12 Table de symboles et arbres binaires de recherche

12.1 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 :

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

⋆ insert(x) : insertion de l'élément x (clé+info)

Opérations parfois supportées :

⋆ delete(k) : supprimer élément avec clé k

⋆ select(i) : sélection de l'i-ème élément (selon l'ordre des clés)

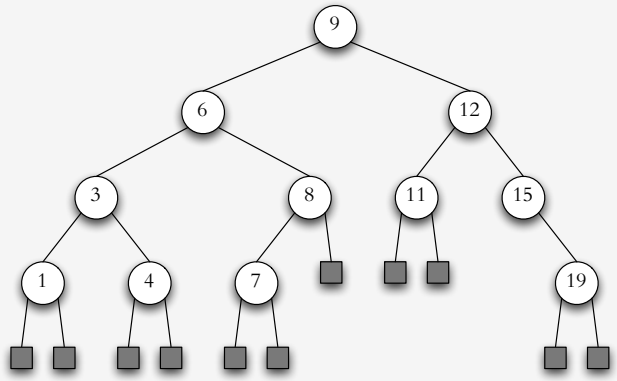
Implantations élémentaires

⋆ 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

12.2 Arbre binaire de recherche

W(fr)



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

Définition 12.1. *Un arbre binaire est un **arbre binaire de recherche** ssi les nœuds inernes sont énumerés lors d'un parcours infixe en ordre croissant de clés.*

L'arbre est stocké 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)

Théorème 12.1. *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.*

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 ← 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
```

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.

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

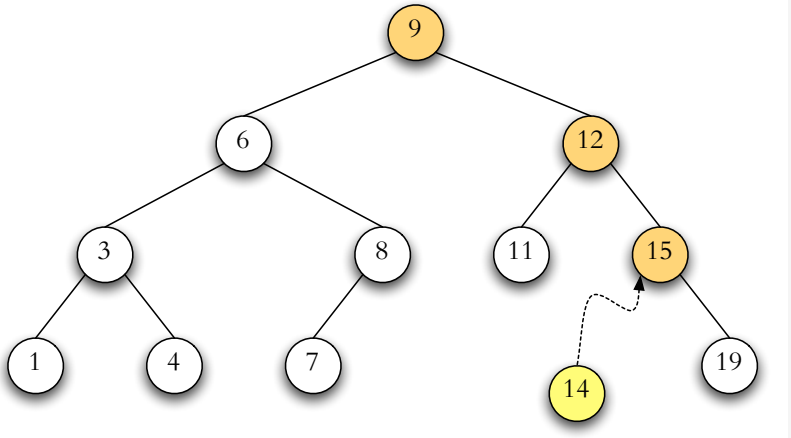
```
SEARCH(x, v) // trouve clé v dans le sous-arbre de x
S1  while x ≠ null et v ≠ x.key do
S2      if v < x.key
S3      then x ← x.left
S4      else x ← x.right
S5  return x
```

Dans un arbre binaire de recherche de hauteur h, MIN, MAX et SEARCH prennent $O(h)$.

12.3 Insertion et suppression

Dans un arbre binaire de recherche de hauteur h, insertion ainsi que la supression prennent $O(h)$.

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



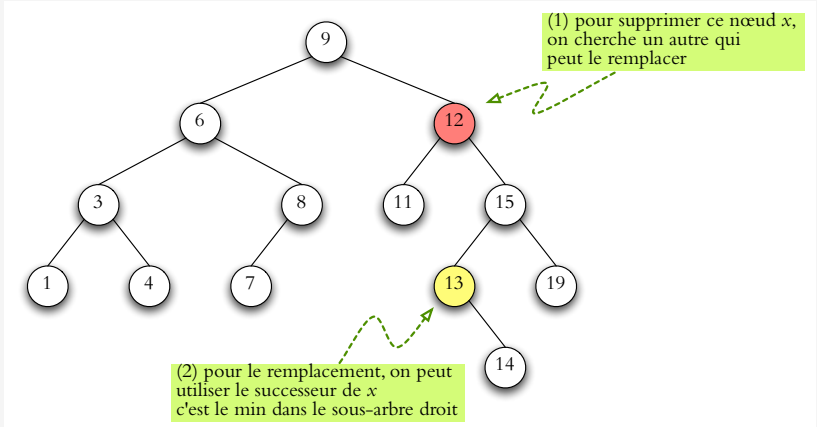
insertion de «14»

Idée : comme en SEARCH, on trouve la place pour v (enfant gauche ou droit manquant)

```
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 les valeurs 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

1. triviale si x n'a pas d'enfants non-null : x.parent.left ← null si x est l'enfant gauche de son parent, ou x.parent.right ← null si x est l'enfant droit
2. facile si x a seulement un enfant : x.parent.left ← x.right ; x.right.parent ← 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)
3. un peu plus compliqué si x a deux enfants : on trouve un 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.*

⇒ 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
```

12.4 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 $1, 2, 3, 4, \ldots, n$ 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 $\{1, 2, \ldots, n\}$)
- ⋆ 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. Insértion/suppression plus compliquées, mais toujours $O(\log n)$.

12.5 Performance moyenne

Théorème 12.3 (Bruce Reed & Michael Drmota)**.** *La hauteur d'un arbre de recherche construit en insérant les valeurs* $1, 2, \ldots, n$ *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 thèorème 12.3 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 exploîte la correspondance à une exécution du tri rapide : le pivot du sous-tableau correspond à la racine du sous-arbre.

Définition 12.2. *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* 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 comme pour Théorème 11.1 (temps de calcul du tri rapide). ■