Cours de Programmation Impérative
Implémenter les piles et les files

Julien David

A101 - david@lipn.univ-paris13.fr

Allocation dynamique

Dans les cours du lundi nous avons vu

- L'allocation d'espace mémoire, permettant de faire des tableaux de taille variable.
- Les listes chaînées

Dans les cours du vendredi nous avons vu

- Les piles
- Les files

Implémentation

Pile et file

Les piles et les files sont des **structures de données linéaires**.

Fonctions

Examinons les fonctions utiles pour manipuler les piles et le files.

Pile	**File**
Créer	Créer
Détruire	Détruire
Tester si vide	Tester si vide
Taille	Taille
Afficher	Afficher
Insérer d'un coté	**Insérer d'un coté**
Extraire du même coté	**Extraire de l'autre côté**

Implémentation

Différence en pile et file

La **seule** différence entre une pile et une file est la manière dont on ajoute/extrait un élément.

- Dans le cas de la pile, on extrait du “même côté” que l’on ajoute.
- Dans le cas de la file, on ajoute par une extrémité et on extrait par l’autre.

Et alors ?

On peut implémenter les piles et les files en utilisant la même structure et les mêmes fonctions, sauf pour l’insertion/extraction.

Tableaux dynamiques ou listes chaînées ?

La structure commune

On peut donc construire une structure de données linéaire et s'en servir ensuite pour définir les piles et les files. On distingue deux cas de figures :

- Les piles et les files n'ont pas une taille bornée **(cas vu en cours le vendredi)**.
- Les piles et les files ont une taille bornée **(cas nouveau)** .

Structures de données linéaires non bornées

On utilise les pointeurs debut et fin dans une liste chaînée

- pour la pile, on va empiler et depiler en utilisant le pointeur **debut**,
- pour la file, on va enfiler en utilisant le pointeur **fin** et defiler avec le pointeur **debut**.
- La seule fonction qui diffère de la pile à la file est donc l'insertion.

On a la structure !

Observons maitenant comment implémenter ses fonctions, puis les piles et les files.

Définition de la structure

```
1  struct maillon_s{
2    int val;
3    struct maillon_s * suiv;
4  };
5  typedef struct maillon_s maillon_t;
6
7  struct liste_s{
8    struct maillon_s * premier;
9    struct maillon_s * dernier;
0    int taille;
1  };
2  typedef struct liste_s liste_t;
```

Déclaration des fonctions

```
1  maillon_t * creer_maillon(int v);
2  void detruire_maillon(maillon_t * m);
3  liste_t * liste_creer();
4  void liste_detruire(liste_t * l);
5  int liste_vide(liste_t l);
6  int liste_taille(liste_t l);
7  void liste_ajouter_debut(liste_t * l,int x);
8  void liste_ajouter_fin(liste_t * l,int x);
9  maillon_t * liste_extraire_debut(liste_t * l);
0  void liste_afficher(liste_t l);
```

Définition des fonctions

```
1  maillon_t * creer_maillon(int v){
2    maillon_t * res=(maillon_t*)malloc(sizeof(maillon_t));
3    res->val=v;
4    res->suiv=NULL;
5    return res;
6  }
7
8  void detruire_maillon(maillon_t * m){
9    free(m);
0  }
```

Définition des fonctions

```
1  liste_t * liste_creer(){
2    liste_t *res=(liste_t*)malloc(sizeof(liste_t));
3    res->premier=NULL;
4    res->dernier=NULL;
5    res->taille=0;
6    return res;
7  }
8
9  void liste_detruire(liste_t * l){
0    while(!liste_vide(l))
1      detruire_maillon(liste_extraire_debut(l));
2    free(l);
3  }
```

Définition des fonctions

```
1  int liste_vide(liste_t * l){
2    if(l->taille==0)
3      return 1;
4    return 0;
5  }
6
7  int liste_taille(liste_t * l){
8    return l->taille;
9  }
0
1  void liste_afficher(liste_t * l){
2    maillon_t * tmp;
3    for(tmp=l->premier;tmp!=NULL;tmp=tmp->suiv)
4      printf("%d ",tmp->val);
5    printf("\n");
6  }
```

Définition des fonctions

```
1 void liste_ajouter_debut(liste_t * l,int x){
2   maillon_t * nvo=creer_maillon(x);
3   if(liste_vide(l))
4     l->dernier=nvo;
5   else
6     nvo->suiv=l->premier;
7   l->premier=nvo;
8   l->taille++;
9 }
```

Définition des fonctions

```
1 void liste_ajouter_fin(liste_t * l,int x){
2   maillon_t * nvo=creer_maillon(x);
3   if(liste_vide(l))
4     l->premier=nvo;
5   else
6     l->dernier->suiv=nvo;
7   l->dernier=nvo;
8   l->taille++;
9 }
```

Définition des fonctions

```
1  maillon_t * liste_extraire_debut(liste_t * l){
2    maillon_t * res=NULL;
3    if(!liste_vide(l)){
4      res=l->premier;
5      l->premier=res->suiv;
6      res->suiv=NULL;
7      l->taille --;
8      if(liste_vide(l))
9        l->dernier=NULL;
0    }
1    return res;
2  }
```

Maintenant que l'on a codé des listes

On va pouvoir définir les piles et les files.

Pile/File avec une liste chaînée

```c
#include "liste.h"

typedef liste_t pile_t;

pile_t * pile_creer();
int pile_vide(pile_t * p);
int pile_taille(pile_t * p);
void empiler(pile_t * p,int x);
int depiler(pile_t * p);
void pile_detruire(pile_t * p);
```

```c
#include "liste.h"

typedef liste_t file_t;

file_t * file_creer();
int file_vide(file_t * f);
int file_longueur(file_t * f);
void enfiler(file_t * f,int x);
int defiler(file_t * f);
void file_detruire(file_t * f);
```

Pile/File avec une liste chaînée

```c
pile_t * pile_creer(){
  return liste_creer();
}

int pile_vide(pile_t *p){
  return liste_vide(p);
}

int pile_taille(pile_t *p){
  return liste_taille(p);
}

void pile_detruire(pile_t * p){
  liste_detruire(p);
}
```

```c
file_t * file_creer(){
  return liste_creer();
}

int file_vide(file_t * p){
  return liste_vide(p);
}

int file_longueur(file_t * p){
  return liste_cardinal(p);
}

void file_detruire(file_t * p){
  liste_detruire(p);
}
```

Pile/File avec une liste chaînée

```c
void empiler(pile_t * p,int x){
  liste_ajouter_debut(p,x);
}

int depiler(pile_t * p){
  maillon_t * m=liste_extraire_debut(p);
  int res=m->val;
  detruire_maillon(m);
  return res;
}
```

```c
void enfiler(file_t * p,int x){
  liste_ajouter_fin(p,x);
}

int defiler(file_t * p){
  maillon_t * m=liste_extraire_debut(p);
  int res=m->val;
  detruire_maillon(m);
  return res;
}
```

Structures de données linéaires bornées

Dans certains contextes. . .

- La taille d'une pile ou d'une file peut-être bornée.
- Dans ce cas, il suffit de faire une unique allocation d'espace mémoire à la création de la structure.
- Il est donc inutile d'utiliser les listes chaînées.
- La structure de données linéaire utilisera donc un tableau.

Définition de la structure

```
1 typedef struct sdl_s{
2   int * tab;
3   int taille_max;
4   int taille;
5   int debut;
6 }sdl_t;
```

Déclaration des fonctions

```
1 sdl_t * creer_sdl(int max);
2 void detruire_sdl(sdl_t * s);
3 int sdl_vide(sdl_t * s);
4 int sdl_taille(sdl_t * s);
5 int sdl_max(sdl_t * s);
6 void afficher_sdl(sdl_t * s);
7 void sdl_ajouter_debut(sdl_t * f,int x);
8 int sdl_extraire_debut(sdl_t * s);
9 int sdl_extraire_fin(sdl_t * s);
```

Définition des fonctions

```
1  sdl_t * creer_sdl(int max){
2    sdl_t * res=(sdl_t *)malloc(sizeof(sdl_t));
3    res->tab=(int *)malloc(sizeof(int)*max);
4    res->taille_max=max;
5    res->taille=0;
6    res->debut=0;
7    return res;
8  }
9
0  void detruire_sdl(sdl_t * s){
1    free(s->tab);
2    free(s);
3  }
```

Définition des fonctions

```
1  int sdl_vide(sdl_t * s){
2    if(s->taille==0)
3      return 1;
4    return 0;
5  }
6
7  int sdl_taille(sdl_t * s){
8    return s->taille;
9  }
0
1  int sdl_max(sdl_t * s){
2    return s->taille_max;
3  }
4
5  void afficher_sdl(sdl_t * s){
6    int i,pos;
7
8    for(i=0,pos=s->debut;i<s->taille;i++,pos=((pos+1)%s->taille_max))
9      printf("%d ",s->tab[pos]);
20   printf("\n");
21 }
```

Définition des fonctions

```
1 void sdl_ajouter_debut(sdl_t * s,int x){
2   if(s->taille!=s->taille_max){
3     s->tab[(s->debut+s->taille)%s->taille_max]=x;
4     s->taille++;
5   }
6   else
7     printf("Erreur: la structure est pleine\n");
8 }
```

Définition des fonctions

```
1  int sdl_extraire_fin(sdl_t * s){
2    int res=0;
3    if(!sdl_vide(s)){
4      res=s->tab[s->debut];
5      s->debut=(s->debut+1)%s->taille_max;
6      s->taille--;
7    }
8    else
9      printf("Erreur: la pile est vide\n");
0    return res;
1  }
```

Définition des fonctions

```
1  int sdl_extraire_debut(sdl_t * s){
2    int res=0;
3    if(!sdl_vide(s)){
4      s->taille--;
5      res=s->tab[s->taille];
6    }
7    else
8      printf("Erreur: la pile est vide\n");
9    return res;
0  }
```

Définition des piles et des files

Comme pour les listes

On va maintenant utiliser la structure `sdl_t` pour définir les piles et les files.

Mais histoire de changer

Plutôt que de définir des fonctions qui vont appeler celles définies pour les listes, on va utiliser des **macros** (des *#define*).

Pile avec un tableau (structure `sdl_t`)

```
1  typedef sdl_t pile_t;
2
3  #define creer_pile(max) creer_sdl((max))
4
5  #define detruire_pile(s) detruire_sdl((s))
6
7  #define pile_vide(s) sdl_vide((s))
8
9  #define pile_taille(s) sdl_taille((s))
0
1  #define pile_max(s) sdl_max((s))
2
3  #define  afficher_pile(s) afficher_sdl((s));
4
5  #define empiler(p,x) sdl_ajouter_debut((p),(x));
6
7
8  #define depiler(p) sdl_extraire_debut((p));
```

File avec un tableau (structure `sdl_t`)

```
1  typedef sdl_t file_t;
2
3  #define creer_file(max) creer_sdl((max))
4
5  #define detruire_file(s) detruire_sdl((s))
6
7  #define file_vide(s) sdl_vide((s))
8
9  #define file_taille(s) sdl_taille((s))
0
1  #define file_max(s) sdl_max((s))
2
3  #define afficher_file(s) afficher_sdl((s));
4
5  #define enfiler(f,x) sdl_ajouter_debut((f),(x));
6
7  #define defiler(f) sdl_extraire_fin((f));
```