INF 421 Luc Maranget

Arbres binaires, ensembles

Luc.Maranget@inria.fr

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421/

Réalisation des ensembles avec les listes

On peut représenter le *ensembles* par des listes.

Il suffit de maintenir la condition :

Les éléments d'un ensemble sont deux à deux distincts.

Il faut donc pouvoir déterminer l'égalité de deux éléments.

Dans la suite, nos éléments sont des **int**, mais cela pourrait facilement être des objets quelconques (par redéfinition de la méthode `equals` des `Object`).

La classe des listes (d'entiers)

Air connu…

```
class List {
  int val ;
  List next ;

  List (int val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

Opérations élémentaires

- Test d'appartenance.

```
static boolean mem(int x, List p) {
  for ( ; p != null ; p = p.next) {
    if (x == p.val) return true ;
  }
  return false ;
}
```

- Ajout

```
static List add(int a, List p) {
  if (mem(a, p)) {
    return p ;
  } else {
    return new List (a, p) ;
  }
}
```

Opération ensembliste : union

- Récursif.

```
static List union(List p, List q) {
  if (p == null) {
    return q ;
  } else {
    return add(p.val, union(p.next, q)) ;
  }
}
```

- Itératif.

```
static List union(List p, List q) {
  List r = q ;
  for ( ; p != null ; p = p.next) {
    r = add(p.val, r) ;
  }
  return r ;
}
```

Bilan des coût

Quel est alors le coût asymptotique dans le cas le pire :

- Du test d'appartenance ? $O(n)$ (penser à l'échec, compter les appels de fonction).
- De l'ajout ? $O(n)$ (comme `mem`).
- De l'union (de deux ensembles de cardinaux n et m) ? $O(n \times (n+m))$ (n fois `add`).

Une idée

Normaliser les listes : représenter un ensemble par la liste triées (ordre croissant) de ses éléments.

Appartenance

On peut utiliser l'ancienne méthode `mem` où une méthode un peu améliorée.

```
static boolean mem(int x, List p) {
  if (p == null || x < p.val) {
    return false ;
  } else {
    return p.val == x || mem(x, p.next) ;
  }
}
```

Ajout et union

- Ajout ? Insertion dans une liste triée (cf. insertion sort).
- Union ? Fusion de deux listes triées (cf. merge sort).

Bilan des coûts

	`mem`	`add`	`union`
Liste	$O(n)$	$O(n)$	$O(n^2)$
Liste triée	$O(n)$	$O(n)$	$O(n)$

Remarquer L'implémentation « liste triée » favorise l'opération ensembliste, mais n'améliore pas les autres opérations.

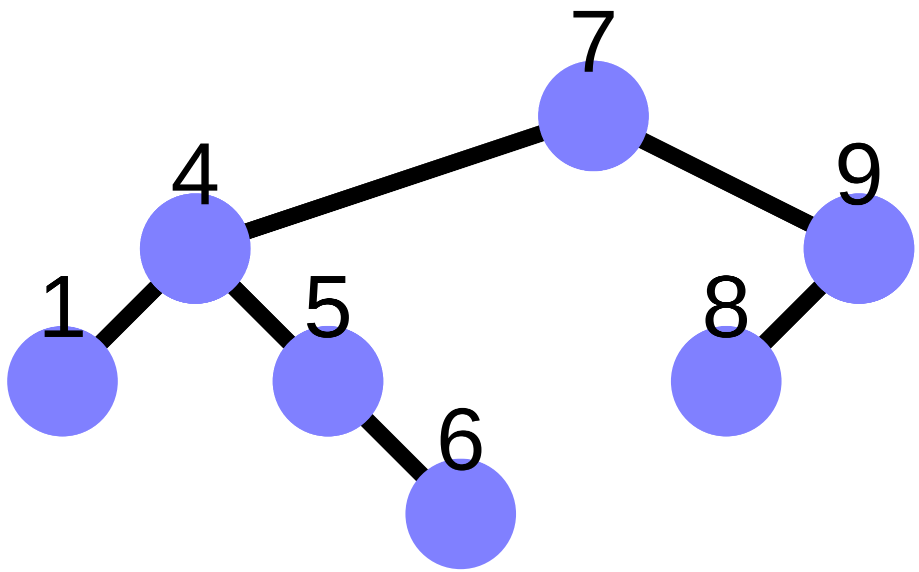
Représenter les ensembles avec les arbres

On définit les *arbres binaires de recherche* :

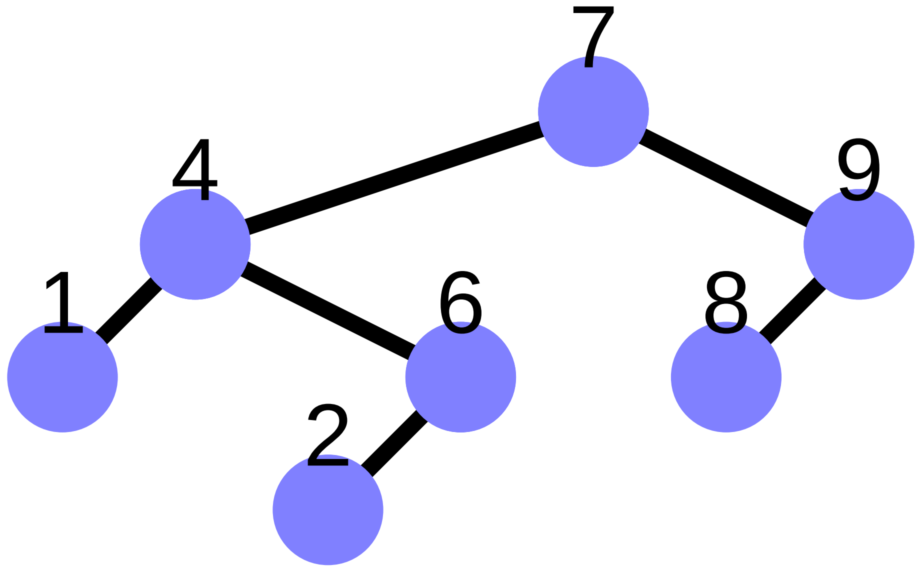
- L'arbre vide est un ABR, ses clefs sont $\emptyset$.
- Si T_0 et T_1 sont des ABR, de clefs respectives C_0 et C_1 et :
  - x majore (strictement) toutes les clefs de C_0 ;
  - x minore (strictement) toutes les clefs de C_1 ;

  alors (T_0, x, T_1) est un ABR et ses clefs sont $C_0 \cup \{x\} \cup C_1$.

Un exemple d'arbre binaire de recherche



Un contre-exemple d'arbre binaire de recherche



Classe Tree des arbres binaires de recherche

```
class Tree {
  int key ;
  Tree left, right ;

  Tree (Tree left, int key, Tree right) {
    this.left = left ; this.right = right ;
    this.key = key ;
  }

  Tree (int key) {
    this.key = key ; left = right = null ;
  }
}
```

Test d'appartenance dans un ABR

C'est simple si on pense récursivement.

```
static boolean mem(int x, Tree t) {
  if (t == null) {
    return false ;
  } else {
    if (x < t.key) {
      return mem(x, t.keft) ; // Chercher à gauche
    } else if (x > t.key) {
      return mem(x, t.right) ; // Chercher à droite
    } else { // x == t.key
      return true ; // Trouvé ici
    }
  }
}
```

Test d'appartenance dans un ABR II

Finalement assez simple à programmer itérativement, penser que l'on suit un chemin dans un arbre.

```
static boolean mem(int x, Tree t) {
  while (t != null) {
    if (x < t.key) {
      t = t.left ;
    } else if (x > t.key) {
      t = t.right ;
    } else { // x == t.key
      return true ;
    }
  }
  return false ;
}
```

Ajouter un élément : insertion dans un ABR

```
static Tree add(int x, Tree t) {
  if (t == null) {
    return new Tree(x) ;
  } else {
    if (x < t.key) {
      return new Tree (add(x, t.left), t.key, t.right) ;
// ajouter à gauche
    } else if (v > t.key) {
      return new Tree (t.left, t.key, add(x, t.right)) ;
// ajouter à droite
    } else { // v == t.key, déjà là
      return t ;
    }
  }
}
```

Programmation itérative possible, mais trop complexe.

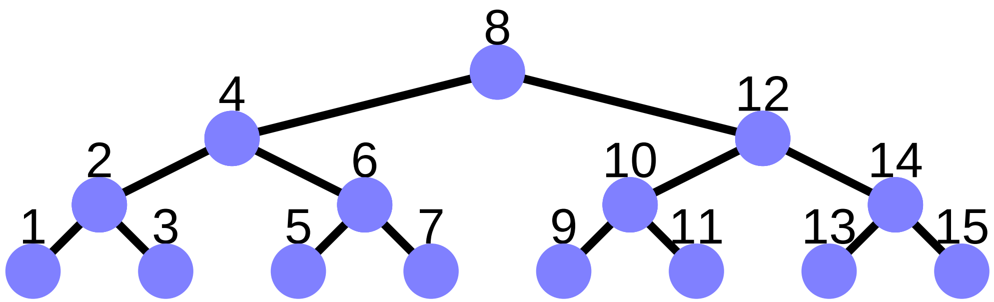
Coût des deux opérations élémentaires

Il est facile de voir que le coût de `mem` et `add`, est en $O(h)$ où h est la hauteur de l'ABR.

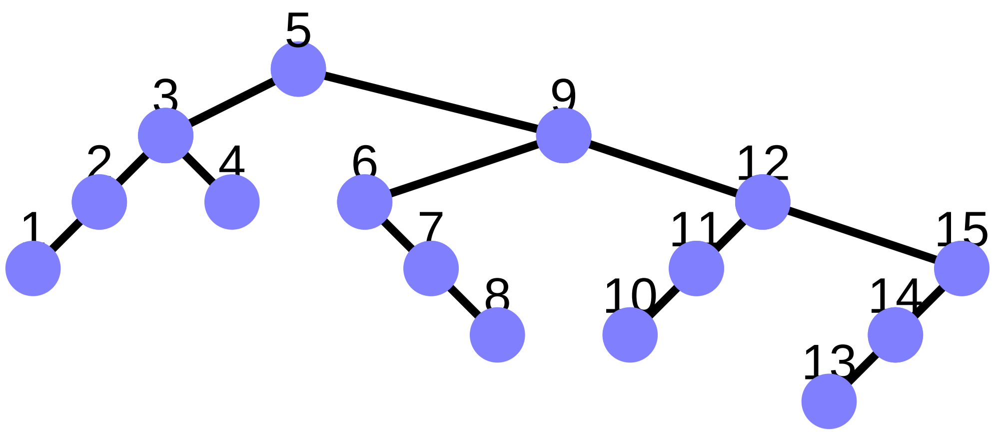
Mais on veut borner le coût fonction de n cardinal de l'ensemble...

Il faut donc exprimer la hauteur h en fonction du cardinal n.

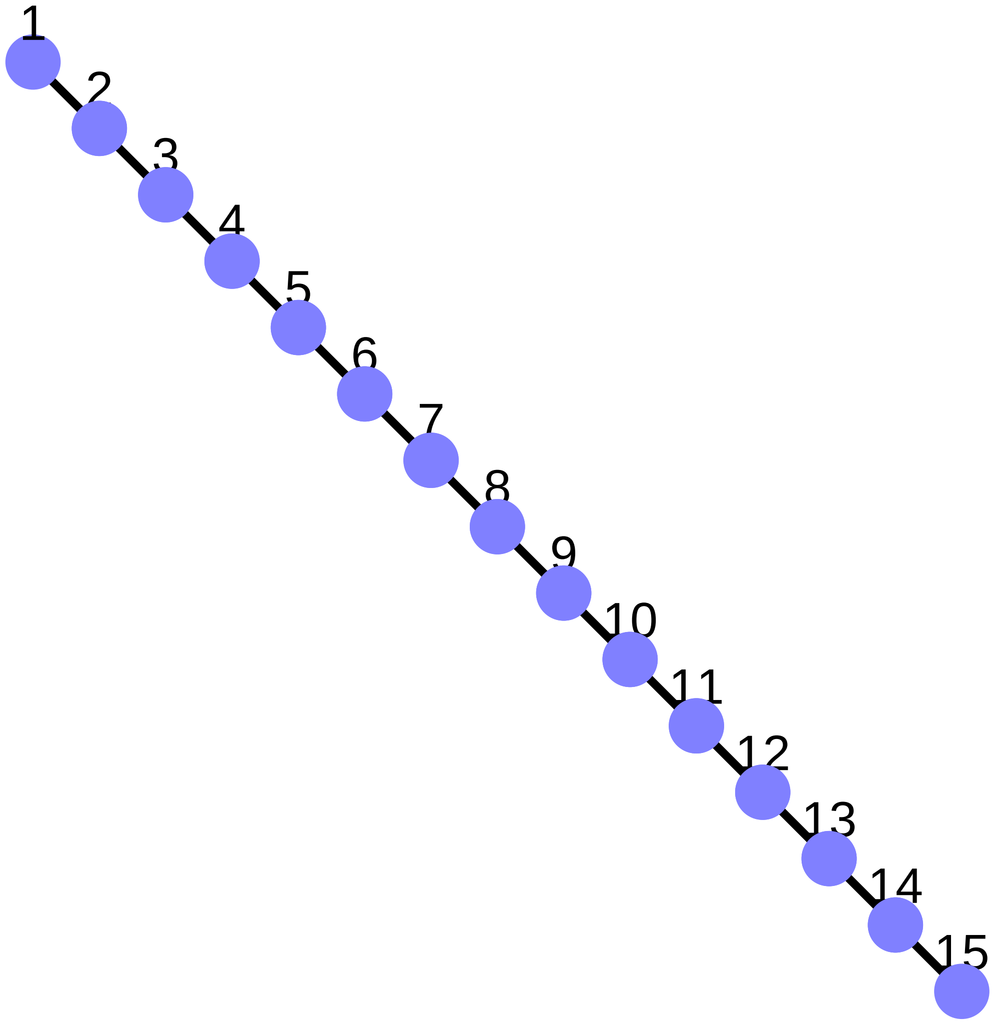
Un arbre (très) équilibré



Un arbre à peu près équilibré

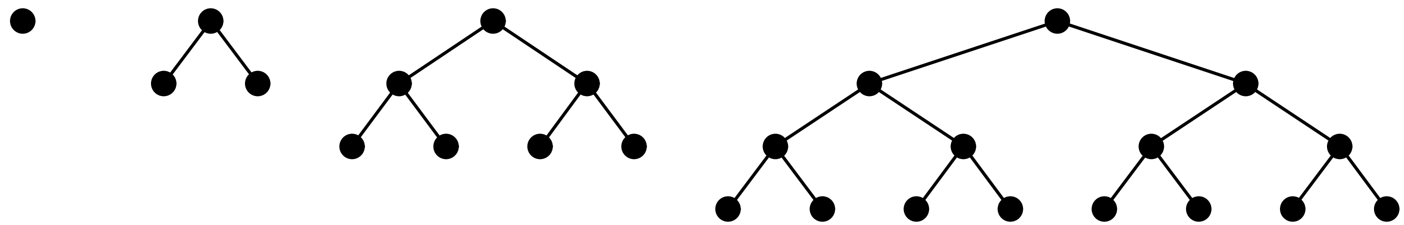


Un arbre (très) déséquilibré



Hauteur minimale d'un arbre binaire

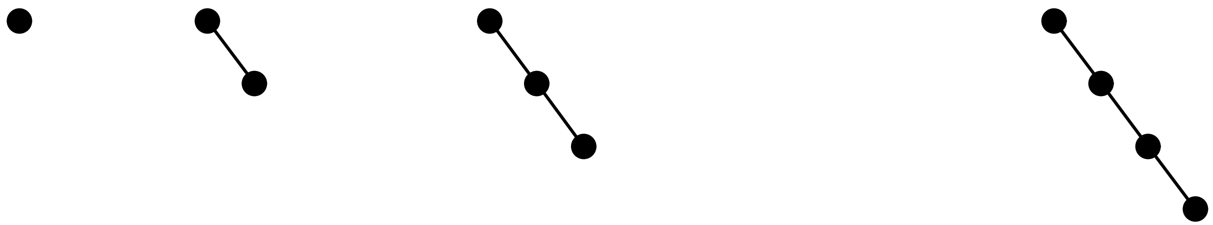
Ou nombre maximal de sommets pour une hauteur donnée : arbre binaire *complet*



$$n \leq 2^{h+1} - 1, \quad h \geq \log_2(n+1) - 1$$

Hauteur maximale d'un arbre binaire

Ou nombre minimal de sommets pour une hauteur donnée : arbre dégénéré en liste.



$$h = n$$

Complexité en moyenne

Sur un univers E qui regroupe des événements e de probabilité $P(e)$, soit une fonction $X(e)$.

L'espérance (moyenne) de X est définie par :

$$E(X) = \sum_{e \in E} \mathrm{Prob}(e) \cdot X(e)$$

Par exemple :

- e est un entier $1 \leq k \leq 2n$,
- X est le nombre d'appels récursifs à `mem`, effectués lors du test d'appartenance de k à la liste (triée) des entiers *pairs* compris entre 2 et $2n$.

Appartenance dans les listes, coût en moyenne

- Cas du `mem` qui ne sait pas que la liste est triée.

$$X(2p) = p, \quad X(2p-1) = n$$

Et donc :

$$E(X) = \sum_{p=1}^{n} \frac{1}{2n} p + \sum_{p=1}^{n} \frac{1}{2n} n = \frac{3n+1}{4}$$

- Cas du `mem` qui sait que la liste est triée.

$$X(2p) = p, \quad X(2p-1) = p$$

Et donc :

$$E(X) = 2 \sum_{p=1}^{n} \frac{1}{2n} p = \frac{1}{2}(n+1)$$

Deux résultats en moyenne, sur les ABR

Pour n tendant vers $+\infty$...

- La hauteur moyenne des arbres à n sommets est de l'ordre de $\sqrt{n}$ (pas de bol).
- La hauteur moyenne des arbres produits par addition de 1, $\ldots n$, dans tous les ordres possibles est elle de l'ordre de $\log(n)$.

Le second résultat nous permet plus ou moins de considérer qu'un arbre pris au hasard est de hauteur $\log(n)$. Mais...

- Le coût dans le cas le pire reste une hauteur en n.
- Et ce cas très déséquilibré est malheureusement assez probable en pratique (addition de $1, 2, \ldots n$).

Les arbres AVL

Les arbres AVL (Adelson-Velinsky-Landis) sont des arbres binaires plus :

- Les hauteurs des sous-arbres gauche et droit diffèrent *au plus* de un

Une conséquence importante :

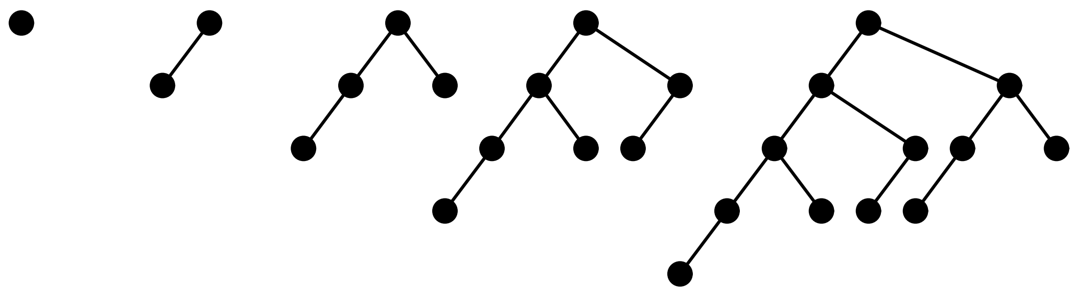
- La hauteur d'un arbre AVL est en $\log(n)$.

$$\log_2(1+n) \leq 1+h \leq \alpha \log_2(2+n)$$

(avec $\alpha \leq 1.44$)

Diversion : explication rapide des bornes

- La borne inférieure $\log_2(1+n) \leq 1+h$ est vraie de tous les arbres binaires (un arbre de hauteur h à au plus $2^{h+1}-1$ nœuds).
- La borne supérieure est plus intéressante. On considère l'arbre AVL F_h le plus petit possible pour une hauteur donnée.
  - Hauteur zéro : arbre vide.
  - Hauteur un : une feuille.
  - Hauteur $h+2$: $F_{h+2} = (F_{h+1}, x, F_h)$.



Arbre équilibré de hauteur maximale

Les équations définissant le nombre de sommets de l'arbre de taille minimale pour h donné, sont donc.

$$F(0) = 0, \quad F(1) = 1, \quad F(h+2) = 1 + F(h+1) + F(h)$$

Posons $G(h) = F(h) + 1$, il vient :

$$G(0) = 1, \quad G(1) = 2, \quad G(h+2) = G(h+1) + G(h)$$

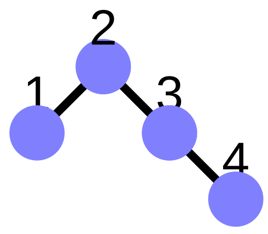
On trouve :

$$G(h) = \frac{5-\sqrt{5}}{2}\Phi^h + \frac{\sqrt{5}-3}{2}\Phi^{-h}, \quad \text{avec}\Phi = \frac{1+\sqrt{5}}{2}$$

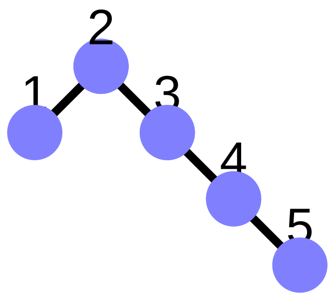
Au final la hauteur maximale d'un AVL est de l'ordre de $\log_\Phi(n)$, donc de l'ordre de $\log_2(n)$.

La vraie question des AVL : l'équilibre

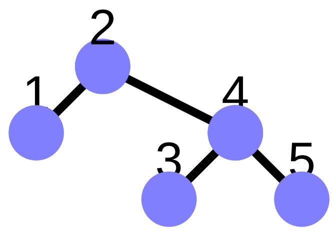
Soit un AVL,



On ajoute l'élément 5



On rétablit l'équilibre.



Déséquilibre (à droite).

Comment garantir l'équilibre ?

Supposons écrite une méthode
`balance(AVL left, int key, AVL right)` qui renvoie un arbre équilibré.

Alors c'est facile, (comme pour les ABR normaux).

```
static AVL add(int x, AVL t) {
  if (t == null) {
    return new AVL(x) ;
  } else {
    if (x < t.key) {
      return balance(add(x, t.left), t.key, t.right) ;
    } else if (v > t.key) {
      return balance(t.left, t.key, add(x, t.right)) ;
    } else {
      return t ;
    }
  }
}
```

Classe des AVL

```
class AVL {
  int key ;
  private int h ; // Champ hauteur (pour éviter le recalcul)
  AVL left, right ;

  static int hauteur(AVL t) {
    if (t == null) { return 0 ; }
    else { return t.h ; }
  }

  AVL (AVL left, int key, AVL right) {
    this.left = left ; this.right = right ; this.key = key ;
    this.h = Math.max(hauteur(left), hauteur(right)) + 1 ;
  }
  AVL (int key) {this.key = key ; left = right = null ; h = 1 ;}
}
```

Il reste à écrire `balance`.

Écrivons `balance`

`balance` prend deux arbres `left` et `right` en argument, avec (par construction)

$$-2 \leq \texttt{hauteur(left)} - \texttt{hauteur(right)} \leq 2$$

On suppose un déséquilibre, c'est à dire par ex.

$$\texttt{hauteur(left)} = \texttt{hauteur(right)} + 2$$

C'est à dire :

- On a ajouté un élément dans le sous arbre de gauche,
- sa hauteur a augmenté (de un),
- alors qu'avant ajout le sous-arbre gauche était déjà plus haut (de un) que le sous-arbre droit

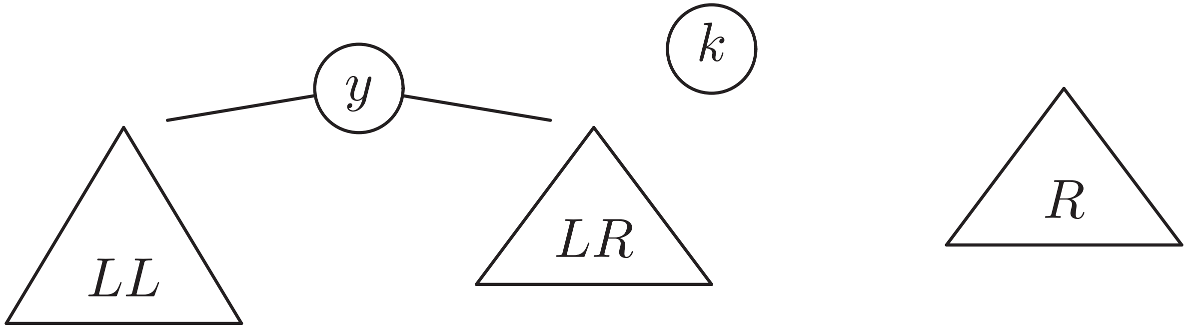
Analysons un peu encore

Notons L (arbre de gauche), k (clé) et R les arguments de `balance`.

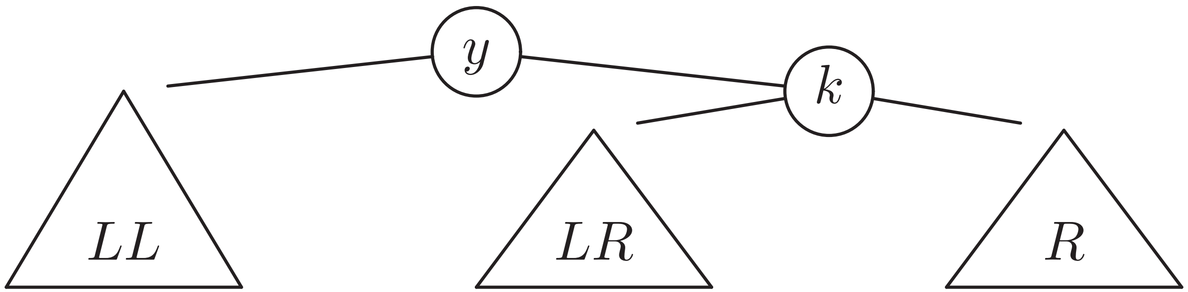
- On note que (L, k, R) est un ABR, mais (possiblement) déséquilibré.
- On suppose le déséquilibre : $h(L) = h(R) + 2$ et donc $L = (LL, y, LR)$. Notons $H(R) = \delta$. Deux sous-cas :
  - Le sous-arbre, LL impose sa hauteur à L, c'est à dire $h(LL) = \delta + 1$ (et $h(LR) = \delta$ ou $h(LR) = \delta + 1$).
  - Ou bien, LL n'impose pas sa hauteur à L, c'est à dire $h(LL) = \delta$ et $h(LR) = \delta + 1$.

Premier cas

Supposons donc $h(LL) = \delta + 1$ (et $h(LR) = \delta + 1$ ou $h(LR) = \delta$).



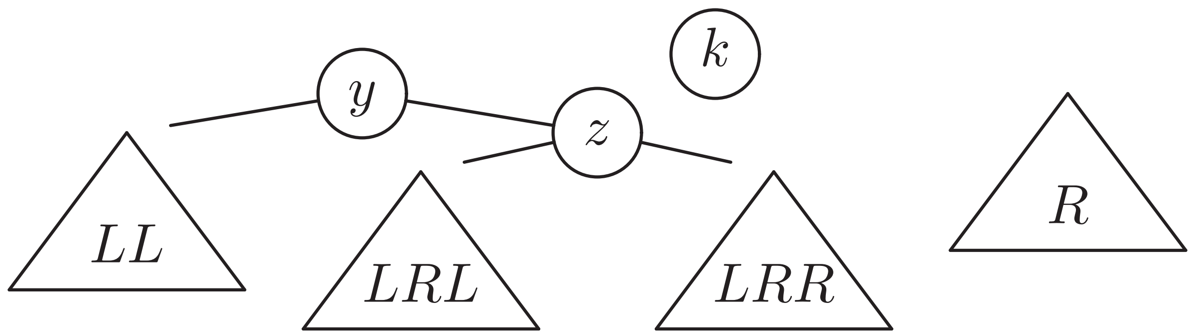
Alors, l'arbre suivant est équilibré :



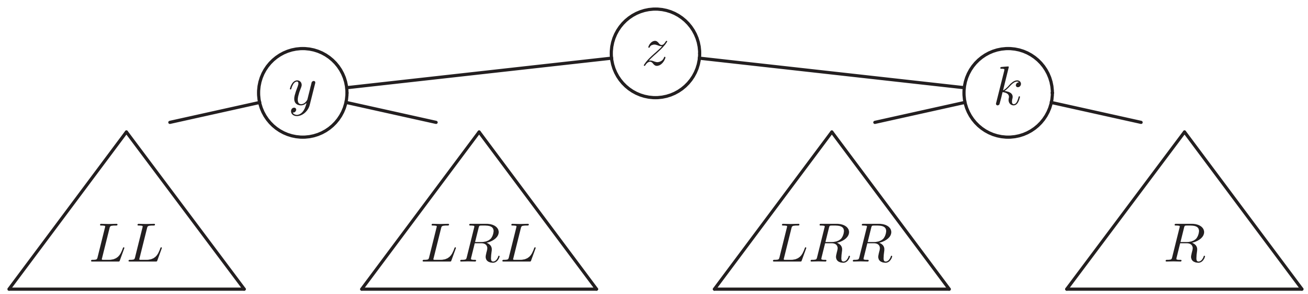
De hauteur $\delta + 2$ ou $\delta + 3$.

Second cas

Ce cas ($\delta = h(LL) < h(LR) = \delta + 1$) entraîne LR non-vide.



Alors, l'arbre suivant est équilibré



De hauteur $\delta + 2$.

Dans un souci de complétude

```
static AVL balance(AVL left, int val, AVL right) {
  int hl = hauteur(left), hr = hauteur(right);

  if (hl > hr + 1) { // => left != null
    if (hauteur(left.left) >= hauteur(left.right))
      return
        new AVL(left.left,left.key,new AVL (left.right,val,right));
    else // => left.right != null
      return
        new AVL(new AVL (left.left,left.key,left.right.left),
                       left.right.key,
                       new AVL (left.right.right,val,right));
  } else if (hr > hl + 1) { // Même chose en symétrique
    ...
  } else
    return new AVL (left,val,right);
}
```

Conclusion temporaire

Le minimum à savoir.

- Les ABR *equilibrés* sont une implémentation très générale et efficace (ajout en $O(\log n)$) des ensembles.
- Pour pouvoir l'employer, il fait un ordre total sur les éléments (pas nécessaire pour les listes non-triées).
- Cette implémentation est persistante, on peut écrire

  ```
  AVL xs = ... ;
  AVL ys = AVL.add(2, xs) ;
  ```

  Et : l'ensemble `xs` ne change pas.

Dans le même ordre d'idée

Les arbres equilibrés permettent une implémentation efficace des associations (amphi 04), cette fois *persistantes*.

- Il suffit d'ajouter un champ `val` dans la définition des cellules d'arbre.

```
class TreeEnv { // Association des chaînes aux entiers
  String key ;
  int val ;
  TreeEnv left, right ;
  ...
}
```

- On retrouver la valeur associée à une clé par un genre de `mem`.

```
static int get(TreeEnv env, String key, int val) {
  ...
}
```

- On ajoute une association par un genre de `add`.

  ```
  static TreeEnv put(TreeEnv env, String key, int val) {
    ...
  }
  ```

  La méthode `put` renvoie `env` augmenté de la nouvelle association, `env` n'est pas modifé.

- Et ici il faut remarquer la différence avec la signature par exemple des tables de hachage.

  void `put(`K `key,` V `value)`

Les ABR de la bibliothèque

- Java fournit les ensembles d'objets réalisés par des arbres équilibrés : la classe `TreeSet`[a] (package **`java.util`**). Il s'agit encore une fois, d'une classe générique `TreeSet<`E`>` est un ensemble de E.

- Les objets-éléments sont ordonnés, c'est-à-dire qu'ils doivent implémenter l'interface `Comparable`[b].

```
interface Comparable<E> {
 int comparesTo(E o) ;
}
```

  C'est par exemple le cas des chaînes `String`.

De façon surprenante, les `TreeSet` de Java, sont des ensembles impératifs (non-persistants), dommage.

[a] http://java.sun.com/j2se/1.5.0/docs/api/java/util/TreeSet.html

[b] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html

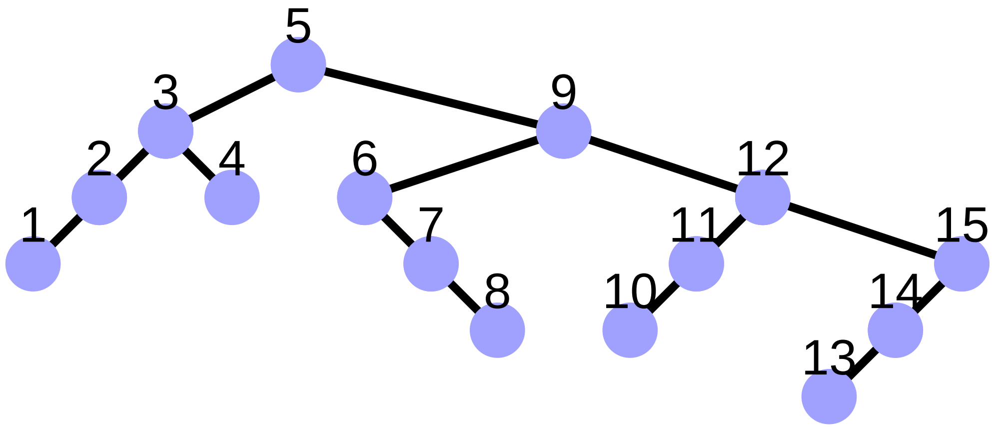
Complément : enlever un élément

Prenons le cas des ABR, on trouve les premiers cas par induction.

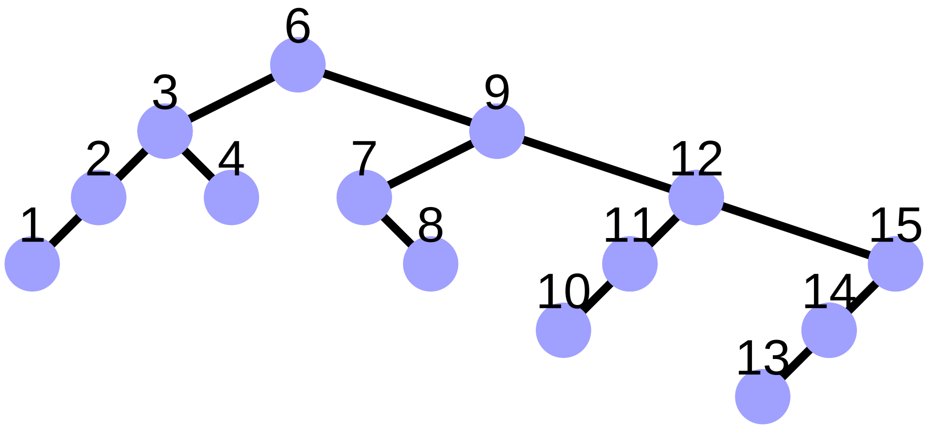
```
static Tree remove(Tree t, int v) {
  if (t == null) {
    return null ;
  } else if (v < t.key) {
    return new Tree (remove(t.left, v), t.key, t.right) ;
  } else if (v > t.key) {
    return new Tree (t.left, t.key, remove(t.right, v)) ;
  } else if (t.right == null) {
    return t.left ;
  } else {// Ici v == t.val, t.right != null
    ...
```

Le dernier cas n'est pas immédiat : il faut mélanger `t.left` et `t.right` en un seul ABR.

Exemple : enlever la racine



L'idée, remplacer la racine par le minimun du sous-arbre droit.



Trouver/enlever le minimum

- Trouver : à gauche toute !

```
static int getMin(Tree t) {
  while (t.left != null) {
    t = t.left ;
  }
  return t.key ;
}
```

- Enlever, à gauche encore !

```
static Tree removeMin(Tree t) {
  if (t.left == null) {
    return t.right ;
  } else {
    return new Tree (removeMin(t.left), t.key, t.right) ;
  }
}
```

Remplacer la racine

Voci le code manquant de `remove`.

```
static Tree remove(Tree t, int v) {
  ...
  } else { // Ici v == t.val, t.right != null
    int min = getMin(t.right) ;
    return new Tree(t.left, min, removeMin(t.right)) ;
  }
}
```

Et les AVL ? Remplacer, dans `remove` et `removeMin`, tous les appels du constructeur **new** `Tree(`ℓ`,` v`,` r`)` par des appels de méthode `balance(`ℓ`,` v`,` r`)`.

Correction ? Le déséquilibre entre ℓ et r est limité à 2 au plus.

Opération ensemblistes

Réaliser par exemple l'union de deux deux ensembres représentés par les ABR equilibrés T_1 et T_2.

Une première méthode simple :

> Parcourir l'arbre T_1 pour ajouter ses éléments à T_2 (avec la méthode `add`).

Coût (T_1 et T_2 possédant n éléments) :$O(n \log n)$ (n fois `add` dans un arbre de taille au plus $2n$)

Une autre méthode (pour les ABR)

Union de T_1 et T_2

- Si T_1 est vide, renvoyer T_2.
- Si T_2 est vide, renvoyer T_1.
- Sinon $T_1 = (L_1, x, R_1)$,
  - Calculer les ABR L_2 et R_2, définis comme formés des éléments de T_2 respectivement $<$ et $>$ à x.
  - Renvoyer l'ABR $(L_1 \cup L_2, x, R_1 \cup R_2)$

Remarquer

- Si T_1 et T_2 sont le même ensemble le coût est en $O(n)$.
- Généralisable aux AVL, avec des résultats satisfaisants en pratique.

Programmation de union

Suit directement l'algorithme.

```
static Tree union(Tree t1, Tree t2) {
  if (t1 == null) return t2 ;
  else if (t2 == null) return t1 ;
  else {
    Tree l1 = t1.left, r1 = t1.right ;
    int x = t1.key ;
    Tree l2 = splitLt(x, t2), r2 = splitGt(x, t2) ;
    return new Tree (union(l1, l2), x, union(r1, r2)) ;
  }
}
```

Programmation de `splitLt`

Renvoie l'ensemble des éléments de `t` qui sont $<$ à `x`.

```
static Tree splitLt(int x, Tree t) {
  if (t == null) return null ;
  else {
    if (t.key < x) { // Ici t.left < x
      return new Tree(t.left, t.key, splitLt(x, t.right))
    } else if (t.key > x) {
      // Cas symétrique
    } else { // t.key == x
      return t.left ;
    }
  }
}
```

Coût, (approche heuristique)

Supposons les arbres équilibrés.

Le coût d'un appel à `union` est dominé par celui des appels à `splitLt` et `splitGt`, qui sont proportionels à la hauteur de `t2`.

$$U(n) \sim \log(n) + 2U(n/2)$$

En posant $n = 2^p$,

$$U(2^p) \sim p + 2U(2^{p-1}).$$

Soit à peu près,

$$U(2^p) \sim p + 2(p-1) + 4(p-2) + \cdots 2^p \sim 2^{p+2}.$$