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.

Solution

L'idée est d'utiliser une pile. À chaque "(", on empile, ce qui représente le fait qu'on attend la fermeture. À chaque ")", on dépile pour montrer que la dernière ouverture vient d'être fermée. Si la pile est vide, il y a un ")" de trop et on retourne false. Si à la fin, la pile est non-vide, il y a un "(" en trop.

Nous donnons deux version: une en pseudo-code pour que vous voyez ce que j'accepte, et une en C++, compilable et testable.

```
fonction estBienFormé(parenthesage)
    pile = Pile()
    pour i = 0..parenthesage.size() − 1 faire
        si parenthesage[i] == '(' alors
            pile.push('(')
        sinon si pile.estVide() alors
            return false
        sinon
            pile.pop()
        fin
    fin
    return pile.estVide()
```

Version C++

```cpp
#include<iostream>
#include<stack>
#include<vector>

bool estBienForme(std::vector<char> parenthesage) {
    std::stack<char> pile;
    for (int i = 0; i < parenthesage.size(); i++) {
        if (parenthesage[i] == '(')
            pile.push(parenthesage[i]);
        else if (pile.empty()) //on a une fermante jamais
   ouverte
            return false;
        else   //on vient de fermer une parenthese
            pile.pop();
    }
    //si la pile est non-vide, on a une ouvrante non-fermee
    return (pile.empty());
}
```

Fin de la solution

(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.

Solution

La version facile: il faut empiler les caractères ouvrants, et quand on dépile on vérifie que ça correspnd.

```
bool estBienForme(std::vector<char> parenthesage)
{
    stack<char> pile;
    for (int i = 0; i < parenthesage.size(); i++)
    {
        if (parenthesage[i] == '('
                    || parenthesage[i] == '{'
                    || parenthesage[i] == '[')
            pile.push(parenthesage[i]);
        else
        {
            if (pile.empty())   //fermante jamais ouverte
                return false;
            char attendu = ")";
            if (pile.top() == "[") attendu = "]";
            if (pile.top() == "{") attendu = "}";
            if (parenthesage[i] != attendu)
                return false;
            pile.pop();
        }
    }
    return (pile.empty());
}
```

Pour la version moins facile, quand on empile, on empile le prochain caractère fermant attendu, qui est spécifié dans les paires en entrée. Quand on dépile, on vérifie qu'on a le bon caractère fermant. Pour que ça aille vite, je trie les paires données en entrée, ce qui fait que la recherche sera en temps $O(\log m)$, où m est le nombre de paires. Je donne la solution en pur pseudo-code.

```
fonction estBienFormé(paren, paires)
    Trier paires selon le premier élément
    pile = Pile()
    pour i = 0..paren.size() − 1 faire
        si il y a (x, y) dans paires tel que paren[i] == x alors
            pile.push(y) //push le prochain fermant attendu
        sinon
            si pile.estVide() alors
                return false
            sinon si pile.top()! = paren[i] alors
                return false
            sinon
                pile.pop()
    fin
    return (pile.estVide())
```

Analysons la complexité. Supposons que $n = paren.size()$ et $m = paires.size()$. Le tri de *paires* peut se faire en temps $O(m \log m)$ (voir std::sort). Ensuite, à chaque tour de boucle, il n'y a que des opérations en temps constant, sauf la partie de trouver (x, y) dans *paires* tel que $paren[i] == x$. Ceci demande de parcourir *paires* et d'y trouver une paire dont le premier élément est x. On peut facilement modifier la recherche binaire vue en classe pour trouver (x, y) en temps $O(\log m)$. Donc, chaque itération prend un temps $O(\log m)$ dans le pire cas, et on fait n itérations.

Le temps total est donc $O(m \log m + n \log m)$. Plus tard dans la session, on verra qu'on peut atteindre $O(n + m)$ (avec forte probabilité).

Fin de la solution

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.

Essayez de faire l'exercice quand $c1$ et $c2$ sont simplement chaînées, puis doublement chaînées.

Solution

L'idée est d'avancer sur $c1$ et $c2$ avec deux pointeurs. Quand on voit un valeur de $c2$ qui diffère du noeud correspndant sur $c1$, on supprime dans $c2$. Quand la valeur est la même, on avance les deux pointeurs. Le plus difficile est de gérer les petits cas limite. Voir prochaine page.

[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.

```
//c1 et c2 sont simplment chainees
bool rend_egal(Noeud* c1, Noeud* c2){
    //cas limites de c2 vide
    if (!c1 && !c2) return true;
    if (c1 && !c2) return false;

    Noeud* p1 = c1;
    Noeud* p2 = c2;
    Noeud* p2_prev = nullptr;

    while (p2){
        if (!p1 || p1->val != p2->val){ //supprimer p2
            if (!p2_prev)
                c2 = p2->next;
            else
                p2_prev->next = p2->next;
            Noeud* tmp = p2->next;
            delete p2;
            p2 = tmp;
        }
        else{
            p1 = p1->next;
            p2_prev = p2;
            p2 = p2->next;
        }
    }
    return (p1 == nullptr); //pourquoi ceci marche??
}
```

Pour la version avec liste doublement chaînée, on n'a pas besoin de mémoriser le p2_prev, mais il y a de la gestion supplémentaire.

```
//c1 et c2 sont doublement chainees
bool rend_egal(Noeud* c1, Noeud* c2){
    //cas limites de c2 vide
    if (!c1 && !c2) return true;
    if (c1 && !c2) return false;

    Noeud* p1 = c1;
    Noeud* p2 = c2;

    while (p2){
        if (!p1 || p1->val != p2->val){ //supprimer p2
            if (!p2->prev)
                c2 = p2->next;
            else
                p2->prev->next = p2->next;
            Noeud* tmp = p2->next;
            if (tmp)
                tmp->prev = p2->prev;
            delete p2;
            p2 = tmp;
        }
        else{
            p1 = p1->next;
            p2_prev = p2;
            p2 = p2->next;
        }
    }
    return (p1 == nullptr); //pourquoi ceci marche??
}
```

Fin de la solution

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)

Solution

```
est_bien_formee(Noeud* v){
    bool vu_rouge = false;
    Noeud* p = v;
    while (p){
        if (!p->est_vert)
            vu_rouge = true;
        else if (vu_rouge)
            return false;
        p = p->next;
    }
    return true;
}
```

Fin de la solution

(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.

Solution

Il faut juste trouver le dernier noeud vert de v et insérer toute la chaîne w entre v et v->next. Il faut tout de même gérer quelques cas limite.

```
if (!v)    //cas limite, v est vide
    v = w;
else if (w){    //si !w, ne rien faire
    Noeud* w_last = w;
    while (w_last->next)
        w_last = w_last->next;

    if (!v->est_vert){   //v n'a aucun vert
        Noeud* v_tmp = v;
        v = w;
        w_last->next = v_tmp;
    }
    else{   //v a au moins un vert: on insere w apres
        Noeud* v_vert = v; //dernier noeud vert de v
        while (v_vert->next->est_vert)
            v_vert = v_vert->next;
        Noeud* tmp = v_vert->next;
        v_vert->next = w;
        w_last->next = tmp;
    }
}
```

Fin de la solution

(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.

Solution

L'idée est qu'on maintient deux sous-listes, une pour les verts et une pour les rouges. À la fin, on ajoute le sous-liste rouge à la fin de la sous-liste verte. Voir prochaine page.

```
Noeud* v = debut; //debut est le input, on suppose non vide
Noeud* first_vert = nullptr, last_vert = nullptr;
Noeud* first_rouge = nullptr, last_rouge = nullptr;
while (v){
    if (v->est_vert){
        if (!first_vert)
            first_vert = v;
        else
            last_vert->next = v;
        last_vert = v;
    }
    else{
        if (!first_rouge)
            first_rouge = v;
        else
            last_rouge->next = v;
        last_rouge = v;
    }
    v = v->next;    //notez que v->next n'a pas changé ci-haut
}
if (!first_vert)   //aucun vert
    return first_rouge;
else if (!first_rouge){
    return last_vert;
else{   //concatener vert -> rouge
    last_vert->next = first_rouge;
    last_rouge->next = nullptr;
    return first_vert;
}
```

Fin de la solution

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
$push(1), push(2), pop(), push(3), pop(), push(4), pop(), pop()$
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

Solution

- 1 2 3 4 est faisable: push 1, pop, push 2, pop, push 3, pop, push 4, pop
- 4 3 2 1 est faisable: push 1,2,3,4 suivi de pop pop pop pop
- 4 2 1 3 n'est pas faisable. Pour avoir ça, il faudrait nécessairement faire un push de 1,2,3,4 puis faire un pop. Le prochain pop produira 3.
- 3 4 2 1 est faisable. push 1,2,3, pop, push 4, pop, pop, pop.
- 3 4 1 2 n'est pas faisable. Il faudrait push 1, 2, 3, pop, push 4, pop, et le prochain pop sera 2.

Fin de la solution

(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.

Solution

Supposons que la séquence est donnée dans un vector<int> v. La séquence nous dit exactement quand on doit empiler et dépiler. Par exemple, pour avoir v[0], il faut empile de 1 à v[0] puis dépiler. Ensuite pour avoir v[1], il faut soit l'avoir au sommet, ou bien empiler jusqu'à v[1] et dépiler.

```
bool est_possible(vector<int>& v){
    stack<int> pile;

    int last_push = 0;  //dernière valeur insérée
    int index_v = 0;

    while (index_v < v.size()){
        //cas 1: le sommet de la pile est le prochain attendu
        //on le dépile et on avance sur v
        if (!pile.empty() && v[index_v] == pile.top()){
            pile.pop();
            index_v++;
        }
        //cas 2: le sommet n'est pas celui attendu.  Il faut
        //pouvoir empiler jusqu'à v[index_v] et dépiler
        else{
            if (v[index_v] < last_push)
                return false;
            for (int i = last_push + 1; i <= v[index_v]; ++i){
                pile.push(i);
                last_push = i;
            }
            pile.pop();
            index_v++;
        }
    }
    return true;
}
```

Fin de la solution

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

Solution

Top facile! Une file est first-in first-out. Si vous insérez $1, 2, \ldots, n$ dans une file, peu importe à quel moment vous faites les pop, le seul ordre possible est $1, 2, \ldots, n$.

Fin de la solution

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?

Solution

Ceci est un classique et il y a plusieurs solutions en ligne. Je vais m'économiser du temps et vous rediriger vers le web.

Exemple: `https://codex.forge.apps.education.fr/en_travaux/file_deux_piles/`.

Le temps d'un push est $O(1)$. Le temps d'un pop est $O(n)$ dans le pire cas, mais est temps $O(1)$ amorti (donc faire n fois push et n fois pop prend un temps total $O(n)$).

Fin de la solution

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 :)

Solution

Voir `https://stackoverflow.com/questions/12054415/get-min-max-in-o1-time-from-a`

Fin de la solution

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