TP7: Pile et file

L'objectif de ce TP est d'implémenter les piles par tableaux et les files par listes chaînées, et d'étudier une variante de l'algorithme de détection des mots bien-parenthésés. Vous pouvez télécharger sur Cahier de Prépa une archive de fichiers pour ce TP.

1 Pile

Structure abstraite

On considère dans cette partie des piles dont les éléments sont des `int`. Le fichier `pile.h` de l'archive déclare les 4 opérations des piles vues en cours, ainsi que des fonctions d'affichage et de libération de mémoire. Le fichier `pile_mystere.c` contient une implémentation complète, vous ne devez pas (et n'avez pas besoin de) regarder le contenu de ce fichier pendant le TP [1].

Q1. Créer un fichier `test_pile.c` dans lequel vous créerez une fonction main initialisant une pile vide, y ajoutant et retirant quelques éléments, en vérifiant à l'aide d'assertions que les éléments dépilés sont ceux attendus. Compiler avec l'implémentation mystère et vérifier que tout fonctionne.

Implémentation par tableaux

Q2. Implémenter dans un fichier `pile_tab.c` les fonctions de `pile.h` en utilisant l'implémentation par tableaux vue en cours. On fixera la taille maximale à 10000. Compiler le programme de test, cette fois-ci en utilisant la nouvelle implémentation, et vérifiez que l'exécution est correcte.

Implémentation par liste chaînée

On utilise le type struct suivant :

```
1 typedef struct maillon {
2   T elem;
3   struct maillon* suivant;
4 } maillon_t;
5
6 struct pile {
7   maillon_t* sommet;
8 };
```

On rappelle que chaque maillon contient un élément, et un pointeur vers le maillon suivant. Cette chaîne de maillons va du sommet de la pile vers la base.

1. Vous pouvez essayer...

Évitez de regarder le cours pour les questions de cette partie, afin de retrouver par vous-même le code.

Q3. Sur papier, représentez une pile avec 2 éléments, implémentée par liste chaînée, puis représentez la même pile sur laquelle on a empilé un nouvel élément. Déduisez-en le code de la fonction d'empilage.

Q4. Implémentez les autres opérations, puis, en reprenant le même fichier `test_pile.c`, vérifiez que votre code fonctionne. Pensez à compiler avec toutes les options de débug/warning, et à exécuter avec valgrind si disponible sur votre machine.

Q5. **Optionnel** Implémentez des versions alternatives des fonctions d'affichage et de libération mémoire, en utilisant de la récursivité plutôt que des boucles.

Reste du TP à faire après le cours sur les tableaux redimensionnables.

Implémentation par tableaux redimensionnables

On rappelle l'existence de la fonction `realloc` :

```
1 int* t = malloc(n * sizeof(int));
2 // réalloue un autre tableau de taille m pour t,
3 // et recopie le contenu de l'ancien tableau dans le nouveau.
4 // libère l'ancien tableau.
5 t = realloc(t, m * sizeof(int));
```

Q6. Utilisez le système de tableaux redimensionnables vu en cours pour modifier votre implémentation des piles par tableaux. Pensez à rajouter à votre fichier de test de quoi vérifier que les redimensionnements fonctionnent.

(Optionnel) Rétrécissement du tableau

On souhaite non seulement agrandir le tableau lorsque la pile devient trop grande, mais aussi rétrécir le tableau lorsque la pile rétrécit, afin de ne pas utiliser trop d'espace superflu. Après chaque dépilage, si le tableau est peu rempli, on réalloue un nouveau tableau plus petit. On ne descendra jamais en dessous de la taille initiale.

Q7. Ajouter à la fonction `depiler` du code permettant de diviser la taille du tableau par 2 si moins de la moitié du tableau est remplie après dépilage.

Q8. La stratégie proposée à la question précédente est-elle adaptée ? Proposer une suite d'opérations où cette stratégie est très coûteuse, par exemple où le coût moyen d'un empilage/dépilage est linéaire.

Q9. Proposer une meilleure stratégie.

2 Application des piles

On considère une chaîne moléculaire [2] , constituée de différents éléments. Chaque élément a deux polarités possibles : positive et négative. On représente les éléments par des lettres, en utilisant les majuscules pour les éléments positifs, et les minuscules pour les éléments négatifs. On appelle une chaîne d'éléments un ***polymère***. Par exemple, **dabAcCaCBAcCcaDA** est un polymère.

Deux éléments de même type mais de polarités opposées peuvent réagir entre eux pour disparaître. Par exemple, si **a** et **A** sont adjacents dans un polymère, ils s'annulent (l'ordre n'importe pas). On dit qu'un polymère est instable s'il contient des éléments pouvant réagir. Un polymère instable va donc se réduire en un autre polymère. On appelle ***forme stable*** d'un polymère le polymère obtenu après avoir fait toutes les réductions possible. On admet que tout polymère admet une forme stable, et que cette forme stable est unique : l'ordre des réductions n'importe pas. Par exemple **dabAcCaCBAcCcaDA** peut se réduire 3 fois avant d'atteindre sa forme stable :

$$\begin{aligned} \textbf{dabAcCaCBA}\underline{\textbf{cC}}\textbf{caDA} &\longmapsto \textbf{dabAcCaCBAcaDA} \\ \textbf{dabA}\underline{\textbf{cC}}\textbf{aCBAcaDA} &\longmapsto \textbf{dabAaCBAcaDA} \\ \textbf{dab}\underline{\textbf{Aa}}\textbf{CBAcaDA} &\longmapsto \textbf{dabCBAcaDA} \end{aligned}$$

On souhaite mettre au point un programme qui détermine efficacement la forme stable d'un polymère.

Q10. Donner la forme stable du polymère **abcdDCaABeEeA**.

Un algorithme naïf calculant la forme stable d'un polymère consiste à appliquer les réductions trouvées une par une :

Entrée(s) : $p = p_0 p_1 \dots p_{n-1}$ polymère
Sortie(s) : La forme stable de p

1 $s \leftarrow p$;
2 **tant que** ***deux éléments*** $s_i s_{i+1}$ ***peuvent réagir*** **faire**
3 $\quad s \leftarrow s_0 \dots s_{i-1} s_{i+2} \dots s_{k-1}$// k `est la taille de` s
4 **retourner** s

La boucle peut s'exécuter au plus $\frac{n}{2}$ fois, et chaque étape coûte $\mathcal{O}(n)$ car il faut recréer la nouvelle chaîne en entier. Au total, l'algorithme coûte donc $\mathcal{O}(n)$.
Nous allons chercher un algorithme en $\mathcal{O}(n)$ dont le principe est similaire à celui de l'algorithme vu en cours pour les mots bien-parenthésés. L'idée est de lire le polymère caractère par caractère, en stockant dans une pile les éléments lus n'ayant pas encore pu réagir, et en testant à chaque caractère lu s'il peut réagir avec le sommet de la pile.

Q11. En pseudo-code, écrire un algorithme implémentant l'idée précédente, et l'exécuter à la main sur le polymère de la question précédente.

Q12. Ajoutez à votre implémentation des piles une opération renvoyant la taille d'une pile.

Q13. Créez un nouveau fichier C, et écrivez-y une fonction C `char* taille_stable(char* polymere)` implémentant votre algorithme. Cette fonction renverra une nouvelle chaîne de caractères donnant la forme stable du polymère d'entrée. [3] Déterminer le nombre d'éléments de la forme stable du polymère écrit dans le fichier "polymere.txt" de l'archive.

2. Le problème dans cette partie est tiré de : adventofcode.com/2018/day/5.
3. **Astuce** : en C, si `c` est une variable de type char contenant le code d'une lettre minuscule, alors `c-'a' + 'A'` donne le code de la lettre majuscule correspondante.

3 File

L'archive du TP contient un fichier `file.h` contenant les déclarations des 4 opérations des files, ainsi que des fonctions d'affichage et de libération.

Implémentation par listes chaînées

On utilise la structure suivante :

```
typedef struct maillon {
  int elem;
  struct maillon* suivant; // de la tete vers la queue
} maillon_t;
54
+
+
struct file {
  maillon_t* tete;
  maillon_t* queue;
};
```

Attention, l'attribut `suivant` pointe de la tête vers la queue, c'est à dire dans le sens **inverse** de la file. Pour s'en souvenir, on peut penser à la file d'attente : quand le caissier dit "au suivant" c'est la personne en tête qui passe, puis la personne derrière elle, etc...

Q14. Implémentez la fonction de création de file et la fonction de vérification de file vide.

Q15. Faites un schéma pour représenter un enfilage dans une file vide, puis dans une file non vide. En déduire le code de la fonction `enfiler`.

Q16. Faites un schéma pour représenter un défilage dans une file avec un seul élément, puis dans une file avec au moins deux éléments. En déduire le code de la fonction `defiler`.

Q17. Implémentez les fonctions d'affichage et de libération, et écrivez dans un nouveau fichier `test_file.c` de quoi tester les fonctions implémentées jusqu'à maintenant.

(Optionnel) Implémentation par tableaux cycliques

On utilise la structure suivante :

```
#define Nmax 10000
struct file {
  int queue; //prochaine case à remplir
  int nb_elem;
  int tab[Nmax];
};
```

L'attribut `.queue` est un indice du tableau, et indique la prochaine case dans laquelle on écrira si l'on enfile un élément. Cet indice est incrémenté à chaque enfilage.

On rappelle qu'en C, l'opérateur `%` renvoie un nombre négatif si on l'applique sur un nombre négatif, par exemple `-10 % 3` vaut -1 et pas 2.

Q18. Proposer une formule qui, étant donné deux entiers $k \in \mathbb{N}$ et $x \in \mathbb{Z}$, permet de calculer le reste de x modulo k, **dans l'intervalle** $[\![0, k-1]\!]$.

Q19. Implémenter les opérations de `file.h` dans un fichier `file_tab.c` en utilisant le principe de tableau circulaire. On garantira que l'indice `.queue` est toujours dans l'intervalle $[\![0, \texttt{Nmax} - 1]\!]$.