IFT339 - Exercices sur les listes chaînées et ses SDD

Exercice 1.

(a) Une expression parenthésée est bien formée si, en lisant de gauche à droite, chaque parenthèse ouvrante a une parenthèse fermante, et s'il n'y a pas de parenthèse fermante qui n'a pas été ouverte.
Par exemple, l'expression suivante est bien formée :
(()(()))
Mais celles-ci ne le sont pas
(()() et ()))
Vous recevez un vecteur<char> dans lequel chaque élément est soit '(' ou bien ')'. Utilisez une structure de données vue en classe pour vérifier qu'un tel vecteur représente un parenthésage bien formé.

(b) Reprenez l'exercice précédent, mais avec plusieurs caractères de parenthésage différents. Par exemple, avec des ( ), [ ] et { }, on pourrait avoir (([( )]{([ ])}){ }).

Version facile: le nombre de paire de caractères ouvrant/fermant est constant et vous pouvez le "hard-coder".

Version moins facile: si l'entrée est un vector<int> et les paires de caractères ouvrant/fermant sont données en argument dans un
vector< pair<int,int> >, quelle complexité pouvez-vous atteindre? Ceci survient par exemple quand il y a beaucoup de caractères possibles comme en UTF16.

Exercice 2.
Vous avez deux listes chaînées $c1$ et $c2$. Vous voulez savoir si $c2$ a été obtenu de $c1$ en insérant des noeuds[1]. Donnez le code d'une fonction qui supprime des noeuds de $c2$ pour que le résultat soit égal à $c1$ et retourne *true* si c'est possible, et si ce n'est pas possible, retourne $false$ (ici, égal au sens == comme dans la série d'exercices précédente). Si ce n'est pas possible, $c2$ sera dans un état non-défini, donc peut avoir certains noeuds supprimés ou non.

[1]Ceci survient, par exemple, si $c1$ représente une liste de lignes de code, et quelqu'un a ajouté des lignes dans votre fichier, ce qui donne $c2$. Vous voulez savoir quelles sont ces lignes.

Exercice 3.
Vous avez une liste chaînée dans laquelle chaque noeud est étiquetté comme "vert" ou "rouge". Ces deux valeurs peuvent, par exemple, représenter deux priorités différentes. Une liste est appelée "bien formée" si tous les noeuds verts apparaissent avant les noeuds rouges.

(a) Donnez le code d'une fonction qui détermine si une liste est bien formée. (ceci devrait être facile)

(b) Écrivez le code manquant dans

```
struct Noeud{
    bool est_vert;  //true si vert, false si rouge
    TYPE val;
    Noeud* next;
}
void combiner(Noeud* v, Noeud* w){
    //code manquant
}
```

pour une fonction qui reçoit le premier noeud v d'une liste bien formée, et le premier noeud w d'une autre liste bien formée. À la sortie, la liste v doit être modifiée: elle doit inclure tous les noeuds de w et être bien formée (la liste w peut aussi être modifiée). En d'autres termes, il faut ajouter à v les éléments de w de façon à ce que le résultat soit bien formé. Évitez de créer des nouveaux noeuds.

(c) On suppose maintenant qu'on reçoit une liste qui n'est pas bien formée (donc il y a des verts et des rouges un peu partout dans n'importe quel ordre). Donnez le code d'une fonction qui réordonne les noeuds pour que la liste soit bien formée.

Version facile. Ceci est assez facile si vous permettez de créer une nouvelle chaîne et de supprimer l'ancienne.

Version difficile. Essayez plutôt de ne pas créer de nouveau noeud, et de seulement réassigner les pointeurs des noeuds actuels. Votre méthode ne devrait déclarer qu'un nombre constant de variables.

Exercice 4.
Vous gérez le journal de la libération de mémoire automatique de variables C++. Rappelons que la mémoire est gérée par une pile. Vous savez ques les variables $1, 2, \ldots, n$ ont été créées et donc empilées (*push*) **dans cet ordre**. Par contre, vous ne savez pas à quel moment elles ont été dépilées (*pop*).

Le journal vous donne l'ordre de dépilement des variables, mais vous suspectez qu'il a été trafiqué. Vous voulez donc savoir si le journal donne une séquence de dépilements possibles.

Par exemple, si $n = 4$, la séquence
$push(1), push(2), pop(), push(3), pop(), push(4), pop()$
donne la séquence de dépilement 2 3 4 1. Une telle séquence est appelée *possible*.

(a) Dites si chacune des lignes suivantes est possible (et pourquoi):
1 2 3 4
4 3 2 1
4 2 1 3
3 4 2 1
3 4 1 2

(b) Donnez le code d'une fonction qui reçoit les entiers 1 à n dans un ordre quelconque, et détermine si la séquence donnée est possible.

(c) Si les *push* et les *pop* se faisaient plutôt sur une file, comment pourriez-vous résoudre le problème?

Exercice 5.
Implémentez une file en utilisant seulement deux piles. Donc, vous avez une classe dont les seules variables membres sont de type *stack*, et vous devez implémenter push_back, pop_front. Quelle complexité pouvez-vous atteindre?

Exercice 6.
(*difficile*) Vous avez une file d'entiers, avec push_back et pop_front en temps $O(1)$, et vous voulez ajouter une fonction get_min, qui retourne la plus petite valeur qui est présentement dans la file. On voudrait que get_min prenne un temps $O(1)$. Comment faire?

Vous pouvez ajouter des variables membre et modifier push_back et pop_front, mais ils doivent demeurer en temps $O(1)$ (ou $O(1)$ amorti sur n push et pop est acceptable).

Suggestion. Il m'a fallu un moment pour trouver comment, alors voici un début d'idée. Dans ma file, j'ai une variable membre qui est une file, disons deque<int> next_mins. À tout moment, next_mins.front() contient le minimum actuel.

Le problème survient quand on pop le minimum de la structure. À ce moment, next_mins.front() n'est plus valide. Il faut qu'on le pop, et que le prochain front soit le prochain minimum.

Pour arriver à ça, je modifie push_front(val). Au push, je regarde next_mins de droite à gauche et je regarde si val rend certains éléments désuets car ils ne seront jamais le prochain min. Si oui, je les pop, et j'insère val à la fin de la liste. Compliqué à expliquer, mais pas beaucoup de lignes de code quand on a compris :)

Exercice 7.
(exercice bonus) Regardez le problème ici, qu eje ne vais pas retranscrire:
`https://leetcode.com/problems/simplify-path/description/?envType=problem-list-v2&envId=stack`