INF421-a
Bases de la programmation et de l'algorithmique

De l'équilibrage des arbres binaires de recherche
(Bloc 8 / 9)

Philippe Baptiste

CNRS LIX, École Polytechnique

14 octobre 2005

THIS

- Dans une classe il est possible de faire référence à l'objet courant en utilisant le mot clef this .
- Ceci est particulièrement utile dans les méthodes dynamiques
- On ne peut jamais changer la valeur de this :

NE JAMAIS ECRIRE

```
if (this == null)
  bla bla bla
```

ou

```
this = this.suivant;
```

Construire un objet

- Sauf pour les variables "primitives", la création passe par `new`

```
class Personne {
  String nom;
  int age;
}
// ...
Personne robert = new Personne();
```

- Un constructeur peut se définir récursivement

Aujourd'hui

Arbre bin. de recherche (rappels)

Arbres de recherche optimaux

Équilibrage d'arbres (AVL)

Rappel : arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire dans lequel pour tout nœud s,

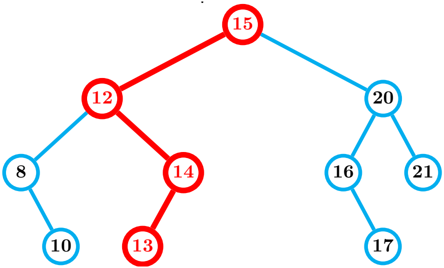
- tous les nœuds du sous-arbre gauche de s sont strictement inférieurs au contenu de s
- tous les nœuds du sous-arbre droit de s sont strictement supérieurs au contenu de s

- Le parcours infixe fournit les contenus des nœuds en ordre croissant
- Recherche, insertion, suppression $= O(h)$

Chercher un élément dans un arbre binaire de recherche

x est-il dans l'arbre t ?

- Si $t = \texttt{null} \rightarrow$ NON
- Si $x = t.val \rightarrow$ OUI
- Si $x < t.val \rightarrow$ Se reposer la question pour x et `t.gauche`
- Si $x > t.val \rightarrow$ Se reposer la question pour x et `t.droite`



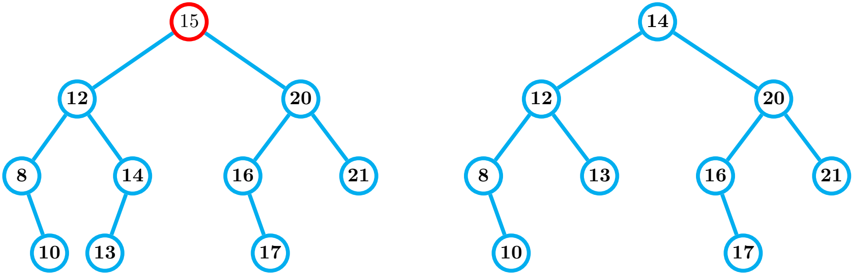
Supprimer un élément d'un arbre binaire de recherche

Nous devons distinguer 3 cas en fonction de l'arité du sommet s à enlever :

- Cas trivial : si le nœud s est une feuille, l'enlever
- Plus ennuyeux : si le nœud s a un fils unique, l'éliminer et remonter le fils
- Très ennuyeux : si le nœud s a deux fils
  - Le nœud s lui-même n'est pas supprimé
  - On échange le contenu de s avec g l'élément maximal du sous-arbre gauche
  - Et on supprime g

  Remarquons
  - g n'a pas de fils droit sinon il serait maximal et donc on retombe sur les cas connus
  - Pour trouver g il suffit d'aller à droite tant qu'on peut !

Supprimer un nœud *s* d'arité 2



De la hauteur d'un arbre binaire de recherche

Nous avons vu (cours 7) :

- La hauteur moyenne d'un arbre binaire de recherche est $O(n \log n)$
- Dans le pire des cas $h = n$

Question fondamentale : Comment garantir une hauteur en $O(n \log n)$ (pire cas) ? $\longrightarrow$ Équilibrage

Question : Comment construire de "bons" arbre binaire de recherche quand les recherches d'éléments ne sont pas "équiprobables" $\longrightarrow$ Arbres optimaux

Aujourd'hui

Arbre bin. de recherche (rappels)

Arbres de recherche optimaux

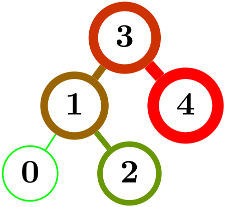
Équilibrage d'arbres (AVL)

Digression : les arbres de recherche optimaux

Problématique

- Soient les clefs $x_1 \leq x_2 \leq \dots \leq x_n$ à ranger dans un arbre binaire de recherche
- Rappel : Le coût de la recherche d'une clef $= O(\text{sa hauteur})$
- On sait que l'on va chercher w_i fois x_i $(1 \leq i \leq n)$
  - w_i est la fréquence de recherche de i
- Quel est le meilleur arbre binaire de recherche ?

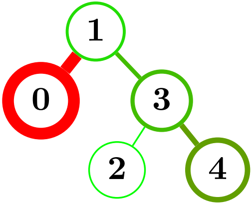
Digression : les arbres de recherche optimaux



i	0	1	2	3	4
w	2	5	4	6	7

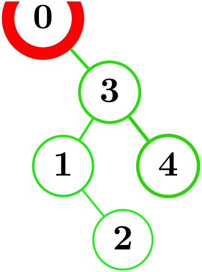
La largeur du trait pour le sommet i est proportionnelle à w (+ code couleur du vert au rouge)

Digression : les arbres de recherche optimaux



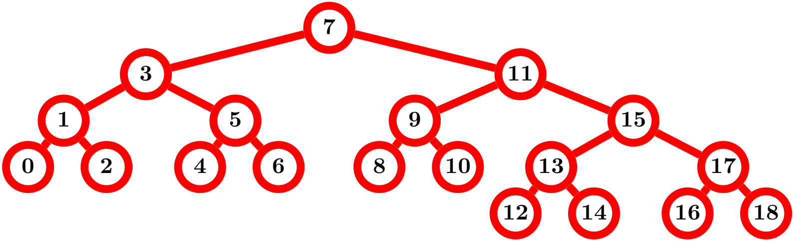
i	0	1	2	3	4
w	12	5	4	6	7

Digression : les arbres de recherche optimaux



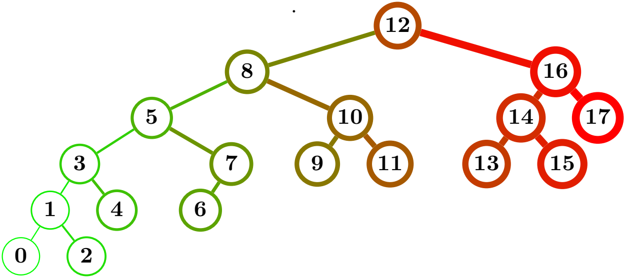
i	0	1	2	3	4
w	25	5	4	6	7

Digression : les arbres de recherche optimaux



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
w	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Digression : les arbres de recherche optimaux



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
w	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Digression : les arbres de recherche optimaux

Diviser pour régner

- Soit W_{jk} le coût associé à la recherche des éléments $j, j+1, ..., k$ dans un arbre de recherche optimal avec les fréquences $w_j, ..., w_k$
  - Rq. On cherche W_{1n}
- Dans un tel arbre, l'un des sommets u est la racine et le coût W_{jk} se décompose alors en
  - le coût associé à la partie gauche $W_{j\ u-1} + \sum_{i=j}^{u-1} w_i$
  - le coût associé à la partie droite $W_{u+1\ k} + \sum_{i=u+1}^{k} w_i$
  - le coût associé à la racine w_u

Digression : les arbres de recherche optimaux

Soit u la racine de l'arbre correspondant à W_{jk}

$$\begin{aligned} W_{jk} &= W_{j\ u-1} + \sum_{i=j}^{u-1} w_i + W_{u+1\ k} + \sum_{i=u+1}^{k} w_i + w_u \\ &= W_{j\ u-1} + W_{u+1\ k} + \sum_{i=j}^{k} w_i \end{aligned}$$

Problème : On ne connaît pas la valeur de u.
Laquelle choisir ? $\rightarrow$ les essayer toutes !

$$W_{jk} = \min_{j \leq u \leq k} \left(W_{j\ u-1} + W_{u+1\ k} + \sum_{i=j}^{k} w_i \right)$$

avec $\forall i, W_{i\ i-1} = 0$

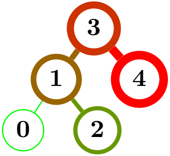
Digression : les arbres de recherche optimaux

De la récurrence à l'algorithme

$$W_{jk} = \min_{j \le u \le k} \left(W_{j\ u-1} + W_{u+1\ k} + \sum_{i=j}^{k} w_i \right)$$

- `W[][]` est un tableau de taille $n * n$ dans lequel on conserve toutes les valeurs de W_{jk}
- Attention, on effectue les calculs en faisant décroître j et en faisant augmenter k (*i.e.*, le tableau est construit à partir de son coin bas-droite)

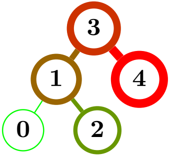
Digression : les arbres de recherche optimaux



i	0	1	2	3	4
w	2	5	4	6	7

$j \backslash k$	0	1	2	3	4
4	0	0	0	0	
3	0	0	0		
2	0	0			
1	0				
0					

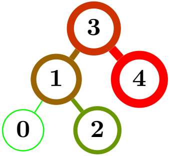
Digression : les arbres de recherche optimaux



i	0	1	2	3	4
w	2	5	4	6	7

$j \backslash k$	0	1	2	3	4
4	0	0	0	0	7
3	0	0	0	6	
2	0	0	4		
1	0	5			
0	2				

Digression : les arbres de recherche optimaux



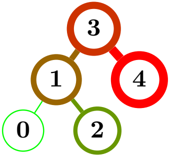
i	0	1	2	3	4
w	2	5	4	6	7

$j \backslash k$	0	1	2	3	4
4	0	0	0	0	7
3	0	0	0	6	19
2	0	0	4	14	28
1	0	5	13	26	42
0	2	9	17	32	48

Digression : les arbres de recherche optimaux

```
// W et root deux tableaux n*n d'entiers
for (int j = n-1; j ≥ 0; j--)
    for (int k = j; k < n; k++) {
        W[j][k] = Integer.MAX_VALUE;
        int sumRoot = 0;
        for (int u = j; u ≤ k; u++)
            sumRoot += w[u]; // ATTENTION notation
        for (int u = j; u ≤ k; u++) {
            int left = 0;
            if (j ≤ u - 1) left = W[j][u-1];
            int right = 0;
            if (u + 1 ≤ k) right = W[u+1][k];
            if (left + right + sumRoot < W[j][k]) {
                W[j][k] = left + right + sumRoot;
                root[j][k] = u;
            }
        }
    }
```

Digression : les arbres de recherche optimaux



i	0	1	2	3	4
w	2	5	4	6	7

```
W[0][0] = 2      root[0][0] = 0
W[0][1] = 9      root[0][1] = 1
W[0][2] = 17     root[0][2] = 1
W[0][3] = 32     root[0][3] = 2
W[0][4] = 48     root[0][4] = 3
W[1][1] = 5      root[1][1] = 1
W[1][2] = 13     root[1][2] = 1
W[1][3] = 26     root[1][3] = 2
W[1][4] = 42     root[1][4] = 3
W[2][2] = 4      root[2][2] = 2
W[2][3] = 14     root[2][3] = 3
W[2][4] = 28     root[2][4] = 3
W[3][3] = 6      root[3][3] = 3
W[3][4] = 19     root[3][4] = 4
W[4][4] = 7      root[4][4] = 4
```

Digression : les arbres de recherche optimaux

- Quelle est la complexité de l'algorithme ?
- A quoi sert le tableau `root` ?

Ce que nous avons vu = Programmation dynamique
[Bellman, 1955] (des détails en IF 431)

Aujourd'hui

Équilibrage

5 ème rappel : les coûts de la recherche, de l'insertion et de la suppression dans un arbre binaire de recherche $= O(h)$

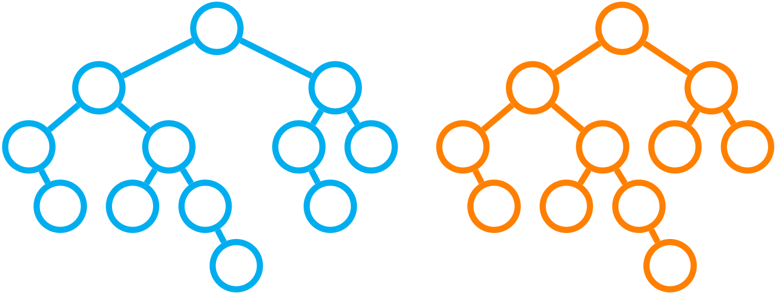
- Le pire cas : pour un arbre à n nœuds, $O(n)$ (arbre = chaîne)
- But du jeu : Éviter que les arbres puissent prendre ces formes
- Comment : opérations peu coûteuses en temps, de transformation d'arbres pour rendre les arbres les plus réguliers possibles
- De nombreuses familles : AVL (Adelson-Velskii et Landis, les arbres 2-3, les arbres rouge et noir, *etc.*.

Équilibrage : AVL

Un arbre binaire est un arbre AVL ssi, pour tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1

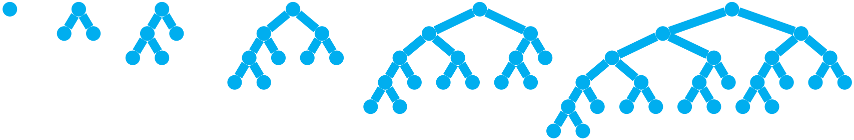
- Une feuille est un arbre de hauteur 0
- l'arbre vide a la hauteur -1
- $\longrightarrow$ L'arbre vide, et l'arbre réduit à une feuille, sont des arbres AVL

Équilibrage : Quels sont les AVLs



Équilibrage : AVL de Fibonacci

Arbres de Fibonacci = arbres binaires A_n avec sous-arbres gauche A_{n-1} et droit A_{n-2}



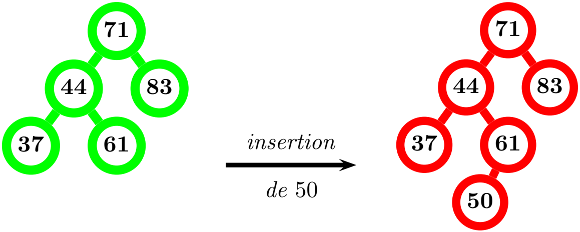
Équilibrage : AVL

Pour un AVL, $\log_2(1+n) \leqslant 1+h \leqslant \alpha \log_2(2+n)$ avec
$$\alpha = \frac{1}{\log_2(\frac{1+\sqrt{5}}{2})} \leqslant 1.44$$

- $n \leqslant 2^{h+1} - 1$, donc $\log_2(1+n) \leqslant 1+h$.
- Soit $\mathcal{N}_h$ nombre min de nœuds d'un AVL de hauteur h
- $\mathcal{N}_h = 1 + \mathcal{N}_{h-1} + \mathcal{N}_{h-2}$
- Avec $\mathcal{F}_h = 1 + \mathcal{N}_h$, $\mathcal{F}_h = \mathcal{F}_{h-1} + \mathcal{F}_{h-2}$ $(\mathcal{F}_0 = 2,\ \mathcal{F}_1 = 3\ )$
  Suite de Fibonacci
- $\mathcal{F}_h = \frac{1}{\sqrt{5}} \cdot (\Phi^{h+3} - \Phi^{-(h+3)})$ avec $\Phi = \frac{1+\sqrt{5}}{2}$
- $1 + n \geq \mathcal{F}_h > \frac{1}{\sqrt{5}} \cdot (\Phi^{h+3} - \Phi^{-(h+3)})$
- $h + 3 < \log_\Phi(\sqrt{5}(2+n)) < \frac{\log_2(2+n)}{\log_2 \Phi} + 2$

Équilibrage : AVL & Rotations

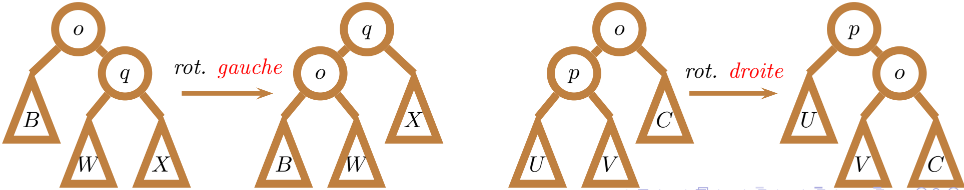
Question : Comment conserver un arbre AVL (après des insertions et des suppressions) ?



L'arbre rouge n'est plus un AVL. Un outil : les rotations

Équilibrage : AVL & Rotations

- $A = (B, o, C)$ un arbre binaire
- Si $C = (W, q, X)$, la rotation gauche est l'opération $(B, o, (W, q, X)) \rightarrow ((B, o, W), q, X)$
- Si $B = (U, p, V)$, la rotation droite est l'opération $((U, p, V), o, C) \rightarrow (U, p, (V, o, C))$
- Attention ! Les rotations gauche (droite) ne sont donc définies que pour les arbres binaires non vides dont le sous-arbre gauche (resp. droit) n'est pas vide.



Équilibrage : AVL & Rotations

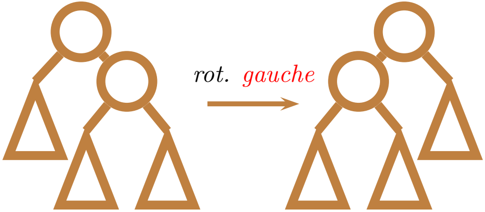
- Les rotations $\longrightarrow$ en temps constant
- Les rotations préservent l'ordre infixe !
- Une suite de rotations gauche ou droite appliqués à un arbre binaire de recherche $\longrightarrow$ encore un arbre binaire de recherche

Rappel codage d'un arbre binaire de recherche

```
class Arbre {
    int val;
    int hauteur; // Utilité ???
    Arbre gauche, droite;
    Arbre pere;
    Arbre (Arbre gauche, int val, Arbre droite) {
        this.gauche = gauche; this.val = val; this.droite = droite;
        hauteur = 1 + Math.max(hauteur(gauche), hauteur(droite));
    }
    static int hauteur(Arbre a) {
        return (a == null) ? -1 : a.hauteur;
    }
    static void recalculHauteur(Arbre a) {
        a.hauteur = 1 + Math.max(hauteur(a.gauche), hauteur(a.droite));
    }
}
```

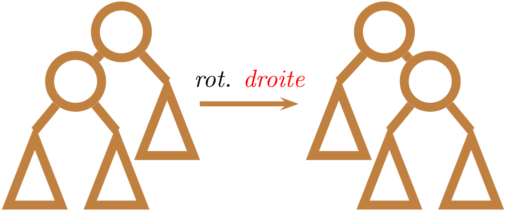
Équilibrage : AVL & Rotation gauche

```
static Arbre rotationGauche(Arbre a) {
  return new Arbre(new Arbre(a.gauche, a.val, a.droite.gauche),
                         a.droite.val,
                         a.droite.droite);
}
```

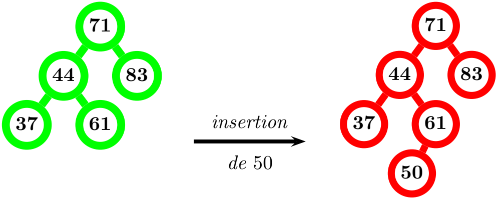


Équilibrage : AVL & Rotation droite

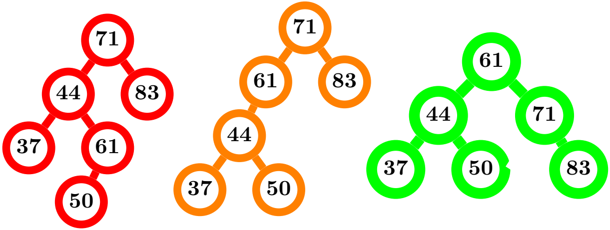
```
static Arbre rotationDroite(Arbre a) {
  return new Arbre (a.gauche.gauche,
                    a.gauche.val,
                    new Arbre(a.gauche.droite, a.val, a.droite));
}
```



Équilibrage : AVL & Rotations



Effectuons une double rotation droite = une rotation gauche du sous-arbre gauche suivie d'un rotation droite



Équilibrage : AVL & Rotations

- On part d'un arbre binaire de recherche `a` déséquilibré à la racine
- On a donc `|hauteur(a.gauche) - hauteur(a.droite)| = 2`
- On suppose sans perte de généralité que `hauteur(a.gauche) = hauteur(a.droite) + 2` (l'autre cas est obtenu par symétrie)
- `|hauteur(a.gauche.gauche) < hauteur(a.gauche.droite)|` $\longrightarrow$ rotation gauche de `a.gauche` puis rotation droite de `a`
- `|hauteur(a.gauche.gauche)` $\geq$ `hauteur(a.gauche.droite)|` $\longrightarrow$ rotation droite de `a`

Équilibrage : AVL & Rotations

- Pour rééquilibrer un arbre AVL après une insertion, une seule rotation ou double rotation suffit.
- Pour rééquilibrer un arbre AVL après une suppression, il faut jusqu'à h rotations ou double rotations (h est la hauteur de l'arbre).

Équilibrage : AVL & Rotations

```
static Arbre equilibrer(Arbre  a) {
    a.hauteur = 1 + Math.max(hauteur(a.gauche), hauteur(a.droite));
    if (hauteur(a.gauche) - hauteur(a.droite) == 2) {
        if (hauteur(a.gauche.gauche) < hauteur(a.gauche.droite))
            a.gauche = rotationGauche(a.gauche);
        return rotationDroite(a);
    }
    if (hauteur(a.gauche) - hauteur(a.droite) == -2) {
        if (hauteur(a.droite.droite) < hauteur(a.droite.gauche))
            a.droite = rotationDroite(a.droite);
        return rotationGauche(a);
    }
    return a;
}
```

Équilibrage : AVL & Rotations

```
static Arbre inserer(int x, Arbre a) {
  if (a == null)
    return new Arbre(null, x, null);
  if (x < a.val)
    a.gauche = inserer(x, a.gauche);
  else if (x > a.val)
    a.droite = inserer(x, a.droite);
  return equilibrer(a);
}
```

Équilibrage : AVL & Rotations

```
static Arbre rechercherEtSupprimer(int x, Arbre a) {
    if (a == null)  return a;
    if (x == a.val) return supprimerRacine(a);
    if (x < a.val) a.gauche = rechercherEtSupprimer(x, a.gauche);
    else a.droite = rechercherEtSupprimer(x, a.droite);
    return equilibrer(a); }
static Arbre supprimerRacine(Arbre a) {
    if (a.gauche == null) return equilibrer(a.droite);
    if (a.droite == null) return equilibrer(a.gauche);
    Arbre f = dernierDescendant(a.gauche);
    a.val = f.val;
    a.gauche = rechercherEtSupprimer(f.val, a.gauche);
    return equilibrer(a); }
static Arbre dernierDescendant(Arbre a) {
    if (a.droite == null) return a;
    return dernierDescendant(a.droite); }
```

Équilibrage : AVL & Rotations