

13 Table de symboles et arbres binaires de recherche

13.1 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).

Opérations principales :

★ $\text{search}(k)$: recherche d'un élément à clé k ← opération fondamentale — peut être fructueuse ou infructueuse.

★ $\text{insert}(x)$: insertion de l'élément x (clé+info)

Opérations parfois supportées :

★ $\text{delete}(k)$: supprimer élément avec clé k

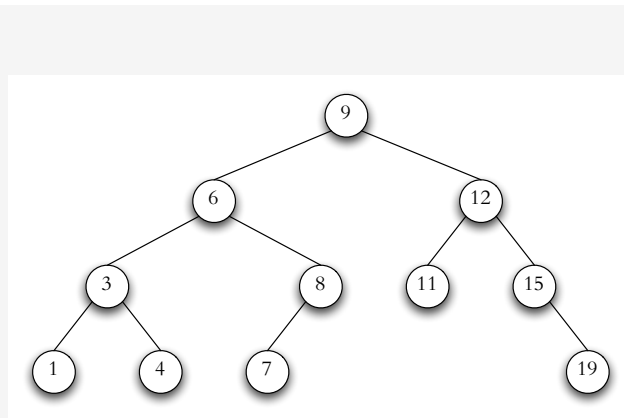
★ $\text{select}(i)$: sélection de l' i -ème élément (selon l'ordre des clés)

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

13.2 Arbre binaire de recherche

$W_{(fr)}$



Dans un arbre binaire de recherche, chaque nœud interne possède une clé.

Définition 13.1. Un arbre binaire est un **arbre binaire de recherche** ssi les nœuds internes sont énumérés lors d'un parcours infixe en ordre croissant de clés.

Accès aux nœuds :

- ★ $x.\text{gauche}$ et $x.\text{droit}$ pour les enfants de x (null si l'enfant est un nœud externe)
- ★ $x.\text{parent}$ pour le parent de x (null à la racine)
- ★ $x.\text{cle}$ pour la clé d'un nœud interne x (en général, un entier dans nos discussions)

Théorème 13.1. Soit x un nœud interne dans un arbre binaire de recherche. Si y est un nœud interne dans le sous-arbre gauche de x , alors $y.\text{cle} \leq x.\text{cle}$. Si y est un nœud interne dans le sous-arbre droit de x , alors $y.\text{cle} \geq x.\text{cle}$.

Les nœuds externes sont null. À l'aide d'un arbre de recherche, on peut implanter une table de symboles d'une manière très efficace.

Opérations : **recherche** d'une valeur particulière, **insertion** ou **suppression** d'une valeur, recherche de **min** ou **max**, et des autres.

Min et max

```
MIN(r) // nœud à clé minimale dans le sous-arbre de r
1   $x \leftarrow r; y \leftarrow \text{null}$ 
2  tandis que  $x \neq \text{null}$  faire
3       $y \leftarrow x; x \leftarrow x.\text{gauche}$ 
4  retourner  $y$ 
```

```
MAX(r) // nœud à clé maximale dans le sous-arbre de r
1   $x \leftarrow r; y \leftarrow \text{null}$ 
2  tandis que  $x \neq \text{null}$  faire
3       $y \leftarrow x; x \leftarrow x.\text{droit}$ 
4  retourner  $y$ 
```

Recherche. SEARCH(racine, 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 .

Solution récursive

```
SEARCH( $x, v$ ) // trouve clé  $v$  dans le sous-arbre de  $x$ 
S1 si  $x = \text{null}$  ou  $v = x.\text{cle}$  alors retourner  $x$ 
S2 si  $v < x.\text{cle}$ 
S3 alors retourner SEARCH( $x.\text{gauche}, v$ )
S4 sinon retourner SEARCH( $x.\text{droit}, v$ )
```

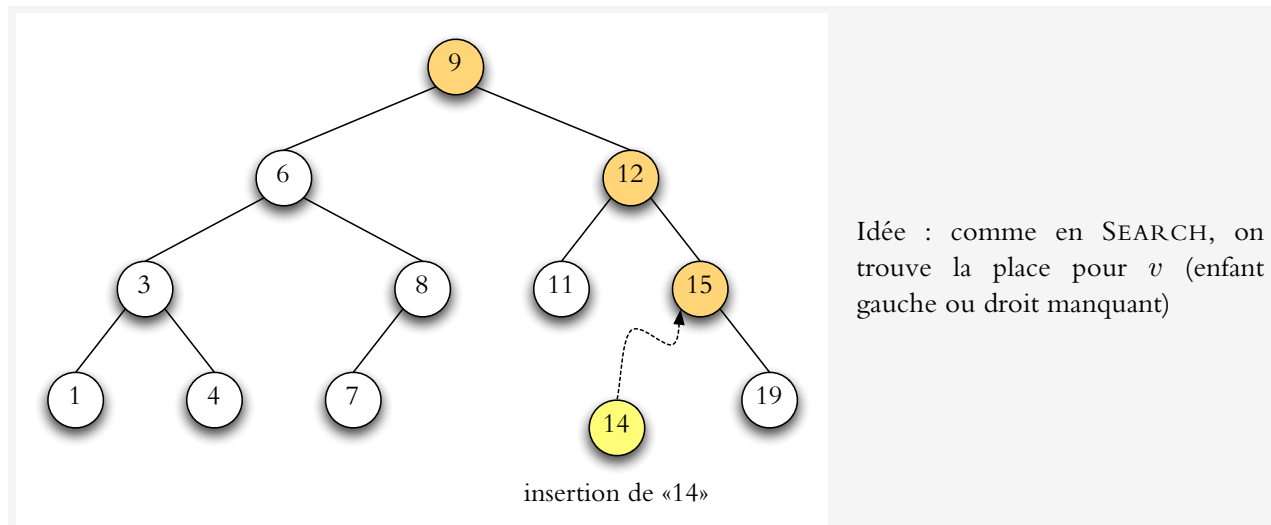
Solution itérative

```
SEARCH( $x, v$ ) // trouve clé  $v$  dans le sous-arbre de  $x$ 
S1 tandis que  $x \neq \text{null}$  et  $v \neq x.\text{cle}$  faire
S2     si  $v < x.\text{cle}$ 
S3     alors  $x \leftarrow x.\text{gauche}$ 
S4     sinon  $x \leftarrow x.\text{droit}$ 
S5 retourner  $x$ 
```

Dans un arbre binaire de recherche de hauteur h :
MIN() prend $O(h)$
MAX() prend $O(h)$
SEARCH(racine, v) prend $O(h)$

13.3 Insertion et suppression

Insertion. On veut insérer une clé v : créer un nœud et l'ajouter à l'arbre.



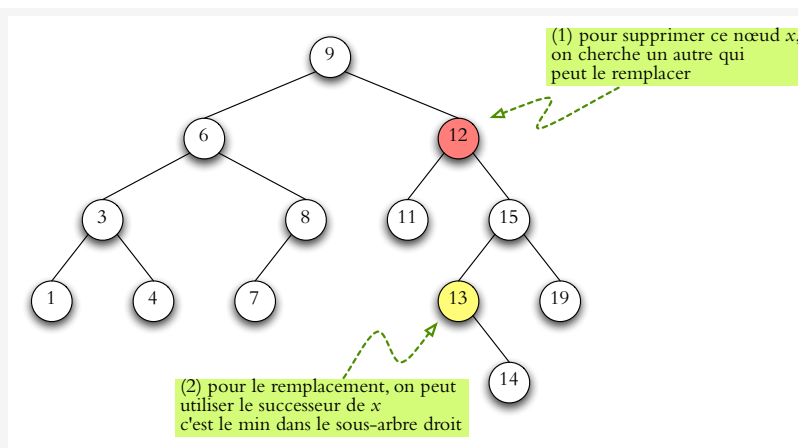
```

INSERT(v) // insère la clé v dans l'arbre
I1 x ← racine
I2 si x = null alors créer nouveau nœud y avec clé v comme racine et retourner y
I3 boucler // conditions d'arrêt testées dans le corps
I4   si v = x.cle alors erreur // on ne permet pas les valeurs dupliquées
I5   si v < x.cle
I6     alors si x.gauche = null
I7       alors attacher nouveau nœud y avec clé v comme enfant gauche de x et retourner y
I8       sinon x ← x.gauche
I9     sinon si x.droit = null
I10      alors attacher nouveau nœud y avec clé v comme enfant droit de x et retourner y
I11      sinon x ← x.droit

```

Suppression du nœud *x*

1. triviale si *x* est n'a pas d'enfants non-null : *x*.parent.gauche ← null si *x* est l'enfant gauche de son parent, ou *x*.parent.droit ← null si *x* est l'enfant droit
2. facile si *x* a seulement un enfant : *x*.parent.gauche ← *x*.droit 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)
3. un peu plus compliqué si *x* a deux enfants : on trouve un remplacement (successeur ou prédécesseur dans le parcours infixe)



Lemme 13.2. Le nœud avec la clé minimale dans le sous-arbre droit de *x* n'a pas d'enfant gauche.

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

```

DELETE(z) // supprime le nœud z
D1 si z.gauche = null ou z.droit = null alors y ← z // cas 1. ou 2.
D2 sinon y ← MIN(z.droit) // cas 3.
   // un des deux enfants de y est null
D3 si y.gauche ≠ null alors x ← y.gauche sinon x ← y.droit
   // on enlève le nœud y de l'arbre
D4 si x ≠ null alors x.parent ← y.parent
D5 si y.parent = null alors racine ← x // y était la racine
D6 sinon // y a un parent
D7   si y = y.parent.gauche alors y.parent.gauche ← x // y est enfant gauche
D8   sinon y.parent.droit ← x // y est enfant droit
D9 si y ≠ z alors remplacer nœud z par y dans l'arbre

```

Exercice 13.1. Écrire l'algorithme $\text{SUCCESEUR}(x)$ qui trouve le successeur du nœud x dans le parcours infixe. Montrer que le temps de calcul de $\text{SUCCESEUR}(x)$ 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 $x \leftarrow \text{SUCCESEUR}(x)$.

Dans un arbre binaire de recherche de hauteur h :
 $\text{INSERT}(v)$, ainsi que $\text{DELETE}(z)$ prennent $O(h)$.

13.4 Hauteur

Toutes les opérations prennent $O(h)$ dans un arbre de hauteur h .

→ **Meilleur cas** Arbre binaire complet : $2^h - 1$ nœuds internes dans un arbre de hauteur h , donc hauteur $h = \lceil \lg(n+1) \rceil$ pour n clés est possible : $h = \Omega(\log n)$

→ **Pire cas** Insertions consécutives de $1, 2, 3, 4, \dots, n$ mènent à un arbre avec $h = n$. Donc $h = \Theta(n)$ au pire cas.

⇒ Est-ce qu'il est possible d'assurer que $h = O(\log n)$ en général ?

- ★ Réponse 1 [randomisation] : la hauteur est de $O(\log n)$ en moyenne (permutations aléatoires de $\{1, 2, \dots, n\}$)
- ★ Réponse 2 [amortisation] : exécution des opérations en temps amorti $O(\log n)$ pour des arbres *splay*
- ★ Réponse 3 [optimisation] : la hauteur est de $O(\log n)$ en pire cas pour beaucoup de genres d'arbres de recherche équilibrés : arbre AVL, arbre rouge-noir, arbre 2-3-4. Insertion/suppression plus compliquées, mais toujours $O(\log n)$.

13.5 Performance moyenne

Théorème 13.3 (Bruce Reed & Michael Drmota). La hauteur d'un arbre de recherche construit en insérant les valeurs $1, 2, \dots, n$ selon une permutation aléatoire est $\alpha \lg n - \beta \lg \lg n + O(1)$ en moyenne où $\alpha \approx 2.99$ et $\beta = \frac{3}{2 \lg(\alpha/2)} \approx 1.35$. La variance de la hauteur aléatoire est $O(1)$.

La preuve du théorème 13.3 est trop compliquée pour les buts de ce cours. On peut analyser le cas moyen en regardant la **profondeur moyenne** (ou niveau moyen) plutôt. Le coût de chaque opération dépend de la profondeur du nœud accédé dans l'arbre. On va démontrer que la profondeur moyenne est $O(\log n)$. Donc le temps moyen d'une recherche fructueuse est en $O(\log n)$. La preuve exploite la correspondance à une exécution du tri rapide : le pivot du sous-tableau correspond à la racine du sous-arbre.

Définition 13.2. 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.

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

Théorème 13.4. Soit $D(n) = \mathbb{E}d(\text{racine})$ l'espérance de la somme des profondeurs dans un arbre aléatoire avec n clés comme en Théorème 13.3. Alors, $D(n)/n = O(\log n)$

Démonstration. Preuve comme pour Théorème 12.1 (temps de calcul du tri rapide). ■