Algorithmique Avancée

Nicolas Labroche

Cours 6 : arbres binaires de recherche et arbres généraux

Université de Tours

Arbres binaires de recherche

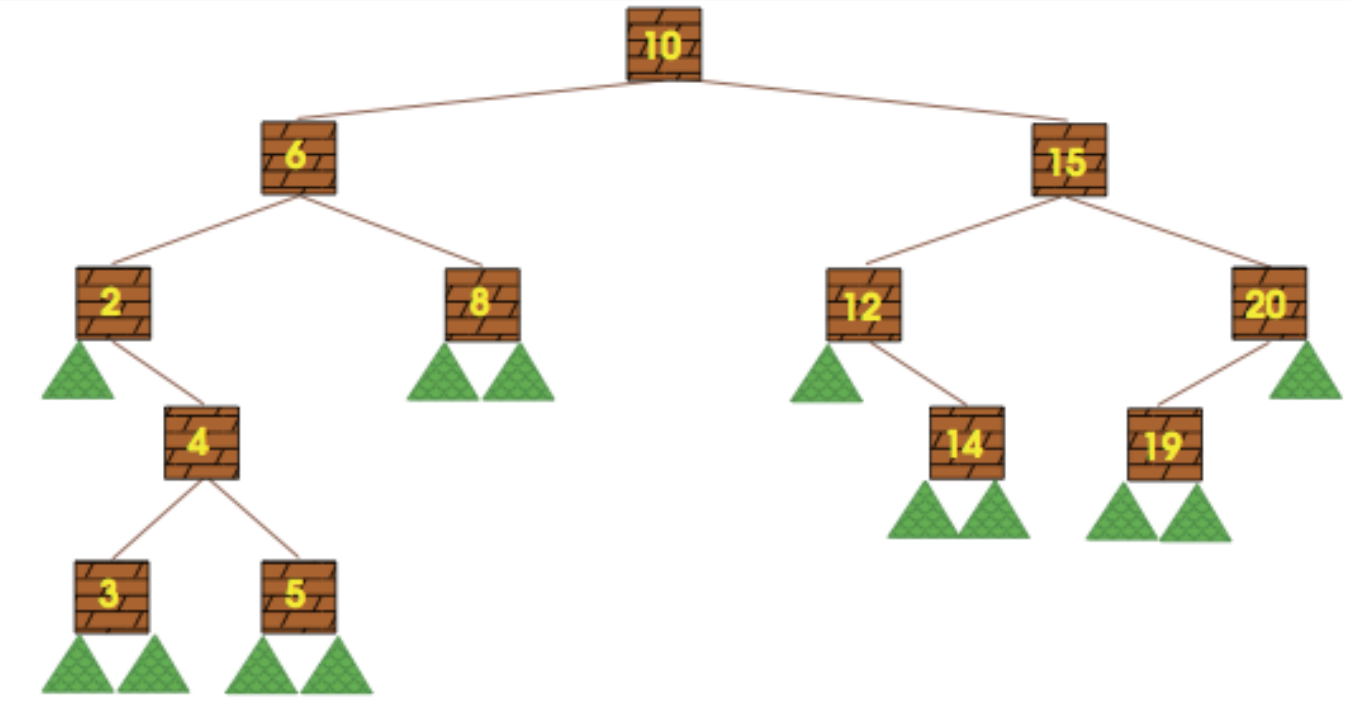
Définition d'un arbre binaire de recherche

Un arbre binaire de recherche (ou ABR) est un arbre binaire qui contient des éléments comparables (ordonnés) et tel que :

- l'étiquette de la racine est
  - supérieure à toutes les étiquettes du sous-arbre gauche,
  - inférieure à toutes les étiquettes du sous-arbre droit
- toutes les étiquettes sont distinctes deux-à-deux
- les sous-arbres gauche et droit sont aussi des ABR

Propriété caractéristique
Un arbre binaire A est un ABR si et seulement si la liste des étiquettes de A obtenues par parcours infixe de l'arbre est classée par ordre croissant.

Exemple



(Illustration issue du cours de LI101 UPMC - C. Marsala)

- Liste infixe triée par ordre croissant :

  `(2 3 4 5 6 8 10 12 14 15 19 20)`

Structure de données en Java

- Structure héritée d'un `AB<T>` classique

```
public class ABR<T extends Comparable<T>> extends AB<T>
```

- Prise en compte des spécificités
  - les étiquettes sont ordonnées strictement
  - pas de doublons
- Représentation d'un ensemble ordonné d'éléments avec les opérations suivantes
  - recherche d'un élément
  - ajout d'un élément : au niveau des feuilles, à la racine de l'arbre
  - suppression d'un élément au niveau des feuilles, à la racine

Structure de données en Java

- 2 constructeurs inspirés de ceux des arbres binaires classiques

```
public class ABR<T extends Comparable<T>> extends AB<T> {

  public ABR(T _label) {
    super(_label);
  }

  public ABR(T _label, ABR<T> _left, ABR<T> _right) {
    super(_label, _left, _right);
  }
}
```

Recherche d'un élément dans un ABR

Une comparaison avec l'étiquette détermine la suite du traitement

- soit l'étiquette est la valeur recherchée
- soit la valeur est plus petite que l'étiquette de la racine
  - $\Rightarrow$ le parcours continue dans le sous-arbre gauche
- soit la valeur est plus grande que l'étiquette de la racine
  - $\Rightarrow$ le parcours continue dans le sous-arbre droit

Ce principe d'aiguillage

- permet d'abandonner un sous-arbre entier (gauche ou droite)
- permet de recommencer le traitement récursivement sur un sous-arbre

Efficacité du parcours d'un ABR

- Rappel : la recherche d'une valeur dans un abre binaire classique est en $O(n)$ avec n nœuds
- Pour un ABR
  - déséquilibré dont les nœuds ne sont pas équitablement répartis autour de la racine
    - pire des cas : 1 seul nœud (la racine) supprimée à chaque itération

      $n \rightarrow n-1 \rightarrow n-2 \rightarrow ... \rightarrow n-(n-1) \Rightarrow O(n)$ opérations
  - équilibré : le nombre de nœuds restants à examiner est divisé par 2 à chaque itération

    $n \rightarrow n \rightarrow \frac{n}{2} \rightarrow ... \rightarrow \frac{n}{2^p} \Rightarrow p = O(log_2(n))$ opérations
    - c'est le principe de la recherche dichotomique

Illustration de la recherche (1)

Exemple : on recherche la valeur 4 dans l'ABR suivant

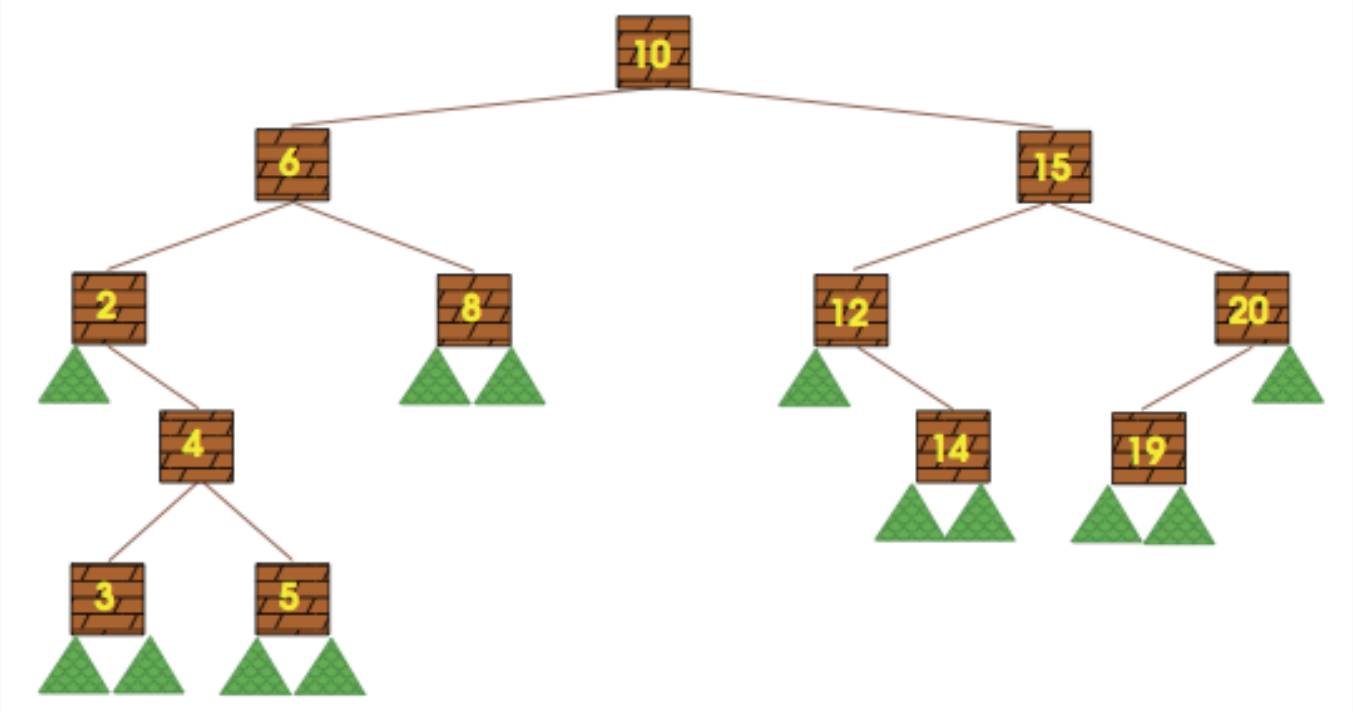


Illustration de la recherche (2)

- La valeur 4 n'est pas égale à l'étiquette de l'ABR courant.
- Elle est plus petite $\Rightarrow$ recherche dans le sous-arbre gauche

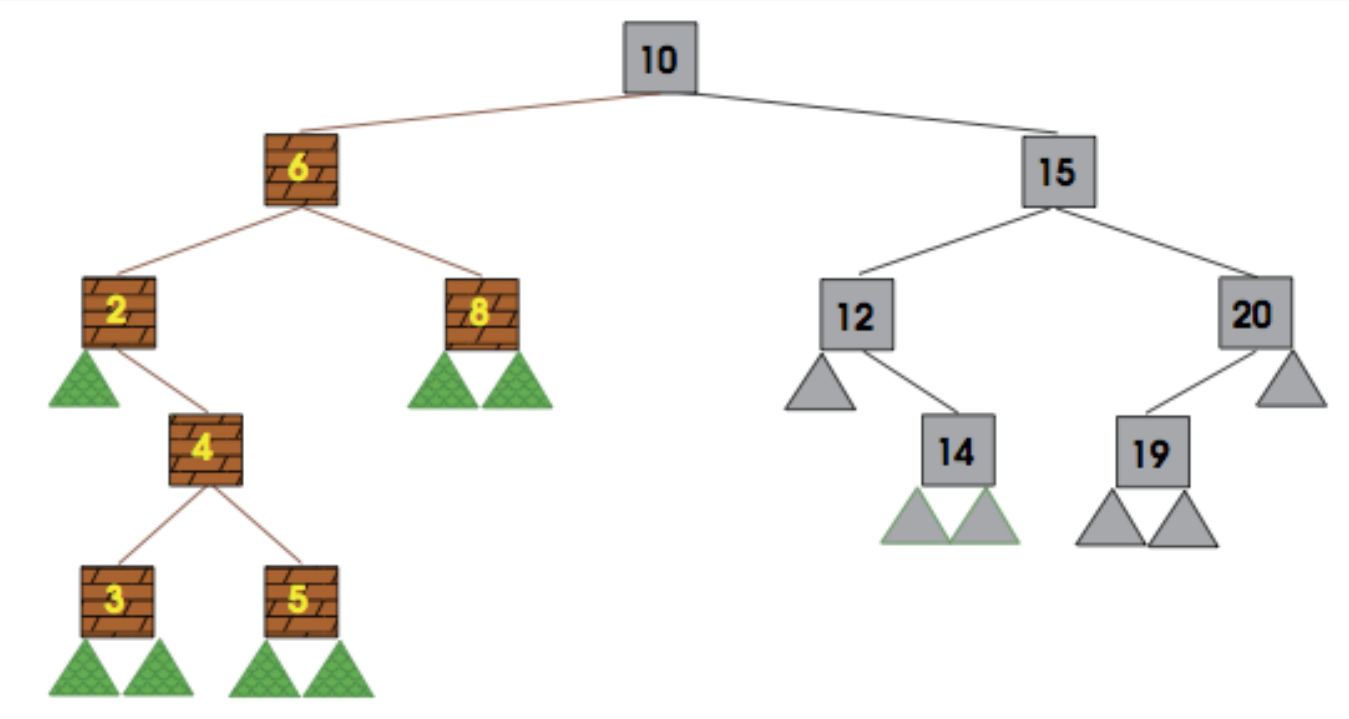


Illustration de la recherche (3)

- La valeur 4 est plus petite que l'étiquette de l'ABR courant.
- $\Rightarrow$ recherche dans le sous-arbre gauche

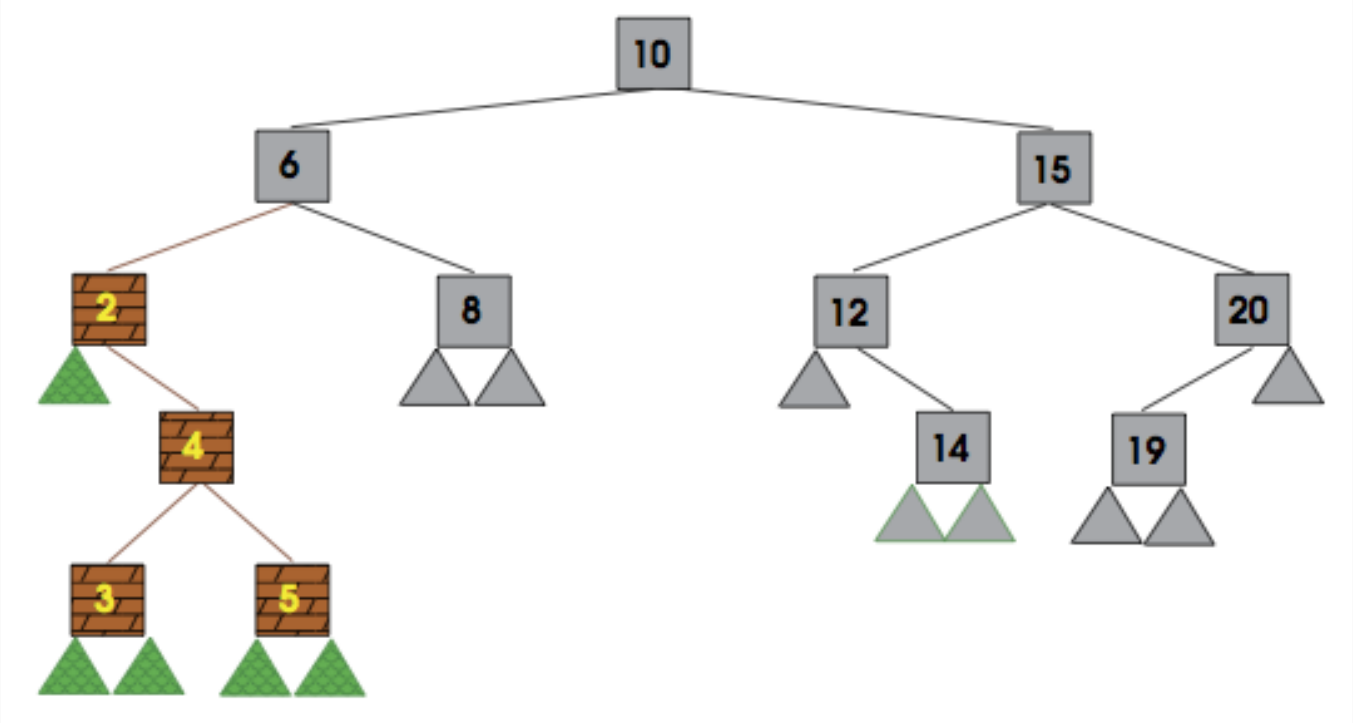
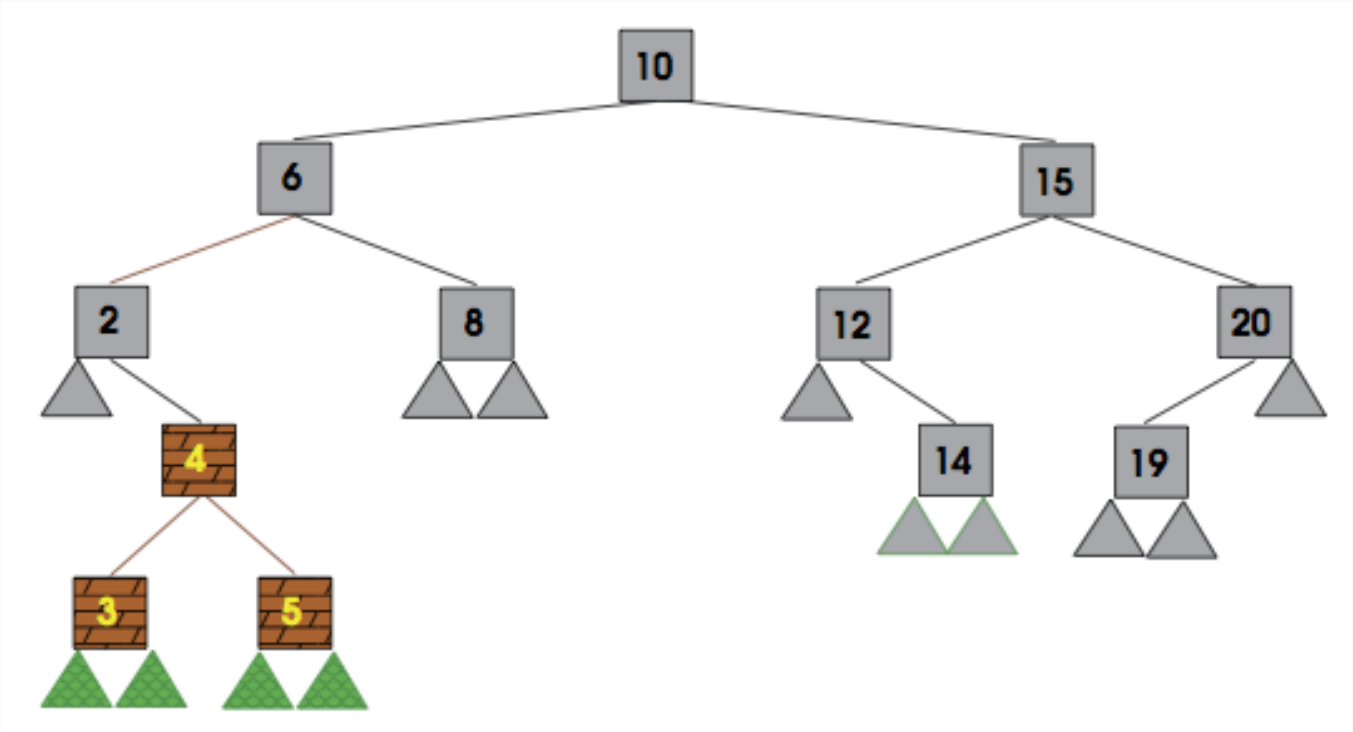


Illustration de la recherche (4)

- La valeur 4 est plus grande que l'étiquette de l'ABR courant
- $\Rightarrow$ recherche dans le sous-arbre droit



- L'arbre coloré est retourné comme résultat

Implémentation de la recherche en Java

```
@Override
public boolean contains (T elem) {
  if (isLeaf()) return this.getLabel().compareTo(elem) == 0;
  else {
    boolean found = this.getLabel().compareTo(elem) == 0;
    if (!found) {
      if (this.getLabel().compareTo(elem) > 0 && this.hasLeft↩
          ()) {
        found = this.getLeft().contains(elem);
      } else if (this.getLabel().compareTo(elem) < 0 && this.↩
          hasRight()) {
        found = this.getRight().contains(elem);
      } else found = false;
    }
    return found;
  }
}
```

Ajout d'un élément au niveau des feuilles d'un ABR

Principe :

- réaliser l'ajout à l'endroit où la recherche de l'élément échoue
- reconstruire l'arbre récursivement pour insérer le nouvel élément

Méthode :

- si l'arbre est vide : retourner une feuille contenant l'élément
- sinon, si l'étiquette = élément à ajouter retourner l'arbre tel quel
- sinon si l'étiquette > élément : insérer la valeur dans le sous-arbre gauche
- insérer la valeur dans le sous-arbre droit

Implémentation de l'insertion au niveau des feuilles

```
public void add(T elem) {
  if (getLabel().compareTo(elem) > 0) {
    // insert to the left
    if (hasLeft()) getLeft().add(elem);
    else this.setLeft(new ABR<T>(elem));
  } else if (getLabel().compareTo(elem) < 0) {
    // insert to the right
    if (hasRight()) getRight().add(elem);
    else setRight(new ABR<T>(elem));
  }
}
```

Suppression d'un élément dans un ABR (1)

Problème de suppression de la racine d'un ABR :

1. soit le nœud contenant l'élément est une feuille : cas simple
2. soit le nœud ne possède qu'un seul fils :
   $\Rightarrow$ on remplace le nœud par son fils
3. soit le nœud possède 2 fils $\Rightarrow$ suppression compliquée
   - rechercher l'élément le plus grand inférieur à la racine
     (M sur le schéma)
   - utiliser ce nœud pour remplacer la racine

Suppression d'un élément dans un ABR (2)

- mécanisme classique de suppression vers les sous-arbres gauche ou droit selon la valeur à supprimer

```
public ABR<T> remove(T elem){
  if (getLabel().compareTo(elem) < 0) {
    // try to remove from right tree if possible
    if (hasRight()) {
      return new ABR<T>(getLabel(), getLeft(), getRight().↩
          remove(elem));
    } else return this;
  } else if (getLabel().compareTo(elem) > 0) {
    // try to remove from left tree if possible
    if (hasLeft()) {
      return new ABR<T>(getLabel(), getLeft().remove(elem), ↩
          getRight());
    } else return this;
[...]
```

Suppression d'un élément dans un ABR (3)

- ensuite traitement du cas de la suppression de la racine
- si un seul fils retourner celui qui n'est pas vide
- sinon, utilisation du max issu du sous-arbre gauche pour remplacer la racine

```
[...]
  } else { // elem == this.getlabel()
    if (this.isLeaf()) return null;
    else if (!hasLeft()) return getRight();
    else if (!hasRight()) return getLeft();
    else { // left and right children
      T mx = getLeft().getMax();
      ABR<T> lTmp = getLeft().remove(mx);
      return new ABR<T>(mx, lTmp, getRight());
    }
  }
}
```

Rotation d'un arbre binaire de recherche

- Opération usuelle dans les arbres binaires de recherche pour les équilibrer qui conserve l'ordre des valeurs de l'ABR
- Faire remonter un nœud à la racine et faire descendre la racine actuelle : la profondeur d'un sous-arbre diminue et l'autre augmente

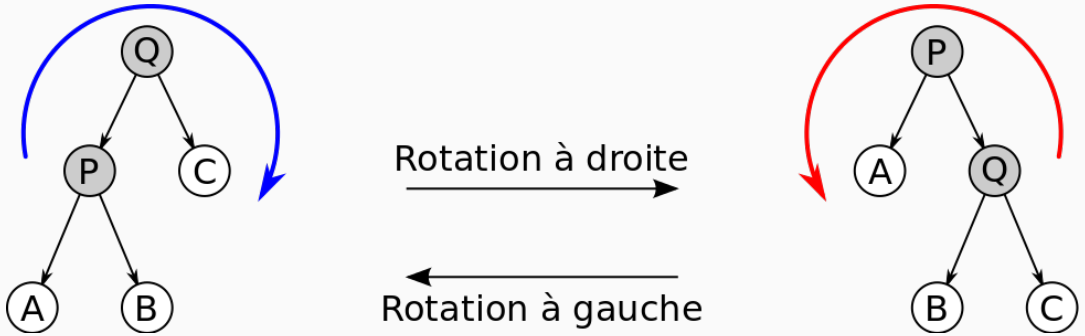
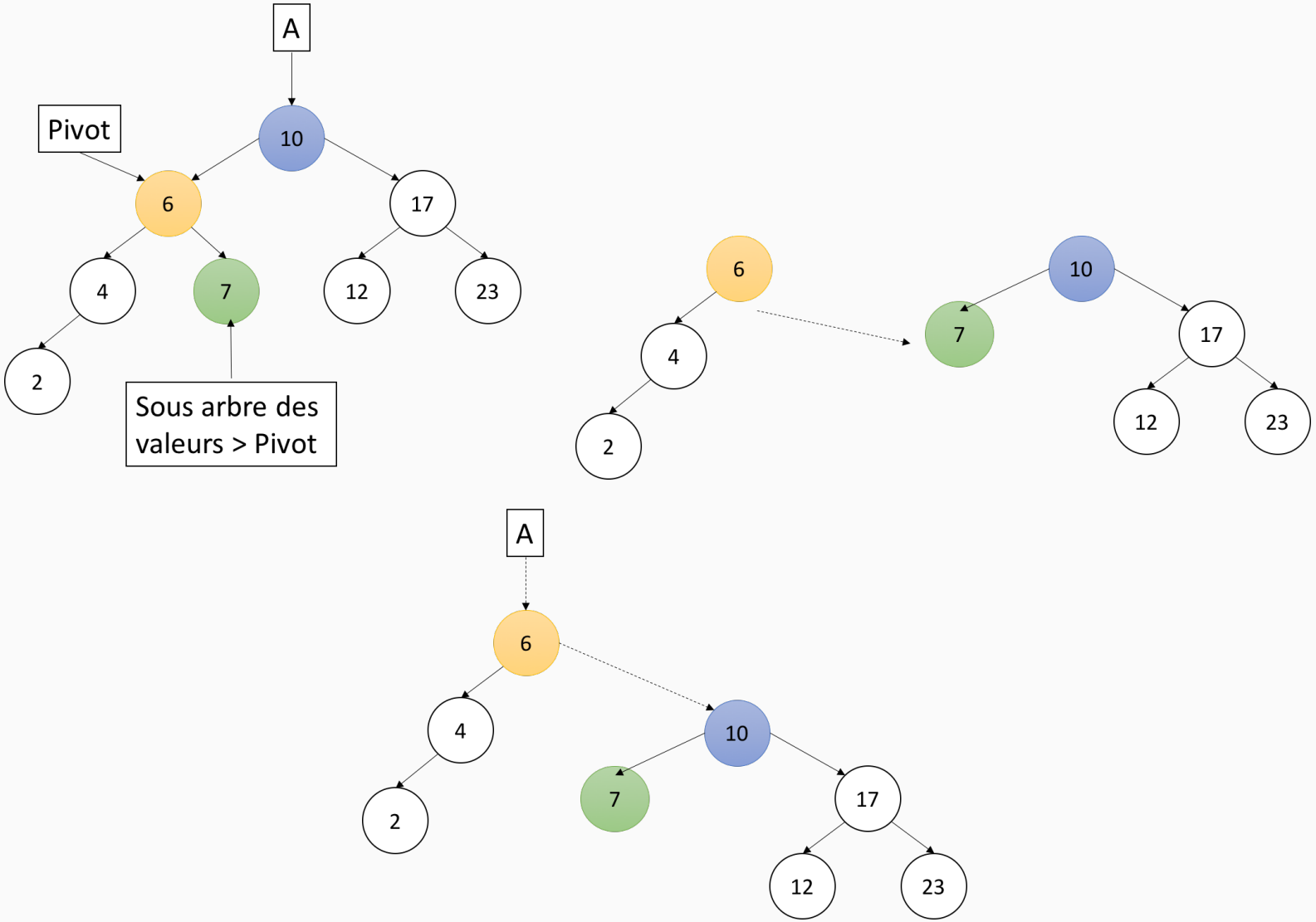


Illustration issue de wikipedia.fr

Illustration étape par étape



Algorithme de rotation droite

```java
/**
 * Hypothesis: left tree not empty
 */
public ABR<T> rightRotation(){
  // memorisation pointeur fils gauche
  ABR<T> pivot = (ABR<T>)this.getLeft();
  // recuperation du sous-arbre des valeurs > pivot.etiquette
  // qui devient la partie gauche du sous arbre A
  this.setLeft(pivot.getRight());
  // A devient le sous-arbre droit du pivot
  pivot.setRight(this);
  // on retourne le pivot
  return pivot;
}
```

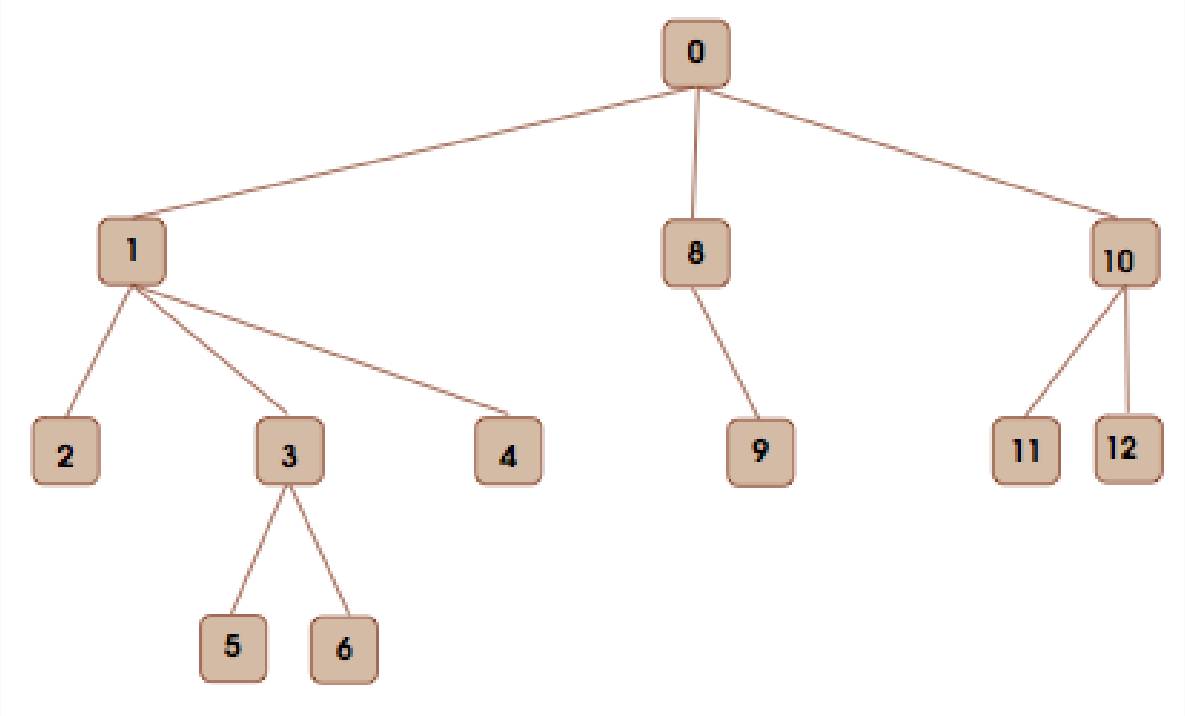
Arbres généraux

Arbres généraux

- Un arbre général (AG) est une structure de données qui permet de représenter des éléments de même type ordonnés de façon hiérarchique
- Un AG est une structure auto-référentielle composée de :
  - un nœud portant une étiquette
  - une liste chaînée de sous-arbres appelée forêt

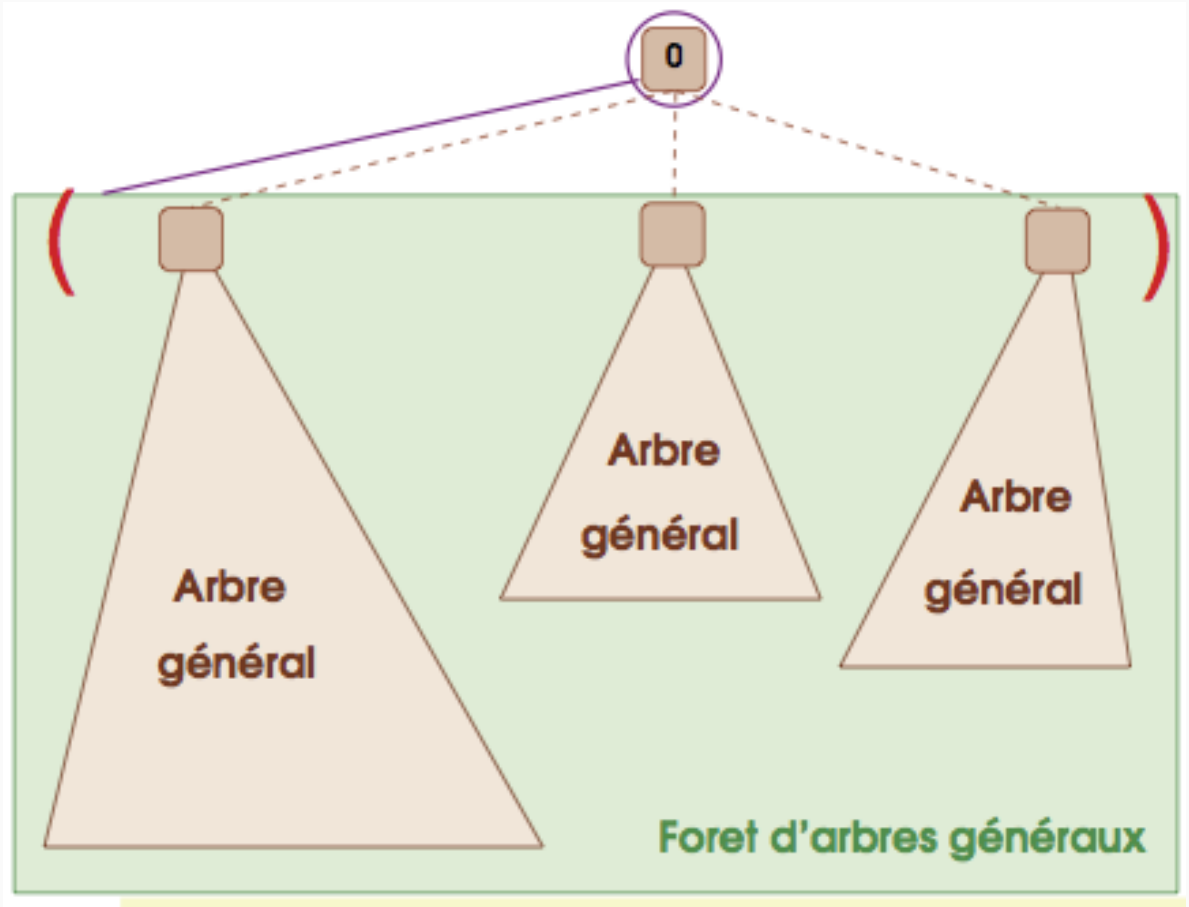
Illustration des arbres généraux

Exemple d'arbre général dont les nœuds contiennent des entiers



(Illustration LI101 UPMC par C. Marsala)

Structure des arbres généraux



(Illustration LI101 UPMC par C. Marsala)

Champs des AG

- Définition d'une classe `AG<T>` avec deux propriétés

```
T label;
IList<AG<T>> forest; // utilisation d'une liste avec ↩
    iterateur !
```

- Remarques
  - exemple de déclaration d'un `AG` vide : `AG a = null;`
  - en pratique, ce cas n'est pas très utile !
  - en revanche, la forêt peut être vide $\Rightarrow$ l'arbre est réduit à une feuille

Constructeurs des AG

- Deux constructeurs principaux

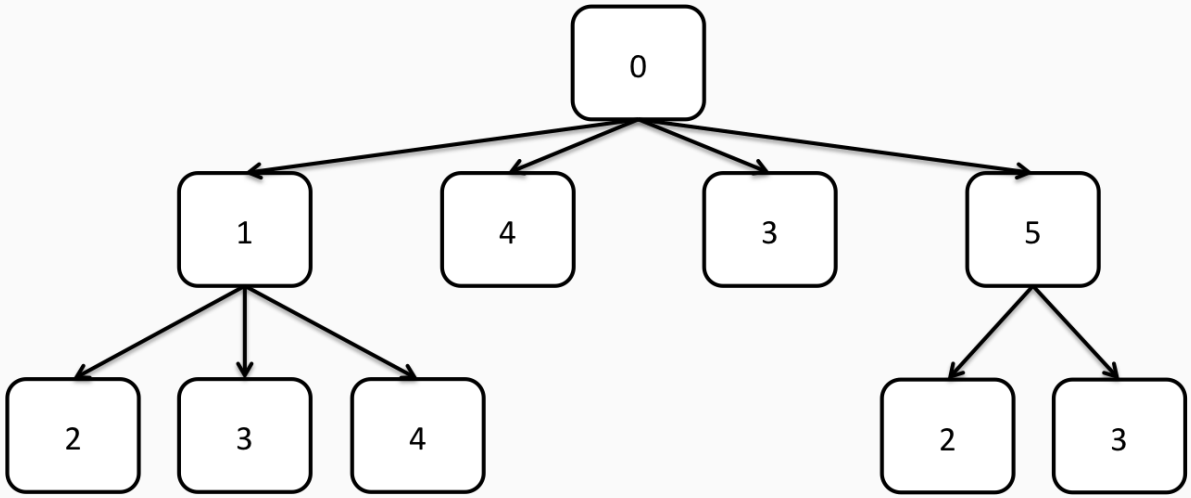
  1. constructeur d'un arbre réduit à une feuille = un seul noeud
  2. constructeur général

```
// construction d'une feuille
public AG(T elem) {
  label = elem;
  forest = new ISList<>();
}

// constructeur general
public AG(T elem, IList<AG<T>> f) {
  label = elem;
  forest = f;
}
```

Exemple de construction en Java

Donner les instructions Java pour construire un tel arbre :



Exemple de construction en Java

```
public static AG<Integer> ag1(){
    AG<Integer> g1 = new AG<Integer>(2);
    AG<Integer> g2 = new AG<Integer>(3);
    AG<Integer> g3 = new AG<Integer>(4);
    ISList<AG<Integer>> f = new ISList<AG<Integer>>(g1);
    f.add(g2); f.add(g3);
    AG<Integer> g4 = new AG<Integer>(1, f);
    AG<Integer> g5 = new AG<Integer>(4);
    AG<Integer> g6 = new AG<Integer>(3);
    AG<Integer> g8 = new AG<Integer>(2);
    AG<Integer> g9 = new AG<Integer>(3);
    ISList<AG<Integer>> f2 = new ISList<AG<Integer>>(g8);
    f2.add(g9);
    AG<Integer> g7 = new AG<Integer>(5, f2);
    ISList<AG<Integer>> f3 = new ISList<AG<Integer>>(g4);
    f3.add(g5); f3.add(g6); f3.add(g7);
    AG<Integer> racine = new AG<Integer>(0, f3);
    return racine;
}
```

Récursion sur les arbres généraux (1)

Un arbre général est constitué :

- d'un nœud racine portant une étiquette
- et d'une forêt (liste) d'arbres généraux (ses fils)

Pour traiter récursivement un arbre général il faut donc :

- traiter son étiquette
- traiter sa forêt, i.e. tous ses descendants directs
    - $\Rightarrow$ traiter chaque arbre de la forêt : parcourir la liste

Récursion sur les arbres généraux (2)

On utilise généralement deux fonctions dédiées :

- une pour le traitement de l'arbre
- une pour le traitement de la forêt

Remarques :

- pas besoin de traiter le cas d'un arbre vide
  - $\Rightarrow$ ce cas est géré au niveau de la forêt
- les fonctions "arbres" et "forêt" se référencent mutuellement
  - $\Rightarrow$ on parle de récursion croisée

Exemple 1 : compter le nombre de nœuds d'un AG

- Fonction récursive sur les arbres

```
public int nbNodes() {
  if (isLeaf()) return 1;
  else return 1 + this.nbNodesForest();
}
```

- Fonction récursive sur les forêts

```
private int nbNodesForest() {
  int sum = 0;
  Iterator<AG<T>> it = this.getForest().iterator();
  while (it.hasNext()) {
    sum += it.next().nbNodes();
  }
  return sum;
}
```

Exemple 2 : afficher un arbre général (1)

- Même schéma de résolution que pour un arbre binaire
  - on ajoute un argument aux fonctions pour mémoriser la profondeur
  - $\Rightarrow$ décalage à l'affichage en fonction de la profondeur
  - appel initial avec une profondeur $= 0$ pour la racine de l'AG

```
public String toString(){
  return aff(0);
}

private String shift(int shift){
  if (shift == 0) return "";
  StringBuilder buffer = new StringBuilder("");
  for (int i = 0; i < shift - 1; i++) buffer.append("    ");
  buffer.append("|-->");
  return buffer.toString();
}
```

Exemple 2 : afficher un arbre général (2)

- Fonction récursive sur les **arbres**

```
private String aff(int level){
  if (this.isLeaf()) return shift(level) + getLabel();
  else {
    return shift(level) + getLabel() + "\n" + aff_Forest(↩
        level + 1);
  }
}
```

Exemple 2 : afficher un arbre général (3)

- Fonction récursive sur les **forêts**

```
private String aff_Forest(int level) {
  StringBuilder sb = new StringBuilder("");
  Iterator<AG<T>> it = getForest().iterator();
  while (it.hasNext()) {
    sb.append(it.next().aff(level));
    sb.append("\n");
  }
  return sb.toString();
}
```

Exemple 2 : afficher un arbre général (4)

Résultat sur l'arbre de test précédent

```
0
|−−>1
    |−−>2
    |−−>3
    |−−>4
|−−>4
|−−>3
|−−>5
    |−−>2
    |−−>3
```