TP7 : Files et application des piles

MP2I Lycée Pierre de Fermat

Vous pouvez télécharger sur Cahier de Prépa une archive pour ce TP. Le but est d'implémenter deux structures : les piles, vues en cours, et les files. Le code est déjà écrit dans le cours, vous devez donc faire ce TP en regardant le moins possible les blocs de code écrits dans le cours, le but étant de retrouver par vous-même comment implémenter les différentes opérations. Il n'y a pas rendu obligatoire pour ce TP, mais vous pourrez déposer une archive de rendu sur Cahier de Prépa avant le 30 décembre si vous souhaitez avoir une correction.

Pile

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

Question 1. 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.

Nous allons tout d'abord implémenter les piles par tableau. On rappelle le principe, qui consiste à utiliser le type struct suivant :

```
#define Nmax
struct pile {
  int nb_elem;
  int tab[Nmax];
};
```

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`.

Question 2. Implémenter dans un fichier `pile_tab.c` les fonctions de `pile.h` en utilisant l'implémentation par tableaux.

Question 3. Créez un autre fichier `test_pile.c` dans lequel vous mettrez la fonction `main`, et où vous testerez bien les différentes opérations des piles.

Passons maintenant à l'implémentation par liste chaînée. On utilise le type struct suivant :

```
typedef struct maillon {
  T elem;
  struct maillon* suivant;
} maillon_t;

struct pile {
  maillon_t* sommet;
};
```

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.

Question 4. Créez un fichier `pile_chaine.c` et implémentez la fonction de création de pile ainsi que la fonction testant si une pile est vide.

Question 5. Représenter 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.

Question 6. Faîtes de même avec le dépilage.

Question 7. Implémentez les fonctions d'affichage et de libération mémoire.

Question 8. 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

Une file représente une suite finie d'éléments de type $\boxed{\mathrm{T}}$. Une file a une tête et une queue, et ses éléments sont rangés, de telle sorte que l'on enfile les nouveaux éléments à la queue de la file, et que l'on défile les éléments par la tête de la file. Visuellement, c'est une file d'attente au supermarché : les nouveaux clients viennent se mettre à la fin de la file d'attente, et sortent de la file lorsqu'ils sont tout devant.

On pourra représenter une file en mettant des flèches indiquant le sens, de la queue vers la tête. Par exemple :

```
2 <- 3 <- 1 <- 7 <- 4
```

L'élément en tête est 2, et celui en queue est 4.
Les opérations des files sont :

— Créer une file vide

— Enfiler un nouvel élément à la queue de la file

— Défiler l'élément à la tête de la file et le renvoyer

— Déterminer si une file est vide

Par exemple, si l'on enfile 8 dans la file précédente :

```
2 <- 3 <- 1 <- 7 <- 4 <- 8
```

Si l'on défile deux fois, on obtiendra 2 puis 3, et la file deviendra :

```
1 <- 7 <- 4 <- 8
```

On dit qu'une file est "premier arrivé premier sorti" (ou FIFO pour First In First Out) car les éléments sont systématiquement défilés dans le même ordre qu'ils sont enfilés, contrairement à une pile où l'ordre est inversé.

Question 1. On considère une file F initialement vide. On applique sur F la suite d'opérations suivante :

```
Enfiler 1
Enfiler 2
Enfiler 3
Défiler
Enfiler 4
Défiler
Défiler
Enfiler 5
Défiler
Enfiler 6
```

Dessinez F après chaque opération, et pour les défilages, noter l'élément renvoyé. Vérifiez que les éléments sont bien traités selon un principe FIFO, i.e. que les éléments défilés apparaissent dans l'ordre où ils ont été enfilés.

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

Nous allons implémenter les files par listes chaînées et par tableaux.

Listes chaînées

On utilise la structure suivante :

```
1  typedef struct maillon {
2    int elem;
3    struct maillon* suivant; // de la tete vers la queue
4  } maillon_t;
5
6
7  struct file {
8    maillon_t* tete;
9    maillon_t* queue;
10 };
```

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...

Question 2. Implémentez la fonction de création de file et la fonction de vérification de file vide.

Regardons maintenant l'implémentation de l'enfilage. Comme pour les piles, on crée un nouveau maillon, et on l'ajoute au niveau au niveau de la queue de file. Dans tous les cas, la queue de la file a changé : c'est le maillon contenant le nouvel élément. Cependant, il se peut que la tête de file ait **aussi** changé, si la file était vide.

Question 3. Dessinez la file `1<-2<-3` implémentée par liste chaînée, et dessinez-la à nouveau après avoir enfilé 4. Identifiez les liens qui ont été changés dans les différentes structs.

Question 4. Faîtes de même avec la file vide.

Question 5. Déduisez-en le code de la fonction `enfiler`.

Question 6. 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.

Question 7. Implémentez la fonction `defiler`, en faisant à nouveau bien attention à ce qu'il se passe si la file est vide une fois que l'on a défilé.

Question 8. Complétez votre fichier de test pour y mettre aussi la fonction `defiler`.

Tableaux

L'implémentation par tableaux des files est assez similaire à celle des piles. 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.

Question 9. Reprendre la suite d'opérations à la question 1 de cette partie, et, à côté de chaque file dessinée, représenter le tableau des éléments si l'on avait implémenté la file par tableaux, en mettant des `?` dans les cases ne contenant pas d'éléments significatifs. On mettra une flèche à l'indice `.queue` du tableau, et une autre à l'indice `.queue` − `.nb_elem`. Que représentent ces indices ?

Question 10. Si au lieu d'avoir `Nmax` qui vaut 10000, on avait Nmax qui vaut 5, quel problème rencontre-t'on dans la question précédente ?

On peut donc se retrouver dans la situation où la file n'est pas pleine, voir même intégralement vide, mais où la structure est inutilisable car on est arrivé à la fin du tableau. Pour résoudre ce problème, on utilise le concept de **tableau circulaire**.

Dans un tableau circulaire de taille n, après la case $n-1$, on retombe sur la case 0. Autrement dit, on fait tous les calculs d'indices modulo n.

Question 11. Avec Nmax qui vaut 5, représenter la dernière étape de la question X si l'on utilise un tableau circulaire.

Question 12. 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

L'archive du TP contient deux fichiers `pile.h` et `pile_tab.c` implémentant la structure de pile par tableau redimensionnable (Cf prochain cours). Vous devez faire cette partie uniquement en vous appuyant sur la spécification de la SDA, c'est à dire en regardant seulement le header et pas l'implémentation, comme si vous utilisiez une bibliothèque comme `<string.h>` ou `<math.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 unique forme stable, et que l'ordre des réductions n'importe pas : la forme stable est unique. 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.

Question 13. Donner la forme stable du polymère **abcdDCaABeEeA**.

L'algorithme naïf de calcul de forme stable consiste à chercher des couples d'éléments pouvant réagir dans le polymère et à supprimer ce couple, jusqu'à atteindre un polymère stable. Cet algorithme prendrait un temps $\mathcal{O}(n^2)$. Nous allons chercher un algorithme plus efficace, en $\mathcal{O}(n)$, utilisant une pile, s'inspirant de l'algorithme vu en cours pour vérifier si un mot est bien parenthésé. Le principe 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.

Question 14. En pseudo-code, écrire un algorithme implémentant l'idée précédente.

Question 15. En suivant cet algorithme, écrire une fonction C `char* stable(char* fn)` qui lit dans un fichier un polymère et renvoie sa forme stable. [2] Déterminer le nombre d'éléments contenus par 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.