IFT2015 hiver 2010 — Notes de cours

Miklós Csűrös

8 février 2010

5 Représentation de listes par tableaux

Pour définir un tableau, on doit spécifier la taille. L'accès au k-ème élément prend $O(1)$. L'accès rapide peut être est exploîté dans l'implantation de listes, mais il faut faire attention à la gestion de la taille.

5.1 Pile par tableau

On peut implanter une pile par un tableau $\mathsf{elements}[0 \ldots \mathsf{taille} - 1]$: on doit maintenir l'indice du sommet séparamment. Sans gestion de taille, on a une limite maximale pour la pile (c'est la taille du tableau alloué au début).

```
Initialisation(N)
1 elements ← tableau de taille N
2 sommet ← 0
3 taille ← N

Opération pop
1 sommet ← sommet − 1
2 x ← elements[sommet]
3 elements[sommet] ← null                    // déstruction explicite de référence
4 retourner x

Opération push(x)
1 elements[sommet] ← x
2 sommet ← sommet + 1
```

Qu'est-ce qui se passe si on a trop d'éléments sur la pile ? → la pile déborde (*overflow*).
Qu'est-ce qui se passe si on essaie de dépiler d'une pile vide ? → la pile déborde négativement (*underflow*).

Gestion dynamique de la taille. On utilise la technique suivante : si le nombre d'éléments sur la pile atteint la taille allouée, on fait une réallocation avec une taille doublée. Similairement, si le nombre d'éléments tombe en-dessous de $1/4$ de la taille, on fait une réallocation avec une taille réduite à moitié.

```
Opération pop
1 sommet ← sommet − 1
2 x ← elements[sommet]
3 elements[sommet] ← null                    // déstruction explicite de référence
4 si sommet < taille/4 alors faire REALLOCATION(⌈taille/2⌉)
5 retourner x

Opération push(x)
1 si sommet = taille alors faire REALLOCATION(2 · taille)
2 elements[sommet] ← x
3 sommet ← sommet + 1
```

Pour la réallocation, on doit créer un tableau de la taille donnée, et copier les éléments sur la pile avant de l'affectation de elements.

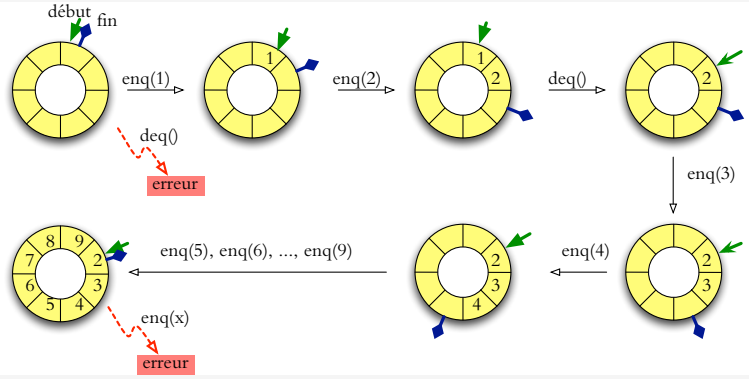
```
REALLOCATION(t)
R1 a[] ← nouveau tableau de taille t
R2 for i ← 0, . . . , sommet − 1 faire
R3     a[i] ← elements[i]
R4 elements ← a
R5 taille ← t
```

Dans le pire cas, une opération de pop ou pop prend maintenant $\Theta(n)$ temps où $n = \text{sommet}$ est le nombre d'éléments sur la pile. Mais cela n'arrive pas trop fréquemment : on a un temps *amorti* de $O(1)$ toujours.

Théorème 5.1. *Une séquence de m opérations de* push *et* pop *prend $O(m)$ temps.*

Démonstration. Le temps d'exécuter une telle séquence est $O(m+r)$ où r est le nombre de fois on exécute la ligne R3. On utilise un système de débits-crédits dans la preuve pour démontrer que $2m \geq r$. Lors d'un push, on met \$2 sur un compte. Lors d'un pop, on y met \$1. On utilise l'argent pour payer la réallocation : on paie \$1 pour copier un élément. À chaque réallocation, on remet la solde à \$0 même s'il reste de l'argent après le copiage. On peut voir que la solde n'est jamais négative. En conséquence, $O(m + r) = O(m + 2m) = O(m)$ temps suffit pour exécuter toute la séquence d'opérations. ■

5.2 Queue par tableau



Idée : utiliser un tableau circulaire («anneau») avec deux indices pour le début et fin de la queue. Anneau en pratique : on utilise $\pmod n$ avec un tableau de taille n.

```
public class Queue
{
   private int debut;
   private int fin;
   private Object[] Q;
   private static final int MAX_TAILLE=2010;
   public Queue()
   {
      debut=fin=0;
      Q=new Object[MAX_TAILLE];
      for (int i=0; i<MAX_TAILLE; i++)
         Q[i] = VIDE;
   }
   private static final Object VIDE=new Object();
   public boolean isEmpty()
   {
      return (Q[debut]==VIDE);
   }
```

On ne veut pas utiliser `null` au lieu de `VIDE` parce qu'on veut permettre `enqueue(null)`...

```
   public Object dequeue()
   {
      Object retval = Q[debut];
      if (retval==VIDE)
         throw new UnderflowException("Rien ici.");
      Q[debut]=VIDE;
      debut = (debut + 1) % MAX_TAILLE;
      return retval;
   }
   static class UnderflowException extends RuntimeException
   { private UnderflowException(String msg){super(msg);}}
```

```
   public void enqueue(Object O)
   {
      if (Q[fin]!=VIDE)
         throw new OverflowException("Queue trop longue.");
      Q[fin]=O;
      fin = (fin+1) % MAX_TAILLE;
   }
   static class OverflowException extends RuntimeException
   {private OverflowException(String msg){super(msg);}}
```

Queue — cases vides. $k = (\texttt{debut} - \texttt{fin}) \bmod n$ peut prendre les valeurs $k = 0, 1, \ldots, n-1$ Si on ne peut pas avoir un élément spécial pour dénoter les cases vides, alors on ne peut stocker que $(n-1)$ éléments dans la queue avec un tableau de taille n (c'est la version du livre)

5.3 Structure exogène

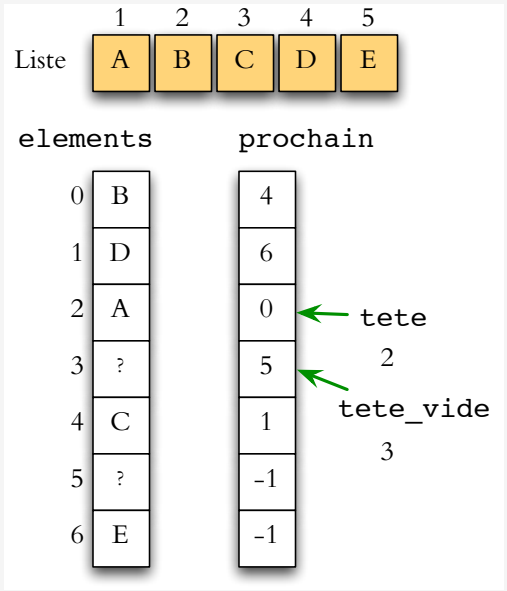
Normalement, l'insertion ou délétion dans le milieu d'un tableau nécessite du décalage d'éléments. Mais on peut aussi créer une structure par deux tableaux pour implanter une liste chaînée : le champs prochain d'un nœud est stocké comme indice dans un tableau elements.

Dans la solution de la liste chaînée (Notes de cours 4), l'information primaire (`valeur`) est stockée avec l'information sur la *structure*, utilisée par l'implantation seulement. Il s'agit d'une structure **endogène** de données. On peut aussi stocker la structure séparamment des éléments. (P.e., en utilisant deux tableaux.) Il s'agirait alors d'une structure **exogène** de données.

Exemple : union-find pour n éléments. Exogène : tableaux `valeur`, `id` et `taille`. Endogène :

```
class Node { private Node id; private int taille;...}
```

Avantage pour endogène : moins d'espace ; avantage pour exogène : même élément peut participer dans plusieurs structures.




```
public class Liste2
{
   private static int MAX_TAILLE = 5555;
   private Object[] elements;
   private int[] prochain;
   private int tete;
   private int tete_vide;
   public Liste2()
   {
      elements = new Object[MAX_TAILLE];
      prochain = new int[MAX_TAILLE];
      tete = -1;
      tete_vide = 0;
      for (int i=0; i<MAX_TAILLE-1; i++)
         prochain[i]=i+1;
      prochain[MAX_TAILLE-1]=-1;
   }
   ...
}
```

Gestion de cases vides. On maintient alors une liste chaînée de cases vides (avec tête `tete_vide`), entrelacée avec la liste chaînée de vrais éléments. En fait, il s'agit d'une *pile* de cases vides : on insère et enlève toujours à la tête.

Ajouter une case vide (remove) :

```
   prochain[case_idx] = tete_vide;
   tete_vide = case_idx;
```

Supprimer une case vide (insert) :

```
   // use elements[tete_vide] pour l'insertion
   tete_vide = prochain[tete_vide];
```