Informatique en MP(2)I

Marc de Falco

Piles et files, structures abstraites et implémentations

I Piles

I.1 Principe

Une pile est une structure de donnée abstraite permettant d'ajouter et de retirer des éléments selon le principe **LIFO : L**ast **I**n **F**irst **O**ut

C'est-à-dire que le premier élément retiré sera celui qui a été ajouté en dernier. La bonne manière de se représenter une pile est donc d'imaginer une pile de dossiers à traiter sur un bureau. Chaque nouveau dossier est empilé et on traite à chaque fois le dossier sur le dessus.

> **Remarque** Ce n'est sûrement pas la manière la plus efficace de gérer des dossiers et l'on risque d'avoir des dossiers empilés depuis très longtemps sans être traités.

On va avoir quatre opérations :

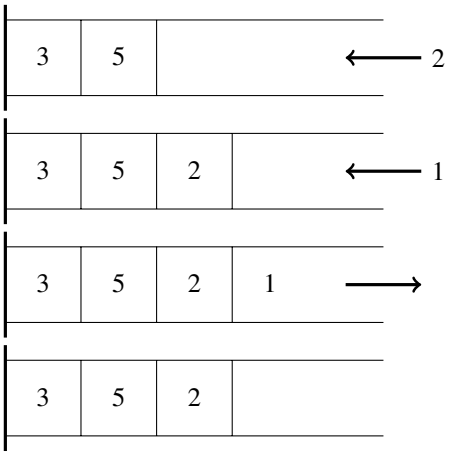
- Créer une pile vide
- Tester si une pile est vide
- Ajouter un élément à une pile, on appelle cette opération *empiler* et en anglais *push*
- Retirer un élément à une pile et le renvoie, on appelle cette opération *dépiler* et en anglais *pop*

Le comportement opérationnel de cette structure abstraite est donné par le principe **LIFO** précédent.

On considère qu'il est très important que ces quatre opérations soient de complexité en temps constante, éventuellement amorti. Par contre, il est possible de limiter le nombre maximum d'éléments. C'est une vision qui est en accord avec le rôle des piles pour gérer des tâches à traiter pour lesquelles on a souvent un majorant sur leur nombre.

I.2 Représentation visuelle

Comme souvent, la représentation visuelle est importante car c'est elle qui permet de raisonner efficacement. Ici, on va représenter une pile verticalement ou horizontalement avec un côté fermé : le *fond* de pile et un côté ouvert permettant d'empiler et de dépiler des éléments.



I.3 Implémentations

I.3.i Dans un tableau borné

Reprenant la remarque précédente, on va présenter une implémentation très standard des piles dans des tableaux de taille fixe. L'idée est de considérer un tableau `t` de taille N muni d'un entier `fond` indiquant l'indice du prochain élément libre.

Au départ, `fond` vaut 0. Chaque fois qu'on empile un élément, on le place à l'indice indiqué par `fond` et on l'incrémente. Pour dépiler, on décrémente `fond` et on renvoie l'élément à cet indice. En fait, les éléments ne sont pas vraiment retirés. Il s'agit exactement du comportement de la pile d'exécution du compilateur vue au chapitre sec:pileexec.

Le choix de la valeur N donnant le nombre maximum d'éléments empilables est critique : il faut qu'elle soit grande devant l'estimation qu'on peut faire du nombre d'empilements. L'implémentation d'une telle pile se fait en général dans des conditions où l'on va éviter de vérifier l'absence de dépassement. Dans le code qui va suivre, on a fait le choix de passer par des `assert` pour vérifier ces conditions uniquement en mode développement.

> **Remarque** En `C`, on passera le plus souvent par un tableau statique avec `N` défini à une grande valeur entière par un `#define`. On présente une version plus dans l'esprit de ces chapitres structures de données.

On va ainsi définir un type `pile` avec des pointeurs :

```c
struct pile {
  int *t;
  int fond;
};

typedef struct pile pile;
```

C

Pour ajouter et retirer un élément, comme on va manipuler l'entier `fond`, il est nécessaire de passer `pile` par pointeur. On fait ainsi ce choix pour l'ensemble des fonctions.

L'opération de création doit faire une allocation et cela va donc nécessiter une opération explicite de destruction :

```c
int N  = 1000; // La pile sera de 1000 éléments au maximum

pile *pile_creer()
{
  pile *p = malloc(sizeof(pile));
  p->t = malloc(sizeof(int) * N);
  p->fond = 0;
  return p;
}

void pile_detruire(pile *p)
```

```c
{
  free(p->t);
  free(p);
}
```
C

Empiler et dépiler correspond directement à la description précédente.

```c
void empiler(pile *p, int x)
{
  assert(p->fond < N);
  p->t[p->fond] = x;
  p->fond++;
}

int retirer(pile *p)
{
  assert(p->fond > 0);
  p->fond--;
  return p->t[p->fond];
}

bool est_vide(pile *p)
{
  return p->fond == 0;
}
```
C

I.3.ii Dans un tableau dynamique

On a vu qu'on pouvait définir des tableaux redimensionnables et que cela permettait d'effectuer des ajouts et des suppressions en complexité amortie constante.

On peut donc directement réaliser une pile à l'aide de cette structure en ajoutant et supprimant des éléments. C'est l'interface usuelle avec le type `list` de `Python` qui propose déjà cela avec les fonctions `append` et `pop`. On peut rajouter une surcouche superficielle pour retrouver la nomenclature précédente :

```python
def creer_pile():
  return []

def est_vide(p):
  return p == []

def empiler(p, x):
  p.append(x)

def depiler(p):
  return p.pop()
```
Python

I.3.iii Avec des listes chaînées

Avec des listes chaînées, l'idée est d'ajouter et de supprimer des éléments en tête. En effet, on a vu que ces deux opérations étaient en temps constant, ce qui est particulièrement adapté aux piles.

En `OCaml`, on remarque que les listes par défaut conviennent, cependant, il est nécessaire de changer de liste pour rajouter ou supprimer un élément, c'est pourquoi on définit le type `'a pile` comme une `'a list ref` c'est-à-dire un pointeur sur un maillon de tête comme on aurait pu le faire en `C`.

On définit ainsi directement les quatre opérations :

```
type 'a pile = 'a list ref

let cree_pile () = ref []

let est_vide p = !p = []

let empile p x = p := x :: !p
```

```OCaml
let depile p =
  match !p with
  | [] -> failwith "Pile vide"
  | x::q -> p := q; x
```

I.4 Applications

I.4.i Parenthésage

I.4.ii Évaluation d'expressions

!subsubsubsection(Expression postfixe) !subsubsubsection(Expression infixe)

II Files

II.1 Principe

Une file est une structure de donnée abstraite permettant d'ajouter et de retirer des éléments selon le principe **FIFO : First In First Out**

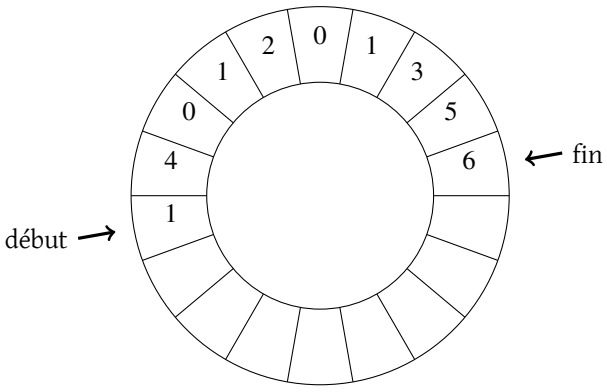
C'est-à-dire que le premier élément retiré sera celui qui a été ajouté le plus anciennement. La bonne manière de se représenter une file est donc d'imaginer la file d'attente à la caisse d'un magasin.

II.2 Implémentations

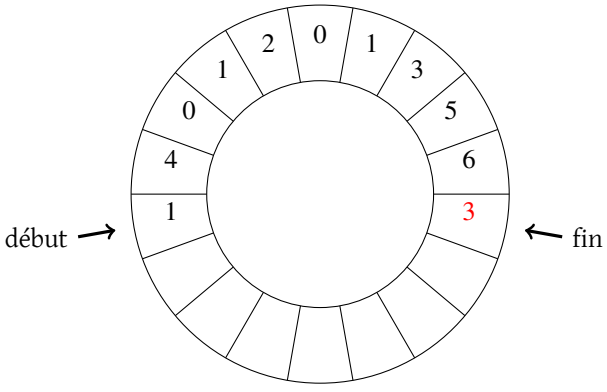
II.2.i Dans un tableau borné

On peut reprendre l'implémentation dans un tableau des piles. Ici, on va conserver deux indices : l'indice de fin et 'indice du début de la file.

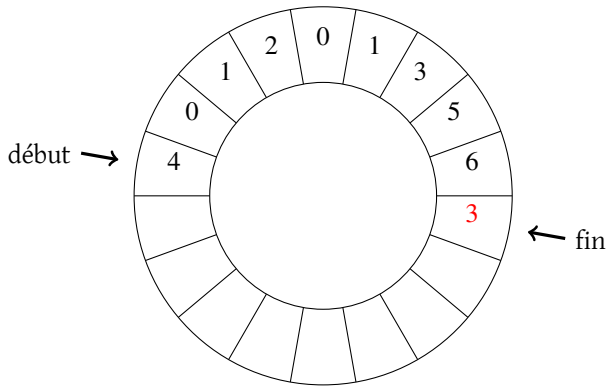
L'idée est de représenter une file comme une plage de cases dans un tableau circulaire :



Pour enfiler, il suffit de décaler fin d'une case vers la droite et d'ajouter à cette case l'élément. Ainsi si on enfile 3 dans l'exemple précédent on obtient :

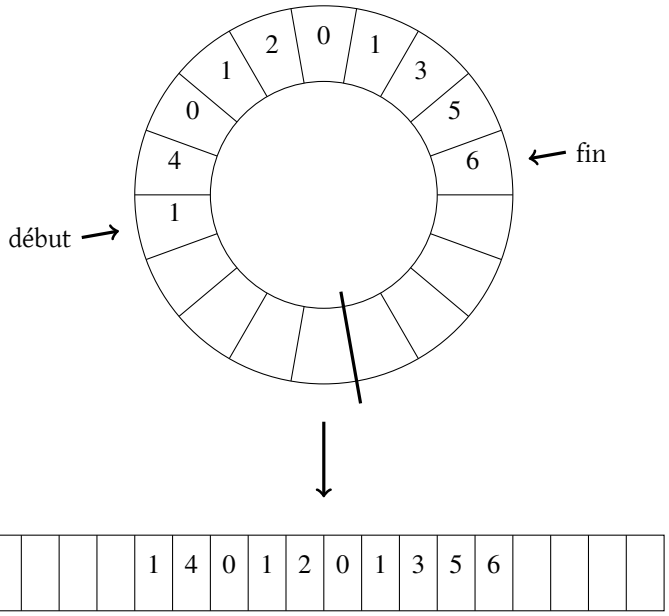


Pour défiler, on va renvoyer l'élément pointé par début et décaler celui-ci d'une case vers la droite. Dans l'exemple précédent, on obtient alors la valeur 1 et le tableau devient :



Si, comme pour les piles, on suppose que la taille de la file ne dépassera jamais le nombre de cases du tableau circulaire, il n'y a pas de risques de dépassement. La question qui se pose alors est celle de l'implémentation d'un tableau circulaire de N cases dans un tableau usuel de N cases.

La technique consiste à couper le tableau arbitrairement entre deux cases et à l'aplatir :



Maintenant, la question du bord se pose : les décalages doivent se faire modulo la longueur du tableau pour avoir le même comportement qu'un tableau circulaire.

On traduit ensuite directement cette représentation en C :

```c
struct file {
  int *t;
  int debut;
  int fin;
};

typedef struct file file;
```

C

II.2.ii Avec des listes chaînées

II.2.iii Avec deux piles

II.3 Applications