TP n°6 - Piles et files

Notions abordées

- manipulation de structures, de pointeurs vers des structures, de l'opérateur `->`
- alias de type avec `typedef`
- implémentation d'une pile avec une liste chaînée améliorée
- implémentation d'une pile avec un tableau
- implémentation d'une file avec un tableau

Toutes les fonctions doivent être commentées et testées. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.

Piles et files implémentées par tableau

Exercice 1 Implémentation de pile grâce à un tableau

Dans cet exercice on cherche à implémenter à l'aide d'un tableau la structure de donnée abstraite de pile présentée en cours. Cette implémentation ayant vocation à être utilisée dans d'autres algorithmes par la suite, on adoptera une structure de code adaptée à la compilation séparée, *i.e.* avec fichiers de déclarations, fichiers de définition et programme de test.

Question 1

Si vous voulez gagner du temps vous pouvez télécharger un squelette du code sur cahier de prépa. Celui-ci contient notamment la déclaration du type `pile_T` qui suit.

```
struct s_pileT{
  int N;// nombre maximal d'éléments
  int n;// nombre d'éléments dans la pile, <=N
  type_elem* tab; //tableau de N cases
  //dont seules les n premières sont significatives
};

typedef struct s_pileT pileT;
```

De plus vous pouvez constater qu'on a mis à part la gestion des éléments, et défini cette fois `type_elem` avec un `type_def`

Question 2
Dans le fichier `pile_tab.h`, expliquez en commentaire quelle hypothèse restrictive importante par rapport à la structure de données abstraite de pile est faite dans cette implémentation par tableau.

Question 3
Codez les fonctions suivantes pour la structure de pile définie ci-avant. *Ou laissez vous guider par le squelette du code, qui propose en plus de créer une fonction d'affichage pour faciliter les tests. Parce qu'évidemment il faut tester ces fonctions, et de préférence au fur et à mesure !*

- `pile_vide`
- `est_pile_non_vide`
- `affiche_pile`
- `sommet`
- `empile`
- `depile`

Exercice 2 Implémentation de file grâce à un tableau

On se limitera ici à des files dont le nombre d'insertion est limité afin de ne pas avoir à gérer une collection d'éléments répartis entre la fin du tableau pour le début de la file, et le début du tableau pour la fin de la file. Ainsi la structure retenue est la suivante :

```c
struct s_fileT{
  int N;// nombre emaximal d'éléments
  int deb;// indice du premier element
  int fin;// indice du dernier elem +1
  type_elem* tab; //tableau de N cases
  //dont seules celle d'indices dans [deb..fin[ sont significatives
};

typedef struct s_fileT fileT;
```

Question 1
Codez les fonctions suivantes pour la structure de file. *Ou à nouveau laissez vous guider par le squelette du code.*

- `file_vide`
- `est_file_non_vide`
- `affiche_file`
- `sommet`
- `enfile`
- `defile`

Piles et files implémentées avec des listes chaînées

Question 1 ✍
Quelles sont les complexités des opérations d'ajout et de suppression, en début et fin de liste pour les listes chaînées et les listes simplement chaînées. Synthétisez les résultats dans un tableau.

Question 2 ✍
En utilisant les listes chaînées ou doublement chaînées, quelle implémentation vous paraît la plus adaptée pour une pile ? Et pour une file ? Justifiez votre réponse en donnant la complexités des opérations élémentaires pour ces deux structures de données.

Question 3 💻
Afin de réutiliser le code de vos fonctions du précédent TP dans de nouveaux programmes, il faut le restructurer comme c'était fait dans le projet. Pour cela :

- copiez votre fichier `xxx.c` dans le dossier TP6
- copiez le deux autres fois, sous les noms `test_xxx.c` et `xxx.h`
- dans le fichier `xxx.c`, supprimez le `main` et les déclarations de type, mais ajoutez la directive `#include "xxx.h"`
- dans le fichier `xxx.h`, supprimez le `main` et les définitions de fonctions, afin d'obtenir un fichier d'en-têtes propre. *Attention, chaque signature de fonction doit être suivie d'un point-virgule*
- afin d'éviter des problèmes d'inclusion multiples par la suite, ajoutez `#ifndef XXX_H` puis `#define XXX_H` sur les deux premières lignes, puis `#endif` en fin de fichier. Cela assure que le contenu de ce fichier n'est pris en compte qu'une seule fois à la compilation d'un fichier. En effet, la première directive `#ifndef XXX_H` est à comprendre comme "if `XXX_H` is not defined", c'est-à-dire que l'on ne rentre dans le corps du "if" que si la variable `XXX_H` n'est pas encore définie, et la directive `#define XXX_H` définit cette variable dès qu'on rentre dans le "if", de sorte qu'on ne rentrera pas une deuxième fois dans ce "if", et donc qu'on ne donnera pas une deuxième fois la déclaration de nos fonctions.
- dans le fichier `test_xxx.c`, ne laissez que le `main`, mais ajoutez la directive `#include "xxx.h"`
- dans le Makefile, ajoutez les lignes de compilation pour `xxx.o` et `test_xxx` *Vous pouvez retrouver plus d'explications dans le sujet du projet, ou procéder par analogie avec ce qui est déjà présent dans le Makefile*
- compilez et exécutez votre programme test pour vérifier que tout fonctionne encore après cette restructuration.

Exercice 3 Implémentation de piles avec des listes chaînes

Question 1 💻
Dans un nouveau fichier appelé `pile_liste.h`, définissez une nouvelle structure `s_pileL` puis un nouveau type `pileL`, pour implémenter les piles avec des listes chaînes. Déclarez les fonctions attendues pour les piles d'après la structure de données abstraite vue en cours. *Ou voir question 3 de l'exercice 2.*

Question 2 💻
Dans un nouveau fichier appelé `pile_liste.c`, dans lequel vous ajouterez bien évidemment la directive `#include "pile_liste.h"`, mais aussi `#include "liste_c.h"`, donnez les implémentations des fonctions attendues pour les piles.

Exercice 4 Implémentation de files avec des listes chaînes

Question 1 💻
Dans un nouveau fichier appelé `file_liste.h`, définissez une nouvelle structure `s_fileL` puis un nouveau type `fileL`, pour implémenter les files avec des listes chaînes. Déclarez les fonctions attendues pour les files d'après la structure de données abstraite vue en cours. *Ou voir question 1 de l'exercice 3.*

Question 2 💻
Dans un nouveau fichier appelé `file_liste.c`, dans lequel vous ajouterez bien évidemment la directive `#include "file_liste.h"`, mais aussi `#include "liste_c.h"`, donnez les implémentations des fonctions attendues pour les files.

Applications

Exercice 5 Détecter un mauvais parenthésage

On dit qu'une expression est **bien parenthésée** si chaque parenthèse ouvrante est suivie d'une parenthèse fermante, (resp. chaque parenthèse fermante est précédée d'une parenthèse ouvrante), telle que l'expression entre les deux est elle-même bien parenthésée, étant entendu qu'une expression sans parenthèse est bien parenthésée. Le but de cet exercice est de décider si une expression est bien parenthésée.

Question 1 ✍
Donnez quelques exemples d'expressions bien parenthésées, mal parenthésées.

Question 2 ✍
Que pouvez vous dire du nombre de parenthèses ouvrantes et du nombre de parenthèses fermantes dans une expression bien parenthésée ? Vous indiquez là une condition nécessaire, est-ce une condition suffisante, c'est-à-dire que cette relation entre les nombres de parenthèses ouvrantes et fermantes assure que l'expression est bien parenthésée ?

Question 3 ✍
Que pouvez vous dire du nombre de parenthèses ouvrantes et du nombre de parenthèses fermantes

dans un préfixe d'une expression bien parenthésée ? Vous indiquez là une condition nécessaire, est-ce une condition suffisante, c'est-à-dire que cette relation entre les nombres de parenthèses ouvrantes et fermantes dans n'importe quel préfixe assure que l'expression est bien parenthésée ?

On s'intéresse maintenant à une généralisation de ce problème, où l'on a plusieurs symboles ouvrants (par exemple `(`, `[` et `{`) associés chacun à un symbole fermant `)`, `]` et `}` dans l'exemple).

Question 4 ✍
Est-ce qu'une expression à la fois bien parenthésée pour les parenthèses, *i.e.* pour `'('` et `')'` et pour les crochets, *i.e.* pour `'['` et `']'` est globalement bien parenthésée ? Si oui le démonter, sinon donner un contre-exemple

Question 5 ✍
Si, alors qu'on lit une expression bien parenthésée de gauche à droite, on trouve une parenthèse fermante (resp. un crochet fermant, une accolade fermante) quelle parenthèse (resp. crochet, accolade) ferme-t-il ? La première ouverte qu'on a trouvée ? La dernière ? Aucune en particulier ?
Et plus généralement, lorsqu'on trouve un symbole fermant, quel symbole ouvrant ferme-t-il ? Le premier trouvé, le dernier trouvé, ou aucun en particulier ?
En déduire quelle structure de données il pourrait être intéressant d'utiliser pour détecter des anomalies de parenthésage .

Question 6 💻
Codez en C une fonction `est_valide` qui teste si une chaîne de caractères est bien parenthésée pour les caractères `'('`, `'('`, `'['`, `']'` `'{'` et `'}'`.
Le but est bien entendu de réutiliser les structures précédemment codées, donc pensez à inclure les bons ***.h****, et à compiler en conséquence. De plus il peut être pratique de créer une fonction qui teste si un caractère est ouvrant, et qui si oui retourne un entier correspondant au type de symbole, et une autre pour les symboles fermant. Ainsi les deux fonctions renverraient le même entier si les deux symboles correspondent.*