Informatique de base-III

9 septembre 2014

Note Dans ce qui suit nous proposons un programme toujours structuré de la même manière (cependant certains éléments ne sont pas toujours présents) : #include, #define, types, prototypes de fonctions, fonction principale, et enfin définitions de fonctions. Cet ordre doit être considéré comme obligatoire. De plus le programme contient des commentaires (/* ... */ (ou simplement //)) et à la fin un exemple d'exécution sous la forme également d'un commentaire.

1 Un type abstrait : la `Pile`

1.1 Le type abstrait `Pile`

Un *type abstrait* représente un ensemble d'informations et est défini par :
- un nom de type
- un ensemble d'opérations représentées chacune par
  - un nom d'opérateur (ou fonction)
  - une signature de l'opérateur
  - des propriétés concernant le résultat de l'opérateur

Un type abstrait ne décrit pas la la manière dont il est représenté dans un algorithme, c'est-à-dire son *implémentation*. Nous décrivons ci-dessous d'abord le type abstrait `Pile`, à travers ses opérateurs et leurs propriétés.

Une pile est un ensemble d'objets ordonnés, le type des objets est un autre type abstrait, supposé déjà défini, le type *élément*. L'idée de pile est très intuitive : un ensemble d'éléments posés l'un au dessus de l'autre, seul l'élément au *sommet* étant directement accessible. Voici les opérateurs, leur signatures et leurs propriétés :
- – nom : *estPileVide*
  - signature : $Pile \rightarrow Booléen$
  - propriétés : *estPileVide(p)* est *Vrai* si la pile p ne contient aucun élément.
- – nom : *sommet*
  - signature : $Pile \rightarrow Élément$
  - propriétés : *sommet(p)* renvoie la valeur de l'élément au sommet de la pile.
- – nom : *empiler*
  - signature : $Élément \times Pile \rightarrow Pile$

  - propriétés : *empiler(e,p)* renvoie la pile après ajout d'un élément de valeur e à son sommet.
- – nom : *dépiler*
  - signature : $Pile \rightarrow Pile$
  - propriétés : *dépiler(p)* renvoie la pile après suppression de l'élément au sommet.
- – nom : *hauteur*
  - signature : $Pile \rightarrow entier$
  - propriétés : *hauteur(p)* renvoie le nombre d'éléments dans la pile.

Il y a un ensemble de propriétés intuitives comme par exemple :

- $hauteur(empiler(e, p)) = hauteur(p) + 1$
- $hauteur(dépiler(p)) = hauteur(p) - 1$

Ce type, aussi appelé FILO [1]est un type plus spécifique que le type `Liste`, qui joue un rôle important en programmation, en particulier pour décrire la manière dont un programme se déroule lorsqu'il fait appel à des fonctions. Les opérateurs définis ci-dessus sont ceux qui définissent le type, et sont appelés des *primitives*. Nous décrivons ci-dessous une implémentation dans laquelle la pile est représentée par un tableau contenant les éléments ainsi qu'un indice indiquant le sommet de la pile.

1.2 Implémentation d'une pile par un tableau et un indice

La correspondance est assez claire entre signatures d'opérateurs et arguments et valeur de retour de la fonction correspondante. Le types à droite de la flèche " $\rightarrow$ " correspond au résultat renvoyé par la fonction, les types à gauche, aux arguments de la fonction. D'autre part, nous n'allons pas implémenter l'opérateur *hauteur*, en tant que *primitive*, pour l'instant, et ajouter une primitive *initPile* permettant d'initialiser une pile.

Ci-dessous, le fichier `pilesTab.h` définissant l'implémentation du type `Pile` par un enregistrement, ainsi que les prototypes des fonctions primitives, plus quelques fonctions supplémentaires. Ici les éléments sont des entiers (int).

```
/*-------------------------------*/
/* declarations */
/*-------------------------------*/
#define NMAXPILE 100

/*-------------------------------*/
/*type */
/*-------------------------------*/
struct pile {
        int sommet;
        int contenu[NMAXPILE];
};

/*-------------------------------*/
```

1. (First In, Last Out)

```
/* prototypes des primitives */
/*------------------------------*/
struct pile      initPile (struct pile p);
int      estPileVide(struct pile p);
struct pile      empiler (int e, struct pile p);
struct pile      depiler(struct pile p);
int sommet(struct pile p);

/*------------------------------*/
/* prototypes des fonctions supplémentaires */
/*------------------------------*/
int      estPilePleine(struct pile p);
void affichePile (struct pile p);
struct pile litPile (struct pile p, int taille );
```

Ici nous avons préféré mentionner la pile comme argument de *initPile* mais ce n'est pas obligatoire. On trouvera ci-dessous le fichier source `pilesTab.c` :

```
#include <stdlib.h>
#include <stdio.h>
#include "pilesTab.h"


/*------------------------------*/
/* definitions des primitives */
/*------------------------------*/


struct pile      initPile (struct pile p){
        /* intialise une pile vide */
        p.sommet=-1;
        return p;
}

int      estPileVide(struct pile p){
        /* renvoie 1 si p est vide */
        return (p.sommet == -1);

}

struct pile      empiler (int e, struct pile p){
        /* ajout d'un élément de valeur e au sommet de p */
        p.sommet=p.sommet+1;
        p.contenu[p.sommet]=e;
        return p;
}

struct pile      depiler(struct pile p){
        /* supprime l'élément sommet de p */
        p.sommet=p.sommet-1;
        return p;
```

```
}

int sommet(struct pile p){
        /* renvoie la valeur de l'élément sommet de p */
        return p.contenu[p.sommet];
}

/*-----------------------------*/
/* definitions des fonctions supplémentaires */
/*-----------------------------*/


int     estPilePleine(struct pile p){
        /* renvoie 1 si la pile est pleine */
        if (p.sommet==NMAX-1) {
                return 1;
        }
        return 0;
}

void affichePile (struct pile p){
        /* affiche les éléments de la pile p de la base au sommet */
        int i;
        printf ("[");
        for (i=0;i<=p.sommet;i=i+1){
                printf("%d ", p.contenu[i]);
        }
        printf ("-");
        return;
}

struct pile litPile (struct pile p, int taille ){
        /* lit taille éléments de la pile p */
        int i;
        int e;
        p=initPile(p);
        for (i=0;i<taille;i=i+1){
                scanf("%d",&e);
                p=empiler(e,p);
        }
        return p;
}
```

On remarquera l'introduction de deux fonctions d'entrée-sortie, `litPile` et `affichePile`, ainsi que d'une fonction `estPilePleine` qui est utile dans cette implémentation, car les tableaux sont de taille limitée à `NMAXPILE`-1 éléments.

Enfin voici un fichier `essaiPile.c` contenant un programme utilisant ces primitives. On trouvera à la fin du fichier, en commentaires, la compilation et l'exécution de ce programme sur un petit exemple.

```
/*
 *  essaiPile.c
 */
#include <stdlib.h>
#include <stdio.h>
#include "pilesTab.h"

/* principale */
int main (void){
        /* définitions de variables */
        struct pile p1,p2;
        int n;
        int e;
        /* lecture et intialisations */
        printf("nb d'éléments de la pile? ");
        scanf("%d", &n);
        printf("éléments ? \n");
        p1=litPile(p1,n);
        p2=initPile(p2);
        /* dépiler p1 et empiler ses éléments dans p2 (donc en ordre inverse) */
        while(! estPileVide(p1)){
                printf("pas vide %d \n",sommet(p1));
                e=sommet(p1);
                p1=depiler(p1);
                p2=empiler(e,p2);
        }
        /*affichage de p1 et p2 */
        printf("la pile p2 (inversée de p1) \n");
        affichePile (p2);
        printf("\n");
        return EXIT_SUCCESS;
}
//dyn246:dufee gcc pilesTab.c essaiPile.c -o essaiPile
//dyn246:dufee ./essaiPile
//nb d'éléments de la pile? 3
//éléments ?
//0 1 2
//pas vide 2
//pas vide 1
//pas vide 0
//la pile p2 (inversée de p1)
//[2 1 0 -
//dyn246:dufee
```

2 Passages d'arguments par adresse

Lorsque l'argument d'une fonction est un objet de grande taille, le passage de l'argument est coûteux : l'objet est recopié dans une variable locale de la

fonction. Dans le cas d'une pile de 10^6 éléments, l'appel `sommet(p1)` dans la fonction principale de `essaiPile.c` du paragraphe 2.2 provoque la recopie des 10^6 éléments dans une pile `p` locale à la fonction `sommet`. Pour éviter cela on peut, comme dans le cas des tableaux, passer l'adresse de la pile plutôt que la pile elle-même. Nous allons voir ci-dessous une nouvelle écriture des fonctions primitives du type *pile* utilisant cette idée.

2.1 Une implémentation efficace des piles : passer les adresses des piles

L'idée ici est de toujours passer les adresses des piles, et de modifier des piles *résultats* dont l'adresse devient un argument de la fonction. Par convention les adresses résultats sont en fin de la liste d'arguments. Voici ci-dessous le fichier `pilesTabE.h` correspondant :

```
/*
 *  pilesTabE.h
 */

/*-------------------------------*/
/* declarations */
/*-------------------------------*/

#define NMAXPILE 100
/*-------------------------------*/
/*type */
/*-------------------------------*/
struct pile {
        int sommet;
        int contenu[NMAXPILE];
};
/*-------------------------------*/
/* prototypes des primitives */
/*-------------------------------*/

void    initPileE(struct pile *ap);
int     estPileVideE( struct pile *ap);//const struct pile *ap serait mieux
void    empilerE (int e, struct pile *ap);
void    depilerE(struct pile *ap);
int sommetE(struct pile *ap);

/*-------------------------------*/
/* prototypes des fonctions supplémentaires */
/*-------------------------------*/
int     estPilePleineE(struct pile *ap);
void affichePileE (struct pile *ap);
void  litPileE  (struct pile *ap, int  taille );
```

On remarque la déclaration `struct pile *ap` signifiant « `ap` est une variable dont la valeur doit être l' adresse d'un objet de type `struct pile` ». On

peut lire cela également : « l'objet `*ap` désignera l'objet de type `struct pile` qui se trouvera à l'adresse `ap` ». Notez le commentaire suivant le prototype de `estPileVideE` : lorsque un argument d'une fonction est l'adresse d'un objet qui ne doit pas être modifié par la fonction, on peut marquer cette interdiction en ajoutant `"const"`devant le type de l'argument. C'est le cas de la pile d'adresse `ap` argument de la fonction `estPileVide` : cette fonction donne une information sur la pile mais ne doit pas la modifier. Les définitions des fonctions dont les en-têtes sont dans `pilesTabE.h`, sont dans le fichier `pilesTabE.c` ci-dessous :

```
/*
 *  pilesTabE.c
  */
#include <stdlib.h>
#include <stdio.h>
#include "pilesTabE.h"


/*-------------------------------*/
/* definitions des primitives */
/*-------------------------------*/


void    initPileE(struct pile *ap){
        /* intialise une pile vide à l'adresse ap*/
        (*ap).sommet=-1;
        return ;
}

int     estPileVideE( struct pile *ap){
        /* renvoie 1 si la pile d'adresse ap est vide */
        return ((*ap).sommet == -1);

}

void    empilerE (int e, struct pile *ap){
        /* ajout d'un élément de valeur e au sommet de la pile d'adresse ap */
        (*ap).sommet=(*ap).sommet+1;
        (*ap).contenu[(*ap).sommet]=e;
        return ;
}

void    depilerE(struct pile *ap){
        /* supprime l'élément sommet de la pile d'adresse ap */
        (*ap).sommet=(*ap).sommet-1;
        return ;
}

int sommetE(struct pile *ap){
        /* renvoie la valeur de l'élément sommet de la pile d'adresse ap */
        return (*ap).contenu[(*ap).sommet];
```

```
}

/*--------------------------------*/
/* definitions des fonctions supplémentaires */
/*--------------------------------*/


int     estPilePleineE(struct pile *ap){
        /* renvoie 1 si la pile est pleine */
        if ((*ap).sommet==NMAXPILE-1) {
                return 1;
        }
        return 0;
}

void affichePileE (struct pile *ap){
        /* affiche les éléments de la pile d'adresse ap de la base au sommet */
        int i;
        printf ("[");
        for (i=0;i<=(*ap).sommet;i=i+1){
                printf("%d ", (*ap).contenu[i]);
        }
        printf ("-");
        return;
}

void litPileE (struct pile *ap, int taille ){
        /* lit les taille éléments de la pile d'adresse ap */
        int i;
        int e;
        initPileE(ap);
        for (i=0;i<taille;i=i+1){
                scanf("%d",&e);
                empilerE(e,ap);
        }
        return ;
}
```

Enfin voici, dans le fichier `essaiPilePetit.c` un exemple simple de programme utilisant cette implémentation des primitives du type *pile* :

```
/*
 *  essaiPilePetitE.c
 Version avec les fonctions Efficaces (accédant aux piles par leur adresse)
 */
#include <stdlib.h>
#include <stdio.h>
#include "pilesTabE.h"

/* principale */
int main (void){
```

```
        /* définitions de variables */
        struct pile p1,p2;
        int e;
        initPileE(&p1);
        if (estPileVideE(&p1)){
                printf("p1 est vide : OK!\n");
        }
        else{
                printf("MARCHE PAS!!!!!\n");
        }
        empilerE (3,&p1);
        printf("on empile 3 dans p1\n");
        if (estPileVideE(&p1)){
                printf("p1 est vide : ERREUR !!!!!!!!! \n");
        }
        else{
                printf("p1 n'est pas vide: OK!\n");
        }
        e=sommetE(&p1);
        printf("le sommet de p1 est %d\n",e);
        depilerE(&p1);
        printf("on a dépilé p1, donc elle devrait être vide\n");
        if (estPileVideE(&p1)){
                printf("p1 est vide : OK!\n");
        }
        else{
                printf("MARCHE PAS!!!!!\n");
        }
        return EXIT_SUCCESS;
}

//bash-3.2 gcc pilesTabE.c -c
//bash-3.2 gcc pilesTabE.c essaiPilePetitE.c -o essaiPilePetitE
//bash-3.2 ../essaiPilePetitE
//p1 est vide : OK!
//on empile 3 dans p1
//p1 n'est pas vide: OK!
//le sommet de p1 est 3
//on a dépilé p1, donc elle devrait être vide
//p1 est vide : OK!
//bash-3.2
```

2.2 Allocation dynamique d'une structure : chaîner des éléments

Il est également très utiles de pouvoir allouer dynamiquement des structures, en particulier lorsqu'on veut construire des listes chaînées d'éléments.

Dans l'exemple ci-dessous nous allons d'abord définir un type `mot` comme un tableau de 20 caractères. Nous supposerons aussi que le dernier caractère d'un

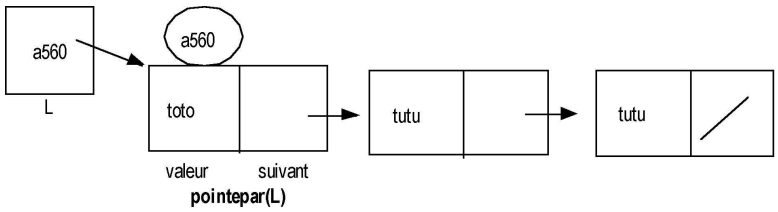

FIGURE 1 – Une chaîne de cellules. La chaîne est représentée par un pointeur `L` contenant l'adresse de la première cellule. ici *pointePar(L)* (notée `(*L)` en C) est la cellule pointée par `L`. De même le champ `suivant` de cette cellule est un pointeur : il contient l'adresse de la cellule suivante, donc `(*L) .suivant`. On a donc `(*L).valeur`="toto" et `(*(*L).suivant).valeur`="tutu". Le champ `suivant` de la dernière cellule contient une adresse particulière, notée `NULL`.

mot est `\0`, et qu'un mot `m` peut être lu par l'instruction `scanf("%s",m);`. Par exemple si lors de l'exécution on tape *"toto"* au clavier, `m[0]` a la valeur `'t'`, `m[1]` a la valeur `'o'`, `m[2]` a la valeur `'t'`, `m[3]` a la valeur `'o'`et `m[4]` a la valeur `'\0'`.

Ensuite nous définissons une *cellule* comme une structure constituée d'un *mot* et d'un *pointeur sur un mot*, le rôle du pointeur étant de permettre d'accéder à la cellule suivante. On obtient alors une structure chaînée dont on trouvera une représentation Figure 1.

Dans le programme ci-dessous, nous allons construire et afficher la liste de la Figure 1.

```
/*
 *  chainee.c
 */
# include <stdio.h>
# include <stdlib.h>
struct  celluleMot{
        char valeur[20];
        struct celluleMot *suivant;
};

int main(void){
        struct celluleMot * listeCellules ; // début de la structure chaînée
        struct celluleMot *aCellule0,*aCellule1,*aCellule2; // pointeurs sur des cellules
        struct celluleMot *aCourante; // pointeur sur une cellule
        struct celluleMot *aRelacher; /// pointeur sur une cellule
        /* on alloue de la mémoire à trois cellules et on remplit leur champ valeur*/
        aCellule0 = (struct celluleMot*) malloc (sizeof(struct celluleMot));
        // aCellule0 contient l'adresse d'une cellule
        printf("premier mot ? \n");
        scanf("%s",(*aCellule0).valeur);
        aCellule1= (struct celluleMot*) malloc (sizeof(struct celluleMot));;
```

```
        // aCellule1 contient l'adresse d'une autre cellule
        printf("deuxième mot ? \n");
        scanf("%s",(*aCellule1).valeur);
        aCellule2= (struct celluleMot*) malloc (sizeof(struct celluleMot));;
        // aCellule2 contient l'adresse d'une autre cellule
        printf("troisième mot ? \n");
        scanf("%s",(*aCellule2).valeur);
        /* la celllule d'adresse aCellule0 sera la première celluleMot de listeCellules */
        listeCellules =aCellule0;
        /* la cellule d'adresse aCellule1 sera la cellule suivant la première cellule */
        (*aCellule0).suivant=aCellule1;
        /* la cellule d'adresse aCellule2 sera la cellule suivant la seconde cellule */
        (*aCellule1).suivant=aCellule2;
        /* la cellule d'adresse aCellule2 est la dernière */
        (*aCellule2).suivant=NULL;
        /* on affiche les mots de la chaîne de cellules */
        printf("La liste de mots est~:\n");
        aCourante=listeCellules; // initialisation de aCourante
        while (aCourante !=NULL){
                printf("%s->",(*aCourante).valeur);
                aCourante=(*aCourante).suivant; // passage à la cellule suivante
                /* s'écrit aussi aCourante=aCourante->suivant;*/
        }
        printf("///\n");
        /* On libère la mémoire : aRelacher mémorise la cellule à libérer*/
        aCourante=listeCellules;
        while (aCourante !=NULL){
                aRelacher=aCourante;
                // passage à la cellule suivante avant de libérer la cellule
                aCourante=(*aCourante).suivant;
                free(aRelacher);
        }
        return EXIT_SUCCESS;
}
//dyn246:dufee soldano ./chainee
//premier mot ?
//toto
//deuxième mot ?
//tutu
//troisième mot ?
//tutu
//La liste de mots est~:
//toto->tutu->tutu->///
```

2.3 Une implémentation des piles en structure chaînée

L'avantage des structures chaînées est que l'on peut facilement ajouter ou supprimer une cellule dont on connaît l'adresse, par exemple la première cellule d'une liste. Nous utilisons ici cette idée pour une nouvelle implémentation des

piles. Voici d'abord ci-dessous le contenu du fichier `pilesChaine.h` où sont définis les types et prototypes de cette implémentation :

```
/*-------------------------------*/
/* types                         */
/*-------------------------------*/
// une cellule contient un élément (un entier) et un pointeur
// vers l'élément suivant de la pile
struct cellule {
        int valeur;
        struct cellule *suivant;
};
// une pile est l'adresse de la première cellule donc de type "struct cellule *"

/*-------------------------------*/
/* prototypes des primitives */
/*-------------------------------*/

struct cellule* initPileC(void);
int     estPileVideC(struct cellule* p);//const struct cellule* p serait mieux
struct cellule* empilerC (int e, struct cellule *p);
struct cellule* depilerC(struct cellule *p);
int sommetC(struct cellule *p);

/*-------------------------------*/
/* prototypes des fonctions supplémentaires */
/*-------------------------------*/
void affichePileC (struct cellule *p);
struct cellule * litPileC ( int taille );
struct cellule* VidePileC(struct cellule* p); // vide la pile
```

Ici un élément de la pile est représenté par le champ `valeur` d'une cellule. Le champ `suivant` permet de chaîner les éléments. La pile est représentée par l'adresse de la cellule représentant le sommet de la pile.

Quelques remarques :

- une pile `p` est de type `struct cellule *`
- `empilerC` et `depilerC` ont comme argument une pile `p` (l'adresse de la cellule au sommet de la pile) et renvoient une adresse de cellule différente : celle de la cellule qui est maintenant au sommet de la pile.
- `initPileC` n'a pas d'argument et renvoie `NULL` qui est considéré comme de type `struct cellule *` et représente une pile vide.
- `VidePileC` a comme argument une pile, dépile tous ses éléments et renvoie la pile vide (c'est-à-dire l'adresse `NULL`) ;
- Il faut bien différencier le rôle de `initPileC` de celui de `VidePileC`. Une pile est une variable contenant une adresse de cellule mais une adresse de cellule n'est une pile que si elle a été initialisée par un appel à `initPileC`. Ainsi faire un appel à `p=empilerC(1,p)` est une erreur si `p` n'a jamais été initialisé. Inversement l'appel à `p=initPileC(1,p)` où `p` représentait déjà une pile contenant des éléments, n'est pas une erreur mais la mémoire

associée aux éléments de la pile `p` ne sera pas restituée.
Voici un programme principal `essaiPilePetitC.c` utilisant ces fonctions :

```
/*
 *  essaiPilePetitC.c
 Version avec l'implémentation des piles en structure chainées */
#include <stdlib.h>
#include <stdio.h>
#include "pilesChaine.h"

/* principale */
int main (void){
        /* définitions de variables */
        struct cellule* p1,p2;
        int e;
        /* */
        p1=initPileC();
        if (estPileVideC(p1)){
                printf("p1 est vide : OK!\n");
        }
        else{
                printf("MARCHE PAS!!!!!\n");
        }
        p1=empilerC (3,p1);
        printf("on empile 3 dans p1\n");
        if (estPileVideC(p1)){
                printf("p1 est vide : ERREUR !!!!!!!!!! \n");
        }
        else{
                printf("p1 n'est pas vide: OK!\n");
        }
        e=sommetC(p1);
        printf("le sommet de p1 est %d\n",e);
        p1=depilerC(p1);
        printf("on a dépilé p1, donc elle devrait être vide\n");
        if (estPileVideC(p1)){
                printf("p1 est vide : OK!\n");
        }
        else{
                printf("MARCHE PAS!!!!!\n");
        }
        /* on vide la pile p1 pour restituer la mémoire de ses éléments */
        p1=videPileC(p1);
        printf("les %d éléments d'une nouvelle pile ? \n",3);
        p1=litPileC(3);
        printf("les %d éléments de la base au sommet ? \n",3);
        affichePileC(p1);
        printf("\n");
        return EXIT_SUCCESS;
}
```

```
//bash-3.2 gcc pilesChaine.c essaiPilePetitC.c -o essaiPilePetitC
//bash-3.2 ./essaiPilePetitC
//p1 est vide : OK!
//on empile 3 dans p1
//p1 n'est pas vide: OK!
//le sommet de p1 est 3
//on a dépilé p1, donc elle devrait être vide
//p1 est vide : OK!
//les 3 éléments d'une nouvelle pile ?
//7 8 9
//les 3 éléments de la base au sommet ?
//[7 8 9 -
//bash-3.2
```

Enfin, voici le fichier `pilesChaine.c` contenant les définitions des fonctions :

```
/*
 * pilesChaine.c : piles d'entiers
 */
#include <stdlib.h>
#include <stdio.h>
#include "pilesChaine.h"
/*-------------------------------*/
/* types                         */
/*-------------------------------*/

// une cellule contient un élément (un entier) et un pointeur vers l'élément suivant de la pile
// une pile est l'adresse de la première cellule donc de type "struct cellule *"



/*-------------------------------*/
/* definitions des primitives */
/*-------------------------------*/



struct cellule* initPileC(){
        /* intialise une pile vide dont l'adresse sera retounrée*/
                return NULL;

}
int     estPileVideC(struct cellule* p){//const struct cellule* p serait mieux
        /* renvoie 1 si la pile p est vide */
        return (p == NULL);
}

struct cellule* empilerC (int e, struct cellule *p){
        /* ajout d'un élément de valeur e au sommet de la pile p */
        struct cellule *ac;
```

```
        /* allocation de la nouvelle cellule */
        ac=(struct cellule*) malloc (sizeof(struct cellule *));
        /* champs valeur (e) et suivant (p) de la cellule */
        (*ac).valeur=e;
        (*ac).suivant=p;
        /* ac est la nouvelle adresse du sommet, donc représente la pile */
        return ac;
}

struct cellule * depilerC(struct cellule *p){
        /* supprime l'élément sommet de la pile p */
        struct cellule *aRelacher;
        aRelacher=p;
        /* dépilage */
        p=(*p).suivant;
        /* restitution de la mémoire de la cellule correspondant à l'ancien sommet */
        free(aRelacher);
        return p;
}

int sommetC(struct cellule *p){
        /* renvoie la valeur de l'élément sommet de la pile p */
        return (*p).valeur;

}
/*-------------------------------*/
/* prototypes des fonctions supplémentaires */
/*-------------------------------*/
void affichePileC (struct cellule *p){
        /* affiche les éléments de la pile p de la base au sommet */
        struct cellule * pI;
        /* on inverse la pile*/
        pI=initPileC();
        while (!estPileVideC(p)){
                pI=empilerC(sommetC(p), pI);
                p=depilerC(p);
        }
        /* on affiche la pile inverse du sommet à la base et on empile dans p */
        /* p est vide */
        printf ("[");
        while (!estPileVideC(pI)){
                printf("%d ", sommetC(pI));
                p=empilerC(sommetC(pI), p);
                pI=depilerC(pI);
        }
        printf ("-");
        /* à l'adresse p on a une pile identique à la pile p en argument */
        return;
}
```

```
struct cellule* litPileC ( int taille ){
        /* lit  taille éléments et contruit une pile */
        struct cellule* p;
        int i;
        int e;
        p=initPileC();
        for (i=0;i<taille;i=i+1){
                scanf("%d",&e);
                p=empilerC(e,p);
        }
        return p;
}

struct cellule* videPileC(struct cellule* p){
        /*vide la pile p */
        while (!estPileVideC(p)){
                p=depilerC(p);
        }
        return p;
        /* p == NULL */
}
```

3 Les Files et leur implémentation en structures chaînées.

Une *file* est un objet abstrait représentant une file d'attente : contrairement à la pile, ici les éléments sortent de la file dans l'ordre de leur entrée (le premier élément entré est le premier sorti). On appelle aussi cet objet une liste F.I.F.O (First In First Out). Le premier élément est en tête de la file et le dernier élément en queue. La file , autre type "abstrait de données", est également définie par un ensemble d' opérations et leurs propriétés.

- creerFile()→file /* Intialise une file vide f */
- estFileVide(file)→ booléen /* Renvoie Vrai si la file est vide */
- enfiler(val, file) → file /* Ajoute en queue de la file f un élément de valeur val */
- defiler(file) → file /* Ôte l'élément en tête de la file f */
- queue(file) → element /* Renvoie la valeur de l'élément au sommet de la pile*/
- tete(file) → element /* Renvoie la valeur de l'element en tete de la file (non vide) */

Exemple : Dans la file f suivante :
`f = ( A,B,C,D,E)` A est l'élément de tete, E est l'élément de queue.
Après enfiler(X, f) on a : `f = ( A,B,C,D,E, X)`
Après defiler(f) on a : `f = ( B,C,D,E, X)`
On a alors tete(f) = B et queue(f) = X

Comme une pile, une file peut également être représentée de manière contiguë, en définissant un enregistrement à trois champs : un tableau contenant les éléments, l'indice de l'élément de tête et l'indice de l'élément de queue. La mise en oeuvre est cependant ici plus difficile (il faut en réalité considérer le tableau comme un tableau circulaire).

3.1 Une implémentation des files en listes chaînées.

La représentation chaînée d'une file est très voisine de celle d'une pile, il faut cependant avoir un accès direct à la queue pour ne pas parcourir la file lorsqu'on veut ajouter un nouvel élément.

Une file est alors représentée par un enregistrement qui contient - un champ Premier qui contient l'adresse de l'élément de tête - un champ Dernier qui contient l'adresse de l'élément de queue.

Nous définissons ci-dessous les procédures et fonctions de base du type abstrait file dans une représentation chaînée. Ici nous faisons le choix de ne pas renvoyer la structure file lorsqu'on modifie la file, mais de passer l'adresse de la file lorsqu'on veut la modifier. Il s'agit là d'une pratique courante lorsqu'il s'agit de structures : donner l'adresse d'un objet plutôt que de passer l'objet et de le renvoyer modifié. Suivant la même idée, nous allons ici également passer l'adresse plutôt que l'objet même si il ne doit pas être modifié. Nous allons également ici systématiquement utiliser la notation adressedobjet→champ au lieu de (*adressedobjet).champ.

La fonction main interactive (ne dépendant pas de l'implémentation mais uniquement des opérations caractéristiques) dans cet exemple est destinée à gérer une file d'attente. Attention ici, on suppose que, quelle que soit l'implémentation, on a toujours passé l'adresse de l'objet représentant la file plutôt que l'objet lui-même. */

```
#include <stdio.h>
#include <stdlib.h>

/* types */
typedef int element;

typedef enum Booleen {Faux, Vrai} booleen;

typedef struct Cellule {
        element valeur;
        struct Cellule *suivant;
} cellule ;

typedef struct File {
         cellule * premier;
         cellule * dernier;
} file ;

typedef enum Reponse {Arrivee, Traitement, Fermeture} reponse;
```

```
/* prototypes */
void creeFile( file * pf);
booleen estFileVide( file *pf );
void enfiler (element valEl , file *pf );
void defiler ( file *pf);
element queue( file * pf);
element tete ( file * pf);
//non primitives
void traiter (element v);
reponse interaction (element *pElement);

/* Fonctions */

int main (int argc, const char * argv[])
{


    /* cette fonction gere une file d'attente de clients.
     La fonction interaction renvoie une valeur de type
     reponse=(arrivee, traitement, fermeture) et modifie son
     argument val si un nouveau client arrive*/
    file f;
    element val;
    reponse rep;

    creeFile(&f);
    do {
        rep=interaction(&val);
        if (rep == Arrivee)// Arrivee==0
            enfiler (val, &f);
        else{
            if (rep==Traitement){
                if (! estFileVide(& f)){
                    traiter (tete(& f));
                    defiler (& f);
                }
                else {
                    printf("pas de client ! \n");
                }
            }
        }
    }
    while (rep!=Fermeture);
    /* */
    /* la file se vide a la fermeture */
    while (! estFileVide(& f)) {
        defiler (& f);
    }
```

```
    return 0;

}

/* Intialise une file vide */
void creeFile( file * pf) {
        /* */
        pf->premier=NULL;
        pf->dernier =NULL;
}

/* Renvoie Vrai si la file d'adresse pf est vide */
booleen estFileVide( file *pf ){
        /* */
        return (pf->premier==NULL);
}

/* Ajoute en queue de la file f un élément de valeur val */
void enfiler (element valEl , file *pf ) {
        /* */
        cellule *pN;
        pN=(cellule*) malloc (sizeof( cellule ));
        pN->valeur = valEl ;
        pN->suivant =NULL;
        if (estFileVide(pf)) { /* pf->premier et pf->.dernier sont NULL */
                pf->premier=pN;
                pf->dernier=pN;
        } else {
                pf->dernier->suivant =pN ;
                pf->dernier=pN;
        }
        return ;
}

/* Ôte l'élément en tête de la file d'adresse pf, qui n'est pas vide */
void defiler ( file *pf) {
        /* */
        cellule *aRelacher;
        aRelacher=pf->premier;
        if (pf->premier==pf->dernier){ /*un seul element dans la file*/
                pf->premier=NULL;
                pf->dernier=NULL;
        } else {
                pf->premier = pf->premier->suivant;
        }
        free (aRelacher);
        return ;
}
```

```
/*Renvoie la valeur de l'élément en queue de la file (la file n'est pas vide) */

element queue( file * pf){
        /* */
        return (pf->dernier->valeur);
}

/* Renvoie la valeur de l'élément en tete de de la file (la file n'est pas vide) */
element tete( file * pf){
        /* */
        return (pf->premier->valeur);
}

/* non primitives */
void traiter (element v){

    printf(" client %d\n", v);

}

/* ------------ */

reponse interaction (element *pElement){
    int irep;
    reponse tabRep[3]={Arrivee, Traitement, Fermeture};

    printf(" interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))\n");
    scanf("%d", &irep);
    if (tabRep[irep]==Arrivee){
        printf(" client ? ( identifiant entier)");
        scanf("%d", pElement);
    }
    return (tabRep[irep]);
}

//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//0
//client ? ( identifiant entier)512
//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//0
//client ? ( identifiant entier)13
//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//0
//client ? ( identifiant entier)24
//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//1
//client 512
//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//0
//client ? ( identifiant entier)24
```

//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//1
//client 13
//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//1
//client 24
//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//1
//client 24
//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//1
//pas de client !
//interaction ? (0 (arrivee), 1 (traitement du premier client), 2 (fin))
//2
//dyn217:file soldano

4 Les Listes et leur implémentation en structures chaînées.

Plus généralement dans une liste on veut pouvoir acceder à un élément à une certaine position p, pour insérer un élément à cette position ou supprimer l'élément à cette position. Une caractéristique de la liste est l'ordre de ses éléments associé aux notions de *successeur* et de *prédécesseur*. On appellera *Hors-liste* une position de dépassement, c'est à dire qui ne permet pas l'accès à un élément de la liste. Voici un ensemble d'opérations caractérisant un type abstrait Liste :

- creerListe()→liste /* intialise une liste vide L */
- estListeVide(liste)→booléen /* renvoie Vrai si la liste est vide */
- premier(liste)→element /* renvoie la position du 1er element de la lite (ou *Hors-Liste* si la liste est vide)*/
- dernier(liste)→element /* renvoie la position du dernier élément de la lite (ou *Hors-Liste* si la liste est vide)*/
- ajouterEnPosition(val, position, liste)→liste /*insere un element de valeur val en position p : attention celle-ci accède à un autre élément après insertion : 1 2 3 (5) devient 1 2 3 (4) 5 : (les parenthèses indiquent à quel élément on a accès à la position p) */
- supprimerEnPosition(position, liste) : /* supprime l' élément à la position p : attention ici aussi celle-ci change après suppression : 1 2 3 (4) 5 devient 1 2 3 (5) */
- valeurElement(position, liste)→element /* renvoie la valeur de l'élément en position p dans la liste */
- successeur(position, liste)→ position /*renvoie la position de l'element qui succède à celui en position p (ou *Hors-Liste*)*/
- fin(position,liste) → booléen /* renvoit Vrai si la position est hors-Liste */

Lorsqu'on considère une représentation chaînée de ce type abstrait Liste, on remarque d'abord que la "position" est un accès à un élément (une adresse) et que le lien "suivant" matérialise la notion de successeur. Cependant l'insertion comme la suppression d'un élément posent un problème du fait de l'absence d'un lien représentant le prédécesseur :

Ainsi si par insertion de 4 la liste 1 2 3 (5) doit se transformer en 1 2 3 (4) 5, alors pour insérer 4 avant 5 il faut connaître le prédécesseur de 5, afin de le lier à 4. De même si la suppression de 4 doit transformer 1 2 3 (4) 5 en 1 2 3 (5), alors il faut connaître le prédécesseur de 4, afin de le lier à 5.

Ce problème se résout de deux manières : ,

- soit en ajoutant un lien "précédent" à chaque cellule, on obtient alors une liste *doublement chaînée*
- soit on accède à un élément par l'adresse de la cellule qui le précède, on garde alors une liste *simplement chaînée*.

Nous décrivons brièvement ci-dessous le principe des listes doublement chaînées avant de détailler une implémentation pour les listes simplement chaînées.

4.1 les listes doublement chaînées.

Cette solution consiste à ajouter un lien "précédent" à l"enregistrement, matérialisant ainsi la notion de prédécesseur. On obtient alors ce que l'on appelle une liste *doublement chaînée*, au prix d'alourdir la représentation, mais avec l'avantage de pouvoir parcourir la liste dans les deux sens. De plus, au lieu de terminer les chaînes de cellules par la valeur NULL, nous définissons ici une chaîne cyclique, ce que l'on appelle une *liste doublement chaînée circulaire*. Dans ce cas le dernier élément pointe sur le premier et réciproquement. Les opérations en tête et en queue sont alors symétriques. La liste sans élément est le pointeur NULL. Le fichier de headers `listesDoublementsChainees.h` est le suivant

```
/*-------------------------------*/
/* declarations */
/*-------------------------------*/

typedef enum BOOL {false,true} bool;
typedef int element;
typedef struct Cellule {
        element valeur;
        struct  Cellule *suiv, *prec;
}  cellule ;

typedef cellule *listeD;
/*-------------------------------*/
/* prototypes */
/*-------------------------------*/
/* primitives */
listeD  creerListe ();
bool     estListeVide(listeD  l);
cellule* premier(listeD  l);
```

```
cellule * dernier(listeD l);
listeD supprimerEnPosition(cellule * p, listeD l);
listeD ajouterEnPosition(element val, cellule * p, listeD l);
element valeurElement( cellule * p, listeD l);
cellule * successeur( cellule * p, listeD l);
bool fin( cellule * p, listeD l);
/* affichage */
void    afficherListe ( listeD l);
void    afficherListePrim ( listeD l);
```

Le fichier `listesDoublementChainées.c` implémentant les fonctions correspondantes :

```
#include <stdlib.h>
#include <stdio.h>
#define Malloc(type) (type *) malloc(sizeof(type))
#include "listesDoublementChainees.h"

/* listes doublement chainees circulaires */
/*-------------------------------*/
/* definitions */
/*-------------------------------*/

/* intialise et retourne une liste vide */
listeD creerListe (){
    listeD l=NULL;
    return l;
}

/* renvoie Vrai si la liste l est vide */
bool    estListeVide(listeD l){
        if (l==NULL)
                return true;
        else
                return false;
}

/* renvoie la position du premier element de la liste */
/* l est non vide */
cellule * premier(listeD l){
    return l;
}

/* renvoie la position du dernier element de la liste */
/* l est non vide */
cellule * dernier(listeD l){
    return l->prec;
}

/* supprime un element à la position p et renvoie la liste */
/* si p est Hors-Liste (NULL) ajout en queue */
```

```
listeD supprimerEnPosition(cellule * p, listeD l){
    if (p==p->suiv){ // un seul element dans liste
        free(p);
        return NULL;
    }
    if (p==l) // le premier element change
        l=l->suiv;
    // le suivant du precedent de p devient le suivant de p
    p->prec->suiv=p->suiv;
    // le precedent du suivant de p devient le precedent de p
    p->suiv->prec= p->prec;
    free(p);
    return l;
}


/* ajoute un élément de valeur val à la position p et renvoie la liste */
/* si p est Hors-Liste (NULL) ajout en queue */
listeD ajouterEnPosition (element val, cellule * p, listeD l){

    // pNew est l'adresse du nouvel element
    // ps est l'adresse de l'element destine a etre
    //       le sucesseur du nouvel element
    // pp est l'adresse de l'element destine a etre
    //       le predecesseur du nouvel element dans la liste circulaire

    cellule *pNew, *ps, *pp, *pos;

    pNew=Malloc(cellule);
    pNew->valeur=val;
    if (l==NULL) {// p = l = NULL
        pNew->suiv=pNew;
        pNew->prec=pNew;
        return pNew;
    }
    if (p==l){
        l=pNew; // ajout en tete, donc la tete sera le nouvel element
    }
    if (p==NULL){
        p= l; // ajout en queue, donc avant le premier element qui est aussi l
    }
    ps=p;
    pp=ps->prec;
    pNew->suiv=ps;
    ps->prec=pNew;
    pNew->prec=pp;
    pp->suiv=pNew;
    return l;
}
```

```
/* valeur de l'element en position p dans l */
/* l n'est pas vide, p n'est pas Hors-Liste */
element valeurElement( cellule * p, listeD l){
    return p->valeur;
}
/* position du successeur de p dans l */
/* l n'est pas vide, p n'est pas Hors-Liste */
cellule * successeur( cellule * p, listeD l){
    if ( p->suiv == l){
        return NULL;
    }
    return p->suiv;
}
/* p est elle la position de fin dans l (Hors-Liste) ? */
bool fin( cellule * p, listeD l){
    return (p==NULL);
}

/* affiche les elements de la liste doublement chainee */
void    afficherListe ( listeD l){
    cellule *pc;
    pc=l;
    printf(" liste ");
    if (pc == NULL){
        printf("vide \n");
        return ;
    }
    printf("%d ", pc->valeur);
    pc=pc->suiv;
    while(pc !=l){
        printf("%d ", pc->valeur);
        pc=pc->suiv;
    }
    printf("\n");
    return ;
}
/* affiche les elements de la liste , version avec fonctions primitives */
void    afficherListePrim ( listeD l){
    cellule *pc;
    printf(" liste ");
    if (estListeVide(l)){
        printf("vide \n");
        return ;
    }
    pc=premier(l);
    while (!fin(pc,l)){
        printf("%d ", valeurElement(pc,l));
        pc=successeur(pc,l);
    }
    printf("\n");
```

```
    return ;
}
```

Un programme simple utilisant ces fonctions ci-dessous, on remarquera que les deux fonctions d'affichage, l'une proche de l'implémentation, l'autre utilisant les fonctions primitives, sont utilisées.

```
#include <stdlib.h>
#include <stdio.h>

#include "listesDoublementChainees.h"

/* listes doublement chainees circulaires */
/*-------------------------------*/
/* principale */
/*-------------------------------*/
int main(){
    listeD l;
    l=creerListe();
     afficherListe (l);
    afficherListePrim(l);
    l=ajouterEnPosition(2, l,l);
     afficherListe (l);
    afficherListePrim(l);
    l=ajouterEnPosition(3, l,l);
    printf("l'element de tete est %d \n", valeurElement(premier(l),l));
    printf("son successeur est %d \n",
            valeurElement(successeur(premier(l),l),l) );
     afficherListe (l);
    afficherListePrim(l);
    l=supprimerEnPosition(dernier(l),l);
     afficherListe (l);
    afficherListePrim(l);
    l=supprimerEnPosition(dernier(l),l);
     afficherListe (l);
    afficherListePrim(l);
    return 0;
}
//compilation
//bash-3.2 gcc listesDoublementChainees.c essaiListesD.c -o essaiListesD
//exemple d'execution
//bash-3.2 ./essaiListesD
//liste vide
//liste vide
//liste 2
//liste 2
//l'element de tete est 3
//son successeur est 2
//liste 3 2
//liste 3 2
//liste 3
```

//liste 3
//liste vide
//liste vide
//bash−3.2