TP5 : Piles et files

MP2I Lycée Pierre de Fermat

Le but de cette séance est d'implémenter les deux versions des piles que l'on a vu en cours, puis de s'inspirer de ce code pour implémenter les files.

Créez pour ce TP deux dossiers : "pile/" et "file/". Pour chacune des deux SDA, tout le code écrit se trouvera dans le dossier correspondant.

Vous devez déposer votre rendu sur cahier de prépa avant mardi 6 décembre à 22h. ***Seule la partie sur les piles est à rendre***, mais vous pouvez déjà avancer sur la partie avec les files qui sera la première partie du prochain TP.

Comme d'habitude, vous devez commenter votre code : Un commentaire de documentation pour chaque fonction (pour les fonctions dont l'en-tête est dans un fichier .h, pas besoin de réécrire le commentaire dans le fichier .c), et des commentaires à l'intérieur du code lorsqu'une étape du code mérite une explication.

Piles

Téléchargez sur cahier de prépa le fichier "pile.h", qui donne les signatures des différentes opérations sur les piles.

En plus des 4 opérations vues en cours, nous allons également implémenter une fonction pour libérer la mémoire allouée pour une pile, et une fonction pour afficher le contenu d'une pile.

La fonction affiche_element déclarée dans "pile.h" sert simplement à afficher un élément de type T. Pour l'instant, le type T est le type char, et donc on définira cette fonction de la même manière dans les deux implémentations des piles :

```c
void affiche_element(T x){
  printf("%c", x);
}
```

L'idée est de rendre le code aussi modulaire que possible : si vous voulez réutiliser le code pour faire des piles de flottants, il suffira de modifier la définition de T et cette fonction d'affichage, et rien d'autre.

Question 1. Pour chaque fonction du fichier "pile.h", écrivez un commentaire de documentation.

Question 2. Écrivez dans un fichier "test.c" un programme permettant de tester les piles, en créant et modifiant une pile. Vous devez utiliser les 4 opérations, l'affichage de pile et la libération de mémoire. Ce fichier devra donc commencer par inclure "pile.h".

Question 3. Essayez de compiler votre programme avec **gcc test.c -o test_pile**. Qu'affiche le compilateur ? Pourquoi ?

Implémentation par tableau

On on utilise pour cette implémentation le type suivant :

```
#define Nmax 10000
typedef struct pile_{
  int nb_elem; // nombre actuel d'éléments dans la pile
  T* tab; // tableau permettant de stocker les éléments.
} PILE;
```

Question 4. Dans un fichier "pile_tab.c", recopiez la définition du type ci-dessus, et implémentez chacune des fonctions présentes dans l'en-tête "pile.h".

Question 5. Compilez à nouveau votre programme, cette fois ci en utilisant main.c et pile_tab.c, et testez le

Implémentation par liste chaînée

On on utilise pour cette implémentation les types suivants :

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

typedef struct pile_{
  MAILLON* sommet;
} PILE;
```

Pour vous aider à manipuler ce type, voici l'implémentation de la fonction d'affichage de pile, qui devrait vous aiguiller. Faîtes la tourner à la main sur un exemple si vous avez du mal à cerner son fonctionnement.

```
void affiche_pile(PILE* p){
  // m représente le maillon actuel que l'on regarde.
  // Initialement c'est le sommet de la pile, puis on
  // avance dans la pile au fur et à mesure en regardant
  // le maillon suivant à chaque fois.
  MAILLON* m = p->sommet;
  while (m != NULL){
    affiche_element(m->elem);
    printf(" "); //sépare les éléments d'un espace pour la lisibilité
    m = m->suivant;
  }
  printf("\n");
}
```

Question 6. Dans un fichier "pile_chaine.c", recopiez la définition du type ci-dessus, et implémentez chacune des fonctions présentes dans l'en-tête "pile.h".

Question 7. Compilez à nouveau votre programme, cette fois ci en utilisant main.c et pile_chaine.c

Application : mots bien parenthésés

Question 8. Implémentez dans un nouveau fichier l'algorithme de vérification des mots bien parenthésés vu en cours, sous la forme d'une fonction :

Entrée(s) : $s = s_0 s_1 \dots s_{n-1}$ un mot sur l'alphabet $\Sigma = \{'(', '[', ')', ']'\}$
Sortie(s) : "Oui" si s est bien parenthésé, "Non" sinon

Question 9. Écrivez un programme C qui lit un mot dans le terminal et détermine s'il est bien parenthésé

On veut maintenant générer des mots bien parenthésés. On utilise également une pile, l'idée étant d'écrire des parenthèses ouvrantes dans le mot généré, en empilant la parenthèse fermante correspondante à chaque fois sur la pile, et de dépiler aléatoirement. On garde un compteur permettant de limiter le nombre d'empilements :

Entrée(s) : $n \in \mathbb{N}^*$
Sortie(s) : Un mot s bien parenthésé de taille au plus $2n$

```
P ← pile_vide() ;
Allouer 2n + 1 cases mémoire pour res une chaîne de caractères ;
P.empiler(')') ;
res[0] ← '(' ;
i ← 1 ;                                          // case actuelle où l'on écrit
N ← n − 1 ;                          // nombre de couples de parenthèses à générer
tant que N > 0 faire
    Tirer a un booléen à pile ou face ;
    si a = 0 ou P est vide alors
        Empiler ')' ou ']' aléatoirement sur P ;
        Écrire la parenthèse ouvrante correspondante dans res[i] ;
        N ← N − 1 ;
    sinon
        Dépiler P et écrire la valeur dépilée dans res[i] ;
    i ← i + 1 ;
tant que P non vide faire
    Dépiler P et écrire la valeur dépilée dans res[i] ;
    i ← i + 1 ;
Écrire '
0' à la dernière case de res ;
retourner res
```

Question 10. Justifier brièvement que $|P|$ la taille de la pile P est un variant de boucle pour la deuxième boucle tant que.

Question 11. Est-ce-que la taille de P est un variant de boucle pour la première boucle tant que ? Pourquoi ?

Question 12. Est-ce-que N est un variant de boucle pour la première boucle tant que ? Pourquoi ?

Question 13. Donner un variant de boucle pour la première boucle tant que.

Question 14. Implémenter cet algorithme en C dans une fonction `generer_bp(int n)`

Question 15. Écrire un programme qui prend en argument un entier k, et qui génère et affiche un mot bien-parenthésé de taille k

Question 16. Avec la syntaxe de redirection des flux standards, vérifiez à l'aide du programme précédent que les mots générés sont effectivement bien-parenthésés.

Files

Cette partie sera plus simple à traiter après le cours de vendredi

On rappelle la spécification des files. Une file sert à stocker des éléments dans un ordre donné. On peut rajouter des éléments à la queue de la file (enfilage) et supprimer des éléments à la tête de la file (défilage). Les 4 opérations sont :

— Créer d'une file vide

— Enfiler un élément à la queue de la file

— Défiler l'élément en tête de file

— Déterminer si une file est vide

Question 1. Téléchargez sur cahier de prépa le fichier "file.h", qui donne les signatures des différentes opérations sur les files. Pour chaque fonction, écrivez un commentaire de documentation.

Comme pour les piles, on implémente en plus des 4 opérations des fonctions de libération de mémoire et d'affichage.

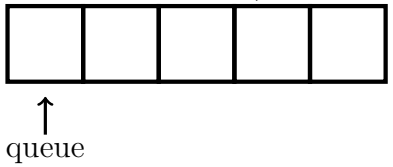
Question 2. Écrivez dans un fichier "test.c" un programme permettant de tester les files, en créant et modifiant une file. Ce fichier devra commencer par inclure "file.h".

Implémentation par tableau

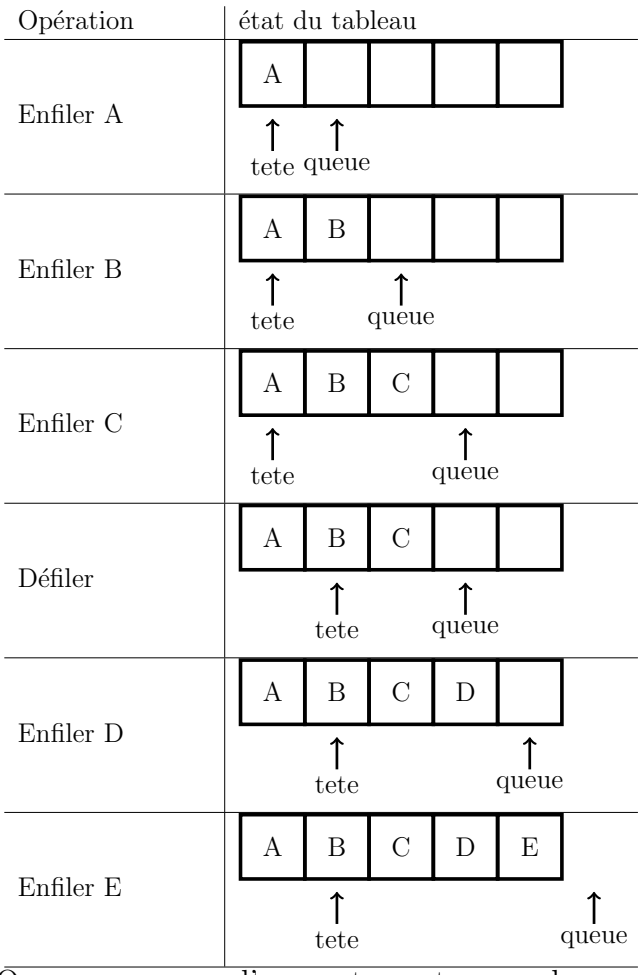
On on utilise pour cette implémentation le type suivant :

```
1 #define Nmax 10000
2 typedef struct file_{
3   int queue; // prochaine case où enfiler un élément
4   int nb_elem; // nombre actuel d'éléments dans la file
5   T* tab; // tableau permettant de stocker les éléments.
6 } FIFO;
```

L'idée est donc d'avoir un indice pour la queue, et de stocker le nombre d'éléments, ce qui permet de retrouver la position de la tête de file facilement. Par exemple, on considère une file F initialement vide. (On prend $N_{max} = 5$) :



Voici l'état du tableau après différentes opérations sur la file F :



On remarque que l'on peut se retrouver dans une situation où l'on ne peut plus enfiler parce que l'on a atteint la dernière case du tableau, même si la file contient moins de N_{max} éléments. En cours et dans le prochain TP, on verra une manière de régler ce problème avec un tableau circulaire, c'est à dire en travaillant modulo N_{max}.

Question 3. Dans un fichier "file_tab.c", recopiez la définition du type ci-dessus, et implémentez chacune des fonctions présentes dans l'en-tête "file.h".

Question 4. Compilez à nouveau votre programme, cette fois ci en utilisant main.c et pile_tab.c

Implémentation par liste doublement chaînée

Comme la file possède une tête et une queue, il est nécessaire de pouvoir parcourir la liste dans les deux sens. On utilise donc une liste ***doublement chaînée***, où chaque maillon connaît son successeur et son prédécesseur. On utilise donc les types suivants :

```
1  typedef struct maillon {
2    T elem;
3    struct maillon* suivant;
4    struct maillon* precedent;
5  } MAILLON;
6
7  typedef struct file_{
8    MAILLON* tete; //pointeur vers le prochain élément à défiler
9    MAILLON* queue; //pointeur vers le dernier élément ajouté
10 } FIFO;
```

Question 5. En vous inspirant de la fonction d'affichage d'une pile, écrivez la fonction affiche_file .

Question 6. Écrivez la fonction de libération de mémoire d'une file

Question 7. Écrivez la fonction de création d'une file vide

Commençons par implémenter la fonction d'enfilage. Pour enfiler l'élément x, La procédure est la suivante :

1. Récupérer le maillon de la queue de la file, que l'on note m. Il peut être NULL si la file est vide.
2. Créer un nouveau maillon q, contenant l'élément x, et n'ayant pas de successeur. Ce maillon sera la nouvelle queue.
3. Indiquer que le prédecesseur de q est m, et que le successeur de m est q.
4. Indiquer que la queue de la file est q. Si la file était vide, aussi indiquer que sa tête est q.

Question 8. Implémentez la fonction enfiler , et testez là.

Question 9. Une fois que cette fonction marche comme vous voulez, implémentez les autres opérations.

Question 10. Compilez à nouveau votre programme, cette fois ci en utilisant main.c et file_chaine.c