Types et structures de données (en C)

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Introduction

Introduction

Structures des données

En informatique, il existe de nombreuses **structures de données** usuelles.

Pour résoudre un problème donné, un informaticien se tournera naturellement vers une structure de données bien connue s'il peut en faire usage efficacement.

Structures de données abstraites

Définition

Une **structure de données abstraite** est la donnée d'un **type**, et des **opérations** que l'on peut effectuer dessus.

On peut regrouper ces opérations en 4 catégories :

- un **constructeur** permet d'initialiser une structure ;
- un **accesseur** permet de récupérer une valeur stockée dans la structure ;
- un **transformateur** permet de modifier l'état de la structure ;
- un **destructeur** permet de supprimer la structure (libérer la mémoire).

Structure abstraite vs implémentation concrète

Cette définition est un peu évasive, mais ce qu'il faut retenir est qu'une structure abstraite est indépendante d'une implémentation concrète.

Ce qui est important est la description des opérations que le type permet, plutôt que la manière dont ces opérations sont réalisées.

Il y a plusieurs avantages à cela.

Avantages

- Si on possède déjà une implémentation de structures de données abstraites, on peut programmer des algorithmes avec sans savoir comment elles sont implémentées : on les voit comme des “boites noires” complexes.
  - $\hookrightarrow$ C’est le point de vue du programmeur utilisateur : il n’a pas besoin de savoir comment ça marche pour utiliser une structure abstraite.
- Si on change l’implémentation d’une structure pour une autre implémentation, les algorithmes faisant usage de la structure que l’on a déjà implémentés seront compatibles avec la nouvelle implémentation.

En pratique

Ce qui permet de juger si une implémentation est meilleure qu'une autre est essentiellement la **complexité**.

Lorsqu'on voudra implémenter concrètement une structure abstraite (on parle de structure concrète), on cherchera à avoir la meilleure complexité (en temps et/ou en mémoire) possible.

Structure de pile

Piles

Pile

Une **pile** est une structure de donnée linéaire, dans laquelle les éléments sont insérés ou supprimés suivant le principe **LIFO** (last in, first out).

Visuellement, le seul élément accessible est celui situé au sommet de la pile, qu'il faudrait retirer pour accéder à l'élément situé en dessous.

Inversement, un élément qu'on rajoute à la pile sera rajouté au sommet.

Piles

Définition (Pile)

Une **pile** est une structure abstraite, supportant les opérations suivantes :

- **création** d'une pile vide ;
- test d'**égalité** au vide ;
- **accès** au sommet d'une pile non vide ;
- **retrait** de l'élément au sommet d'une pile non vide ;
- **ajout** d'un élément au sommet de la pile.

Piles

Exemple

La pile est un outil de base en informatique, dont les utilisations sont multiples. Voici quelques exemples :

- la gestion des appels de fonctions dans l'exécution d'un programme se fait via une **pile d'appels** : lorsqu'une fonction est appelée, les informations relatives à l'appel (adresse de retour, variables locales, . . .) sont empilées sur la pile d'appels, et seront dépilées lorsque la fonction terminera son exécution ;
- les boutons "page précédente" et "page suivante" d'un navigateur utilise deux piles ;
- le **parcours en profondeur** est un algorithme classique pour explorer un graphe, où l'on stocke les sommets visités dans une pile.

Implémentation d'une pile de capacité finie

Pile de capacité finie

Commençons par une implémentation simple : nos piles auront ici une **capacité finie**, c'est-à-dire que lors de la création de la pile, nous spécifierons un entier c, et la pile ne sera pas capable de contenir plus de c éléments.

Nous allons voir comment implémenter une pile capable de stocker des **entiers**, mais bien sûr nous pouvons adapter le code pour le faire fonctionner avec d'autres types si besoin.

Implémentation d'une pile de capacité finie

Pile de capacité finie

On crée un **type structuré `pile`** contenant les champs suivants :

- un `int` **`capacite`** indiquant la capacité maximale de notre pile ;
- un `int` **`nb`** indiquant le nombre d'éléments stockés dans la pile ;
- un tableau `int *`**`contenu`** de taille **`capacite`**, dont les **`nb`** premières cases contiendront les éléments de la pile.

Implémentation d'une pile de capacité finie

pile.c (1/2)

```c
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>

struct pile
{
  int *contenu;
  int nb;
  int capacite;
};
typedef struct pile pile;

pile creer_pile(int c)
{
  pile p;
  p.contenu = (int *)malloc(c * sizeof(int));
  p.nb = 0;
  p.capacite = c;
  return p;
}

bool pile_est_vide(pile p)
{
  return (p.nb == 0);
}
```

pile.c (2/2)

```c
int sommet(pile p)
{
  assert(!pile_est_vide(p));
  return p.contenu[p.nb-1];
}

int depiler(pile *p)
{
  assert(!pile_est_vide(*p));
  p->nb--;
  return p->contenu[p->nb];
}

void empiler(pile *p, int x)
{
  assert(p->nb < p->capacite);
  p->nb++;
  p->contenu[p->nb-1] = x;
}

void detruire_pile(pile p)
{
  free(p.contenu);
}
```

Implémentation d'une pile de capacité finie

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>
#include "pile.h"

int main()
{
  pile p = creer_pile(10);
  empiler(&p,0);
  empiler(&p,1);
  empiler(&p,2);
  empiler(&p,3);
  printf("sommet : %d\n", sommet(p));
  printf("depiler : %d\n", depiler(&p));
  printf("depiler : %d\n", depiler(&p));
  empiler(&p,4);
  empiler(&p,5);

  while (!pile_est_vide(p))
  {
    printf("depiler : %d\n", depiler(&p));
  }

  detruire_pile(p);
  return EXIT_SUCCESS;
}
```

Output

```
sommet : 3
depiler : 3
depiler : 2
depiler : 5
depiler : 4
depiler : 1
depiler : 0
```

Implémentation d'une pile de capacité finie

Remarque

Les **transformateurs** ont besoin de prendre en argument un **pointeur** vers la **structure** pour pouvoir la modifier par **effets de bords**.

Implémentation d'une pile de capacité finie

Complexité

Toutes les fonctions de cette implémentation ont une complexité temporelle en $O(1)$.

Remarque

Il est très difficile d'analyser la complexité d'un appel à **`malloc`** ou à **`free`** (en particulier car cela dépend du compilateur utilisé).

Pour simplifier, on considèrera qu'un appel à ces fonctions se fait en temps constant ($O(1)$).

Exemple : vérifier si une expression est bien parenthésée

Exemple

Déterminer si un mot est bien parenthésé et indiquer (via des `printf`) les couples de positions des parenthèses ouvrantes et fermantes.

Idée de l'algorithme

Idée

- On crée une pile (initialement vide), et on parcourt la chaine de caractères passée en entrée de gauche à droite.
- Lorsqu'on trouve une parenthèse ouvrante, on empile sa position sur la pile.
- Lorsqu'on trouve une parenthèse fermante, deux cas peuvent se produire...

Idée de l'algorithme

Idée

- Lorsqu'on trouve une parenthèse fermante, deux cas peuvent se produire :
  - si la pile est non vide, on dépile l'élément en haut de la pile : c'est l'indice de la parenthèse ouvrante correspondante ;
  - si la pile est vide, alors la parenthèse fermante n'a jamais été ouverte, donc le mot n'est pas bien parenthésé.
- À la fin du parcours de la chaine, on teste si la pile est vide. Si ce n'est pas le cas, alors une parenthèse ouvrante n'a jamais été fermée, donc le mot n'est pas bien parenthésé.

Exemple : vérifier si une expression est bien parenthésée

```
1  bool bien_parenthesee(char *s)
2  {
3    pile p = creer_pile(strlen(s));
4    for (int i = 0; i < strlen(s); i++)
5    {
6      if (s[i] == '(') {
7        empiler(&p,i);
8      } else if (s[i] == ')') {
9        if (pile_vide(p)) {
10         printf("La parenthese %d n'a jamais ete ouverte.\n", i);
11         return false;
12       } else {
13         int x = depiler(&p);
14         printf("La parenthese ouverte en %d est fermee en %d.\n", x, i);
15       }
16     } else {
17       printf("La chaîne ne contient pas que des parentheses.\n");
18       exit(EXIT_FAILURE);
19     }
20   }
21   if (pile_vide(p)) {
22     printf("La chaîne est bien parenthesee.\n");
23     return true;
24   } else {
25     printf("La parenthese %d n'a jamais ete fermee.\n", sommet(p));
26     return false;
27   }
28 }
```

Exemple : vérifier si une expression est bien parenthésée

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include "pile.h"

bool bien_parenthesee(char *s);

int main()
{
  char *s1 = "(())()";
  printf("%s\n",s1);
  bien_parenthesee(s1);
  char *s2 = "(()";
  printf("%s\n",s2);
  bien_parenthesee(s2);
  char *s3 = "(()()))(()";
  printf("%s\n",s3);
  bien_parenthesee(s3);
}
```

Output

```
(())()
La parenthese ouverte en 1 est fermee en 2.
La parenthese ouverte en 0 est fermee en 3.
La parenthese ouverte en 4 est fermee en 5.
La chaîne est bien parenthesee.
(()
La parenthese ouverte en 1 est fermee en 2.
La parenthese 0 n'a jamais ete fermee.
(()()))(()
La parenthese ouverte en 1 est fermee en 2.
La parenthese ouverte en 3 est fermee en 4.
La parenthese ouverte en 0 est fermee en 5.
La parenthese 6 n'a jamais ete ouverte.
```

Les listes chaînées

Liste chaînée

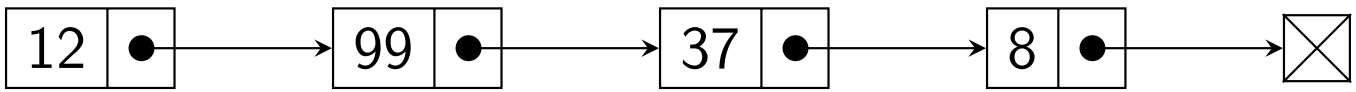
Une **liste chaînée** est définie de manière récursive :

- la liste vide est une liste chaînée ;
- si elle n'est pas vide, une liste chaînée est constituée d'un élément, suivi d'une liste chaînée.

Exemple

Voici une liste constituée des éléments 12, 99, 37 et 8.

La liste est constituée de l'élément 12, puis de la liste constituée des éléments 99, 37 et 8.



Implémentation en C

Pour implémenter cette structure en C, il sera plus pratique que notre liste commence par un **maillon factice**, dont on n'utilisera pas la valeur, mais qui pointera vers le "**vrai**" **premier maillon** de la chaîne.

De cette façon, on pourra représenter la **liste vide** par un **maillon factice** qui pointe vers `NULL`.

Les listes chaînées

liste.h

```c
#include <stdbool.h>

struct maillon;
typedef struct maillon maillon;
typedef struct maillon *liste;

liste creer_liste();
bool liste_est_vide(liste l);
int liste_tete(liste l);
liste liste_suivant(liste l);
void liste_insertion(liste l, int x);
void liste_suppression(liste l);
void detruire_liste(liste l);
```

liste.c (1/3)

```c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "liste.h"

struct maillon {
    int val;
    liste suivant;
};
```

liste.c (2/3)

```c
liste creer_liste()
{
  liste l = (liste)malloc(sizeof(maillon));
  l->suivant = NULL;
  return l;
}

bool liste_est_vide(liste l)
{
  return l->suivant == NULL;
}

int liste_tete(liste l)
{
  assert(l->suivant != NULL);
  return l->suivant->val;
}

liste liste_suivant(liste l)
{
  return l->suivant;
}
```

Les listes chaînées

liste.c (3/3)

```c
void liste_insertion(liste l, int x)
{
  maillon *m = (maillon *) malloc(sizeof(maillon));
  m->val = x;
  m->suivant = l->suivant;
  l->suivant = m;
}

void liste_suppression(liste l)
{
  assert(l->suivant != NULL);
  liste s = l->suivant;
  l->suivant = s->suivant;
  free(s);
}

void detruire_liste(liste l)
{
  while (!liste_est_vide(l))
  {
    liste_suppression(l);
  }
  free(l);
}
```

Les listes chaînées

main.c (1/2)

```c
#include <stdio.h>
#include "liste.h"

void liste_affiche(liste l)
{
  liste iter;
  for (iter = l; !liste_est_vide(iter); iter = liste_suivant(iter))
  {
    printf("%d -> ", liste_tete(iter));
  }
  printf("NULL\n");
}
```

Exemple

La fonction ci-dessus affiche le contenu de la liste **`l`**.

Les listes chaînées

main.c (2/2)

```c
int main()
{
  liste l = creer_liste();
  for(int i = 0; i < 5; i++)
  {
    liste_insertion(l, i);
  }
  liste_affiche(l);

  liste iter = l;
  while (!liste_est_vide(iter))
  {
    if (liste_tete(iter) % 2 == 0)
    {
      liste_suppression(iter);
    }
    else
    {
      iter = liste_suivant(iter);
    }
  }
  liste_affiche(l);

  detruire_liste(l);
}
```

Console

```
eleve@mpi2:~$ gcc main.c liste.c -o main
eleve@mpi2:~$ ./main
4 -> 3 -> 2 -> 1 -> 0 -> NULL
3 -> 1 -> NULL
```

Exemple

La boucle **while** ci-contre supprime les maillons de **l** contenant des valeurs paires.

Complexité

La fonction **`detruire_liste`** a une complexité en $O(n)$, où n est la taille de la liste.

Les autres fonctions de **`liste.c`** s'effectuent en temps constant ($O(1)$).

Les listes chaînées

Longueur d'une liste chaînée

```
1  int longueur(liste l)
2  {
3    if (liste_est_vide(l))
4    {
5      return 0;
6    }
7    else
8    {
9      return 1 + longueur(liste_suivant(l));
10   }
11 }
```

Exemple

La fonction ci-dessus renvoie le nombre d'éléments de la liste **`l`**.

Sa complexité est **linéaire** en la taille de la liste ($O(n)$).

Structure de file

Files

File

La file est une autre structure linéaire, assez semblable à la pile, mais qui fonctionne sur le principe FIFO (first in, first out).

Lorsqu'on insère un élément dans une file, celui-ci ne pourra être retiré qu'après que tous les éléments insérés avant l'aient été.

Files

Définition (File)

Une **file** est une structure abstraite, supportant les opérations suivantes :

- **création** d'une file vide ;
- test d'**égalité** au vide ;
- **retrait** de l'élément en tête d'une file non vide ;
- **ajout** d'un élément en queue de file.

Files

Exemple

La file n'est pas d'un usage aussi courant que la pile en informatique, citons néanmoins quelques exemples :

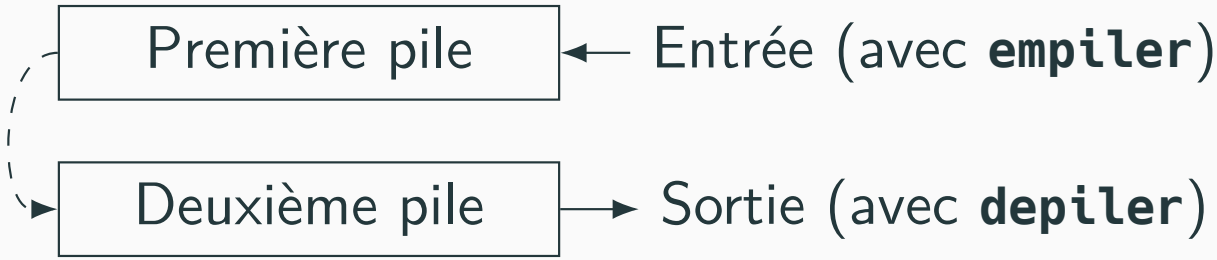
- une imprimante "bas de gamme" (qui ne gère pas les priorités, cf. section suivante) à qui l'on envoie des documents à imprimer les traitera séquentiellement : le premier document à être envoyé sera imprimé en premier ;
- le parcours en largeur d'un graphe traite les sommets par distance à l'origine (nombre minimal d'arêtes à parcourir depuis l'origine) croissante : il suffit de rajouter les sommets dans une file lorsqu'ils sont découverts pour respecter cet ordre dans le parcours.

Implémentation d'une file à l'aide de piles

Implémentation

Pour implémenter une file, on peut utiliser deux piles.

Implémentation d'une file à l'aide de piles



Principe

- L'ajout d'un élément à la file se fait uniquement avec **`empiler`** dans la première pile.
- La suppression se fait avec **`depiler`** dans la deuxième pile.
- Lorsque la deuxième pile est vide et qu'on veut défiler (sortir un élément de la liste), il faut transférer les éléments de la première pile dans la deuxième.

↪ Pour conserver l'ordre des éléments de la file, il faut alors inverser l'ordre des éléments de la pile lors du transfert, ce qui se fera naturellement.

Implémentation d'une file à l'aide de piles

pile.h

```c
#include <stdbool.h>

struct maillon;
typedef struct maillon maillon;
typedef struct maillon *pile;

pile creer_pile();
bool pile_est_vide(pile p);
int sommet(pile p);
void empiler(pile p, int x);
pile pile_suivant(pile p);
int depiler(pile p);
void detruire_pile(pile p);
```

file.h

```c
#include <stdbool.h>
#include "pile.h"

struct file
{
  pile p1;
  pile p2;
};
typedef struct file file;

file creer_file();
bool file_est_vide(file f);
void enfiler(file f, int x);
int defiler(file f);
void detruire_file();
```

Implémentation

Supposons que l'on dispose du fichier **pile.h** ci-dessus.

On peut alors utiliser le type **pile** (sans savoir exactement comment il a été implémenté) pour définir notre type **file**.

Implémentation d'une file à l'aide de piles

pile.h

```c
#include <stdbool.h>

struct maillon;
typedef struct maillon maillon;
typedef struct maillon *pile;

pile creer_pile();
bool pile_est_vide(pile p);
int sommet(pile p);
void empiler(pile p, int x);
pile pile_suivant(pile p);
int depiler(pile p);
void detruire_pile(pile p);
```

file.h

```c
#include <stdbool.h>
#include "pile.h"

struct file
{
  pile p1;
  pile p2;
};
typedef struct file file;

file creer_file();
bool file_est_vide(file f);
void enfiler(file f, int x);
int defiler(file f);
void detruire_file();
```

Remarque

J'ai testé le programme qui va suivre avec une implémentation des piles via des listes chaînées, mais tout devrait fonctionner si jamais on changeait cette implémentation par une autre.

Implémentation d'une file à l'aide de piles

file.c (1/2)

```c
#include <assert.h>
#include "file.h"

file creer_file()
{
  file f;
  f.p1 = creer_pile();
  f.p2 = creer_pile();
  return f;
}

bool file_est_vide(file f)
{
  return (pile_est_vide(f.p1)
      && pile_est_vide(f.p2));
}

void enfiler(file f, int x)
{
  empiler(f.p1, x);
}
```

file.c (2/2)

```c
int defiler(file f)
{
  assert(!file_est_vide(f));
  if (pile_est_vide(f.p2))
  {
    /* si f n'est pas vide et que p2 est vide
     * c'est que p1 n'est pas vide :
     * il faut alors vider p1 dans p2 */
    while (!pile_est_vide(f.p1))
    {
      empiler(f.p2, depiler(f.p1));
    }
  }
  return depiler(f.p2);
}

void detruire_file(file f)
{
  detruire_pile(f.p1);
  detruire_pile(f.p2);
}
```

Implémentation d'une file à l'aide de piles

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "file.h"

int main()
{
  file f = creer_file();
  for (int i = 0; i < 5; i++)
  {
    enfiler(f, i);
  }
  printf("%d\n", defiler(f));
  printf("%d\n", defiler(f));
  for (int i = 5; i < 8; i++)
  {
    enfiler(f, i);
  }
  while (!file_est_vide(f))
  {
    printf("%d\n", defiler(f));
  }
}
```

Console

```
eleve@mp2i:~$ gcc main.c file.c pile.c -o main
eleve@mp2i:~$ ./main
0
1
2
3
4
5
6
7
```

Implémentation d'une file à l'aide de piles

Complexité

- La fonction **`detruire_file`** a la même complexité que la fonction **`detruire_pile`**.
- Les fonctions **`creer_file`**, **`file_est_vide`**, et **`enfiler`** ont toutes une complexité en $O(1)$.
- La fonction **`defiler`** à une complexité en $O(1)$ dans le meilleur des cas, et $O(n)$ dans le pire des cas, où n est le nombre d'éléments présents dans la file.
  - $\hookrightarrow$ Si le cas pire se produit, on va passer dans la boucle **`while`**, et **`f.p2`** contiendra alors les $n-1$ éléments de la file : les $n-1$ prochains appels à **`defiler`** se feront donc en $O(1)$.
  - $\hookrightarrow$ **`defiler`** a donc une complexité amortie en $O(1)$.

Listes doublement chaînées

On peut également implémenter une structure de **file** à l'aide d'une généralisation des **listes chaînées** : les **listes doublement chaînées**.

Dans une telle structure, chaque maillon possède :

- une **valeur** ;
- un pointeur vers le maillon **suivant** ;
- un pointeur vers le maillon **précédent**.

Il faudra également qu'on garde un **pointeur** vers le **début de la chaîne**, et un autre **pointeur** vers la **fin de la chaîne**. Pour cela, il sera plus pratique d'avoir **deux maillons fictifs** : un au **début** et un à la **fin**.

Les listes doublement chaînées

double_linked.h

```c
#include <stdbool.h>

struct maillon;
typedef struct maillon maillon;
struct file;
typedef struct file file;

file *creer_file();
bool file_est_vide(file *f);
void enfiler(file *f, int x);
int defiler(file *f);
void detruire_file(file *f);
```

double_linked.c (1/4)

```c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "double_linked.h"

typedef struct maillon maillon;
struct maillon
{
  int val;
  maillon *suiv;
  maillon *prec;
};
```

double_linked.c (2/4)

```c
struct file
{
  maillon *fictif_d;
  maillon *fictif_f;
};
typedef struct file file;

file *creer_file()
{
  file *f = (file *)malloc(sizeof(file));
  maillon *debut = (maillon *)malloc(sizeof(maillon));
  maillon *fin = (maillon *)malloc(sizeof(maillon));
  debut->suiv = fin;
  debut->prec = NULL;
  fin->suiv = NULL;
  fin->prec = debut;
  f->fictif_d = debut;
  f->fictif_f = fin;
  return f;
}

bool file_est_vide(file *f)
{
  return f->fictif_d->suiv == f->fictif_f;
}
```

Les listes doublement chaînées

double_linked.c (3/4)

```c
void enfiler(file *f, int x)
{
  maillon *m = (maillon *)malloc(sizeof(maillon));
  m->val = x;
  m->suiv = f->fictif_d->suiv;
  m->prec = f->fictif_d;
  f->fictif_d->suiv = m;
  m->suiv->prec = m;
}

int defiler(file *f)
{
  assert(!file_est_vide(f));
  maillon *m = f->fictif_f->prec;
  int x = m->val;
  f->fictif_f->prec = m->prec;
  m->prec->suiv = f->fictif_f;
  free(m);
  return x;
}
```

double_linked.c (4/4)

```c
void detruire_file(file *f)
{
  while (!file_est_vide(f))
  {
    defiler(f);
  }
  free(f->fictif_d);
  free(f->fictif_f);
  free(f);
}
```

Les listes doublement chaînées

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "double_linked.h"

int main()
{
  file *f = creer_file();
  for (int i = 0; i < 5; i++)
  {
    enfiler(f, i);
  }
  printf("%d\n", defiler(f));
  printf("%d\n", defiler(f));
  for (int i = 5; i < 8; i++)
  {
    enfiler(f, i);
  }
  while (!file_est_vide(f))
  {
    printf("%d\n", defiler(f));
  }
  detruire_file(f);
}
```

Console

```
eleve@mp2i:~$ gcc main.c double_linked.c -o main
eleve@mp2i:~$ ./main
0
1
2
3
4
5
6
7
```

Complexité

Avec cette implémentation :

- la fonction **`detruire_file`** a une complexité en $O(n)$, où n est le nombre d'éléments dans la file ;
- toutes les autres fonctions ont une complexité en $O(1)$ dans le pire des cas.

Structure de liste (ou tableau dynamique)

Structure de liste (ou tableau dynamique)

Listes Python

Pour finir ce chapitre, nous allons essayer d'implémenter la structure des listes Python. Les listes Python sont une structure hybride entre une pile et un tableau.

Nous allons ainsi créer un type abstrait **`liste`** qui devra disposer des opérations suivantes :

- `liste *creer_liste();`
  crée une liste vide (`[]` en Python) ;
- `bool liste_est_vide(liste *L);`
  teste si la liste **`L`** est vide (`L == []` en Python) ;
- `int longueur(liste *L);`
  renvoie la longueur de **`L`** (`len(L)` en Python) ;

Structure de liste (ou tableau dynamique)

Listes Python

Pour finir ce chapitre, nous allons essayer d'implémenter la structure des listes Python. Les listes Python sont une structure hybride entre une pile et un tableau.

Nous allons ainsi créer un type abstrait **`liste`** qui devra disposer des opérations suivantes :

- **`void append(liste *L, int val);`**
  crée une nouvelle case à la fin de **`L`** contenant **`val`** (`L.append(val)` en Python) ;
- **`int pop(liste *L);`**
  renvoie la dernière valeur de la liste **`L`** et supprime cette case (`L.pop()` en Python) ;

Structure de liste (ou tableau dynamique)

Listes Python

Pour finir ce chapitre, nous allons essayer d'implémenter la structure des listes Python. Les listes Python sont une structure hybride entre une pile et un tableau.

Nous allons ainsi créer un type abstrait **`liste`** qui devra disposer des opérations suivantes :

- `void modifier(liste *L, int i, int x);`
  modifie la i-ème valeur de la liste **`L`** par **`x`** (`L[i] = x` en Python) ;
- `int valeur(liste *L, int i);`
  renvoie la i-ème valeur de la liste **`L`** (`L[i]` en Python) ;

Structure de liste (ou tableau dynamique)

Listes Python

Pour finir ce chapitre, nous allons essayer d'implémenter la structure des listes Python. Les listes Python sont une structure hybride entre une pile et un tableau.

Nous allons ainsi créer un type abstrait **`liste`** qui devra disposer des opérations suivantes :

- `void detruire_liste(liste *L);`
  détruit la liste `L` (libère la mémoire utilisée).

Structure de liste Python (avec des listes doublement chaînées)

Implémentation

Puisque certaines opérations (**pop** et **append**) correspondent à des opérations sur des piles, on pourrait essayer d'utiliser une structure de liste doublement chaînée comme précédemment.

Notre structure de liste pourrait également avoir un champ **`longueur`**, pour pouvoir obtenir la longueur efficacement.

Structure de liste Python (avec des listes doublement chaînées)

liste_python.h

```c
#include <stdbool.h>

struct maillon;
typedef struct maillon maillon;
struct liste;
typedef struct liste liste;

liste *creer_liste();
bool file_est_vide(liste *L);
int longueur(liste *L);
void append(liste *L, int val);
int pop(liste *L);
void modifier(liste *L, int i, int x);
int valeur(liste *L, int i);
void detruire_liste(liste *L);
```

liste_python.c (1/5)

```c
typedef struct maillon maillon;
struct maillon
{
  int val;
  maillon *suiv;
  maillon *prec;
};
```

liste_python.c (2/5)

```c
struct liste
{
  maillon *fictif_d;
  maillon *fictif_f;
  int longueur;
};
typedef struct liste liste;

liste *creer_liste()
{
  liste *L = (liste *)malloc(sizeof(liste));
  maillon *debut = (maillon *)malloc(sizeof(maillon));
  maillon *fin = (maillon *)malloc(sizeof(maillon));
  debut->suiv = fin;
  debut->prec = NULL;
  fin->suiv = NULL;
  fin->prec = debut;
  L->fictif_d = debut;
  L->fictif_f = fin;
  L->longueur = 0;
  return L;
}

bool liste_est_vide(liste *L)
{
  return L->longueur == 0;
}
```

Structure de liste Python (avec des listes doublement chaînées)

liste_python.c (3/5)

```c
int longueur(liste *L)
{
  return L->longueur;
}

void append(liste *L, int x)
{
  maillon *m = (maillon *)malloc(sizeof(maillon));
  m->val = x;
  m->suiv = L->fictif_f;
  m->prec = L->fictif_f->prec;
  L->fictif_f->prec = m;
  m->prec->suiv = m;
  L->longueur++; // nouveauté par rapport à la partie précédente
}

int pop(liste *L)
{
  assert(!liste_est_vide(L));
  maillon *m = L->fictif_f->prec;
  int x = m->val;
  L->fictif_f->prec = m->prec;
  m->prec->suiv = L->fictif_f;
  free(m);
  L->longueur--; // nouveauté par rapport à la partie précédente
  return x;
}
```

Structure de liste Python (avec des listes doublement chaînées)

liste_python.c (4/5)

```c
/* renvoie un pointeur
   vers le i-ème maillon
  de la liste L */
maillon *trouver(liste *L, int i)
{
  assert(i < L->longueur);
  maillon *m = L->fictif_d->suiv;
  for (int j = 0; j < i; j++)
  { /* m pointe vers le j-ème maillon de la liste */
    m = m->suiv;
  }
  return m;
}

void modifier(liste *L, int i, int x)
{
  maillon *m = trouver(L, i);
  m->val = x;
}

int valeur(liste *L, int i)
{
  maillon *m = trouver(L, i);
  return m->val;
}
```

liste_python.c (5/5)

```c
void detruire_liste(liste *L)
{
  while (!liste_est_vide(L))
  {
    pop(L);
  }
  free(L->fictif_d);
  free(L->fictif_f);
  free(L);
}
```

Structure de liste Python (avec des listes doublement chaînées)

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "liste_python.h"

void affiche_liste(liste *L)
{
  for (int i = 0; i < longueur(L); i++)
  {
    printf("%d ", valeur(L, i));
  }
  printf("\n");
}

int main()
{
  liste *L = creer_liste();
  for (int i = 0; i < 5; i++)
  {
    append(L, i);
  }
  affiche_liste(L);
  modifier(L, 2, 12);
  pop(L);
  append(L, 42);
  append(L, 100);
  affiche_liste(L);
  detruire_liste(L);
}
```

Console

```
eleve@mp2i:~$ gcc main.c liste_python.c -o main
eleve@mp2i:~$ ./main
0 1 2 3 4
0 1 12 3 42 100
```

Structure de liste Python (avec des listes doublement chaînées)

Complexité

Avec cette implémentation :

- les fonctions **`creer_liste`**, **`liste_est_vide`**, et **`longueur`** ont une complexité en $O(1)$;
- grâce à l'utilisation des listes doublement chaînées, les fonctions **`append`** et **`pop`** ont également une complexité en $O(1)$;
- la fonction **`detruire_liste(L)`** a une complexité en $O($**`L->longueur`**$)$;
- en revanche, les fonctions **`modifier(L, i, x)`** et **`valeur(L, i)`** ont une complexité en $O(i)$, car la fonction **`trouver(L, i)`** n'a pas d'autre choix que de partir du début de la liste doublement chaînée, et d'avancer de maillon en maillon jusqu'au i-ème maillon.

Structure de liste Python (avec des listes doublement chaînées)

Problème

Les complexités des fonctions **`modifier`** et **`valeur`** ne sont pas très satisfaisantes, car parcourir tous les éléments d'une liste de longueur n s'effectue alors en $O(n^2)$.

C'est par exemple le cas de la fonction **`affiche_liste`**.

Structure de liste Python (avec des tableaux dynamiques)

Tableaux dynamiques

Le langage Python est codé en C. Mais le type `list` de Python est implémenté d'une autre manière, pour garantir que l'accès ou la modification de `L[i]` s'effectue en $O(1)$.

Cette implémentation est une version améliorée de notre première tentative d'implémentation de piles avec un **tableau**.

Structure de liste Python (avec des tableaux dynamiques)

Tableaux dynamiques

- On va créer un tableau, dont les **`L->longueur`** premières cases contiendront les éléments de notre liste.
- Ainsi, il sera facile (et rapide) d'accéder au i-ème élément de la liste : c'est le i-ème élément du tableau.
- Pour les fonctions **pop** et **append**, nous ferons comme au début de ce chapitre, avec une petite amélioration.

Structure de liste Python (avec des tableaux dynamiques)

Tableaux dynamiques

Si on appelle la fonction **append** alors que le tableau (de taille n) est plein :

- on crée un nouveau tableau de taille $2n + 1$;
- on recopie les n valeurs de l'ancien tableau au début du nouveau tableau ;
- on libère l'ancien tableau de la mémoire, et notre liste utilise désormais le nouveau tableau, dans lequel on a la place de faire le **append**.

Structure de liste Python (avec des tableaux dynamiques)

Complexité de append

Avec cette implémentation, **append** a une complexité dans le pire des cas en $O(n)$ (car il faut recopier les n valeurs de l'ancien tableau).

L'intérêt de créer un nouveau tableau de taille $2n+1$ est d'avoir une complexité amortie en $O(1)$.

En effet, si le cas pire se produit, alors les n dernières cases du nouveau tableau sont inutilisées, donc on a la garantie que les n prochains **append** se feront en $O(1)$.

Complexité

Toutes les autres opérations auront une complexité en $O(1)$.

Structure de liste Python (avec des tableaux dynamiques)

dyna.h

```c
#include <stdbool.h>

struct liste;
typedef struct liste liste;

liste *creer_liste();
bool file_est_vide(liste *L);
int longueur(liste *L);
void append(liste *L, int val);
int pop(liste *L);
void modifier(liste *L, int i, int x);
int valeur(liste *L, int i);
void detruire_liste(liste *L);
```

dyna.c (1/4)

```c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "liste_python.h"

struct liste
{
  int *tab;
  int n_max;
  int longueur;
};
typedef struct liste liste;
```

dyna.c (2/4)

```c
liste *creer_liste()
{
  liste *L = (liste *)malloc(sizeof(liste));
  L->n_max = 1;
  L->tab = (int *)malloc(L->n_max * sizeof(int));
  L->longueur = 0;
  return L;
}

bool liste_est_vide(liste *L)
{
  return L->longueur == 0;
}

int longueur(liste *L)
{
  return L->longueur;
}

int pop(liste *L)
{
  assert(!liste_est_vide(L));
  int x = L->tab[L->longueur - 1];
  L->longueur--;
  return x;
}
```

Structure de liste Python (avec des tableaux dynamiques)

dyna.c (3/4)

```c
void append(liste *L, int x)
{
  if (L->longueur == L->n_max)
  {
    L->n_max = 2 * L->n_max + 1;
    int *new_tab = (int *)malloc(L->n_max * sizeof(int));
    for (int i = 0; i < L->longueur; i++)
    {
      new_tab[i] = L->tab[i];
    }
    free(L->tab);
    L->tab = new_tab;
  }
  L->longueur++;
  L->tab[L->longueur - 1] = x;
}
```

dyna.c (4/4)

```c
void modifier(liste *L, int i, int x)
{
  assert(i < L->longueur);
  L->tab[i] = x;
}

int valeur(liste *L, int i)
{
  assert(i < L->longueur);
  return L->tab[i];
}

void detruire_liste(liste *L)
{
  free(L->tab);
  free(L);
}
```

Structure de liste Python (avec des tableaux dynamiques)

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "dyna.h"

void affiche_liste(liste *L)
{
  for (int i = 0; i < longueur(L); i++)
  {
    printf("%d ", valeur(L, i));
  }
  printf("\n");
}

int main()
{
  liste *L = creer_liste();
  for (int i = 0; i < 5; i++)
  {
    append(L, i);
  }
  affiche_liste(L);
  modifier(L, 2, 12);
  pop(L);
  append(L, 42);
  append(L, 100);
  affiche_liste(L);
  detruire_liste(L);
}
```

Console

```
eleve@mp2i:~$ gcc main.c dyna.c -o main
eleve@mp2i:~$ ./main
0 1 2 3 4
0 1 12 3 42 100
```