

# **Informatique 05**

**Listes : exercices**

**Arbres binaires de recherche**

**François Bourdoncle**

`Francois.Bourdoncle@ensmp.fr`

`http://www.ensmp.fr/~bourdonc/`

# Exercices

- Listes
  - Insertion d'un élément dans une liste triée
  - Suppression d'un élément d'une liste triée
  - Concatenation impérative
  - Inversion impérative
- Arbres binaires de recherche
  - Recherche
  - Insertion
  - Suppression
  - Arbres de recherche équilibrés

# Listes triées

```
class List {
    // Champs des listes non vides
    int head;
    List tail;

    // Liste vide
    static List empty = null;

    // Prédicats
    static boolean isEmpty(List l) {
        return (l == null);
    }

    static boolean isCons(List l) {
        return (l != null);
    }

    // Constructeur
    static List cons(int head, List tail) {
        List l = new List();
        l.head = head;
        l.tail = tail;
        return l;
    }
}
```

## Insertion d'un élément

```
// Version sans effet de bord
static List insert(int x, List l) {
    if (isEmpty(l) || x < l.head) {
        return cons(x, l);
    } else {
        return cons(l.head, insert(x, l.tail));
    }
}
```

```
// Version avec effets de bord
static List insert(int x, List l) {
    if (isEmpty(l) || x < l.head) {
        return cons(x, l);
    } else {
        l.tail = insert(x, l.tail);
        return l;
    }
}
```

## Suppression d'un élément

```
// Version sans effets de bord
static List delete(int x, List l) {
    if (isEmpty(l) || x < l.head) {
        return l;
    } else if (x == l.head) {
        return l.tail;
    } else {
        return cons(l.head, delete(x, l.tail));
    }
}
```

```
// Version avec effets de bord
static List delete(int x, List l) {
    if (isEmpty(l) || x < l.head) {
        return l;
    } else if (x == l.head) {
        return l.tail;
    } else {
        l.tail = delete(x, l.tail);
        return l;
    }
}
```

## Inversion destructive d'une liste

```
static List reverse(List l) {  
    List r = empty;  
    while (isCons(l)) {  
        List t = l.tail;  
        l.tail = r;  
        r = l;  
        l = t;  
    }  
    return r;  
}
```

## Concaténation destructive de deux listes

```
// Version récursive
static List concat(List l1, List l2) {
    if (isEmpty(l1)) {
        return l2;
    } else {
        l1.tail = concat(l1.tail, l2);
        return l1;
    }
}
```

```
// Version itérative
static List concat(List l1, List l2) {
    if (isEmpty(l1)) {
        return l2;
    } else {
        List l = l1;
        while (isCons(l.tail)) {
            l = l.tail;
        }
        l.tail = l2;
        return l1;
    }
}
```

# Arbres binaires de recherche

- Domaine

$$\begin{aligned} \textit{Tree} &= \textit{Empty} + \textit{int} \times \textit{Tree} \times \textit{Tree} \\ \textit{Empty} &= \{\textit{empty}\} \end{aligned}$$

- Recherche de la présence d'un élément  
`static Tree find(Tree t, int x)`
- Insertion d'un élément  
`static Tree insert(Tree t, int x)`
- Suppression d'un élément  
`static Tree delete(Tree t, int x)`

## Arbres binaires de recherche — Java

```
class Tree {  
    int value;  
    Tree left;  
    Tree right;  
  
    static Tree empty = null;  
  
    Tree(int value) {  
        this.value = value;  
        this.left = empty;  
        this.right = empty;  
    }  
  
    Tree(int value, Tree left, Tree right) {  
        this.value = value;  
        this.left = left;  
        this.right = right;  
    }  
  
    static boolean isEmpty(Tree t) {  
        return t == empty;  
    }  
}
```

## Recherche

```
static Tree find(Tree t, int x) {  
    if (isEmpty(t) || t.value == x) {  
        return t;  
    } else if (x < t.value) {  
        return find(t.left, x);  
    } else {  
        return find(t.right, x);  
    }  
}
```

# Insertion

```
// Version sans effets de bord
static Tree insert(Tree t, int x) {
    if (isEmpty(t)) {
        return new Tree(x);
    } else if (x < t.value) {
        return new Tree(t.value,
                        insert(t.left, x),
                        t.right);
    } else {
        return new Tree(t.value,
                        t.left,
                        insert(t.right, x));
    }
}
```

```
// Version avec effets de bord
static Tree insert(Tree t, int x) {
    if (isEmpty(t)) {
        return new Tree(x);
    } else if (x < t.value) {
        t.left = insert(t.left, x);
        return t;
    } else {
        t.right = insert(t.right, x);
        return t;
    }
}
```

## Suppression de la racine

```
// L'arbre est supposé non-vide
static Tree deleteRoot(Tree t)
{
    if (isEmpty(t.right) {
        return t.left;
    } else if (isEmpty(t.left) {
        return t.right;
    } else if (isEmpty(t.right.left) {
        t.right.left = t.left;
        return t.right;
    } else {
        Tree r = t.right;
        Tree l = r.left;
        while (!isEmpty(l.left)) {
            r = l;
            l = r.left;
        }
        r.left = l.right;
        l.left = t.left;
        l.right = t.right;
        return l;
    }
}
```

## Suppression d'une valeur

```
static Tree delete(Tree t, int x) {  
    if (isEmpty(t)) {  
        return t;  
    } else if (x == t.value) {  
        return deleteRoot(t);  
    } else if (x < t.value) {  
        t.left = delete(t.left, x);  
        return t;  
    } else {  
        t.right = delete(t.right, x);  
        return t;  
    }  
}
```

# Arbres équilibrés

- Arbres 2-3-4
- Arbres AVL
  - Différence de hauteur  $\leq 1$
  - Recherche/insertion/suppression en  $O(\log n)$
  - Rotations

