# Grand Devoir 1

## Structures de données et algorithmes
## Piles, Files d'attente et Listes (Stacks, Queues & Lists)

**Mentions générales**

• Pour ce projet vous allez travailler par équipes de 2 personnes (ou seuls).

• Le projet est à rendre sur la plateforme Moodle (curs.upb.ro). S'il y a des problèmes lors du téléchargement vers ou depuis la plateforme, contactez par Teams ou par mail votre assistant des travaux pratiques.

• Le projet est à rendre **au plus tard le 22.04.2024, à 8h**. Aucun retard ne sera accepté.

• Vous recevrez des questions concernant votre solution durant la séance suivante du TP. **Les projets non présentés au laboratoire suivant ne seront pas notés !**

• La rendue finale du projet comportera une archive nommée Etudiant1Nom_Etudiant1Prenom_Etudiant2Nom_Etudiant2Prenom _GD1 avec :

1. **les fichiers du code source (.cpp et .h)** et non pas des fichiers objets (i.e. *.o) ou des fichiers exécutables (i.e. *.exe) ; **SVP mettez chaque exercice dans un dossier séparé !**
2. **un fichier de type README** où vous allez spécifier (dans quelques mots) toutes les fonctionnalités de votre projet, avec les instructions pour les employer. Au contraire, s'il y aura des exigences qui ne seront pas fonctionnelles, vous pouvez proposer des idées pour une solution possible (et peut-être obtenir des points supplémentaires).

• Pour toute question concernant le sujet du projet ou les exigences, envoyer un mail/message sur Teams à vos assistants ou utiliser le forum de la plateforme Moodle (il est recommandé de créer un nouveau post dans le groupe Teams afin que tout le monde puisse voir la question et la réponse). N'oubliez pas de mentionner @General (pour notifier tout le monde) ou @NomDuProf.

• Attention : notre équipe utilise des logiciels détectant le plagiat (Moss du Stanford). Dans le cas malheureux de plagiat, **le projet sera noté par un 0 (zéro)**.

• Attention x2 : Il est nécessaire de tester tous les exercices dans le main. Dans le cas où vous ne le faites pas, la note maximale pour chaque exercice sera de 1.

! Observation : Vous pouvez utiliser les structures de données et autres exercices que nous avons utilisés en classe. Alternativement, vous pouvez utiliser les implémentations standard de C++, MAIS pas d'autres implémentations personnalisées trouvées sur internet.

**1. Listes Circulaires Liées (Linked Lists) (4p)**
**Système de réservation de vols (Flight Booking System)**

Vous êtes chargé de développer une partie d'un système de réservation de vols pour une petite compagnie aérienne. La compagnie opère plusieurs vols par jour, et chaque vol a un nombre limité de sièges. Pour gérer efficacement la réservation et l'annulation de sièges sur plusieurs vols, vous décidez d'implémenter une structure de données qui reflète les besoins de la compagnie aérienne.

Structure de Données : **FlightBookingSystem**

Votre structure de données, **FlightBookingSystem**, gérera une série de vols, chaque vol ayant un nombre limité de sièges (pour simplifier - jusqu'à un maximum de 5 sièges par vol). Chaque vol est représenté par un tableau dans un nœud d'une liste chaînée, où chaque élément du tableau correspond au statut de réservation d'un siège (réservé/non réservé).

Méthodes à implémenter Les projets non présentés au laboratoire suivant ne seront pas notés !

1. **FlightBookingSystem()**: Initialise le système de réservation.
2. **void bookSeat(String passengerName)**: Réserve un siège pour un passager en ajoutant son nom au dernier siège disponible du dernier vol dans le système. Si le dernier vol est complet, un nouveau vol (nœud) est créé, et le passager est réservé sur ce nouveau vol.
3. **void cancelBooking(int seatIndex)**: Annule la réservation pour le siège à l'index global (sur tous les vols) indexSiège. Cette opération ne doit pas laisser de places vides dans les tableaux (vols). Par conséquent, les réservations suivantes doivent être décalées vers la gauche, éventuellement sur différents vols.
4. **void displaySystem()**: Affiche l'état actuel des réservations de tous les vols, chaque réservation de vol étant affichée sur une nouvelle ligne.
5. **String getPassenger(int seatIndex)**: Retourne le nom du passager réservé à l'index global de siège indexSiège.

Example :

1. Les réservations sont initialement vides.
2. **bookSeat("Sergiu")** ajoute Sergiu au premier vol.
3. **bookSeat("Ana")** ajoute Ana au même vol que Sergiu.
4. **displaySystem()** affiche l'état des réservations sur tous les vols.
5. **getPassenger(0)** retourne "Sergiu".
6. **cancelBooking(0)** efface Sergiu du premier vol, décale Ana au premier siège, et déplace les réservations suivantes en conséquence.
7. **displaySystem()** affiche l'état des réservations mis à jour.

Exigences du Problème :

- Le nombre maximum de sièges par vol est de 5. (0.5p)
- Implémentez la classe FlightBookingSystem avec les méthodes spécifiées. (2p)
- Assurez-vous que la méthode cancelBooking décale correctement les passagers sans laisser de sièges libres. (0.5p)
- Gérez les cas limites, tels que l'annulation d'une réservation lorsqu'il n'y a aucune réservation ou la tentative de réserver un siège lorsque tous les vols sont complets. (1p)

**2. File d'attente (Queue) (3p):**
**Voyage en oasis à travers le désert (Oasis Journey through Desert)**

Vous partez en voyage à travers un vaste désert, mais votre véhicule ne peut transporter qu'une quantité limitée d'eau. Le long de la route, il y a plusieurs oasis, chacune dotée d'une pompe à eau. Votre objectif est de **trouver l'oasis de départ optimale** à partir de laquelle **vous pouvez terminer le voyage sans manquer d'eau**.

Chaque oasis est représentée par **une structure** contenant deux attributs :

- **water** : la quantité totale d'eau disponible à cette oasis.
- **distanceToNext** : la distance jusqu'à la prochaine oasis.

**w** est la quantité totale d'eau qui existe dans le véhicule.

Dans **queueInit**, nous mettons l'oasis dans l'ordre croissant à partir de 0.

Nous utilisons la **queueAux** pour tester si nous pouvons terminer le voyage en partant de l'oasis qui est en tête de la file d'attente. Ainsi, nous **dequeue** chaque élément et voyons si nous réussissons à arriver à l'oasis suivante. Si nous ne pouvons pas (D < nextDistance), nous arrêtons le processus.

Si la file d'attente est vide, cela signifie que nous avons réussi à terminer le voyage (nous avons défilé toutes les oasis) -> nous avons le bon point de départ.
Sinon, nous partons de la prochaine oasis en **dequeuing** le premier élément et en le déplaçant à la fin de la file d'attente (nous avons une file d'attente circulaire) et répétons le processus.

**Notez** que queueAux est une copie de queueInit afin de ne pas la perdre.
Il est recommandé (mais pas obligatoire) de lire l'entrée et d'imprimer la sortie en utilisant des **fichiers**.

**Example**
**Entrée**
Le nombre d'oasis : 3
Le taux de consommation : 5
Eau des oasis et distance jusqu'au suivant :
3 5 // premiere ligne contient no oasis et taux de consommation
1 25
10 15
3 20

**Sortie**
1 est l'oasis à partir de laquelle nous pouvons commencer notre voyage

Points :
1p - lecture de l'entrée et construction de la file d'attente correctement
1.5p - algorithme principal pour trouver l'oasis correcte
0.5p - gestion des cas particuliers, affichage du résultat / erreur

**3. Pile (Stack) (3p) :**
**Calculatrice**

Écrivez un programme qui simule une calculatrice, en utilisant une **pile**. La calculatrice lira l'entrée à partir d'un fichier **("input.txt")** et imprimera la sortie dans un fichier **("output.txt")**.

Le programme va contenir les éléments suivants :
- **classe / struct Pair** - contient des paires de type variable (char) et valeur (int)
- **classe Calculator** - Initialise la calculatrice pour le programme.
  - **Attributs** :
    - **Stack<char> expressionStack** - enregistre les caractères de l'expression dans une pile ;
    - **tableau/liste de variables** – enregistre les noms de variables et leurs valeurs ;
  - **Méthodes** :
    - **void readInput()** - lit le fichier d'entrée;
    - **int execute()** - exécute l'expression et retourne le résultat;
    - **void writeOutput()** - écrit le résultat dans le fichier de sortie, avec la syntaxe Result=%expression_result%

**Opérateurs possibles : + - * / %**

**Noms de variables possibles :** lettres de A à Z, **uniquement en majuscules**, suivies de '=' et d'une valeur entière.

Dans le fichier d'entrée, sur des lignes séparées, les déclarations de variables seront d'abord, suivies par **une seule expression à gauche** utilisant les variables, **entre parenthèses**. La calculatrice devrait également être capable de gérer toute erreur d'entrée potentielle.

**Note :** Vous pouvez ajouter toutes autres méthodes/classes dont vous avez besoin, et vous pouvez utiliser “MyStack.h” implémenté et utilisé pendant les travaux pratiques.

**Gestion des erreurs** (Vous pouvez ajouter plus d'erreurs) :
- Opérateur invalide
- Valeurs numériques et/ou noms de variables invalides
- Variable utilisée dans l'expression n’est pas définie
- Division par zéro

**Entrée Example 1 :**
A=10
B=2
(A+B)

**Sortie :**
Résultat =12

**Entrée Example 2 :**
A=2
B=10
C=3
D=4
E=2
((((A+B)*C)+D)/E)

**Sortie :**
Résultat=20

**Entrée Example 3 :**
A$10
B$2
(A+B)

**Sortie :**
L'opérateur '$' n'est pas valide.

**Entrée Example 4 :**
A=0
B=9
(B/A)

**Sortie :**
Division par zéro.

Points :
0.5p - implémentation des classes et méthodes/constructeurs
1p - évaluation correcte de toutes les expressions mathématiques

0.5p - gestion des erreurs
1p - utilisation de fichiers pour l'entrée et la sortie

Liens utiles pour l'utilisation de fichiers pour l'entrée/sortie :
https://cplusplus.com/doc/tutorial/files/
https://www.w3schools.com/cpp/cpp_files.asp