tour que de nombre de cartes.

Après un passage complet, les joueurs n'ont plus de cartes, le jeu est fini. Il est temps de désigner un vainqueur. Le score est calculé de la façon suivante. Une carte rouge compte une fois et demie alors qu'une carte noire compte pour sa propre valeur seulement. La valeur de la carte est multipliée aussi par le bonus. Dans l'exemple plus haut, la carte de gauche est comptabilisée pour 9 *1*2 = 18 alors que celle de droite est comptabilisée pour 5 *1.5*3 = 22.5. Le score final de chaque joueur sera la somme de toutes les cartes qu'il a dans sa pile de gain.

Vous utiliserez pour cela les piles telles qu'enseignées en classe avec les structures adéquates.

On demandera à l'utilisateur à la fin de chaque passage si oui ou non il voudra en effectuer un autre.

Pour les fins de cet exercice vous devrez utiliser la structure Pile suivante :

```
template <typename T>
class Pile
{
public:
        // constructeurs et destructeurs
        Pile(int max = MAX_PILE); //constructeur
        Pile(const Pile&); //constructeur copie
        ~Pile();      //destructeur
        // Modificateurs
        void empiler(const T&);
        T depiler();
        //Sélecteurs
        bool estVide()const;
        int taille() const;
        constT& sommet() const; // consulte l'élément au sommet
        //surcharge d'opérateurs
        const Pile<T>& operator = (const Pile<T);
        template <typename U> friend std::ostream& operator<<
                (std::ostream& , const Pile<U>& );

private: …    //Modèle d'implantation

};
```

Exercice 2 : L'usine

À l'aide de C++, vous devez écrire un programme qui va simuler une chaîne de production qui assemble des pistons. Le fonctionnement est décrit ci-dessus. En effet, une usine fait appel à vous afin de simuler le fonctionnement de son usine de recyclage de pistons.

Un piston est composé de trois pièces : la tête du piston, la jupe du piston et finalement son axe. L'assemblage des trois pièces se fait au niveau d'une machine principale MP.

L'arrivée des trois pièces à l'usine se fait sur le même dock et elles arrivent mélangées dans un même carton. Nous ne savons pas ce que contient un carton. La première opération consiste donc à trier les pièces pour les séparer. Chaque pièce passe ensuite sur une machine qui va procéder à son usinage pour les rendre de meilleure qualité et ce, dépendamment de sa nature. Nous aurons ainsi une machine MT pour les têtes, MJ pour les jupes et MA pour les axes. Les temps d'usinage respectifs sont de 2, 3 et 2.5 minutes pour les têtes, les jupes et les axes. La machine MP traite les pièces en 1 minute. Cependant, les 4 machines dont vous disposez sont presque toujours en panne et nécessitent des réparations. En effet, pour chaque pièce, une machine aurait jusqu'à 25% de chance de tomber en panne et la réparation nécessite entre 5 et 10 minutes.

Il est évident que la machine MP nécessite la présence des 3 pièces pour fabriquer un piston.

Nous vous demandons de reproduire la chaîne de montage et de déterminer le temps nécessaire pour monter 100 pistons. Le tout devra être effectué en utilisant les files.

Pour les fins de cet exercice vous devrez utiliser la structure File suivante (vous pourrez y apporter quelques modifications si nécessaire) :

```
template<typename T>
class File
{
public: // constructeurs et destructeurs:
       File() ;
       File(const File &);
       ~File();
       // modificateurs
       void enfiler(const T&);
       T defiler();
       // sélecteurs
       int taille() const;
       bool estVide() const;
       bool estPleine() const;
       const T& premier() const; //  tête de la file
       const T& dernier() const; // queue de la file
       // surcharges d'opérateurs
       const File<T>& operator = (const File<T>&) throw (bad_alloc);
       template <typename U> friend std::ostream& operator <<
       (std::ostream& f, const File<U>& q);
private:     // ...Modèle d'implantation
};
```

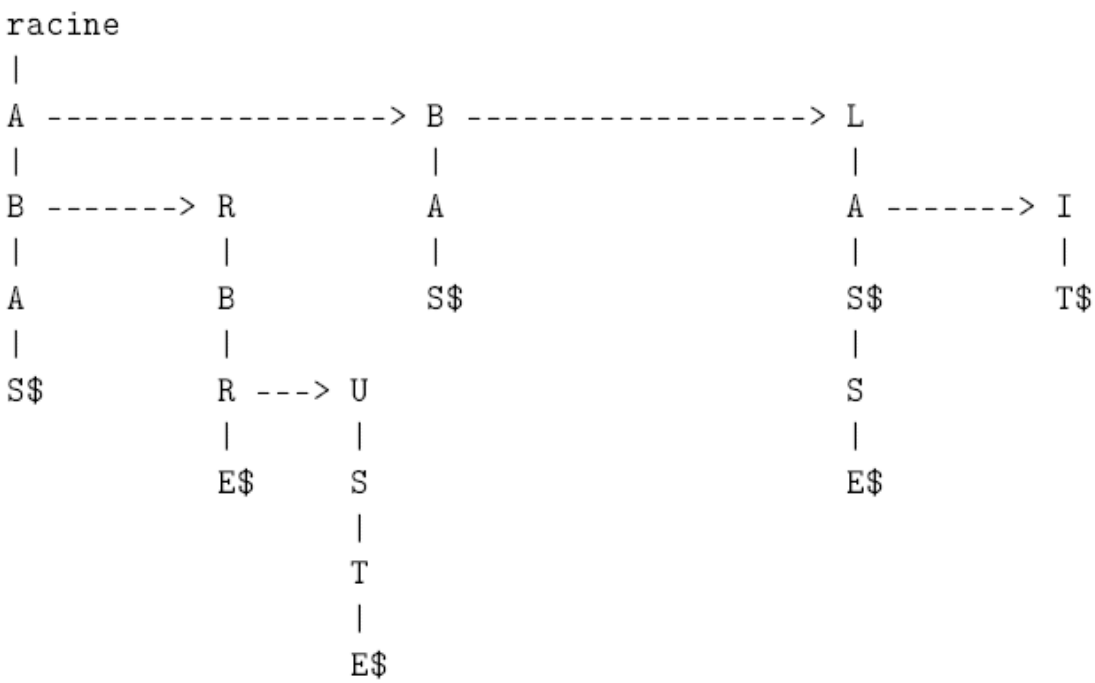
Exercice 3 : Dictionnaire

Il existe plusieurs manières de stocker des mots dans un dictionnaire. La méthode classique consiste à utiliser un arbre pour stocker les mots. On se propose d'implémenter en C++ une structure particulière de dictionnaire.

Cette structure utilise des arbres binaires.

L'arbre binaire que l'on cherche à produire fonctionne de la manière suivante : chaque nœud contient une lettre, deux pointeurs, et éventuellement d'autres informations (si on est à la fin d'un mot par exemple). Le premier pointeur sert à indiquer la prochaine lettre dans le mot. Le second indique les alternatives.

Voici un exemple :



Le dictionnaire que l'on voit ici contient les mots abas, arbre, arbuste, bas, las, lasse, lit. Les flèches vers la droite indiquent le fils droit, correspondant aux alternatives, le fils gauche, représenté par la barre verticale, la prochaine lettre du mot. Le nœud racine étant le premier nœud de l'arbre, correspondant au nœud racine de l'arbre.

À l'aide des fichiers fournis, vous devez implémenter en C++ une structure d'arbre binaire pouvant servir à stocker un dictionnaire. Il vous sera demandé de respecter l'ordre lexicographique quand vous ajouterez un mot dans le dictionnaire. Vous devrez implémenter plusieurs fonctionnalités à la structure de dictionnaire dont les fonctions suivantes :

1. ajouterMot(string s) : ajouter le mot s dans le dictionnaire.
2. enleveMot(string s) : enlève le mot s du dictionnaire.
3. affcherDict() : affiche le dictionnaire, selon l'ordre lexicographique.
4. chercheMot(string s) : si le mot appartient ou non au dictionnaire.

D'autres fonctionnalités peuvent être ajoutées, toute initiative sera la bienvenue.

Exercice 4 : Arbre généalogique

À l'aide de C++, vous devez écrire un programme qui permet de gérer un arbre généalogique d'une famille.

Pour chaque membre d'une famille, nous sauvegarderons le nom, le prénom, l'année de naissance et la couleur des yeux.

Vous devrez d'abord décrire le fonctionnement de votre arbre et la structure des nœuds utilisée.

Votre programme devra permettre de :

- Calculer la taille de l'arbre généalogique
- Lister la descendance d'une personne (in-order, pre-order and post-order)
- Ajouter un membre de la famille
- Pour une couleur d'yeux entrée par l'utilisateur, lister les personnes ayant cette couleur.
- Pour une couleur d'yeux entrée par l'utilisateur, permet de lister tous les ancêtres (ainsi que lui-même) qui ont la même couleur.
- Calculer la moyenne d'âge.

Exercice 5 : Liste distribuée

Il vous est demandé de mettre en œuvre une liste doublement chaînée distribuée où chaque nœud possède deux pointeurs vers son prédécesseur et son successeur et une méthode send(int). Chaque nœud ne peut communiquer qu'avec son successeur et son prédécesseur uniquement. Différentes instances des mêmes threads sont en cours d'exécution. Implémentez la classe thread qui permet à chaque nœud de calculer la somme des id de tous les nœuds.