TP7 : Piles et files

MP2I Lycée Pierre de Fermat

L'objectif de ce TP est d'implémenter les piles par tableaux redimensionnables 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.

Pile

On considère dans cette partie des piles dont les éléments sont des `int`.

Q1. Regarder le contenu du fichier `pile.h` de l'archive et vérifier que ce header déclare bien les 4 opérations des piles vues en cours, ainsi que des fonctions d'affichage et de libération de mémoire.

Implémentation par tableaux redimensionnables

On utilise le type struct suivant :

```
1 struct pile {
2   int nb_elem;
3   int taille_max;
4   int* tab;
5 };
```

Les cases $0, 1, \ldots,$ `.nb_elem` $- 1$ du tableau `.tab` contiennent les éléments de la pile, de la base vers le sommet. Lorsque l'on empile, on écrit donc le nouvel élément dans la case d'indice `.nb_elem`. Si cette case dépasse du tableau, on commence par redimensionner le tableau en lui allouant deux fois plus de taille.
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));
```

Q2. Implémenter dans un fichier `pile_tab.c` les fonctions de `pile.h` en utilisant l'implémentation par tableaux redimensionnables. On fixera arbitrairement la taille initiale à 10.

Q3. Créez un autre fichier `test_pile.c` dans lequel vous mettrez la fonction `main`, et où vous testerez bien les différentes opérations des piles.

(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.

Q4. 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.

Q5. 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.

Q6. Proposer une meilleure stratégie.

(Optionnel) Implémentation par listes chaînées

Passons maintenant à l'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 };
```

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.

Q7. Créez un fichier `pile_chaine.c` et implémentez la fonction de création de pile ainsi que la fonction testant si une pile est vide.

Q8. 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.

Q9. Même question pour le dépilage.

Q10. Implémentez les fonctions d'affichage et de libération mémoire.

Q11. Implémentez des versions alternatives des fonctions de la question précédente, en utilisant de la récursivité plutôt que des boucles.

Q12. En reprenant le même fichier `test_pile.c`, vérifiez votre implémentation. Pensez à compiler avec toutes les options de débug et à exécuter avec valgrind si possible.

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 :

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


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...

Q13. Implémentez la fonction de création de file et la fonction de vérification de file vide.

Q14. 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`.

Q15. Faites un schéma pour représenter un défilage dans une file avec une seul élément, puis dans une file avec au moins deux éléments. En déduire le code de la fonction `defiler`.

Q16. 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.

Q17. Implémentez la fonction `defiler`, en faisant à nouveau bien attention à ce qui se passe si la file est vide une fois que l'on a défilé.

Q18. Complétez votre fichier de test pour y mettre aussi la fonction `defiler`.

(Optionnel) Implémentation par tableaux cycliques

On utilise la structure suivante :

```c
#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.

Q19. Implémenter les opérations de `file.h` dans un fichier `file_tab.c` en utilisant le principe de tableau circulaire. Il pourra être pratique de toujours garder l'indice `.queue` dans l'intervalle $[\![0, \texttt{Nmax} - 1]\!]$. N'oubliez pas 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.

Application des piles

Reprenez votre implémentation des piles par tableaux redimensionnables pour cette partie. Créez un nouveau fichier C, et commencez par inclure `"pile.h"`.

On considère une chaîne moléculaire [1] , 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 :

$$\begin{aligned} \textbf{dabAcCaCBAcCcaDA} &\longmapsto \textbf{dabAcCaCBAcaDA} \\ \textbf{dabAcCaCBAcaDA} &\longmapsto \textbf{dabAaCBAcaDA} \\ \textbf{dabAaCBAcaDA} &\longmapsto \textbf{dabCBAcaDA} \end{aligned}$$

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

Q20. Donner la forme stable du polymère **abcdDCaABeEeA**.

Nous allons chercher un algorithme en $\mathcal{O}(n)$ utilisant une pile. Le principe, similaire à celui de l'algorithme vu en cours pour les mots bien-parenthésés, 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.

Q21. En pseudo-code, écrire un algorithme implémentant l'idée précédente, et l'exécuter à la main sur le polymère de la question précédente.

Q22. Ajoutez à votre implémentation des piles une opération renvoyant la taille d'une pile.

Q23. Écrire une fonction C `int taille_stable(char* polymere)` implémentant votre algorithme. Cette fonction renverra la taille de la forme stable d'un polymère. [2] Déterminer le nombre d'éléments de la forme stable du polymère écrit dans le fichier "polymere.txt" de l'archive.

1. Le problème dans cette partie est tiré de : adventofcode.com/2018/day/5.
2. **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.