# 15 Table de symboles et arbres binaires de recherche

## 15.1 Table de symboles

Type abstrait **table de symboles** (*symbol table*) ou **dictionnaire** : ensemble d'objets avec clés. Typiquement (mais pas toujours !) les clés sont comparables (abstraction : nombres naturels).

Opération principale :

⋆ search$(k)$ : recherche d'un élément à clé $k$ ← opération fondamentale — peut être fructueuse ou infructeuse.

Opérations souvent supportées :

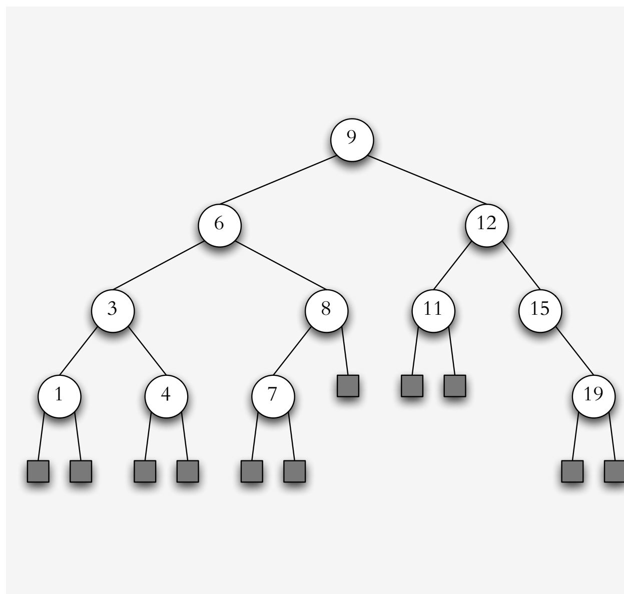
⋆ insert$(x)$ : insertion de l'élément $x$ (clé+info)
⋆ delete$(k)$ : supprimer élément avec clé $k$
⋆ select$(i)$ : sélection de l'$i$-ème élément (selon l'ordre des clés)

### Implantations élémentaires

⋆ liste chaînée ou tableau non-trié : recherche séquentielle — temps de $\Theta(n)$ au pire (même en moyenne), mais insertion/suppression en $\Theta(1)$ [si non-trié]
⋆ tableau trié : recherche binaire — temps de $\Theta(\log n)$ au pire, mais insertion/suppression en $\Theta(n)$ au pire cas

## 15.2 Arbre binaire de recherche (ABR)

W(fr)




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

**Définition 15.1.** *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 15.1** (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.*

## 15.3 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.

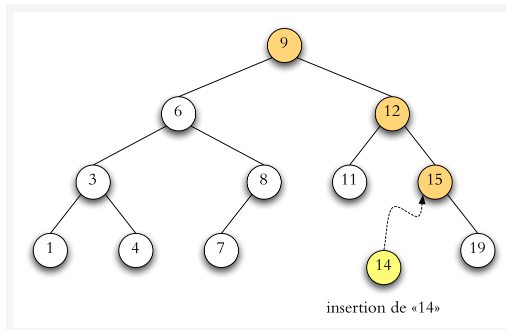
**Recherche.** SEARCH(root, $v$) retourne (a) soit un nœud dont la clé est égale à $v$, (b) soit null s'il n'y a pas de nœud avec clé $v$. Théorème 15.1 mène aux algorithmes suivants basés sur la même logique.

**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** $x \neq$ null et $v \neq x$.key **do**
S2     **if** $v < x$.key
S3     **then** $x \leftarrow x$.left
S4     **else** $x \leftarrow x$.right
S5 **return** $x$



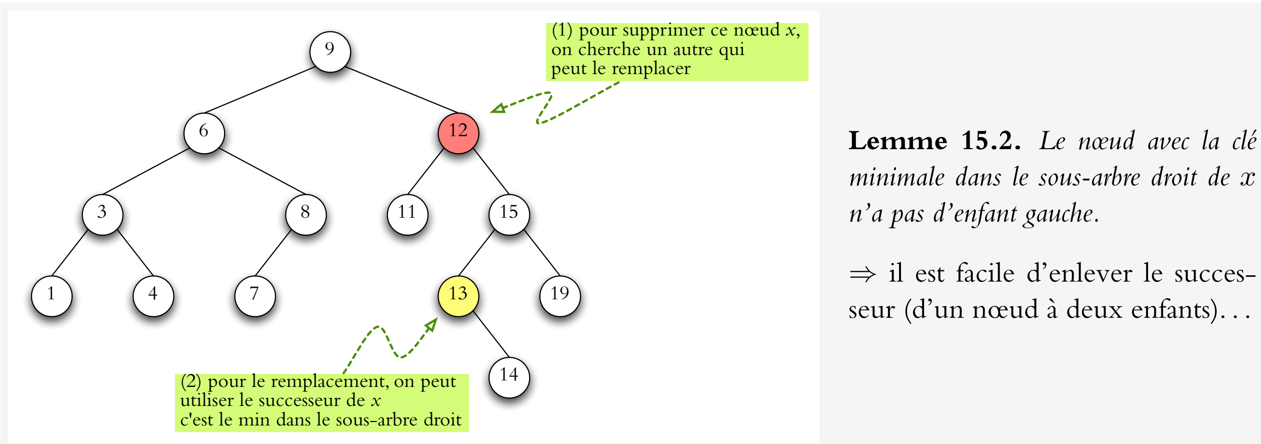
insertion de «14»

**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 $x \leftarrow$ root ; $y \leftarrow$ nouveau nœud ; $y$.key $\leftarrow v$
I2 **if** $x$ = null **then** root $\leftarrow y$ ; **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 $\leftarrow y$; $y$.parent $\leftarrow x$ ; **return** // *attacher $y$ comme enfant gauche de $x$*
I8         **else** $x \leftarrow x$.left
I9     **else if** $x$.right = null
I10         **then** $x$.right $\leftarrow y$; $y$.parent $\leftarrow x$ ; **return** // *attacher $y$ comme enfant droit de $x$*
I11         **else** $x \leftarrow x$.right

**Suppression du nœud $x$.**

0. triviale si $x$ n'a **pas d'enfants** internes : faire $x$.parent.left $\leftarrow$ null si $x$ est l'enfant gauche de son parent, ou $x$.parent.right $\leftarrow$ null si $x$ est l'enfant droit
1. facile si $x$ a seulement **1 enfant** : faire $x$.parent.left $\leftarrow$ $x$.right ; $x$.right.parent $\leftarrow$ $x$.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 15.2.** *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
```

**Sélection** Théorème 15.1 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
```

## 15.4 Temps de calcul des opérations

Les opérations prennent $O(h)$ au pire dans les implantations de §15.3.

**Hauteurs extrêmes.** Par Théorème 6.2, la hauteur $h$ d'un arbre binaire avec $n$ nœuds internes est bornée comme $\lceil \lg(n+1) \rceil \le h \le n$ 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 15.2.** *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 15.3** (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 15.3 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 15.3.** *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 15.4.** *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 15.3. Alors, $D(n)/n = O(\log n)$*

*Démonstration.* Preuve identique à celle du Théorème 14.1 (temps de calcul moyen du tri rapide). ∎