Cours 2 – informatique théorique : structures de données dynamiques

Structures de données statiques

- Les structures de données statiques (de type tableau) occupent une grande partie de la mémoire (du segment de données, c.à.d. de la pile)

- Si la quantité d'éléments n'est pas connue avec précision, il devient nécessaire de sur-dimensionner le tableau, puisqu'il n'est pas possible de réajuster sa taille en cours d'exécution

- Si on veut supprimer ou ajouter un élément, il faut décaler tout le reste

Structures de données statiques : exemple

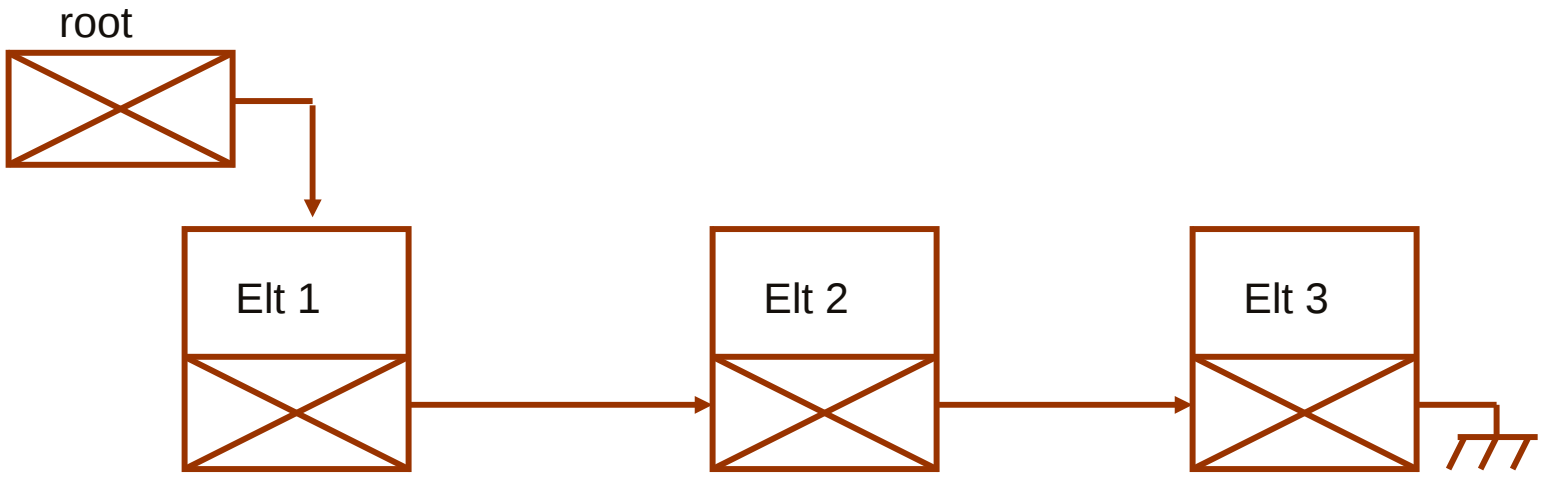
- Utilisation d'un tableau

```
//Déclarer, allouer et remplir un tableau
int[] arr = new int[5];
arr[0] = 34;
arr[1] = 42;
arr[2] = 17;
arr[3] = 163;

//Insertion au milieu (indice 2) : décalage nécessaire
for (int i = 3; i >= 2; --i)
  arr[i+1] = arr[i];

arr[2] = 78;

//Le tableau est plein, plus aucune insertion est possible
```

Les listes chainées

Une liste est une suite d'éléments telle que :

- le premier élément est connu par son adresse (pointeur/référence de tête),
- chaque élément connaît son suivant.



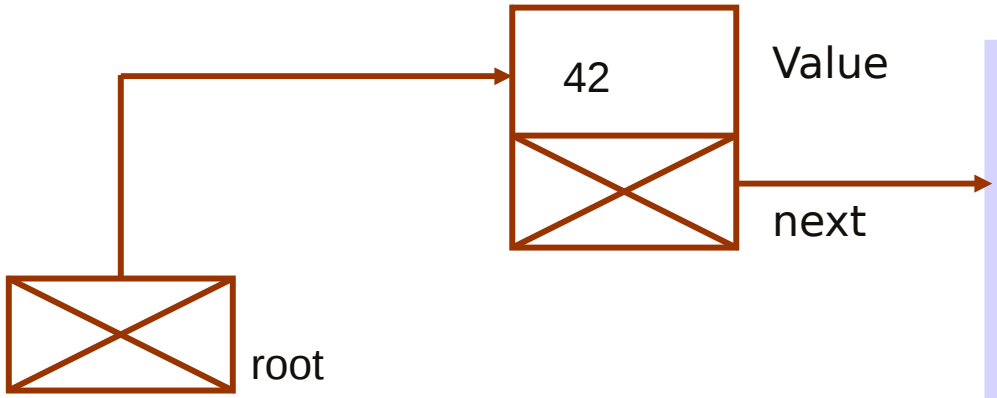
Implémentation (manuelle) d'une liste

- Remarque : ces structures existent « clé en main » en java.

```
//Création d'une liste chainée
LinkedList<Integer> LL = new LinkedList<Integer>();
//Ajouter une valeur
LL.add(34)        //à la fin
//chercher une valeur
if (LL.contains(34))
   System.out.println("Yes.");
```

- Ici nous implémenterons une version « manuelle » pour des raisons pédagogiques.
- Création d'une classe « Llist »

Listes : Déclaration




```
public class LList {

   //Une liste est caractérisée
   //par une référence vers
   //le premier élément
   private LElement root;

   (...)
}
```

```
public class LElement {
   //La valeur a stocker
   private int value;

   //Référence vers le
   //prochain élément
   private LElement next;

   //Constructeur
   public LElement(int v) {
      value= v;
      next= null;
   }
   public void setNext(LElement nxt){
         next= nxt;
   }
   public LElement getNext() {
         return next;
   }
```

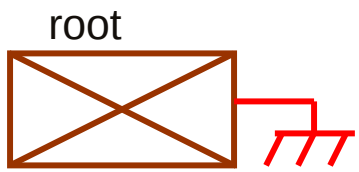
Listes: Algorithmes essentiels

- Principales fonctionnalités
  - Création, initialisation
  - Parcours séquentiel
  - Insertion d’un élément
  - Suppression d’un élément

Algorithmes (1) : initialisation d'une liste vide

- Initialisation d'une liste vide

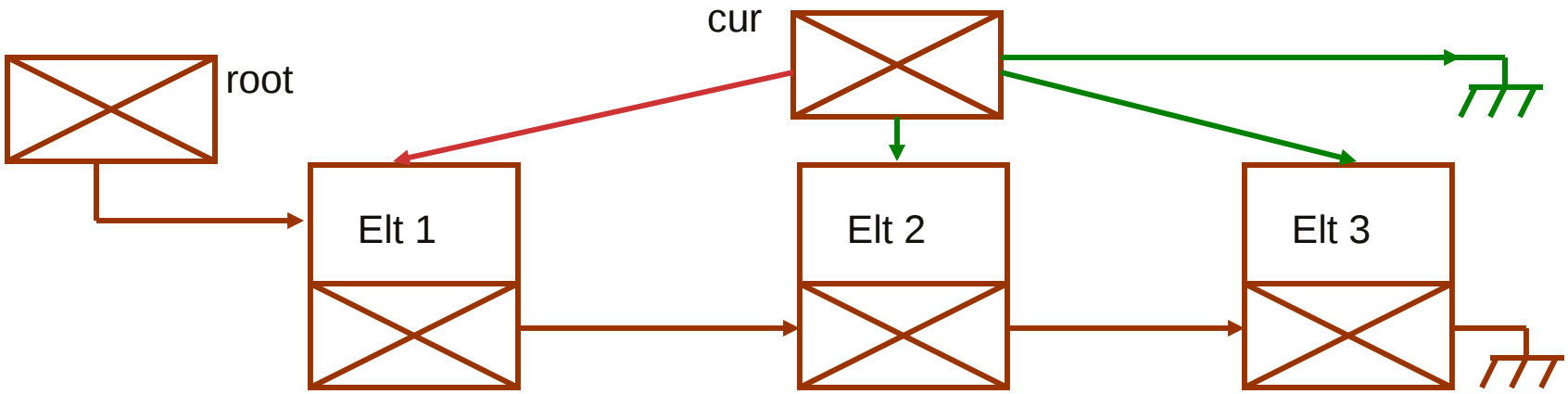
```
public LList() {
   root = null;
}
```



Algorithmes (2) : parcours séquentiel

- Traiter tous les éléments d'une liste (ici : affichage)

```
public void display (){
   //Initialiser une référence vers le début
   LElement cur= root;

   //La référence va parcourir la liste
   while (cur != null){
      System.out.println (cur.getValue());

          //faire avancer la référence
      cur= cur.getNext();
   }
}
```

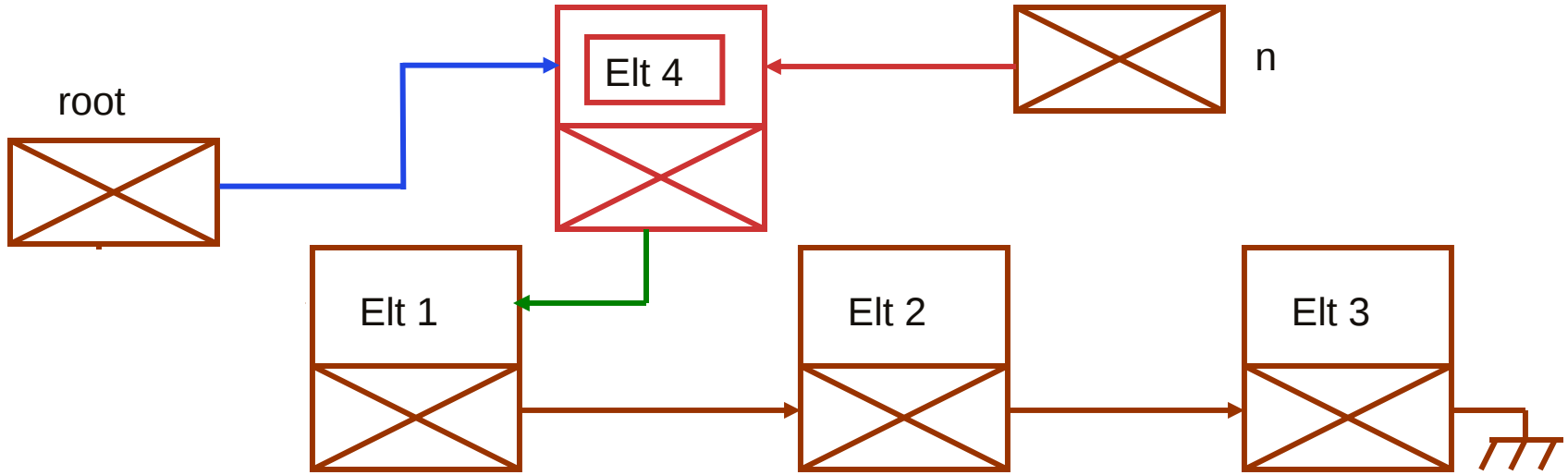


Algorithmes (3) : parcours, cas de la liste vide

```
public void display (){
   //Initialiser une référence vers le début
   LElement cur = root;

   //Vérifier si la liste est vide
   if (root == null)
      System.out.println ("The list is empty.");

   else
         //La référence va parcourir la liste
      while (cur != null){
         System.out.println (cur.getValue());

            //faire avancer la référence
         cur = cur.getNext();
      }
}
```

Algorithmes (4) : insertion d'un élément en tête

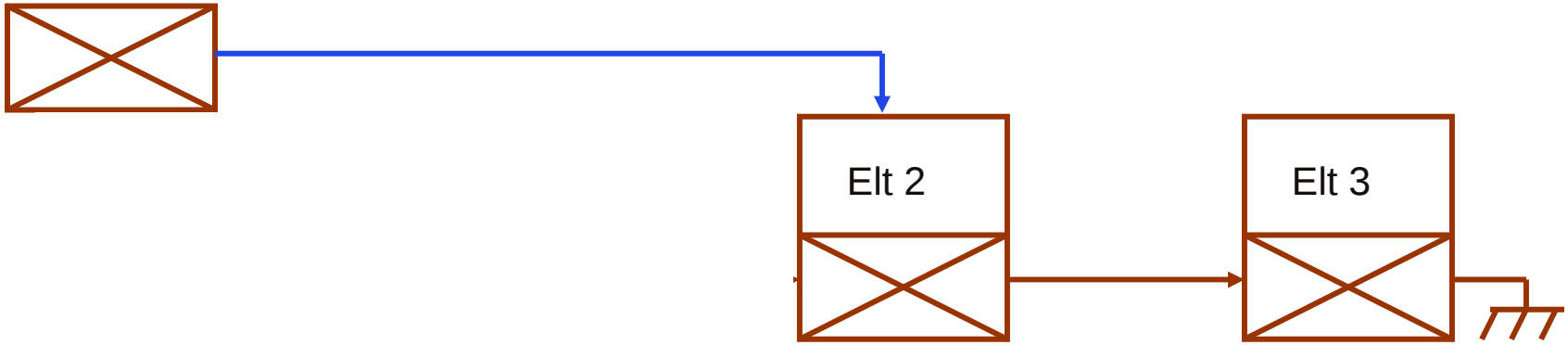
```
public void insertHead (int newvalue){
        //Créer un nouvel élément
    LElement n = new LElement(newvalue);

    //son successeur est l'ancien premier élément
    n.setNext(root);

    //le nouvel premier élément est le nouvel élément
    root= n;
}
```



Algorithmes (6) : suppression en tête, avec ou sans renvoi

```
public void deleteHead (){
    root= root.next;
  }
```

```
public int deleteAndReturnHead (){
    //Sauvegarder la premi ère valeur de la liste
    int value = root.value;

    //Le nouveaux premi er élém ent est l'ancien deuxièm e
    root= root.next;

    //Renvoyer la sauvegarde
    return value;
  }
```



Recherche d'un élément

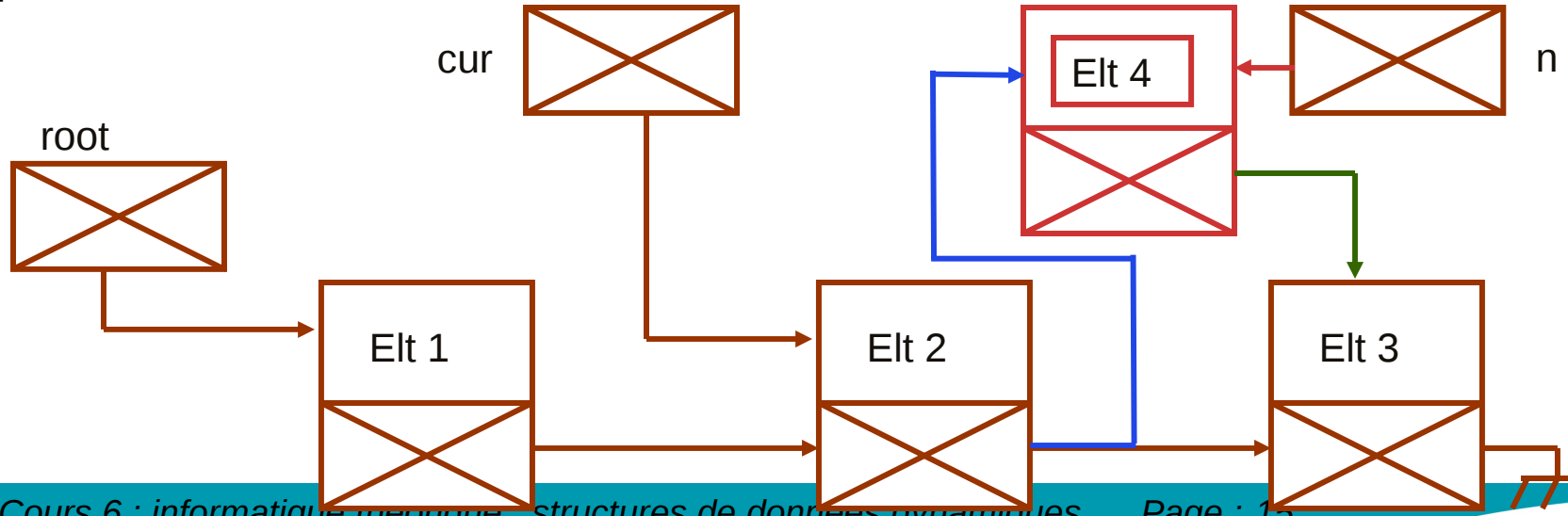
```
//Renvoyer true si la valeur existe dans liste,
//false sinon
public boolean find(int value){
        boolean trouve = false;
        //Initialiser une référence vers le début
    LElement cur= root;

    //La référence va parcourir la liste
    while (cur != null){

      //s'agit-il de la valeur cherchée?
      if (cur.getValue()==value)
        trouve = true;
      cur= cur.getNext();
    }


    return trouve;
  }
```

Insertion d’un élément au milieu

- Objectifs : insérer un nouvel élément après un élément existant de la liste (à chercher)

- Trois étapes :
  - Recherche de l’élément
  - Création d’un nouvel élément
  - Chainage avec l’élément trouvé et son suivant

```
public void insertValueAfterValue (int findvalue, int newvalue){
   LElement cur = root;
   boolean trouve = false;
   //Parcourir la liste et chercher la valeur
   while (!trouve & cur != null){
         //Trouvé
          if (cur.getValue()==findvalue){
            //Créer un nouvel élément
          LElement n = new LElement(newvalue);
            //Son prochain est l'ancien prochain de l'élément cherché
            n.setNext(cur.getNext());
            //Le nouveau prochain de l'élément cherché est le nouvel élément
          cur.setNext(n);
          trouve = true;
          }
          cur = cur.getNext();
      }
   }
```

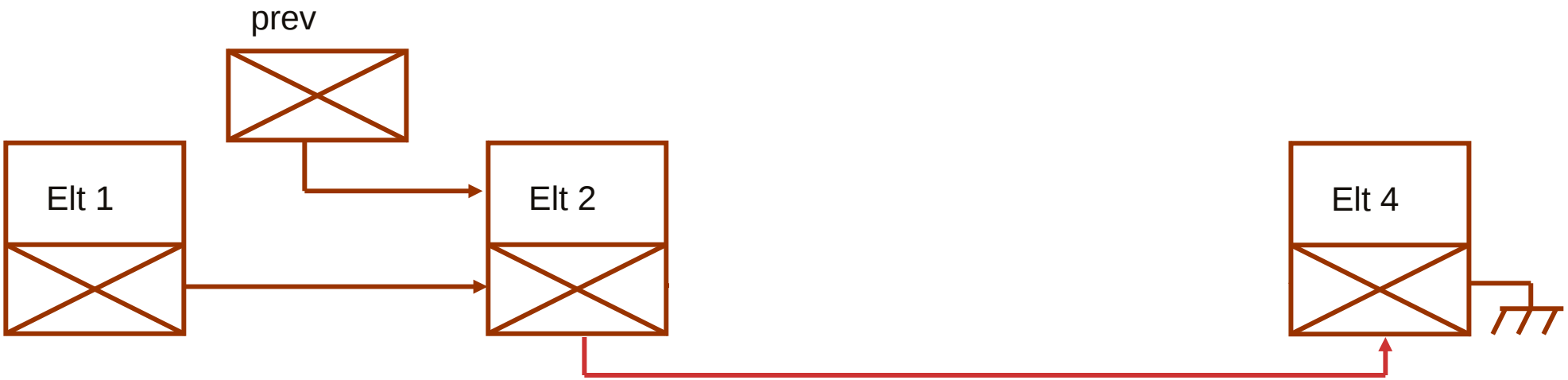


Algorithmes (7) : suppression au milieu

- Première étape : chercher l'élément à supprimer, tout en gardant un pointeur vers l'élément précédent

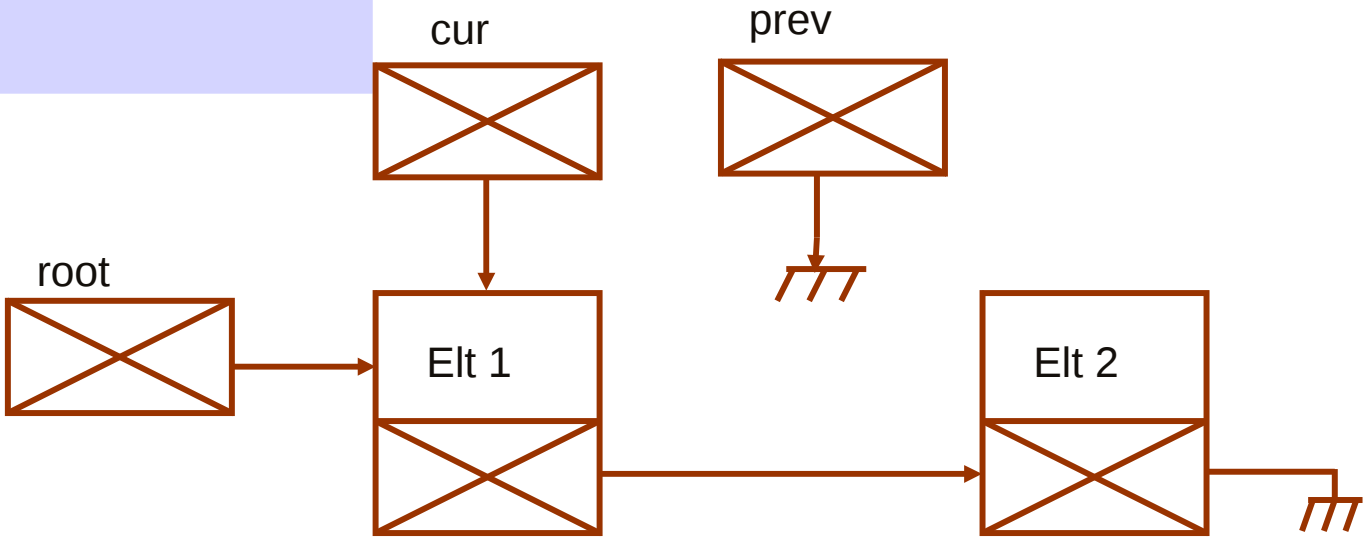
```
public void deleteValue (int value){
        LElement cur = root;
        LElement prev = null;

        //Parcourir la liste et chercher la valeur
        while (cur != null){

          //Trouvé!
          if (cur.getValue()==value){
            //Code de suppression ici
            (...)
          }

                  //Faire avancer les références vers
                  //l'élément actuel et le précédent
          prev = cur;
          cur = cur.getNext();
        }
}
```

```
public void deleteValue (int value){
        LElement cur = root;
        LElement prev = null;
        while (cur != null){

            //Trouvé!
            if (cur.getValue()==value){
                    //Faire sauter l'élément dans la chaine
              prev.setNext(cur.getNext());
                    return;
                }
                    prev = cur;
            cur = cur.getNext();
        }
}
```



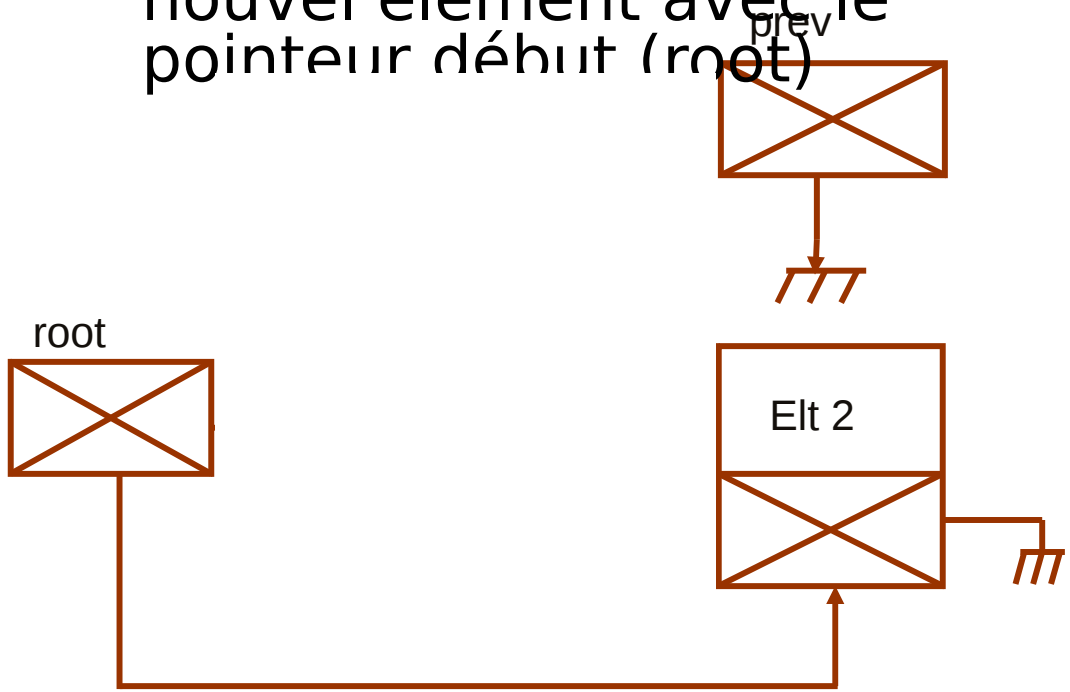
```
public void deleteValue (int value){
   LElement cur = root;
   LElement prev = null;

   while (cur != null){

      //Trouvé!
      if (cur.getValue()==value){
            //Faire sauter
        //l'élément dans la chaine
            prev.setNext(cur.getNext());
          }
      prev = cur;
      cur = cur.getNext();
   }
}
```

- Si l'élément à supprimer est le premier élément de la chaine, le programme plante :
  - L'élément précédent n'existe pas (prev==null)
  - Prev.next est illégal!



```
public void deleteValue (int value){
   LElement cur= root;
   LElement prev= null;

   while (cur != null){
         //Trouvé
      if (cur.value==value){
            //le précédent n'existe pas :
            //cas spécial de la
            //suppression du
            //premier élément
            if (prev==null)
                  //chainer avec root
             root= cur.getNext();
            else {
                  //chainer avec le
                  //précédent
                  prev.setNext
(cur.getNext());
            }
        return;
      }
      prev = cur;
      cur= cur.getNext();
   }
}
```

- Solution: gestion du cas spécial
- Si le précédent existe, on chaine le nouvel élément avec l'élément précédent
- S'il n'existe pas, on chaine le nouvel élément avec le pointeur début (root)



Les listes linéaires : trois types particuliers

Les piles (en gestion LIFO)

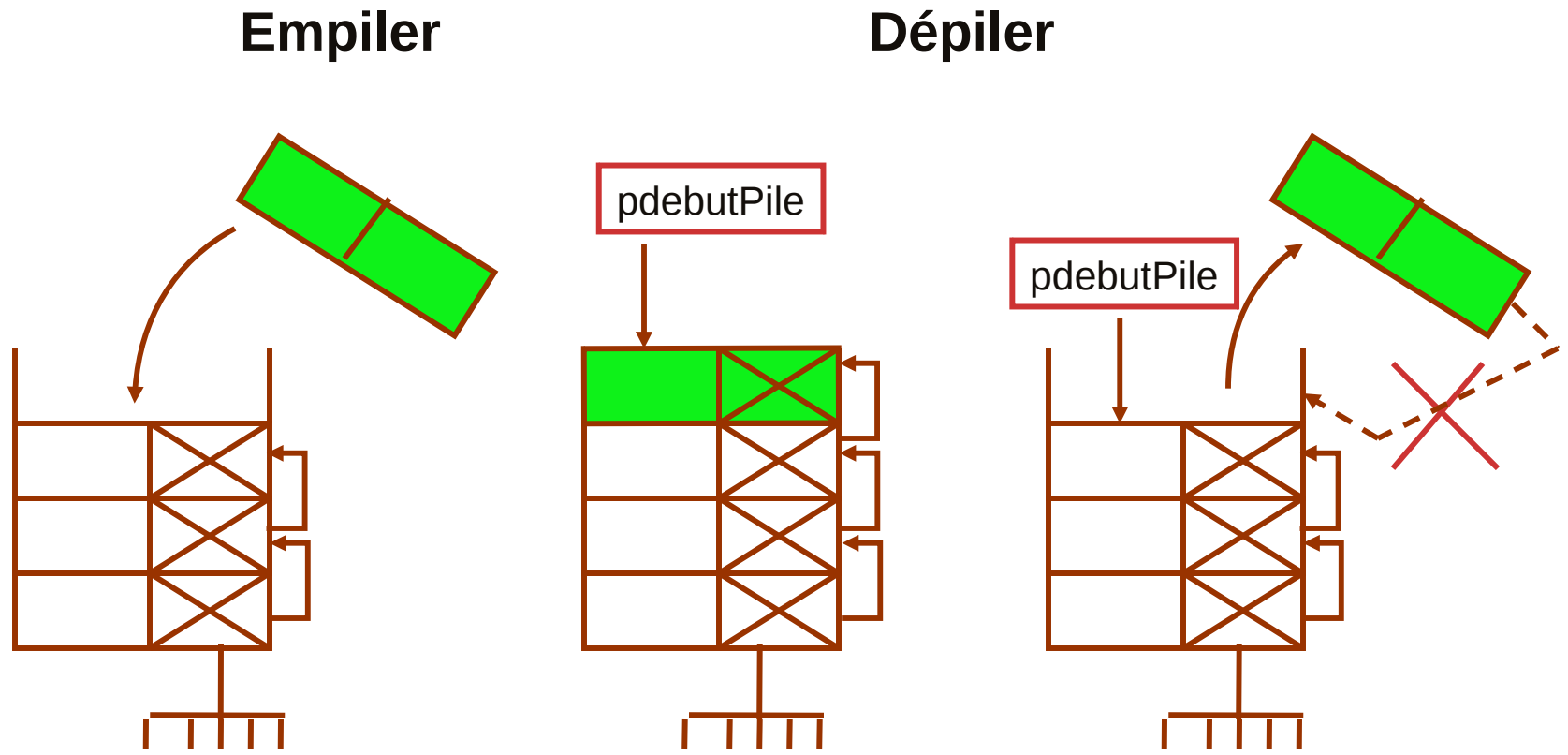
Les files d'attente (en gestion FIFO)

Les listes à double chaînage

La pile

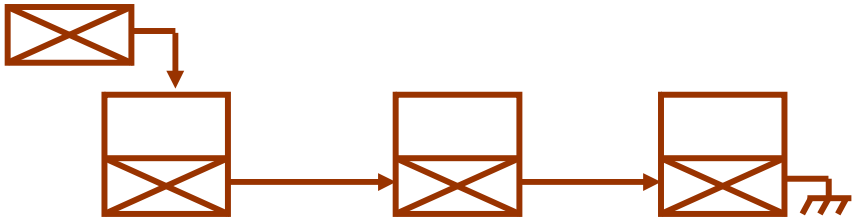
Stratégie LIFO (« *last in – first out* »)

Empilage des assiettes



Réalisation d’une pile

- Une pile (LIFO) est une structure de données logique
- Elle peut être réalisée avec des structures de données physiques différentes :
  - Avec un tableau (mémoire statique)
  - Avec une liste chainée (mémoire dynamique)
  - …



Réalisation d'une pile avec une liste

Les 3 fonctionnalités fondamentales correspondant à une pile:

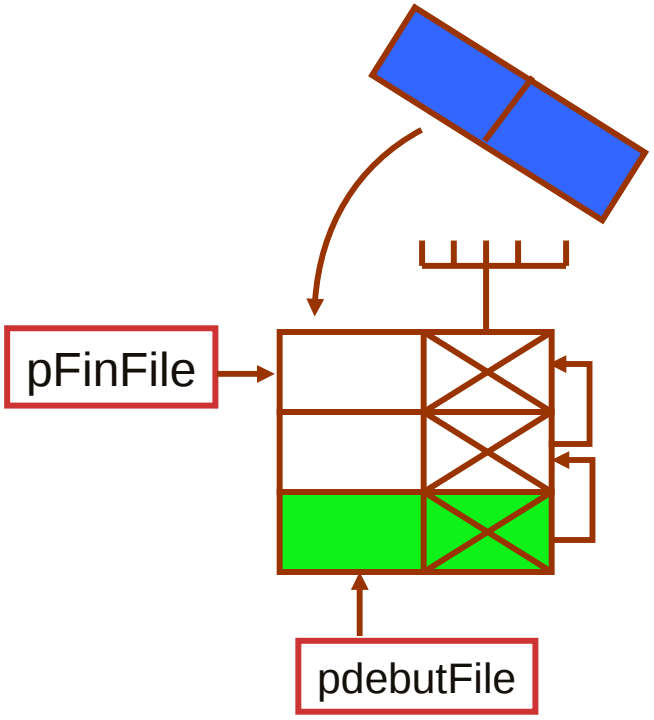
- Initialiser = Initialisation d'une liste
- Empiler = Insertion en tête dans une liste
- Dépiler = Suppression en tête de la liste

```
public class LList{
   private LElement root;

   public LList(){
      root= null;
   }

   //Empiler
   public void push (int value){
      insertHead (value);
   }

   //Dépiler
   public int pop (){
      return deleteAndReturnHead();
   }
   (...)
```

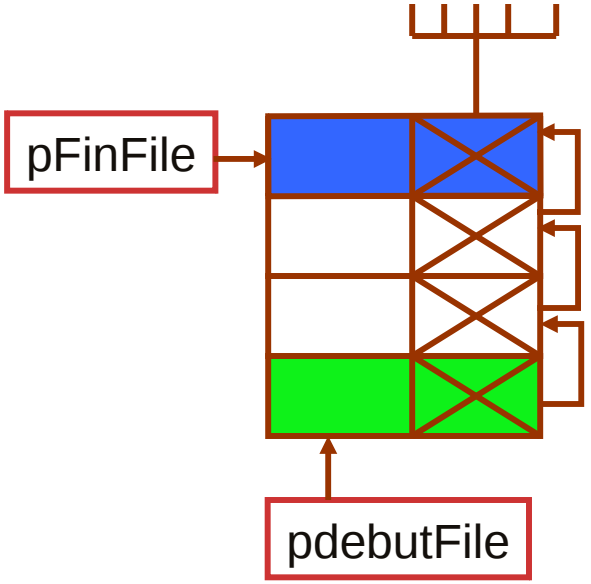
La file d'attente

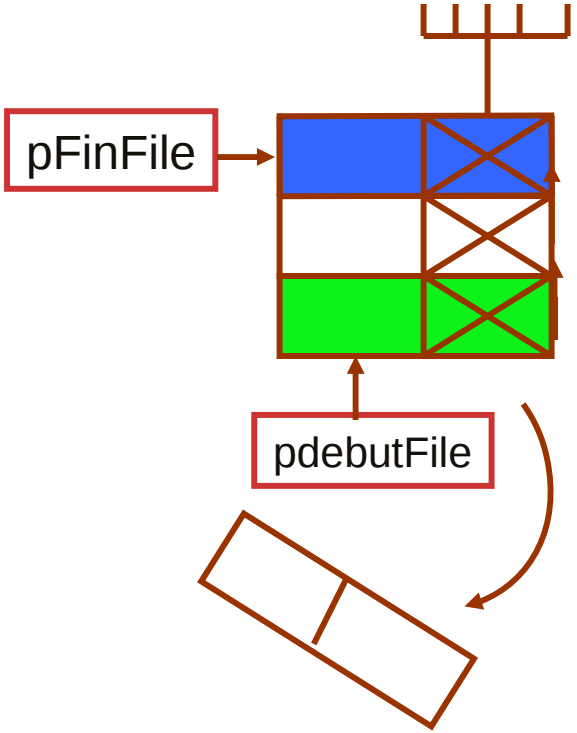
- Stratégie FIFO (« *first in – first out* ») :
- *Queue à la caisse*

Remplissage



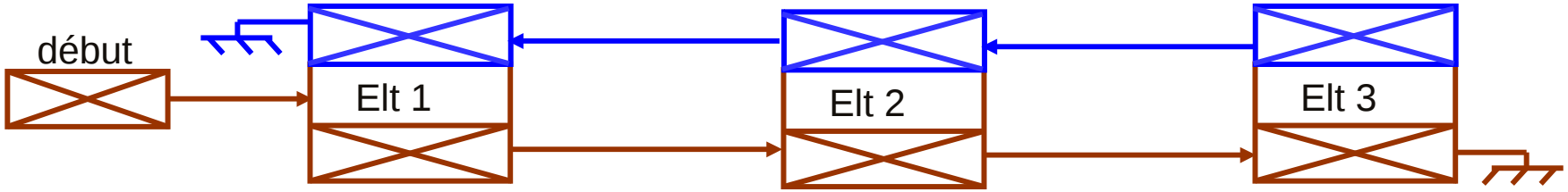
Vidage





Les Listes à double chainage

- Chaque élément comprend un pointeur supplémentaire sur l'élément précédent
- Peu de différences avec les listes à simple chaînage.
- Il est possible d'insérer ou de supprimer un élément **situé avant** celui sur lequel est positionné le pointeur de recherche



Complexité de calcul : notation grand O

- Mesurer la complexité d'un algorithme en fonction de la taille des données d'entrée (comparaison asymptotique)

- On dit que la complexité d'un algorithme est de O(g(N)), parlé « Ohh de g de N », où
  - N est le nombre de données à l'entrée
  - g(.) est une fonction

quand il existe des constantes T et C telles que

$$\forall N > T \quad \text{complexite} \leq C|g(N)|$$

intuitivement cela signifie que la durée d'éxecution ne croît pas plus vite que g().

Listes chainées : complexité des algorithmes

	List chainée	List à double chainage
Insertion en tête :	O(1)	O(1)
Insertion à la fin :	O(N)	O(1)
Suppression en tête :	O(1)	O(1)
Suppression du dernier élément :	O(N)	O(1)
Parcours de la liste :	O(N)	O(N)
Recherche d'un élément :	O(N)	O(N)
Parcours en sens inverse :	O(N2)	O(N)
Parcours en sens inverse (récursif) :	O(N)	

(Mais : besoin excessif de mémoire)

Remarque : chaque opération individuelle est légèrement plus complexe pour la liste chainée (une référence supplémentaire à gérer).

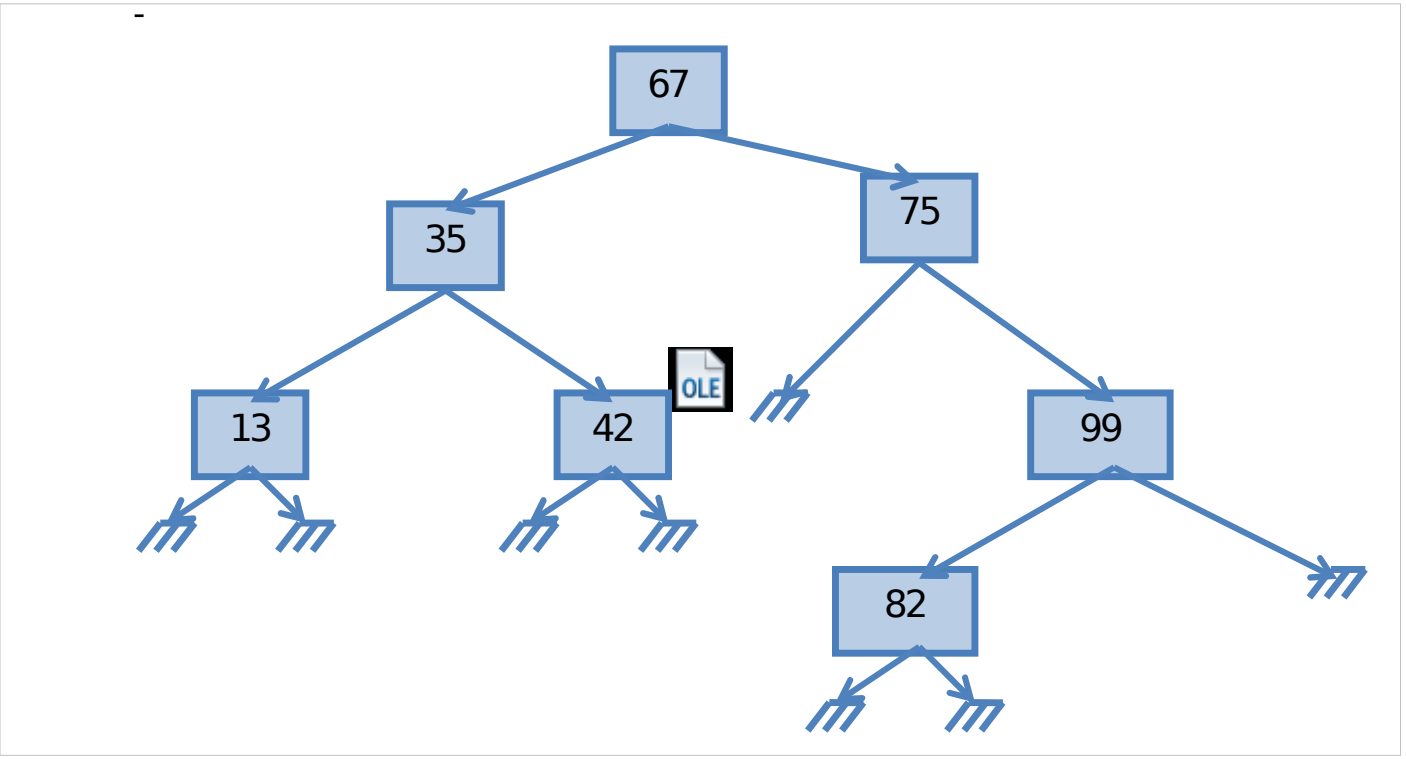
Autre structure : motivation

- La recherche d'un élément dans une liste est de complexité O(N)
- En moyenne, la recherche nécessite N/2 étapes
- Peut-on faire mieux?
- Motivation : conception d'une structure de recherche permettant une complexité « sous-linéaire »

- -> Arbres!

Arbres binaires

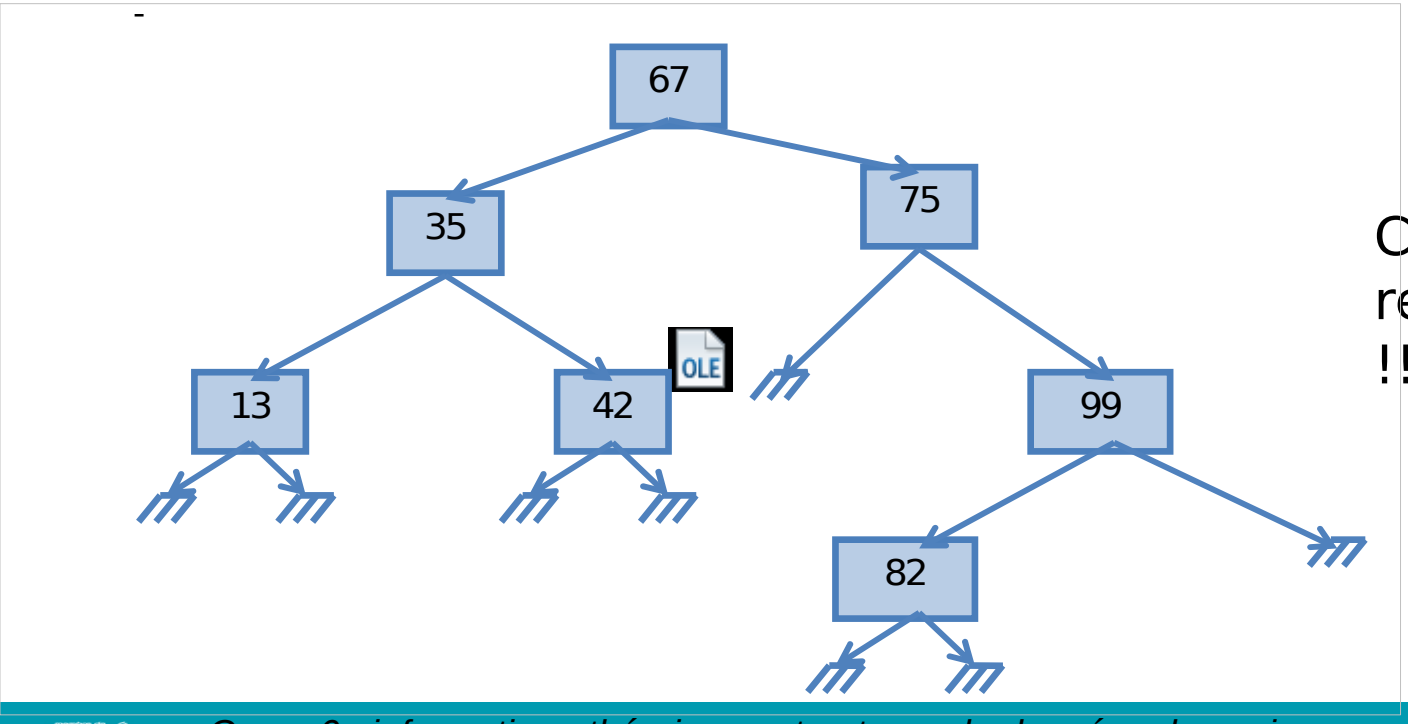
- **Pour chaque sommet :**
  - **chacune des valeurs du sous-arbre gauche est plus petite que la valeur de l'élément actuel;**
  - **chacune des valeurs du sous-arbre droite est plus grande que la valeur de l'élément actuel.**



Recherche dans un arbre binaire

Parcourir l'arbre de la racine jusqu'à la feuille. À chaque itération :

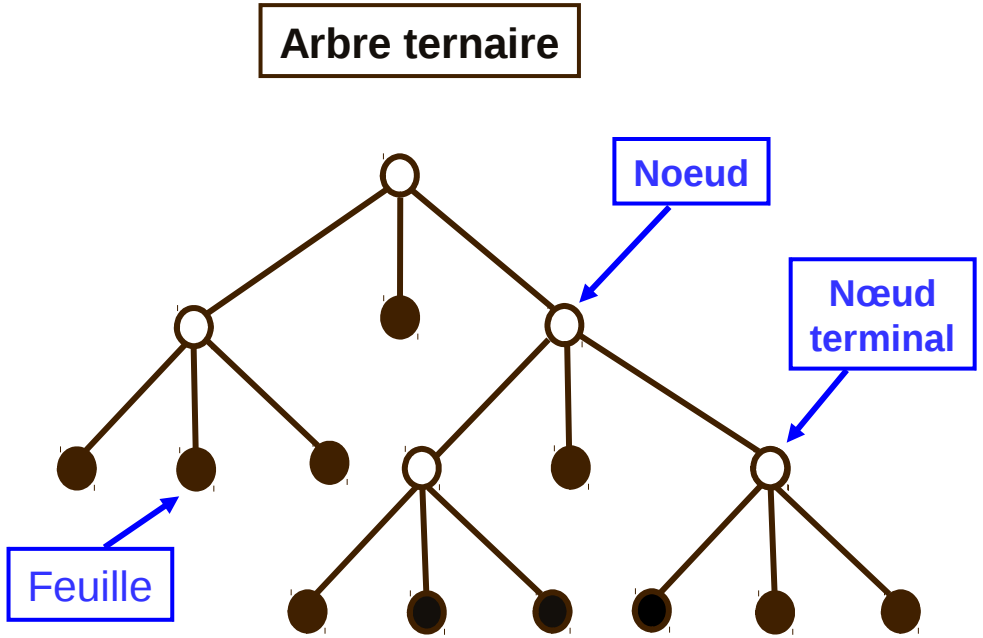
- Vérification si la valeur de l'élément examiné est égale à la valeur cherchée -> réponse positive
- Sinon, continuer avec une des deux références gauche ou droite, selon la comparaison
- tant que la référence actuelle est non-null. Si la référence est null, la recherche est interrompue avec une réponse négative.

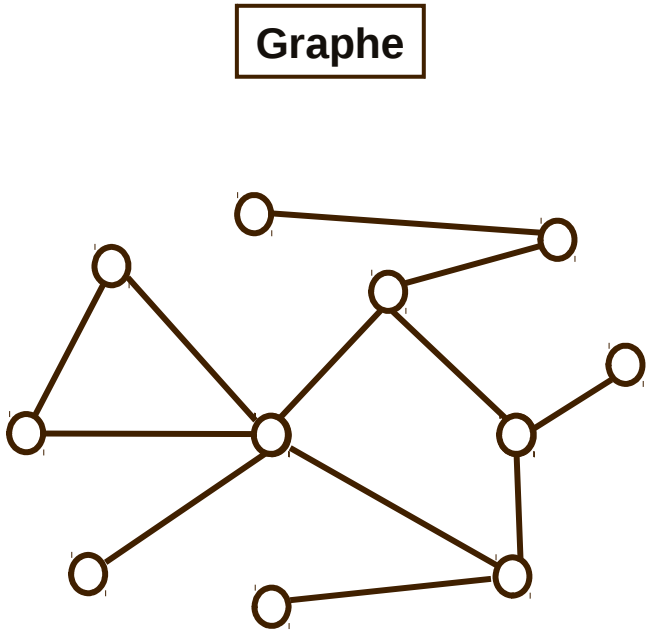


Complexité de la recherche : O(log2(N)) !!

Autres structures dynamiques

- L'utilisation des pointeurs permet de représenter de nombreuses autres structures.





Démonstrations animées : listes, arbres, graphes ...

David Galles, University of San Francisco

http://www.cs.usfca.edu/~galles/visualization/StackLL.html