IFT2015 hiver 2010 — Notes de cours

Miklós Csűrös

3 février 2010

4 Listes, piles et files

4.1 Types

Définition 4.1. *Un **type** est un ensemble (possiblement infini) de valeurs et d'opérations sur celles-ci.*

En Java :
- ⋆ types primitifs (`int`, `double`, `boolean`, ...)
- ⋆ types agrégés (tableaux et ceux définis par les classes)

En Java, la valeur d'une variable de type agrégé est une référence. Une **référence** (ou pointeur) est une adresse d'emplacement mémoire contentant de l'information (ou elle est nulle). En Java, les variables de types simples donnent l'information directement.

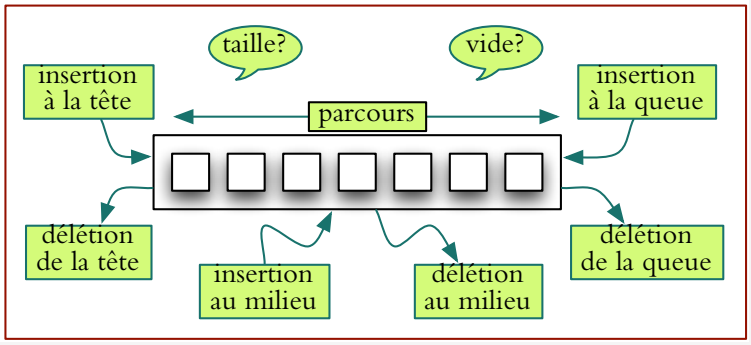
Rappel : variable = abstraction d'un emplacement en mémoire (von Neumann)
nom + adresse (lvalue) + valeur (rvalue) + type + portée

Définition 4.2. *Un **type abstrait** est un type accessible* uniquement *à travers une interface.*

- ⋆ **client** : le programme qui utilise le TA
- ⋆ **implantation** : le programmme qui spécifie le TA
- ⋆ **interface** : contrat entre le client et l'implantation

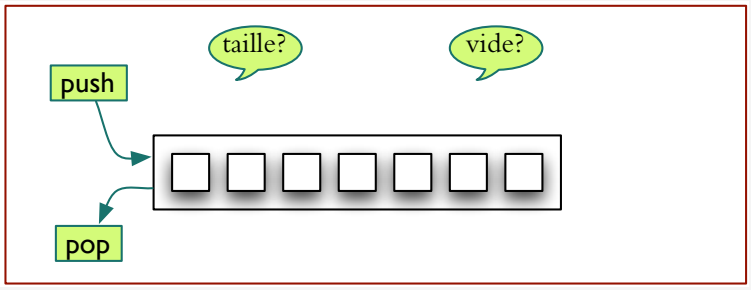
En Java :
- ⋆ interface et implémentation souvent dans le même fichier
- ⋆ interface défini par la signature des méthodes et variables non-privées
- ⋆ «clients» avec droits différents (sous-classe, package)
- ⋆ `interface` n'est pas exactement l'interface de notre définition (ne définit pas le syntaxe des constructeurs)

4.2 Liste

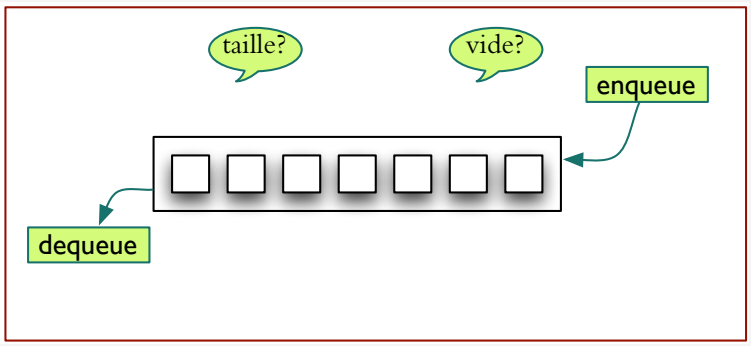


Une **liste** (comme type) est un ensemble d'éléments dans un arrangement séquentiel. Opérations typiques incluent l'insértion et délétion aux deux extremités ou au milieu, ainsi que le parcours, est des quêtes de taille.

Les sous-types abstraits des listes utilisent un sous-ensemble de ces opérations.



Une **pile** permet l'accès à la tête mais pas ailleurs. Les opérations de base sont push («empiler») et pop («dépiler»).



Une **queue** ou file FIFO permet l'insertion à la queue et la délétion à la tête. Les opérations de base sont enqueue («enfiler») et dequeue («défiler»).

4.3 Structure de données

Des structures spécifiques permettent l'organisation de grande quantitiés de données. Deux structures de base pour représenter des listes :

- ⋆ **tableaux** *(arrays)* — accès facile, manipulation inefficace
- ⋆ **listes chaînées** — manipulation efficace, accès difficile

Tableau.

- ⋆ taille fixe, allocation explicite : `int[] T = new int[12];`
- ⋆ acccès rapid au k-ème élément : `int z=2*T[k];`
- ⋆ manipulation difficile : pour insérér ou supprimer un élément il faut décaler les éléments dans la reste du tableau

Liste chaînée. Une liste chaînée est un ensemble d'éléments conservés chacun dans un nœud qui contient aussi un ou deux liens sur le nœud suivant et/ou précédent dans la liste. Chaque élément de la liste est stocké comme un nœud formé par (valeur, prochain) ou (valeur, prochain, précédent) dans le cas d'une liste **doublement chaînée**. Dans une **liste circulaire**, on a queue.prochain = tête et/ou tête.précédent = queue.

4.4 Liste chaînée

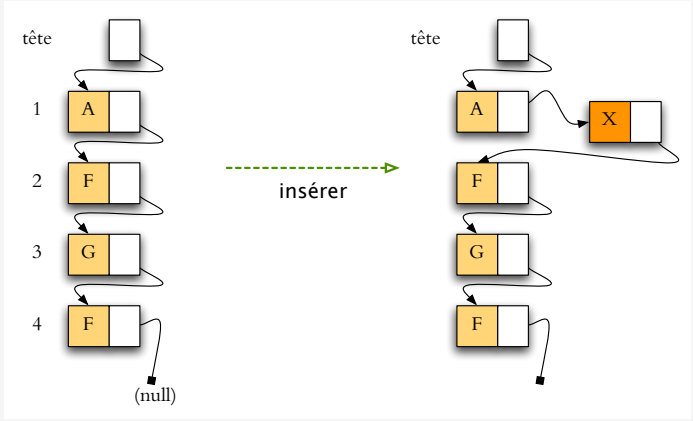
En Java, on peut implanter une liste chaînée à l'aide d'une classe pour représenter les nœuds (`LinkedList.Noeud`). La liste est spécifiée par la tête de la liste (de type `Noeud`).

```
public class LinkedList
{
   public static class Noeud
   {
      private Object valeur;
      private Noeud prochain;
      private Noeud(Object valeur)
      {
         this.valeur=valeur; this.prochain=null;
      }
      public Object getValue(){ return valeur;}
      public Noeud getNext(){ return prochain;}
      public void setNext(Noeud prochain)
      {
         this.prochain = prochain;
      }
   }
   private Noeud tete;
   public Noeud getFirst(){ return tete;}
   ...
}
```

Parcours. Le parcours se fait par une boucle ou par récursion.

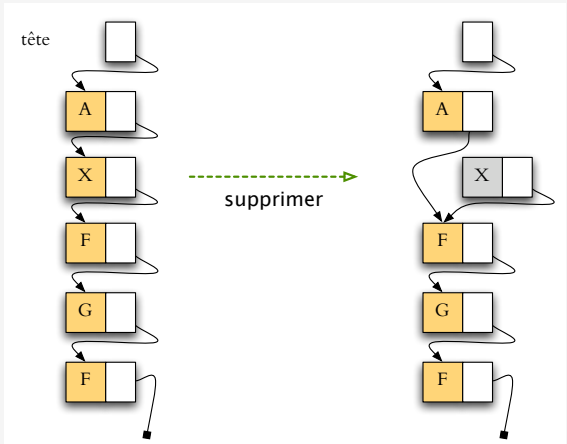
```
   private Noeud Keme(int pos) // itération
   { // trouve l'élément en position pos
       Noeud N = tete;
       while (N != null && pos>0)
       {
           N = N.getNext();
           pos--;
       }
       return N;
   }
```

```
   private Noeud Keme(int pos){ return Keme(tete, pos);}
   private Noeud Keme(Noeud N, int pos) // récurrence
   { // trouve l'élément en position pos après N
       if (pos==0 || N==null) return N;
       else return Keme(N.getNext(), pos-1);
   }
```



Insertion se fait par l'affectation de deux références.

```
public void insertFirst(Object e)
{ // insertion à la tête
  Noeud N = new Node(e);
  N.setNext(tete);
  tete = N;
}
private void insert(Noeud P, Object e)
{ // insertion après P
  Noeud N = new Node(e);
  N.setNext(P.prochain);
  P.setNext(N);
}
```



Suppression se fait par l'affectation d'une seule référence.

```
public void deleteFirst()
{ // délétion de la tête
  if (tete==null)
   throw new NoSuchElementException();
  tete = tete.getNext();
}
private void delete(Noeud P)
{ // délétion après P
  if (P.getNext()==null)
   throw new NoSuchElementException();
  P.setNext(P.getNext().getNext());
}
```

4.5 Sentinelles

Définition 4.3. *Une **sentinelle** est un nœud «factice» sur la liste pour dénoter la tête et/ou la queue.*

«factice» : n'est pas vu dehors de l'implantation, ne contient pas d'élément
Avantage : code plus clair, exécution un peu plus rapide (mais pas en asymptotique)

```
   private Noeud TETE_SENTINELLE = new Noeud(null);
   private tete = TETE_SENTINELLE;
   public Noeud getFirst(){return tete.getNext();}
   public void deleteFirst()
   { // délétion de la tête
       delete(tete);
   }
   public void insertFirst(Object e)
   { // insertion à la tête
      insert(tete, e);
   }
}
```

Désavantage : espace pour un élément de plus
$\rightarrow$ n listes de longueur totale ℓ nécessitent un espace de $O(n+\ell)$ au lieu de $O(\ell)$ ... problème si on a beaucoup de listes courtes