2 Listes

2.1 Types

Définition 2.1. *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.

Définition 2.2. *Un **type abstrait** (TA) est un type accessible* uniquement *à travers une interface.*

Exemple. Le type de *dictionnaire* ou *table de symboles* représente un ensemble d'associations (clé, info) avec clés uniques. L'interface (minimale) contient l'opération essentielle search(k) qui retourne l'info associée avec la clé k, et l'opération d'ajouter un nouveau paire add(clé, info).

Notions.
- ⋆ **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 définie 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 (en Java, elle n'inclut pas le syntaxe des constructeurs)

2.2 Liste chaînée

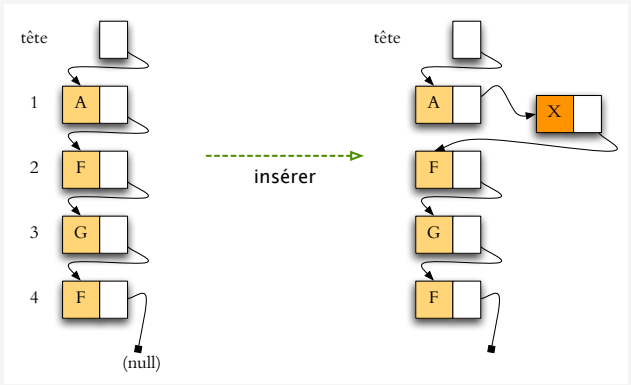
Des structures spécifiques permettent l'organisation de grande quantités de données.

La structure appellée **liste chaînée** (*linked list*) 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 le paire (info, next), ou, dans le cas d'une liste **doublement chaînée**, par le triple (info, next, previous). Dans une **liste circulaire**, on a tail.next = head et/ou head.previous = tail. W(fr)

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

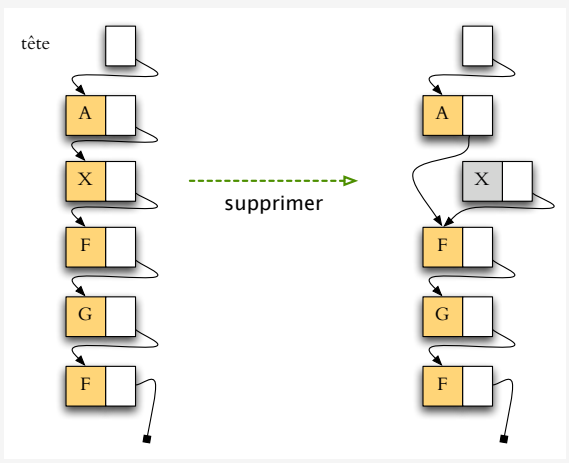
```
public class LinkedList
{
   private static class Node
   {
      private Object info;
      private Node next;        // prochain élément
      private Node(Object info) // création d'un nouveau noeud sans successeur
      {
         this.info=info; this.next=null;
      }
   }
   private Node head=null;   // tête de la liste; liste vide au début
   public boolean isEmpty(){ return head==null;} // si liste vide
   ...
}
```

2.3 Insertion et suppression



Insertion se fait par l'affectation de deux références.

```
public void insertFirst(Object e)
{ // insertion à la tête
  Node N = new Node(e);
  N.next=head;
  head = N;
}
private void insertAfter(Node P, Object e)
{ // insertion après P
  Node N = new Node(e);
  N.next = P.next;
  P.next = N;
}
```



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

```
public void deleteFirst()
{ // suppression de la tête
  if (head==null)
   throw new NoSuchElementException();
  head = head.next;
}
private void deleteAfter(Node P)
{ // suppression après P
  if (P.next==null)
   throw new NoSuchElementException();
  P.next=P.next.next;
}
```

2.4 Parcours

Le parcours se fait par une boucle ou par récursion.

```
private Node Kth(int pos) // itération
{ // trouve l'élément en position pos
    Node N = head;
    while (N != null && pos>0)
    {
        N = N.next;
        pos--;
    }
    return N;
}
```

```
private Node Kth(int pos){ return Kth(head, pos);}
private Node Kth(Node N, int pos) // récurrence
{ // trouve l'élément en position pos après noeud N
    if (pos==0 || N==null) return N;
    else return Kth(N.next, pos-1);
}
```

Recherche séquentielle. On peut utiliser une liste chaînée pour implanter beaucoup de types différents. Par exemple, pour implanter le TA dictionnaire, on peut utiliser des nœuds avec champs (key, info, next).

W(en)

	SEARCH(x) *// recherche d'un élément avec clé x sur une liste non-triée*
S1	$N \leftarrow$ head *// (parcours doit commencer à la tête)*
S2	**while** $N \neq$ null *// (fin de la liste dénotée par* null*)*
S3	**if** N.key $= x$ **then return** N.info */ (recherche fructueuse)*
S4	$N \leftarrow N$.next
S5	**return** null *// (recherche infructueuse)*

Théorème 2.1. *Pour un dictionnaire implanté par une liste chaînée, la recherche infructueuse prend un temps linéaire (dans le nombre d'éléments).*

Curseurs. Pour pouvoir manipuler la liste lors du parcours, il est nécessaire de maintenir une référence au nœud précédent. Le code suivant maintient les entrées dans le dictionnaire selon l'ordre de clés.

```
   ADD(k, x)        // ajout d'un paire (k, x) dans une liste triée selon les clés
I1 N ← head                                  // (parcours doit commencer à la tête)
I2 P ← null                                  (P stocke le nœud précédent)
I3 while N ≠ null et N.key < k do            // (parcours de la liste)
I4     P ← N; N ← N.next
I5 if P = null then insertFirst(⟨k, x⟩)
I6 else insertAfter(P, ⟨k, x⟩)
```

2.5 Sentinelles

On peut utiliser une sentinelle pour dénoter la tête et/ou la queue. W(en)

Définition 2.3. *Une **sentinelle** est un élément «factice» dans une structure de données.*

```
   private Node HEAD_SENTINEL = new Node(null); // on ne stocke aucune info ici
   private head = HEAD_SENTINEL; // ne changera jamais
   public void deleteFirst()
   { // suppression à la tête
       deleteAfter(head);
   }
   public void insertFirst(Object e)
   { // insertion à la tête
      insertAfter(head, e);
   }
   public boolean isEmpty(){ return head.next==null;} // si liste vide
}
```

Avantage : code plus clair, exécution un peu plus rapide (mais pas en asymptotique)
Désavantage : un nœud de plus → n listes de longueur totale ℓ nécessitent $n + \ell$ nœuds au lieu de ℓ