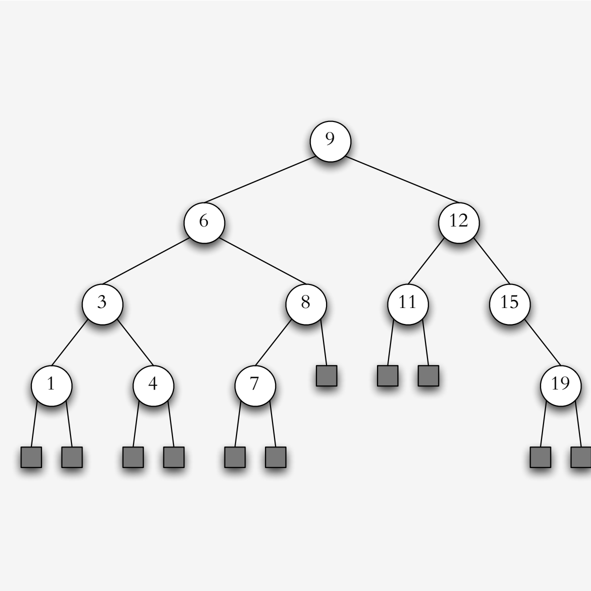
# 12 Arbres binaires de recherche

## 12.1 Arbre binaire de recherche (ABR)

W(fr)




Chaque nœud interne possède une clé : les clés sont comparables.

**Définition 12.1.** *Dans un **arbre binaire de recherche** (ABR), les nœuds internes possèdent des clés comparables, en respectant un ordre parmi les enfants gauches et droits : le parcours infixe énumère les nœuds internes dans l'ordre croissant de clés.*

On spécifie l'ABR par sa racine root. Accès aux nœuds :

- ⋆ $x$.left et $x$.right pour les enfants de $x$ (null si l'enfant est un nœud externe)
- ⋆ $x$.parent pour le parent de $x$ (null à la racine)
- ⋆ $x$.key pour la clé d'un nœud interne $x$ (en général, un entier dans nos discussions)

Normalement, on indique les nœuds externes par null (donc, p.e., $x$.parent n'est pas valide quand $x$ est un nœud externe).

**Théorème 12.1** (Ordre d'ABR)**.** *Soit $x$ un nœud interne dans un arbre binaire de recherche. Si $y \neq x$ est un nœud interne dans le sous-arbre gauche de $x$, alors $y$.key $<$ $x$.key. Si $y \neq x$ est un nœud interne dans le sous-arbre droit de $x$, alors $y$.key $>$ $x$.key.*

## 12.2 Opérations sur l'ABR

Opérations : la structure ABR permet l'implantation efficace de **recherche** d'une valeur particulière (opération principale du TAD dictionnaire), **insertion** et **suppression** d'éléments (opérations optionnelles pour dictionnaires dynamiques), et même recherche de **min** ou **max** (permettant l'implantation de file à priorité), et beaucoup d'autres.

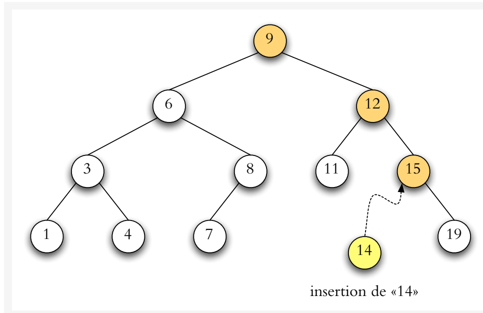
**Recherche.** SEARCH(root, $v$) retourne (a) soit un nœud dont la clé est égale à $v$, (b) soit null s'il n'y a pas de nœud avec clé $v$. Théorème 12.1 mène àux algorithmes suivants basés sur la même logique.

**Solution récursive**

```
   SEARCH(x, v) // trouve clé v dans le sous-arbre de x
S1 if x = null ou v = x.key then return x
S2 if v < x.key
S3 then return SEARCH(x.left, v)
S4 else return SEARCH(x.right, v)
```

**Solution itérative**

```
   SEARCH(x, v) // trouve clé v dans le sous-arbre de x
S1 while x ≠ null et v ≠ x.key do
S2     if v < x.key
S3     then x ← x.left
S4     else x ← x.right
S5 return x
```



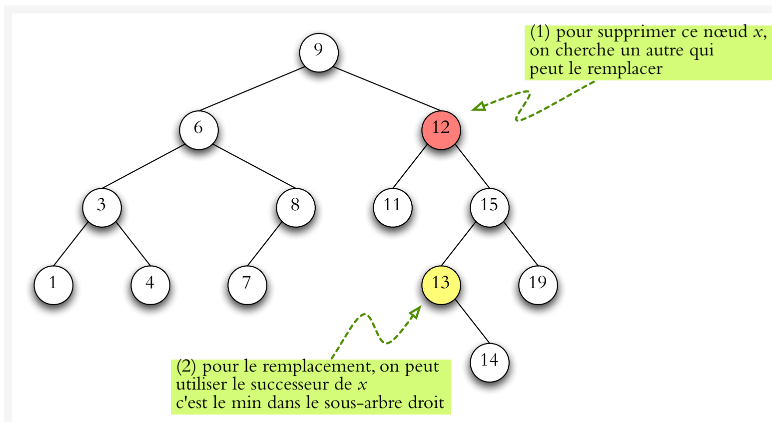
insertion de «14»

**Insertion.** Pour insérer une clé $v$ il suffit d'attacher son nœud interne en remplaçant un seul nœud externe, identifié à l'échec de SEARCH($v$).

```
    INSERT(v)                                              // insère la clé v dans l'arbre
I1  x ← root ; y ← nouveau nœud ; y.key ← v
I2  if x = null then root ← y ; return
I3  loop                                   // boucler : conditions d'arrêt testées dans le corps
I4      if v = x.key then erreur                      // on ne permet pas de clés dupliquées
I5      if v < x.key
I6      then if x.left = null
I7          then x.left ← y; y.parent ← x ; return     // attacher y comme enfant gauche de x
I8          else x ← x.left
I9      else if x.right = null
I10         then x.right ← y; y.parent ← x ; return    // attacher y comme enfant droit de x
I11         else x ← x.right
```

**Suppression du nœud $x$.**

0. triviale si $x$ n'a **pas d'enfants** internes : faire $\boxed{x\text{.parent.left} \leftarrow \text{null}}$ si $x$ est l'enfant gauche de son parent, ou $\boxed{x\text{.parent.right} \leftarrow \text{null}}$ si $x$ est l'enfant droit
1. facile si $x$ a seulement **1 enfant** : faire $\boxed{x\text{.parent.left} \leftarrow x\text{.right}\,;\, x\text{.right.parent} \leftarrow x\text{.parent}}$ si $x$ a un enfant droit et $x$ est l'enfant gauche de son parent (il y a 4 cas en total dépendant de la position de $x$ et celle de son enfant)
2. un peu plus compliqué si $x$ a **2 enfants** : on trouve d'abord remplacement (successeur ou prédecesseur dans le parcours infixe)




**Lemme 12.2.** *Le nœud avec la clé minimale dans le sous-arbre droit de $x$ n'a pas d'enfant gauche.*

$\Rightarrow$ il est facile d'enlever le successeur (d'un nœud à deux enfants)...

```
    DELETE(z)                                                // supprime le nœud z
D1  if z.left = null ou z.right = null alors y ← z                    // cas 1. ou 2.
D2  else y ← MIN(z.right)                                                  // cas 3.
                   // c'est le nœud y qu'on enlève physiquement : un de ses enfants est externe
D3  if y.left ≠ null then x ← y.left else x ← y.right   // le nœud x remplace y à son parent
D4  if x ≠ null then x.parent ← y.parent
D5  if y.parent = null then root ← x                                 // y était la racine
D6  else                                                                // on remplace
D7      if y = y.parent.left then y.parent.left ← x                 // y est enfant gauche
D8      else y.parent.right ← x                                      // y est enfant droit
D9  if y ≠ z then remplacer nœud z par y dans l'arbre      // copier contenu : z.key ← y.key
```

**Exercice 12.1.** ▶ Écrire l'algorithme SUCCESSEUR$(x)$ qui trouve le successeur du nœud $x$ dans le parcours infixe. ▶ Montrer que le temps de calcul de SUCCESSEUR$(x)$ est $\Theta(h)$ dans le pire cas où $h$ est la hauteur de l'arbre. ▶ Montrer que le temps de calcul est $\Theta(1)$ en moyenne (quand $x$ est un nœud aléatoire dans l'arbre). **Indice** : considérer un parcours infixe en utilisant l'itération $x \leftarrow$ SUCCESSEUR$(x)$.

**Sélection** Théorème 12.1 suggère immédiatement la démarche pour trouver le minimum ou le maximum.

```
MIN(r) // nœud à clé minimale dans le sous-arbre de r
1 x ← r; y ← null
2 while x ≠ null do y ← x; x ← x.gauche
3 return y
```

```
MAX(r) // nœud à clé maximale dans le sous-arbre de r
1 x ← r; y ← null
2 while x ≠ null do y ← x; x ← x.droit
3 return y
```

**Exercice 12.2.** Pour implanter l'opération select$(i)$ (qui retourne le $i$-ème élément selon l'ordre de clés), on a besoin de stocker la taille de chaque sous-arbre à sa racine par la variable $x$.size : $x$.size est le nombre d'internes dans le sous-arbre enraciné au nœud interne $x$. ▶ Donner une définition récursive pour $x$.size. ▶ Donner un algorithme récursif pour initialiser $x$.size partout dans un ABR donné. ▶ Montrer comment mettre à jour $x$.size lors d'une insertion ou suppression.

## 12.3 Temps de calcul des opérations

Les opérations prennent $O(h)$ au pire dans les implantations de §12.2.

**Hauteurs extrêmes.** Par Théorème 6.2, la hauteur $h$ d'un arbre binaire avec $n$ nœuds internes est bornée comme $\lceil \lg(n+1) \rceil \leq h \leq n$ avec égalités dans le cas d'un arbre binaire complet à la borne inférieure, et l'arbre résultant de l'insértion succéssive d'éléments $1, 2, 3, 4, \ldots, n$.

**Arbre «moyen».**

**Définition 12.2.** *Un* ***ABR aléatoire*** *se construit en insérant les valeurs $1, 2, \ldots, n$ selon une permutation aléatoire, choisie à l'uniforme.*

REMARQUE. Notez que cette notion est tout à fait différente de celle d'une structure choisie à l'uniforme : les 6 permutations des clés $\{1, 2, 3, \}$ mènent à seulement 5 arbres possibles.

**Théorème 12.3** (Bruce Reed & Michael Drmota)**.** *La hauteur d'un ABR aléatoire sur $n$ clés est $\mathbb{E}h = \alpha \lg n - \beta \lg\lg n + O(1)$ en espérance où $\alpha \approx 2.99$ et $\beta = \frac{3}{2\lg(\alpha/2)} \approx 1.35$. La variance de la hauteur aléatoire est $O(1)$.*

Le théorème 12.3 applique au pire cas des opérations (nœud externe le plus distant) d'un ABR aléatoire. Il montre que les opérations prennent $O(\log n)$ en moyenne. La preuve du thèorème est trop compliquée pour les buts de ce cours.

**Profondeur moyenne.** Le temps moyen de la recherche fructueuse (ou d'insertion) correspond au niveau moyen de nœuds internes parce que c'est où la recherche se termine. On va démontrer que la profondeur moyenne est $O(\log n)$. La preuve exploîte la correspondance à une exécution du tri rapide : le pivot du sous-tableau correspond à la racine du sous-arbre.

**Définition 12.3.** *Soit $x$ un nœud interne d'un ABR, et soit $T_x$ le sous-arbre enraciné à $x$. Pour tout nœud interne $y \in T_x$, la distance $d(x, y)$ est définie comme la longueur du chemin de $x$ à $y$. On définit $d(x) = \sum_{y \in T_x} d(x, y)$ comme la somme des profondeurs des nœuds internes dans le sous-arbre $T_x$ enraciné à $x$.*

Avec cette définition, $d(\mathsf{root}, y)$ est la profondeur (ou niveau) du nœud $y$ et $\frac{d(\mathsf{racine})}{n}$ est la moyenne des profondeurs dans l'arbre.

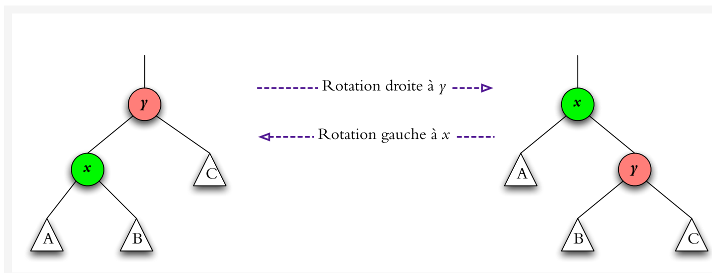
**Théorème 12.4.** *Soit $D(n) = \mathbb{E}d(\mathsf{root})$ l'espérance de la somme des profondeurs dans un arbre aléatoire avec $n$ clés comme en Théorème 12.3. Alors, $D(n)/n = O(\log n)$*

*Démonstration.* Preuve identique à celle du Théorème 11.2 (temps de calcul moyen du tri rapide). ■

## 12.4 Rotations

W(en)

Vu que la performance d'un ABR est déterminée par sa hauteur, des implantations efficaces visent à maintenir un arbre équilibré.




La technique principale dans l'établissement de l'équilibre est la **rotation**. Les rotations (gauche ou droite) — préservent la propriété des arbres de recherche et prennent seulement $O(1)$.

```
ROTR(y)                                // rotation droite à y
R1  x ← y.left ; B ← x.right ; p ← y.parent
R2  x.right ← y ; y.parent ← x
R3  y.left ← B ; if B ≠ null then B.parent ←
    y
R4  x.parent ← p
R5  if p ≠ null then
R6      if y = p.left then p.left ← x
R7      else p.right ← x
```

```
ROTL(x)                                // rotation gauche à x
L1  y ← x.right ; B ← y.left ; p ← x.parent
L2  y.left ← x ; x.parent ← y
L3  x.right ← B ; if B ≠ null then B.parent ←
    x
L4  y.parent ← p
L5  if p ≠ null then
L6      if x = p.left then p.left ← y
L7      else p.right ← y
```

Il existe de nombreuses implantations efficaces qui utilisent des rotations :

**(Randomisation)** En maintenant une variable $x.\mathsf{size}$ à chaque nœeud interne $x$ qui stocke le nombre de nœuds internes dans le sous-arbre de $x$, on peut simuler l'ABR aléatoire de Déf. 12.2, indépendamment de l'ordre des insertions. L'idée est de décider lors de la descente dans INSERT$(v)$ au hasard que $v$ devienne la racine du sous-arbre[1] Pour un tel arbre, les opérations prennent $O(\log n)$ *en moyenne* (individuellement). (Mais un temps de calcul de $\Theta(n)$ arrive avec une probabilité positive.)

**(Amortisation)** Dans un arbre *splay*, on remonte un nœud jusqu'à la racine par des rotations selon une logique analogue à celle de compression de chemin par réduction à moitié d'Union-find (*path halving*, v. §6.5). Dans un tel arbre, une séquence quelconque de $m$ opérations s'exécute en $O(m \log n)$ au pire cas[2], donc les opérations prennent $O(\log n)$ de *temps amorti*.

**(Optimisation)** La hauteur est $O(\log n)$ *au pire cas* (individuellement) pour beaucoup de genres d'arbres de recherche équilibrés : arbre AVL, arbre rouge-noir, arbre 2-3-4. Il faut toujours stocker au moins une variable additionnelle à chaque nœud interne pour guider la démarche de rotations.

[1] Plus précisément, au nœud interne $x$, on performe une «insertion à la racine» avec probabilité $1/(x.\mathsf{size} + 1)$ : après avoir inseré $v$ dans le sous-arbre de $x$ selon la démarche usuelle, on remonte à $x$, en performant des rotations jusqu'à $x$.

[2] ici, $n$ est la taille maximal de l'arbre pendant la séquence