Listes

Types

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

En Java :

- types primitifs (`int`, `double`, `boolean`, ...)

- types agrégés (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

Structure de données

organisation de grande quantitiés de données

deux méthodes simples d'organisation :

tableaux *(arrays)* — accès facile, manipulation inefficace

listes — manipulation efficance, accès difficile

Tableaux

- 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

Listes

l'organisation de la liste permet de manipuler un ensemble dynamique d'éléments dans un arrangement séquentiel

Déf. Une liste chaînée est un ensemble d'éléments conservés chacun dans un nœud qui contient aussi un lien sur un autre nœud.

On veut une liste de taille illimitée. . .

Chaque élément de la liste est stocké comme une paire de (valeur, prochain élément)

Implantation en Java

(1) une classe `List.Element`

(2) classe `List` doit maintenir une variable pour la tête de la liste (de type `Element`)

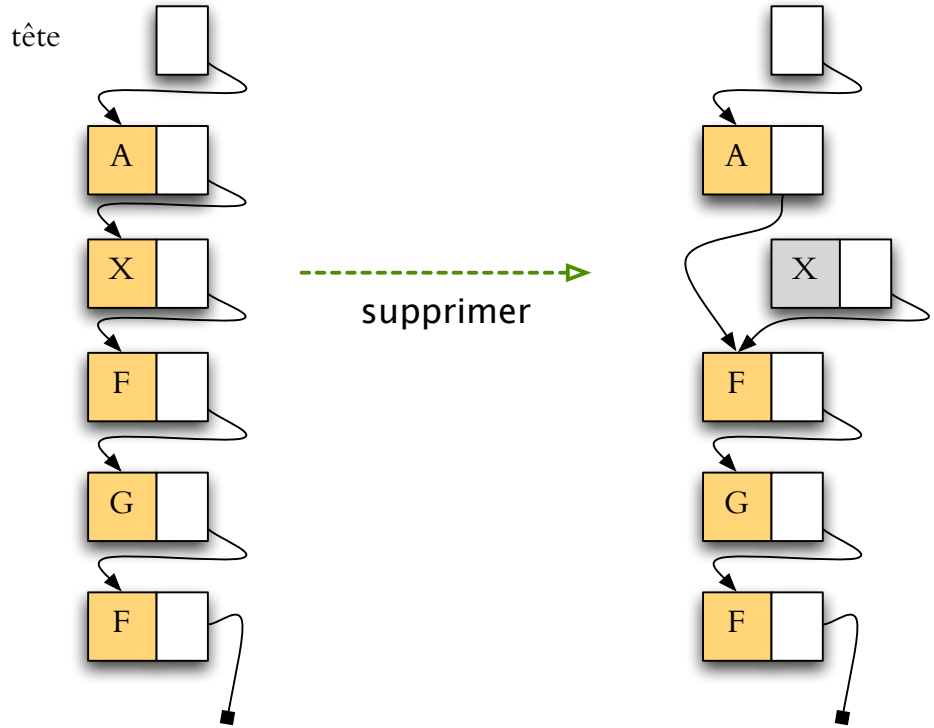
```
public class Liste {
   public class Element {
      private Object valeur;
      private Element prochain;
      private Element(Object valeur) {
         this.valeur=valeur; this.prochain=null;
      }
      public Object getValue(){ return valeur;}
      public Element getNext(){ return prochain;}
      public void setNext(Element prochain){
         this.prochain = prochain;}
   }
   private Element tete;
   ...
}
```

Liste — parcours

```
public Object Keme(int pos){ // trouve l'élément en position pos
    if (pos<=0)
        throw new IndexOutOfBoundsException("pos<=0");
    int k=1;
    Element E = tete;
    while (E != null && k<pos){
        k++;
        E = E.getNext();
    }
    if (E == null)
        throw new IndexOutOfBoundsException("pos trop grand");
    return E.getValue();
}
```

Ici, la fin de la liste est indiquée par `prochain=null` au dernier élément.

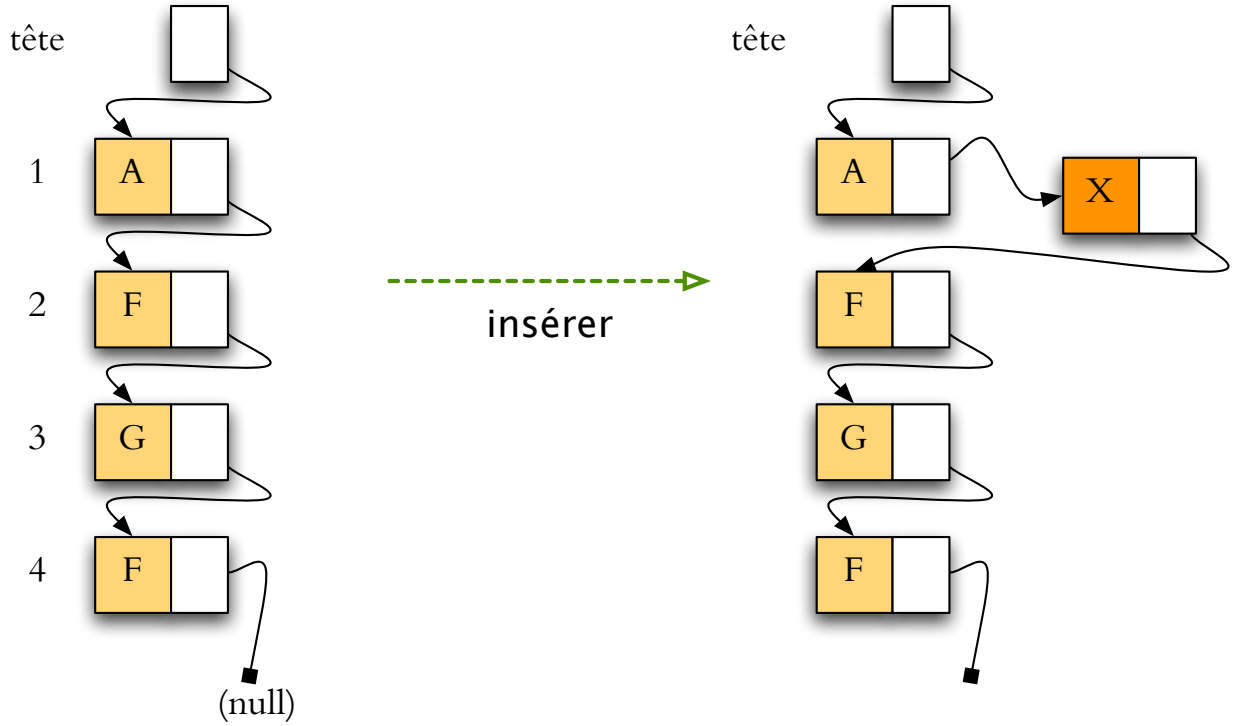
Liste — suppression



suppression de l'élément après `a` : `a.prochain=a.prochain.prochain`

le mémoire occupé par l'objet de `a.prochain` sera recuperé automatiquement en Java

Liste — insérer



insertion de l'élément x après a : `x.prochain=a.prochain; a.prochain=x;`

Attention : traitement spécial pour la suppression de la tête ou l'insertion avant la tête

Sentinelles

sentinelle : 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)
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

Traitement de listes

parcours :

```
for (E = tete; E.prochain != null; E = E.prochain)
   visiter(E);
```

tri par insertion

Efficacité sur une liste de ℓ éléments :

insertion et suppression et de $O(\ell)$ en pire temps

parcours et Keme prennent $O(\ell)$ au pire cas

Variantes

1. liste circulaire : le dernier élément de la liste pointe vers le premier

2. Liste doublement chaînée : les éléments pointent vers tous les deux voisins (champs `prochain` et `precedent`)
avec ou sans sentinelle

Liste doublement chaînée

Avantage : on peut supprimer et insérer un élément dans le milieu de la liste sans traverser tous les éléments pour y arriver (avec une liste simple, il faut connaître l'élément précédent)

suppression et insertion : il faut changer tous les pointeurs affectés

Keme(i) : si on stocke aussi la taille de la liste, on peut rapidement trouver les éléments vers la fin aussi

Efficacité : en asymptotique comme liste simple, mais
- insertion et suppression prennent un peu plus de temps (plus de pointeurs)
- pire temps de Keme est à peu près la moitié de celui de la liste simple

Structure

Dans la solution présentée, l'information primaire (`valeur`) est stockée avec l'information sur la *structure*, utilisée par l'implantation seulement. Il s'agit d'une structure de donnée endogène.

On peut aussi stocker la structure séparamment des éléments. (P.e., en utilisant deux tableaux.) Il s'agirait alors d'une structure de donnée exogène.

Exemple : union-find pour n éléments

exogène : tableaux `valeur`, `id` et `taille`

endogène :

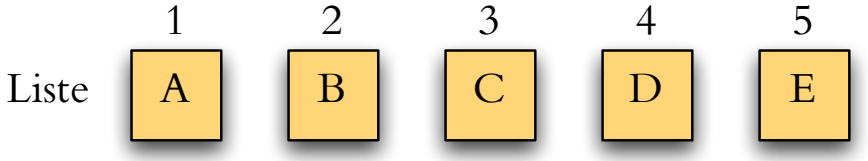
```
class Node { private Node id; private int taille;...}
```

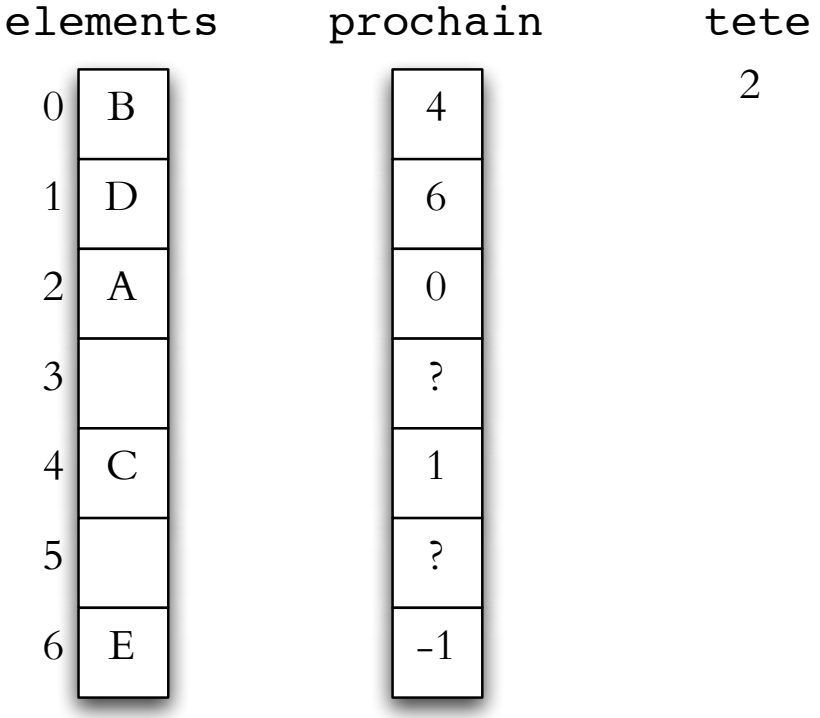
avantage pour endogène : moins d'espace
avantage pour exogène : même élément peut participer dans plusieurs structures

Liste — implantation exogène

```
public class Liste2 {
   private Object[] elements;
   private int[] prochain;
   private int tete;
   private static int MAX_TAILLE = 5555;
   public Liste2(){
      elements = new Object[MAX_TAILLE];
      prochain = new int[MAX_TAILLE];
      tete = -1; // liste vide
   }
   ...
}
```

Liste2 (cont.)


1
2
3
4
5
Liste
A
B
C
D
E


elements
prochain
tete
2
0
B
4
1
D
6
2
A
0
3
?
4
C
1
5
?
6
E
-1

Liste2 — cases vides

Problème : où placer le nouvel élément ?

Solution 1 : objet `VIDE` pour dénoter une case vide $\Rightarrow O(M)$ pour trouver une case libre dans un tableau de taille max. M

Solution 2 : maintenir la liste de cases libres
on peut utiliser `prochain[]` pour ceci avec une variable `tete_vide`

```
public class Liste2 {
   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;
   }
   ...
}
```

Liste2 — cases vides

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];
```