Structures récursives : listes et arbres

Au delà des tableaux

Conception des structures

⋆ tableau (accès par indice mais décalage couteux)

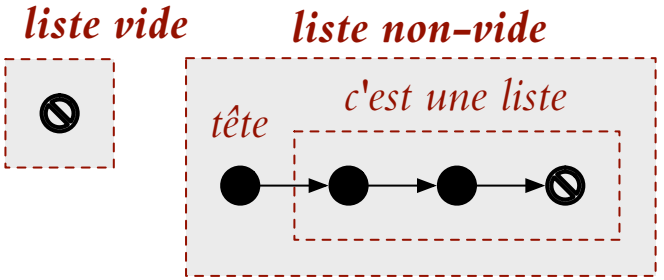
⋆ récursion

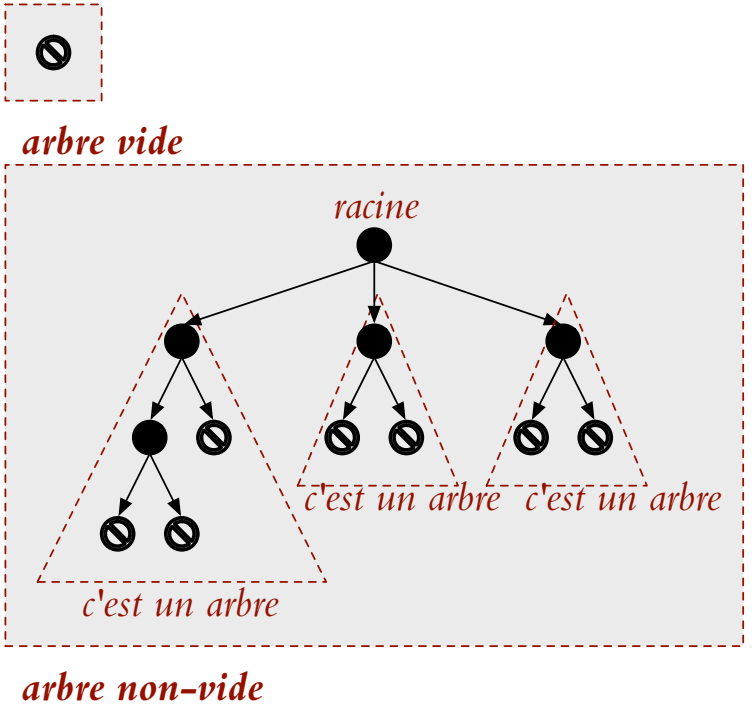
⋆ non-Turing (p.e. quantique)

Liste : organization séquentielle (1 sous-structure dans la récursion)

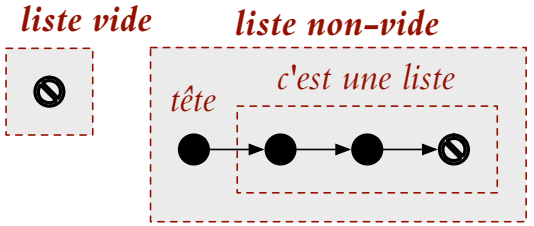
Arbre : organization hiérarchique (≥ 1 sous-structures dans la récursion)

Structures récursives





Liste

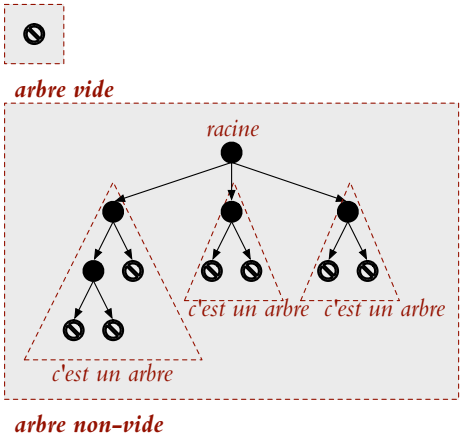


Une liste est soit une référence null (un seul nœud externe), soit une paire formée par un nœud interne (la tête) et d'une autre liste.

Définition. La **liste** est une structure récursive construite par l'application des règles suivantes.

$$\mathsf{liste} \longrightarrow \langle \textit{nœud externe} \rangle \qquad \text{liste vide} = \mathsf{null}$$
$$\mathsf{liste} \longrightarrow \Big(\langle \textit{noeud interne} \rangle, \mathsf{liste} \Big) \qquad (\text{tête}, \text{successeurs})$$

Arbre



Un arbre est soit null (nœud externe), ou il est composé d'un nœud interne (la racine) et un ensemble d'arbres (les enfants).

Définition. L'**arbre enraciné** (ou «arboresence») est une structure récursive construite selon les règles suivantes.

$$\mathsf{arbre} \longrightarrow \langle \textit{nœud externe} \rangle \qquad \text{arbre vide} = \mathsf{null}$$

$$\mathsf{arbre} \longrightarrow \Big(\langle \textit{noeud interne} \rangle, \underbrace{\mathsf{arbre}, \ldots, \mathsf{arbre}}_{d > 0 \text{ fois}} \Big) \qquad (\text{racine}, d \text{ enfants})$$

(Le degré d peut varier chez les nœuds ; $d > 1$ est en général assumé.)

Liste chaînée

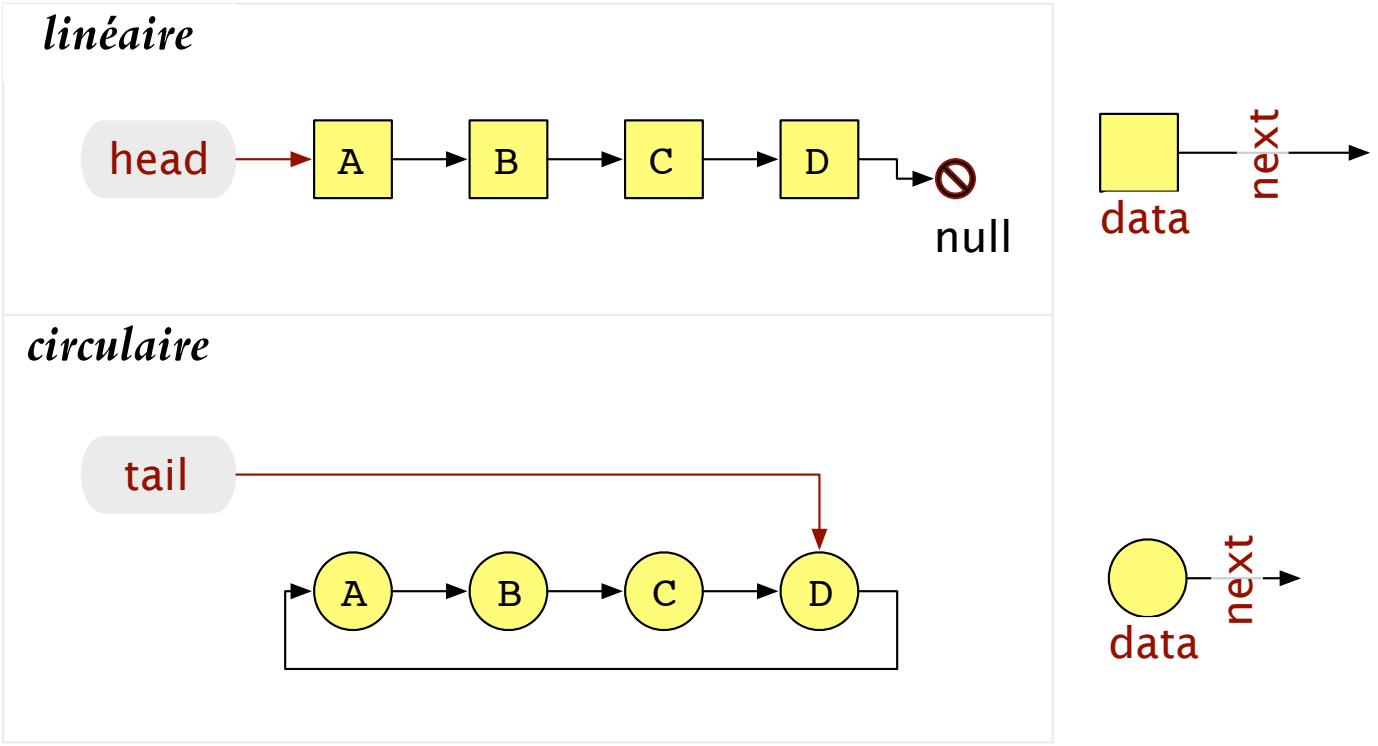
Définition. La structure appellée **liste chaînée** (*linked list*) est une liste d'éléments conservés chacun dans un nœud qui contient aussi un ou deux liens sur le nœud suivant et/ou précédent.

⋆ liste identifiée par son premier nœud, ou la **tête** (*head*)
⋆ le dernier nœud interne s'appelle la **queue** (*tail*) de la liste

Variantes :
doublement chaînée (précédent et suivant)
circulaire (accès à la queue)

Liste chaînée

2 variables par nœud

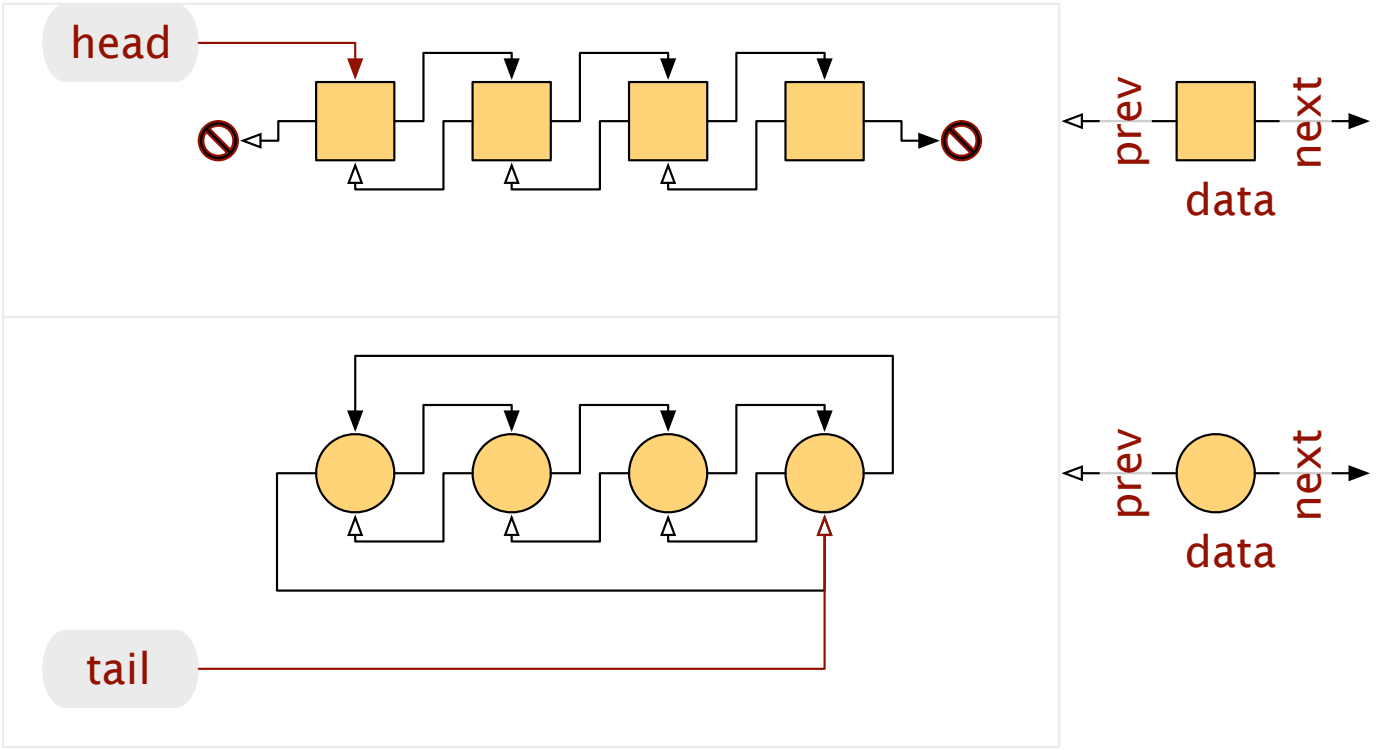


opérations : accès à la tête, parcours en avant, insertion/ suppression après

avec circularisation : accès à la queue

Liste doublement chaînée

3 variables par nœud



opérations : parcours en arrière, insertion/ suppression avant

avec circularisation : accès à la queue

Implémentation Java

avec type paramétrisé

```
/**
 * Noeud de liste avec payload typisé
 * @param <Data> type du payload
 */
class ListNode<Data>
{
   private final Data data;  // inséparables
   private ListNode<Data> next;        // prochain élément
   ListNode(Data data) // nouveau noeud sans successeur
   {

      this.data=data; this.next=null;
   }
   Data getData(){return this.data;}
   ...
}
```

Rappel : on déclare un type parametrisé par `C<A, B, ...>` où `C`

Génériques de Java

```
class C<A, B>
{
    ...
}
```

- ⋆ `C` : classe générique
- ⋆ `A`, `B`, ... : types paramétrisés (arguments dans le code source lors d'instanciation et appel de méthodes)

Génériques de Java

⋆ Les paramètres dans la déclaration de classe appartiennent à une instance et n'existent pas dans un contexte statique.

⋆ Les arguments aux types prametrisés sont vérifiés lors de compilation mais l'information n'est nullement disponible à runtime. Donc,

  ∗ on ne peut pas instancier un objet avec le paramètre-type (**new** `A`() ne marche pas),

  ∗ on on ne peut pas vérifier si un objet en est membre (x **instanceof** `A` ne marche pas), et

  ∗ **getClass**() ne donne aucune information sur les arguments utilisés lors de l'instanciation.

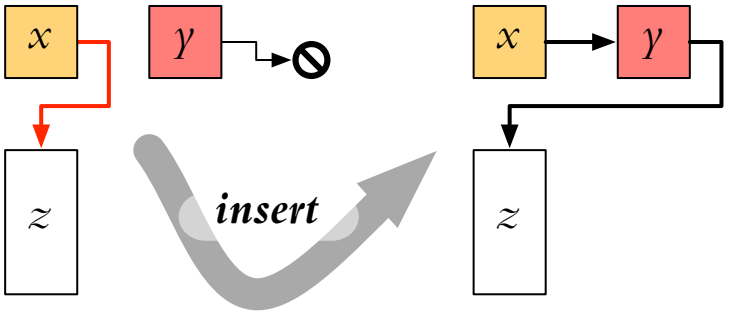
  Pour la même raison, on doit éviter des tableaux de types parametrisés (comme `LinkedList<String>[10]`) car le typage ne peut pas être enforcé aux membres du tableau.

⋆ L'héritage ne se transfère pas à travers des classes génériques : même si E est une sous-classe de F, `LinkedList`$\langle E \rangle$ n'est pas une sous-classe de `LinkedList`$\langle F \rangle$.

Opérations sur la liste chaînée

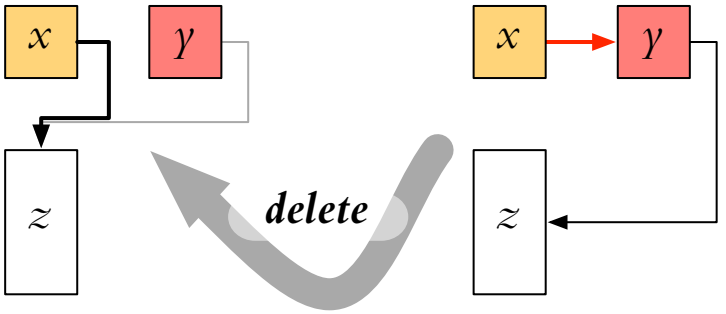
opérations	*liste chaînée*		*liste doublement chaînée*	
	lin.	*circ.*	*lin.*	*circ.*
accéder à la tête TA pile	+			
accéder à la queue TA queue concaténer	-	+	-	+
prochain insérer/supprimer après	+			
précédent insérer/supprimer avant	-		+	
renverser	-			+

Insertion




```
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   void insertNext(ListNode<Data> y){
      ListNode<Data> z = next;
      y.next = z;
      next = y;
   }
```

Suppression




```
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   void deleteNext(){
      ListNode<Data> y = next;
      ListNode<Data> z = y.next;
      next = z;
   }
}
```

Récursions

op sur la liste = combinaison(op sur la tête, op sur les successeurs)

```
\begin{verbatim}
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   /* accès aux noeuds par indice */
   ListNode<Data> getNode(int i)
   {
      if (i=0) return this;
      else return next.getNode(i-1);
   }
   /* recherche */
   boolean contains(Data key)
   {
       return data.equals(key)
         || (next != null && next.contains(key));
   }
```

Récursions 2

```
\begin{verbatim}
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   /* suppression par indice; retourne la nouvelle tête */
   ListNode<Data> deleteAt(int i)
   {
      if (i==0) { return node.next;}
      else {next=next.deleteAt(i-1); return this;}
   }
   /** longueur de la liste
    * @param x tête de la liste (possiblement null)
    * @param L 0 au premier appel
   */
   static int length(ListNode<?> x, int L)
   {
      if (x==null) return L;
      return length(x.next, L+1);
   }
```

Itérations

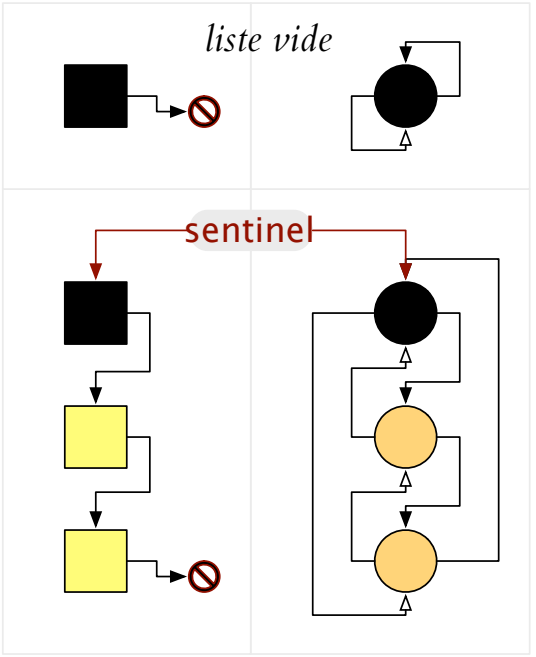
```
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   static <D> ListNode<D> getNode(ListNode<D> head, int i)
   {
      ListNode<D> node = head;
      while (i>0) {node = node.next; --i;}
      return node;
   }
```

Itérations 2

```
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   static boolean contains(ListNode<?> head, Object key)
   {
      ListNode<?> node=head;
      while (node != null && !node.data.equals(key))
         node = node.next;
      return (node!=null);
   }
   static int length(ListNode<?> x)
   {
      int L = 0; for (; x!= null; x=x.next) L++;
      return L;
   }
```

Sentinelle

Définition. Une **sentinelle** est un élément factice dans une structure de données.



Deux implémentations

Spécification de l'API Java définit l'interface pour `java.util.LinkedList`, supportée par une liste doublement chaînée :

```
package java.util;
public class LinkedList<T> extends AbstractSequentialList<T>
   implements List<T>, Deque<T>,
      Cloneable, java.io.Serializable;
```

openjdk : LinkedList.java

classe imbriquée pour nœud, liste circulaire avec sentinelle

GNU classpath : LinkedList.java

liste linéaire avec head+tail