Cours: ASD 2 & Complexité

Chapitre 5: Les Piles

Réalisé par:
Dr. Sakka Rouis Taoufik

I. Introduction

On appelle pile un ensemble de données, de taille variables, éventuellement nul, sur lequel on peut effectuer les opérations suivantes :

- **creer_pile** : elle permet de créer une pile vide
- **pile_vide** : elle permet de voir est ce qu'une pile est vide ou non ?
- **empiler** : elle permet d'ajouter un élément de type T à la pile
- **depiler** : elle permet de supprimer un élément de la pile
- **dernier** : elle retourne (ou rend) le dernier élément empilé et non encore dépilé.

I. Introduction

OPERATIONS ILLEGALES :

Certaines opérations définies sur une SD exigent des préconditions: on ne peut pas appliquer systématiquement ces opérations sur une SD.

Illustration :

dépiler exige que la pile soit non vide de même pour l'opération dernier.

II. Propriétés

Les opérations définies sur la SD pile obéissent aux propriétés suivantes :

- (P1) creer_pile permet la création d'une pile vide.
- (P2) si on ajoute un élément (en utilisant empiler) à une pile alors la pile résultante est non vide.
- (P3) un empilement suivi immédiatement d'un dépilement laisse la pile initiale inchangée.
- (P4) On récupère les éléments d'une pile dans l'ordre inverse ou on les a mis (empiler).
- (P5) Un dépilement suivi immédiatement de l'empilement de l'élément dépilé laisse la pile inchangée.

Ch 5: Les Piles

II. Propriétés

REMARQUE : ces cinq propriétés permettent de cerner la sémantique (comportement ou rôle) des opérations applicables sur la SD pile.

En conclusion : la SD pile obéit à la loi LIFO (Last In, First Out) ou encore DRPS (Dernier Rentré, Premier Sortie).

5

Ch 5: Les Piles

III. Représentation physique

Pour pouvoir matérialiser (concrétiser, réaliser ou implémenter) une SD on distingue deux types de représentation :

- **Représentation chaînée** : les éléments d'une SD sont placés à des endroits quelconques (bien entendu dans la MC) mais chaînés (ou reliés).

  Idée : pointeur.

- **Représentation contiguë** : les éléments d'un SD sont rangés (placés) dans un espace contiguë.

  Idée : tableau.

6

Ch 5: Les Piles

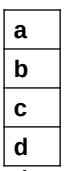
III. Représentation physique

3.1 Représentation chaînée

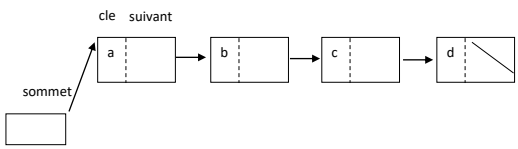
Utilisation des pointeurs pour relier explicitement les éléments formant la SD.

La représentation chaînée comporte plusieurs nœuds. Chaque nœud regroupe les champs suivants :

- Le champ « cle » permet de mémoriser un élément de la pile.
- Le champ « suivant » permet de pointer sur l'élément précédemment empilé.
- La variable « sommet » permet de pointer ou de repérer l'élément au sommet de la pile.



Représentation abstraite



7

Ch 5: Les Piles

III. Représentation physique

3.1 Représentation chaînée

```
struct element {
   int cle ;
   struct element * suivant ;
} ;
struct element * sommet ;
```

- Les éléments formant la pile sont créées grâce à empiler qui fait appel à **malloc**. Cependant, les éléments de la pile peuvent être supprimés grâce à dépiler qui fait appel à **free**.
- Les éléments d'un SD matérialisée par une représentation chaînée résident dans un espace appelé **heap** ou **tas**.

8

Ch 5: Les Piles

III. Représentation physique

3.1 Représentation chaînée

Rq:

- Variable globales résident dans un espace mémoire appelé **segment de donnée**
  → **durée de vie :** celle du programme
- Variable locales résident dans un espace mémoire appelé **stock empile**
  → **durée de vie :** celle de son sous programmes
- Variable dynamique résident dans un espace mémoire appelée **Heap** (ou **tas**)
  → **durée de vie :** création →malloc
  suppression →free

→Pour pouvoir repérer une variable dynamique, nous avons besoin de son adresse souvent stockée, dans une variable de type pointeur.

9

Ch 5: Les Piles

III. Représentation physique

3.2 Représentation contiguë

Pour pouvoir représenter la structure de données Pile d'une façon contiguë, on fait appel à la notion du tableau.

Puisque le nombre d'éléments d'un tableau doit être fixé avant l'utilisation, on aura deux parties :

- **Partie utilisée** : elle est comprise entre 0 et sommet.
- **Partie non utilisée** : elle est comprise entre sommet+1 et n-1.
- Avec sommet est un indice compris entre -1 et n-1.

→ sommet == -1 indique que la pile est vide

a
b
c
d

Sommet = 3

d	0
c	1
b	2
a	3
	n-1

utilisé

non utilisée

10

Ch 5: Les Piles

III. Représentation physique

3.2 Représentation contiguë

Solution 1 en C	Solution 2 en C
#define n 100 int cle [n] ; /*tableau pour mémoriser des entiers*/ int sommet ;	# define n 100 struct pile { int cle[n] ; int sommet ; } ; struct pile p ;

- **Critique :** la solution 1 est mauvaise car elle dispose des éléments étroitement liés (ici clé et sommet). D'où la solution 2.

11

Ch 5: Les Piles

III. Représentation physique

3.3 Évaluation

- Quels sont les problèmes posés par la représentation contiguë ?

Estimation de n : (trop grand / trop petit)

➔ Cas n trop grand➔ perte mémoire

➔ Cas n trop petit➔ risque que des demandes d'adjonction (ajout) ne sont pas satisfaire (cas d'empiler sur les SD pile)

12

Ch 5: Les Piles

III. Représentation physique

3.3 Évaluation

- Quels sont les problèmes posés par la représentation chaînée ?

Hypothèse : La pile contient à un instant donné n éléments

La représentation chaînée occupe plus de mémoire que la représentation contiguë ceci est justifié par le coût mémoire de pointeurs présents dans la représentation chaînée .

N=100 et la taille d'un pointeur est deux octets

- Représentation contiguë : espace occupé est de taille =100*sizeof(int)=100*2=200 octets
- Représentation chaînée : espace occupé est de taille =100*2+100*2=400 octets

13

Ch 5: Les Piles

IV. Matérialisation de la SD Pile

- Un programme écrit en C peut être reparti physiquement sur plusieurs fichiers. On distingue :
- Les fichiers ayant l'extension (**.h**)
- Les fichiers ayant l'extension (**.c**)
- Un fichier ayant l'extension .h regroupe des déclarations : constantes, types, variables et sous-programmes.
- Un fichier ayant l'extension **.c** regroupe des déclarations (constantes, types, variables, sous-programmes) et des réalisations des sous-programmes.

Rq. Un fichier ayant l'extension **.c** peut ne pas comporter les sous-programmes main. Dans ce cas, ce fichier ne peut pas être exécuté. Afin que cette application soit exécutable, il faut qu'un parmi les fichiers d'extension **.c** englobe main.

14

Ch 5: Les Piles

IV. Matérialisation de la SD Pile

- On va proposer une **interface pile (pile.h)** regroupe les services exportées par cette SD.
- Chaque service correspond à une opération applicable sur la SD (ici la SD Pile). Et il est fourni sous forme d'un sous-programme.

- Opérations applicable sur la SD Pile définies:

  - Opération de création :(procédure ou fonction)
  - Opération de modification : procédure
  - Opération de consultation : fonction

15

Ch 5: Les Piles

IV. Matérialisation de la SD Pile

Matérialisation de la SD Pile : → **Objet abstrait (OA)**

Matérialisation de la SD Pile : → **Type de données Abstrait (TDA)**

- **Objet abstrait (OA)➔** un seul exemplaire (ici une seul pile)→ unique →implicite.

- **Type de données Abstrait (TDA) ➔** plusieurs exemplaires → il faut mentionner ou rendre explicite l'exemplaire courant.

16

IV. Matérialisation de la SD Pile

4.1 Structure de données Pile comme OA

- En supposant que les éléments de la Pile sont des entiers, celle-ci se déclare de la façon suivante :

```
/*Fichier pile.h*/

    void ceer_pile (void) ;  /*  void creer_pile( )*/

     /*opérations de consultation*/
    unsigned vide (void) ;  /* 1 si la pile est vide sinon 0*/
    int dernier (void) ;

    /*opérations de modification*/
    void empiler (int) ;  /* Le paramètre est la valeur à empiler*/
    void depiler(void) ;
```


IV. Matérialisation de la SD Pile

4.1 Structure de données Pile comme OA

```
/*Fichier pile.c*/
#include <stdio.h>/* librairie spécifique définie par le programmeur*/
#include <alloc.h>/* librairie spécifique définie par le programmeur*/
#include <assert.h>/* librairie spécifique définie par le programmeur*/
#include " pile.h"  /*pile.h est censée être stocké dans le répertoire courant
                        appelé répertoire de travail*/
 /*représentation physique*/
struct element {
        int cle ;
        struct element * suivant ;
} ;
 static struct element *sommet ;
/*Pour restreindre la portée de cette variable à ce fichier, il faut utiliser le
mot réservé static on dit que la variable sommet n'est pas destiné à
l'exportation */
```

IV. Matérialisation de la SD Pile

4.1 Structure de données Pile comme OA

```
void creer_pile (void) {
      sommet=NULL ;
}
unsigned vide (void) {
     return(sommet==NULL) ;
}
```

```
int dernier (void){
     assert( ! vide()) ;
     return (sommet →cle) ;
}
```

```
void empiler(int info){
    struct element *p ;
    p= (struct element*) malloc (sizeof (struct element)) ;
    p→cle=info;
    p→suivant=sommet;
    sommet=p; /*mettre à jour du sommet*/
}
```

19

IV. Matérialisation de la SD Pile

4.1 Structure de données Pile comme OA

```
void depiler (void) {
      struct element *p ;
      assert( !vide()) ;
      p=sommet ;
      sommet = sommet →suivant ;
      free(p) ;
}
```

20

IV. Matérialisation de la SD Pile

4.1 Structure de données Pile comme OA

/* Exemple d'utilisation fichier exemple.c */

```
#include <stdio.h>
#include  "pile.h"        /*on compte utilise des services offerts par
                          l'interface pile.h */

void main(void) {
    unsigned i;
    creer_pile();
    assert(vide());
    for(i=1;i<=10;i++)
          empiler(i);
    assert(!vide());
    for(i=1;i<=10;i++) {
       printf("%d \n", dernier());
       depiler();
    }
    assert (vide());

}
```

21

IV. Matérialisation de la SD Pile

4.2 Structure de données Pile comme TDA

Dans cette partie, on va concrétiser la SD pile sous forme d'un **Type de données Abstrait** (TDA) capable de gérer plusieurs exemplaires de la SD pile et non pas un seul exemplaire (Objet Abstrait voir 4.1).

22

IV. Matérialisation de la SD Pile
4.2 Structure de données Pile comme TDA

```
/*Partie interface :  fichier pile.h */
/*représentation physique*/
struct element {
        int cle ;
        struct element * suivant ;
} ;
/*opérations  ou services exporté*/
struct element* creer_pile(void) ;
unsigned vide (struct element*) ;
int dernier  (struct element*) ;
void empiler  (int, struct element**) ;
void depiler  (struct element**) ;
```


IV. Matérialisation de la SD Pile
4.2 Structure de données Pile comme TDA

/*Partie implémentation : fichier pile .c */

```
#include <stdio.h>
#include <alloc.h>
#include <assert.h>
#include "pile.h"

struct element * creer_pile (void){
   return NULL;
}

unsigned vide (struct element *p){
   return (p==NULL);
}

int dernier (struct element *p) {
   assert( !vide(p)) ;
   return (p→cle) ;
}
```

IV. Matérialisation de la SD Pile
4.2 Structure de données Pile comme TDA

/*Partie implémentation : fichier pile .c */

```
void empiler (int info, struct element ** p) {
    struct element* q ;
    q=(struct element*) malloc (sizeof (struct element)) ;
    q→cle=info;
    q→suivant=*p;
    *mise à jour*/
    *p=q;
}
```


IV. Matérialisation de la SD Pile
4.2 Structure de données Pile comme TDA

/*Partie implémentation : fichier pile .c */

```
void depiler (struct element **p) {
     struct element *q;
     assert (!vide(*p));
     q=*p ;
     *p=q → suivant ;     /* <==>  *p=(*p) → suivant ;   */
     free(q) ;
}
```

IV. Matérialisation de la SD Pile

4.2 Structure de données Pile comme TDA

/* Exemple d'utilisation : fichier test.c */

```
#include<stdio.h>
#include "pile.h"
void main(void)  {
struct element *p1;
struct element *p2;
unsigned i;
p1=creer pile();
for(i=1;i<=10;i++)
     empiler(i, &p1);
p2=creer pile();
for(i=11;i<=20;i++)
     empiler(i, &p2);
/*affichage de p1 et p2*/
for(i=1 ;i<=10 ;i++) {
    printf(" %d\t ", dernier (p1));
    printf(" %d\n ", dernier (p2));
    depiler(&p1);
    depiler(&p2);
}
if(vide(p1)&&vide(p2))
        printf(" quelle joie!! ");
else
        printf(" problème!?!? ");
}
```

27

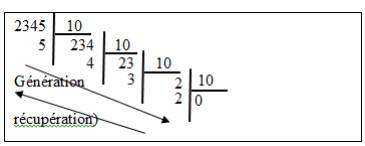
V. Exercices d'application

Exercice 1 :

Ecrire un algorithme qui permet de lire un entier et d'afficher tous les chiffres qui le composent.

Exemple :

Si le nombre proposé est 2345 alors le programme souhaité doit afficher dans l'ordre : 2, 3, 4, 5



28

V. Exercices d'application

Exercice 2 :

Enrichir l'objet abstrait PILE, concrétisé par une représentation chainée, en intégrant les opérations suivantes :
nb_element : renvoie le nombre d'éléments de la pile.
remplace_sommet : change le sommet de la pile. Elle exige que la pile soit non vide.
effacer : efface tous les éléments de la pile.

V. Exercices d'application

Exercice 3 :
Écrire un programme C permettant de lire une expression avec parenthèses supposée valide (nombre des parenthèses ouvrantes=nombre de parenthèses fermantes) et d'afficher toutes les sous-expressions entre parenthèses en commençant par la plus interne. Par exemple, si l'expression soumise au programme est :
a+(b-(c*d)+8.14)-(d*k) alors le programme demandé doit afficher :
(c*d)
(b-(c*d) +8.14)
(d*k)