TP 10 : Piles et files de cartes

Notions : Type abstrait, Files, Piles, Structures, Allocation dynamique, pointeurs

1 Structures de données

L'objectif de cette séance est de définir les fonctions de base sur les files et les piles de cartes qui seront utilisées lors de la simulation de la bataille. Il permet aussi de voir comment réutiliser du code déjà existant.

Le fichier `element.h` définit le type réel des éléments des listes, piles et files de carte est est le suivant ;

```
#ifndef _ELEMENT
#define _ELEMENT

typedef enum { TREFLE,CARREAU,COEUR,PIQUE} COULEUR;

typedef struct { int rang; char visible; COULEUR couleur; } CARTE;

typedef CARTE element_t;

void element_print (CARTE e);
void element_init (CARTE* e);
CARTE element_empty();
int element_is_empty(CARTE e);
int element_equal(CARTE*, CARTE*);
#endif
```

Vous avez déjà défini le type `list_t` et les fonctions de base sur les listes lors du TP précédent. Votre fichier `list.h` décrivant les types et prototypes de vos listes ressemble à ceci

```
// Fichier list.h
#ifndef _LIST_H
#define _LIST_H
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "element.h"

// 1. Définition des types maillon (link_t) et liste (list_t)
typedef struct _link {
  element_t val; /* un element de la liste*/
  struct _link *next; /* l'adresse du maillon suivant */
} link_t, *list_t;

// 2. Protoype des fonctions realisant les opérations essentielles du type abstrait Liste
list_t list_new() ;
int list_is_empty(list_t l);
element_t list_first(list_t l);
list_t list_next(list_t l);
list_t list_add_first(element_t e, list_t l);
list_t list_del_first(list_t l);
int list_length(list_t l);
list_t list_find(element_t e, list_t l);
```

```
void list_print(list_t l);
list_t list_delete(list_t l);
int list_count(element_t e, list_t l) ;
list_t list_add_last(element_t e, list_t l) ;
list_t list_concat(list_t l1, list_t l2);
list_t list_copy(list_t l);
list_t list_remove_n(int n, list_t l) ;
#endif
```

2 Piles de cartes

Dans notre cas, les piles sont identiques aux listes chainées dynamiques, que vous avez déjà implémentées, en limitant les opérations à empiler, dépiler. Vous définissez le type `lifo_t` à partir du type `list_t` en le rendant similaire à l'aide de l'instruction `typedef`.

Votre fichier `lifo.h` doit ressembler à ceci :

```
// fichier lifo.h pour l'implantation du TAD pile au moyen d'une liste chaînée dynamique
#ifndef LIFO_H_
#define LIFO_H_

#include "list.h"  // On importe le TAD liste chaînée dynamique

// 1. declaration du type lifo_t.
// Il suffit de déclarer que "une pile est une liste" !
typedef list_t lifo_t ;

// 2. prototype des fonctions de l'API du TAD Pile
// Crée et retourne un pile vide ou NULL en cas d'erreur
lifo_t lifo_new();
// Supprime le premier element de la pile
lifo_t lifo_del_first(lifo_t p);
// Détruit la pile et retourne une pile vide
lifo_t lifo_delete(lifo_t stack);
// Retourne 1 si la pile stack est vide, 0 sinon
int lifo_is_empty(lifo_t stack);
// Ajoute l'élément e à la pile stack et retourne la nouvelle pile
lifo_t lifo_push(element_t e, lifo_t stack);
// Retourne l'élément en tête de pile (sans l'enlever de la pile)
// PRECONDITION : la pile stack ne doit pas être vide
element_t lifo_peek(lifo_t stack);
// Enlève l'élément en tête de la pile, et retourne cet élément
// PRECONDITION : la pile pointee par p_stack ne doit pas etre vide
element_t lifo_pop(lifo_t * p_stack);
// Supprime tous les maillons
lifo_t lifo_delete(lifo_t stack);
// Affiche la pile
void lifo_print(lifo_t stack);
// Nombre d'elements de la pile
int lifo_length(lifo_t stack);
#endif
```

2.1 Travail à réaliser sur les Piles

1. Télécharger le fichier `tp9.zip` sur le site et le décompresser. Il contient les fichiers d'implémentation des listes `list.h`, `list.c`, des piles `lifo.h`, `lifo.c`, des files `fifo.h`, `fifo.c` ainsi que des fichiers de tests `test_lifo1.c`, `test_lifo2.c`, `test_lifo2.c`, `test_fifo1.c`, `test_fifo2.c`, `test_fifo3.c` et un fichier `Makefile`.
2. En utilisant au maximum les fonctions réalisées la séance précédente sur les listes, compléter le fichier `lifo.c` en complétant le code des fonctions pour les piles :

— `lifo_t lifo_push(element_t e, lifo_t stack);` Ajoute une carte a la pile

— `void lifo_print(lifo_t stack);`

— `element_t lifo_peek(lifo_t stack);` Renvoie la carte du sommet SANS la supprimer. La pile ne doit pas être vide.

— `element_t lifo_pop(lifo_t* ap);` Renvoie la carte du sommet ET la supprime de la pile. La pile n'est pas vide.

— `lifo_t lifo_delete(lifo_t p);` Libère la memoire allouée à tous les éléments de la pile et retourne une pile vide

3. Compiler le fichier `test_lifo1.c` à l'aide de la commande `make test_lifo1` pour tester vos fonctions. Ce programme teste vos fonctions avec des cas simples. Vérifier que les résultats soient corrects en vérifiant la sémantique du programme de test et en comparant la sortie écran de votre execution avec celle contenue dans le fichier `resultatsTestLifo1.txt`. Vous pouvez utiliser la commande `diff` pour cela.
4. Compiler le fichier `test_lifo2.c` à l'aide de la commande `make test_lifo2` pour tester vos fonctions. Ce programme teste les fonctions `lifo_push` et `lifo_pop` à l'aide d'un tirage aléatoire. Il peut ainsi mettre en évidence des cas que vous n'auriez pas prévu. Vérifier que les résultats soient corrects en comparant la sortie écran de votre execution avec celle contenue dans le fichier `resultatsTestLifo2.txt`.
5. Compiler le fichier `test_lifo3.c` à l'aide de la commande `make test_lifo3` pour tester interactivement vos fonctions.

3 Files de cartes

Dans notre cas, les files sont implémentées sous forme de listes chainées dynamiques, **circulaires**

3.1 Travail à réaliser sur les Files

1. En utilisant la représentation par une **liste circulaire**, compléter le fichier `fifo.c` avec le code des fonctions utiles pour les files :
   — `fifo_t fifo_enqueue(element_t , fifo_t );` Ajoute une carte à la file par la queue.
   — `void fifo_print(fifo_t );`
   — `element_t fifo_peek(fifo_t f);` retourne la premiere carte de la file SANS la supprimer.
   — `fifo_t fifo_del_head(fifo_t f);` Supprime la premiere carte de la file et retourne la nouvelle file obtenue
   — `element_t fifo_dequeue(fifo_t* );` Retourne la carte en tete de file ET la supprime de la file en utilisant les 2 fonctions précédentes
2. Compiler le fichier `test_fifo1.c,` à l'aide de la commande `make test_fifo1` pour tester vos fonctions. Ce programme teste les fonctions `fifo_enqueue` et `fifo_dequeue` sur des cas assez simples. Vérifier que les résultats soient corrects avec la sémantique du code fourni et comparer la sortie écran de votre execution avec celle contenue dans le fichier `resultatsTestFifo1.txt`.
3. Compiler le fichier `test_fifo2.c,` compiler à l'aide de la commande `make test_fifo2` pour tester vos fonctions. Ce programme teste les fonctions `fifo_enqueue` et `fifo_dequeue` à l'aide d'un tirage aléatoire. Il peut ainsi mettre en evidence des cas que vous n'auriez pas prévu. Vérifier que les résultats soient corrects en comparant la sortie écran de votre execution avec celle contenue dans le fichier `resultatsTestFifo2.txt`.

4 Distribution de cartes

Dans l'utilisation que nous allons faire de ces piles et files (jeu de cartes tel que celui de la bataille), la main d'un joueur est représenté par une file car le joueur prend les cartes sur le dessus de son jeu

et remet les cartes gagnées par le dessous de son jeu. Le fichier `distribution.c` contient une fonction `int distribution(fifo_t* aj1, fifo_t *aj2, int alea, int nbcarteparcouleur)` qui permet de distribuer toutes les cartes d'un jeu de cartes entre 2 joueurs.

- Les 2 premiers paramètres `aj1` et `aj2` sont des pointeurs sur les files représentant les jeux de 2 joueurs.
- Le paramètre `nbcarteparcouleur` indique le nombre de cartes par couleur choisie (inférieur à 14, supérieur à 0). Ce paramètre est utile pour vérifier plus facilement vos programmes avec un nombre de cartes limité.
- Le paramètre `alea` sert à initialiser le générateur pseudo aléatoire. Un générateur pseudo aléatoire est en réalité une suite $U_n = f(U_{n-1})$ difficile à prévoir et de distribution uniforme (toutes les valeurs sont tirées avec la meme probabilité). Ce paramètre fixe la valeur U_0 de cette suite. On aura donc des distributions de cartes toujours identiques pour une valeur donnée de U_0, mais différentes pour des valeurs de U_0 différentes.

4.1 Travail à réaliser

1. Compiler le fichier `test_file3.c` à l'aide de la commande `make test_fifo3` pour tester la distribution. Vérifier les résultats.
2. Le programme ne se termine pas proprement car les files n'ont pas été libérées.
   Ecrire une fonction `fifo_t fifo_delete(fifo_t f)` et modifier le programme pour qu'il libère les jeux à la fin. Vérifier que la mémoire est bien libérée à l'aide de la commande `valgrind ./test_fifo2`.