STRUCTURE DES DONNEES

RAPPORT:

IMPLÉMENTATION D'UNE PILE PAR UNE LISTE DE JOUEURS

Réalisé par:

- Amal Aderdour
- Laila Hamza
- Asma Elfahim
- Asma Asaad
- Maria Ejjbiri

Encadré par:

- Mr Yousse Es-saady

Ecole Supérieure d'Education et de Formation d'Agadir
Université IBN ZOHR
Année universitaire: 2019 / 2020

SOMMAIRE

INTRODUCTION

Dans le cadre de notre deuxième année en **E**cole **S**upérieure de **F**ormation et d'**E**ducation, nous avons eu pour tâche la réalisation d'un programme en **langage C**.
Pour poursuivre la découverte des différentes structures de données, nous allons maintenant nous attarder sur les **piles**.
Le principe des piles se base sur l'empilage et le dépilage en utilisant l'algorithme **LIFO: Last In First Out**, qui sera mieux expliqué ci-dessous.
Pour ce faire, on a créer une équipe de joueurs pour implémenter la pile, en utilisant le langage C.
La première chose à faire est de créer les structures suivantes:

- Pile
- Joueur

Ensuite, on a écrit un ensemble de fonctions pour faciliter la gestion de l'équipe.
Après des nombreuses analyses, nous avons abouti à un code, qui sera bien détaillé et expliqué dans les chapitres suivants.

Chapitre 1 : Créations des structures

1. Structure d'un joueur (Player) :

Pour chaque joueur de l'équipe, on va définir un nouveau type Player dans une structure Player et qui contiendra trois éléments :

```
12      typedef struct Player
13          {
14              char nom[25];
15              char prenom[25];
16              int numero;
17          }Player;
```

- ➔ Un nom
- ➔ Un prénom
- ➔ Un numéro d'indentification

2. Structure d'une pile (Stack) :

Nous avons créé dans cette structure un élément d'une pile (Stack), qui contient :

- ➔ Une donnée de type joueur
- ➔ Un pointeur vers un élément du même type appelé Next

C'est ce qui permet de lier les éléments les uns aux autres : chaque élément « sait » où se trouve l'élément suivant en mémoire.

```
23      typedef struct StackElement
24          {
25              Player P;
26              struct StackElement *next;
27          }StackElement, *Stack;
```

Chapitre 2 : Les fonctions :

1. Création d'une pile :

Cette fonction permet de créer une nouvelle pile de type **Stack** et retourne une pile vide.

```
7    Stack new_stack(void)
8        {
9            return NULL;// Retourne une Pile vide
10       }
```

2. Vérifier si la pile est vide :

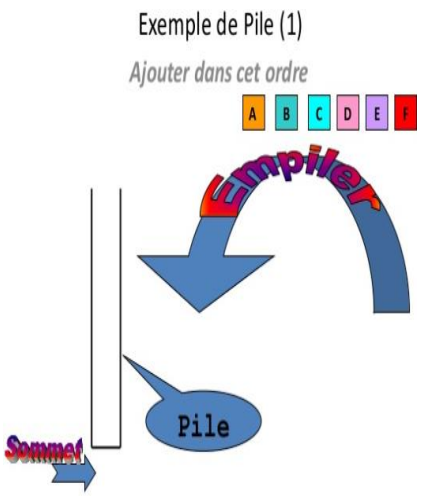
Notre fonction **Is_empty_stack** permet de vérifier si une pile est vide.

Elle doit prendre en paramètre la structure de contrôle de la pile (de type Stack) et retourner un booléen (vrai (1) si la Pile est vide, faux (0) sinon) qu'on a défini dans une énumération « **Bool** ».


```
2    typedef enum
3        {
4            false, //0
5            true //1
6        }Bool;
```

```
15   Bool is_empty_stack(Stack st)
16       {
17           if(st == NULL)
18           return true;
19           return false;
20       }
```

3. Empiler :



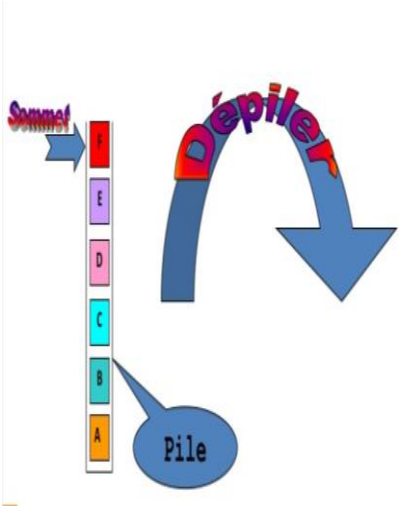
La chose la plus importante qu'il faut prendre en considération est que l'insertion dans une pile se fait toujours en haut (au début). Ce qui explique que le premier élément saisi est le dernier dans la pile.

Pour ce faire, on commence par créer un nouvel élément de type **Stack** et on lui alloue un espace dans la mémoire, au cas où l'allocation n'est pas réussie, un message d'alerte s'affichera.

Ensuite, on passe à préparer remplissage de la pile avec les informations du joueur (la fonction **strcpy** est utilisée pour copier le contenu du nom et prénom, étant donnés des chaînes de caractères). Et on finit par lier le nouvel élément avec la pile (la suite de la pile sera elle-même) et retourner la pile avec l'élément ajouté.

```
25  Stack push_stack(Stack st, Player joueur)
26      {
27          StackElement *element;
28          element = malloc(sizeof(*element));
29          if(element == NULL)
30              {
31                  fprintf(stderr, "Probleme allocation dynamique.\n");
32                  exit(EXIT_FAILURE);
33              }
34          strcpy(element->P.nom, joueur.nom);
35          strcpy(element->P.prenom, joueur.prenom);
36          element->P.numero = joueur.numero;
37          element->next = st;
38          return element;
39      }
```

4. Dépiler :



Le rôle de la fonction **Pop_Stack** est de supprimer l'élément tout en haut de la pile. Mais elle doit aussi retourner l'élément qu'elle dépile, c'est-à-dire dans notre cas le joueur qui était stocké en haut de la pile. Toujours en respectant l'algorithme **LIFO**.

C'est comme cela que l'on accède aux éléments d'une pile : en les enlevant un à un. On ne parcourt pas la pile pour aller y chercher le second ou le troisième élément. On demande toujours à récupérer le premier.

Elle doit prendre en paramètre la structure de contrôle de la pile (de type Stack).

Voici l'état de la liste avant l'appel de la fonction :

```
44  Stack pop_stack(Stack st)
45      {
46          StackElement *element;
47          if(is_empty_stack(st))
48          return new_stack();
49          element = st->next;
50          free(st);
51          return element;
52      }
```

➔ Le pointeur element représente l'argument de la fonction et donc l'élément à supprimer
➔ Vérifie si la Pile est vide
➔ Sauvegarde de l'élément précédent celui à supprimer grâce au pointeur element=st->next
➔ Libération de la mémoire
➔ Retourner les éléments sauf le joueur retiré

5. Afficher la pile :

Pour afficher les éléments de la pile, on vérifie tout d'abord si la pile n'est pas vide. Si c'est le cas, le parcours commence du premier élément et on affiche le contenu de chaque élément de la pile (les informations du joueur). On se sert du pointeur ***suivant*** pour passer à l'élément qui suit à chaque fois.

Valeur initiale	st
Instruction à répéter	st->P.nom, st->P.prenom, st->P.numero. st->next
Condition d'arrêt	st->next==NULL

```
57  void print_stack(Stack st)
58      {
59          if(is_empty_stack(st))
60              {
61                  return;
62              }
63          while(!is_empty_stack(st))
64              {
65                  printf("[%s %s- numero %d]\n", st->P.nom, st->P.prenom,st->P.numero);
66                  st = st->next;
67              }
68      }
```

6. Afficher le sommet de la pile :

D'après sa définition, le sommet de la pile est le dernier élément empilé. Cette fonction retournera un élément P de typer **player**, c'est-à-dire un joueur avec ses informations.

Premièrement on teste : si la pile est vide, on arrête immédiatement le programme en faisant appel à exit ().

Sinon on affiche le contenu du premier élément grâce à **return st -> P**.

```
73    Player top_stack(Stack st)
74        {
75            if(is_empty_stack(st))
76                {
77                    printf("Aucun sommet, la Pile est vide.\n");
78                    exit(EXIT_FAILURE);
79                }
80            return st->P;
81        }
```

7. La longueur de la pile :

Cette fonction retourne la hauteur (longueur) de la pile. Elle prend en paramètre le premier élément de la pile (st) de type **Stack**.

On initialise la langueur à **0** et on commence par parcourir la pile à partir de son premier élément jusqu'au dernier, grâce aux pointeurs (next). A chaque fois que st prend la valeur de successeur on y ajoute 1 a la langueur.

.

Valeur initiale	st
Instruction à répéter	st->next longueur++
Condition d'arrêt	st->next==NULL

```
86  int stack_length(Stack st)
87  {
88      int length = 0;
89      while(!is_empty_stack(st))
90          {
91              length++;
92              st = st->next;
93          }
94      return length;
95  }
```

8. Supprimer tous les éléments de la liste :

Cette fonction a pour but de vider la pile. Pour ce faire, on va tester :

- ∇ si la pile est vide, on return la fonction **new_stack()** (Retourne une Pile vide).
- ∇ Sinon. On supprime l'élément tout en haut de la pile, c'est-à-dire le sommet.

Le processus se répète jusqu'à ce que la pile soit complétement vide.

```
100 Stack clear_stack(Stack st)
101     {
102         while(!is_empty_stack(st))
103         st = pop_stack(st);
104         return new_stack();
105     }
```

Les prototypes :

- ✓ Stack new_stack(void) ;
- ✓ Bool is_empty_stack (Stack st) ;
- ✓ void print_stack (Stack st) ;
- ✓ Stack push_stack (Stack st, Player joueur) ;
- ✓ Stack pop_stack (Stack st) ;
- ✓ Player top_stack (Stack st) ;
- ✓ int stack_length (Stack st) ;
- ✓ Stack clear_stack (Stack st) ;

Conclusion :

Le programme a été organisé en trois fichiers : « **main.c** », « **MesFonctions.h** » et « **MesFonctions.c** ». Vous trouverez, ci-joint, le code principal dans un fichier zip « **Piles** ».

A l'aide de ce projet nous avons pu comprendre et expérimenter les différentes étapes de l'implémentation d'une pile en commençant par l'analyse des différentes fonctions dont on aura besoin pour enrichir le code.

De plus la programmation nous a permis d'améliorer nos connaissances du langage C.

Vu les circonstances qu'on est entrain de vivre ces derniers temps, nous avons fait de notre mieux pour pouvoir concevoir ce programme.

Cependant, malgré les nombreuses fonctions que nous avons créées, nous ne pensons pas avoir épuisé la question, et nous espérons sincèrement susciter d'autres fonctions pour mieux maîtriser les piles.