Arbres de recherche

Introduction

Structures de données représentant un objet de sorte à supporter efficacement des opérations de certain type

• Exemples: représentation de graphes, CLASS-UNION

• Cas *statique* : les opérations ne modifient pas l'objet (requêtes uniquement)
• Cas *dynamique* : les opérations peuvent modifier l'objet (par ex ajouter ou supprimer des éléments de l'ensemble). Cas *on-line* vs. *off-line*.

Maintenir un ensemble ordonné

Maintenir un ensemble ordonné S par rapport aux opérations suivantes :

• TROUVER(*e*) : retourner le pointeur à *e* (si $e \in S$)
• AJOUTER(*e*) : ajouter *e* à *S*
• ENLEVER(*e*) : supprimer *e* de *S*
• MAX(*S*) : retourner l'élément maximal de *S*
• MIN(*S*) : retourner l'élément minimal de *S*
• SUCC(*e*) : retourner l'élément minimal de *S* plus grand que *e*
• PRED(*e*) : retourner l'élément maximal de *S* plus petit que *e*

Solutions directes

1) stocker les éléments dans un tableau dans l'ordre croissant
   - TROUVER, SUCC, PRED : O(log(*n*))
   - MAX, MIN : O(1)
   - AJOUTER, ENLEVER : O(*n*)

2) stocker les éléments dans une liste doublement chainée
   - AJOUTER : O(1)
   - TROUVER, SUCC, PRED, MAX, MIN, ENLEVER : O(*n*)

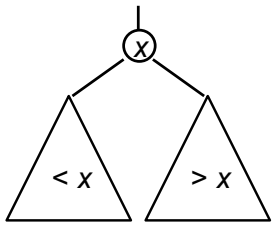
3) même chose mais maintenir la liste ordonnée
   - MAX, MIN : O(1)
   - TROUVER, AJOUTER, ENLEVER, SUCC, PRED : O(*n*)

Arbres binaires de recherche

UMLV ©

Solution proposée : stocker les éléments dans les noeuds d'un arbre binaire vérifiant la *propriété d'arbre de recherche* :

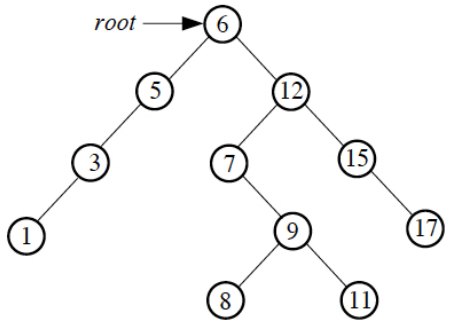
pour tout nœud *x* :
- tous les éléments dans le sous-arbre gauche de *x* sont < *x*,
- tous les éléments dans le sous-arbre droite de *x* sont > *x*



Pour simplicité, dans ce qui suit on va supposer que tous les éléments sont distincts

Arbres binaires de recherche : exemple

UMLV ©



Convention: l'ensemble vide est représenté par *nil*. Chaque nœud stockant un élément possède les deux fils qui peuvent être *nil* (feuille). Les nœuds *nil* (feuilles) ne sont pas montrés sur les images.

Arbres binaires de recherche : application

UMLV ©

Il est facile de sortir tous les éléments dans l'ordre croissant (ou décroissant) par le parcours en profondeur (parcours *symétrique* ou *infixe*) :

```
SORTIR-CROISSANT(x)
        si x≠nil faire
        SORTIR-CROISSANT(left(x))
        sortir x
        SORTIR-CROISSANT(right(x))

SORTIR-CROISSANT(root)
```


Arbres binaires de recherche : TROUVER

UMLV ©

Trouver un élément : en partant de la racine, descendre dans l'arbre en se guidant par les comparaisons :

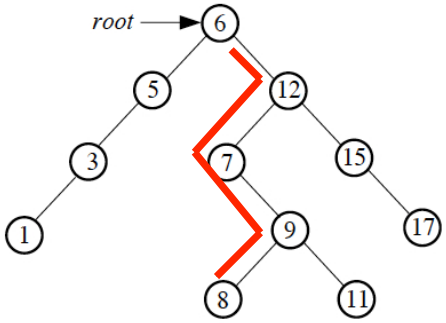
```
TROUVER(e,x)
        si e=x faire
                retourner x; STOP
        si e<x faire
                TROUVER(left(x),e)
        sinon TROUVER(right(x),e)

TROUVER(e,root)
```

Arbres binaires de recherche : TROUVER

UMLV ©

TROUVER(8,*root*) :

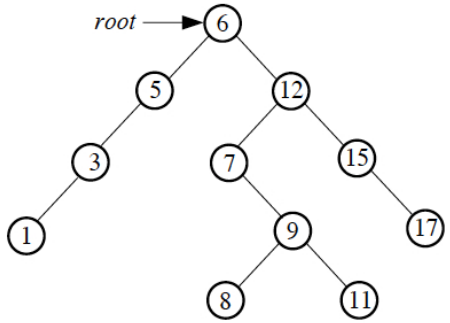


Arbres binaires de recherche : MIN, MAX

UMLV ©

- MIN : la feuille « la plus à gauche »
- MAX : la feuille « la plus à droite »

MIN(*root*)=1
MAX(*root*)=17



Arbres binaires de recherche : SUCC, PRED

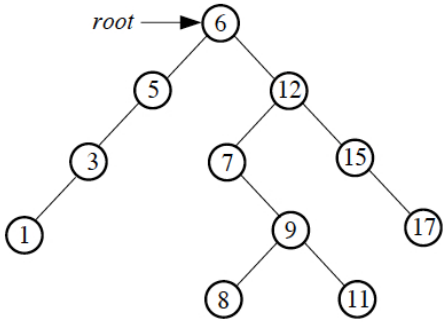
UMLV ©

SUCC(*e*)= { MIN(*right*(*e*)) si *right*(*e*)≠*nil*, sinon
le plus proche ancêtre de *e* dont *e* est dans le sous-arbre gauche,
nil sinon /* *e*=MAX(*root*) */

SUCC(7)=8
SUCC(11)=12
SUCC(17)=*nil*

PRED(*e*) est calculé symétriquement

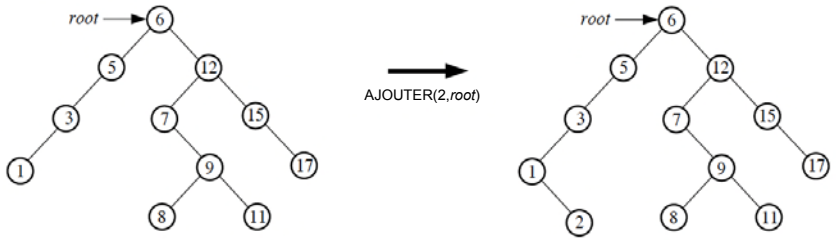
NB: pas de comparaisons non plus !



Arbres binaires de recherche : AJOUTER

UMLV ©

Ajouter un élément : chercher cet élément (comme dans TROUVER) et l'insérer comme fils correspondant du nœud terminal



Arbres binaires de recherche : ENLEVER

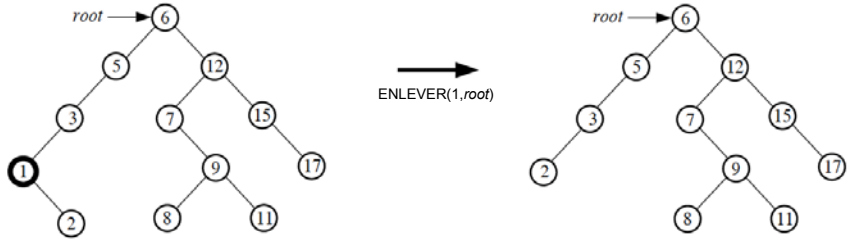
UMLV ©

Enlever un élément *e* :
1) si *e* est une feuille, supprimer-la

Arbres binaires de recherche : ENLEVER

UMLV ©

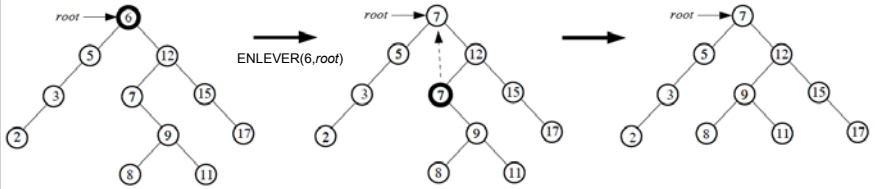
Enlever un élément *e* :
1) si *e* est une feuille, supprimer-la
2) si *e* n'a qu'un fils, attacher le à l'ancêtre de *e* (s'il existe), supprimer le nœud de *e*



Arbres binaires de recherche : ENLEVER

UMLV ©

Enlever un élément *e* :
1) si *e* est une feuille, supprimer-la
2) si *e* n'a qu'un fils, attacher le à l'ancêtre de *e* (s'il existe), supprimer le nœud de *e*
3) si *e* a les deux fils,
   - trouver *s*=SUCC(*e*)=MIN(*right*(*e*)),
   - copier *s* dans *e* et enlever *s*



Complexité

UMLV ©

Toutes les opérations TROUVER, AJOUTER, ENLEVER, SUCC, PRED, MAX, MIN s'exécutent en temps O(*h*), où *h* est la hauteur de l'arbre.

L'objectif est alors de maintenir la hauteur petite, idéalement O(log *n*) (*n* nombre d'éléments courant), donc maintenir l'arbre *équilibré.*

Il existe plusieurs solutions à ce problème : arbres rouge-noirs, arbres AVL, arbres 2-3, *splay trees*, …

Arbres rouge-noirs
(*red-black trees*)

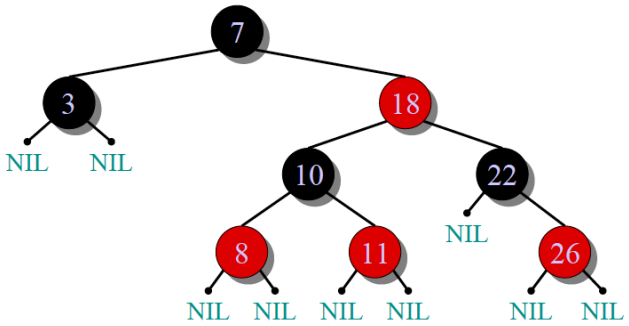
Arbres rouge-noirs : définition

Arbre rouge-noir est un arbre de recherche binaire où chaque nœud possède une « couleur » rouge ou noir qui vérifient les propriétés suivantes :

① le *root* est noir, les feuilles *nil* sont noires
② un nœud rouge a le parent noir
③ tous chemins de *root* à une feuille ont le même nombre de nœuds noirs

(2) implique que un nœud rouge a les fils noirs
(3) implique que tous chemins de *un nœud interne* à une feuille ont le même nombre de nœuds noirs

Arbres rouge-noirs : exemple



Arbres rouge-noirs : hauteur

Theorème : La hauteur d'un arbre rouge-noir avec n nœuds (hors *nil*) est bornée par $2 \cdot \log(n+1)$

Preuve : $h_{max} \leq 2 \cdot h_{min}$, où h_{max} et h_{min} sont les longueurs maximale et minimale d'un chemin de *root* à une feuille. Comme un arbre de hauteur h a 2^h-1 nœuds, on a

$$h_{min} \leq \log(n+1) \leq h_{max} \leq 2 \cdot h_{min}$$

$$1/2 \cdot \log (n+1) \leq h_{min} \leq \log(n+1)$$

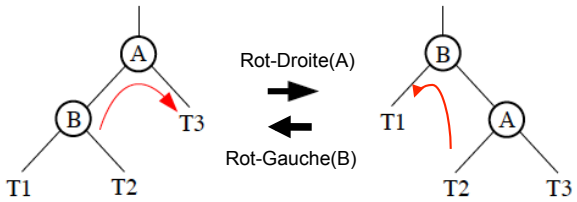
$$h_{max} \leq 2 \cdot h_{min} \leq 2 \cdot \log (n+1)$$

Arbres rouge-noirs : AJOUTER

UMLV ©

On ajoute d'abord un élément comme précédemment. Il se peut que son coloriage est impossible sans violer les propriétés (2) ou (3). On va le colorier en vert (« couleur à définir ») et modifier l'arbre pour rendre le coloriage possible.

La modification va consister en une série de « rotations » :



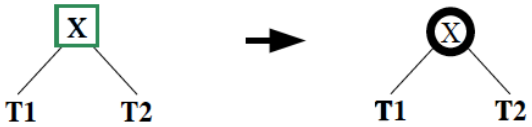
Arbres rouge-noirs : AJOUTER

UMLV ©

Les rotations vont propager le nœud vert de feuille à la racine jusqu'à le faire disparaître

Soit X le nœud vert. Cas faciles:

Cas 1: X est la racine ⇨ colorier X en noir



Arbres rouge-noirs : AJOUTER

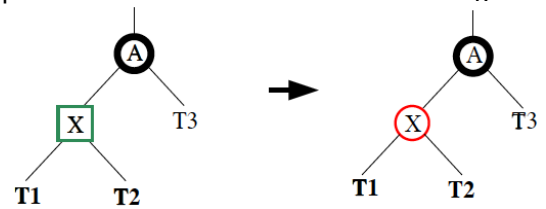
UMLV ©

Les rotations vont propager le nœud vert de feuille à la racine jusqu'à le faire disparaître

Soit X le nœud vert. Cas faciles:

Cas 1: X est la racine ⇨ X est le seul nœud de l'arbre ⇨ colorier X en noir

Cas 2: Le parent de X est noir ⇨ colorier X en rouge

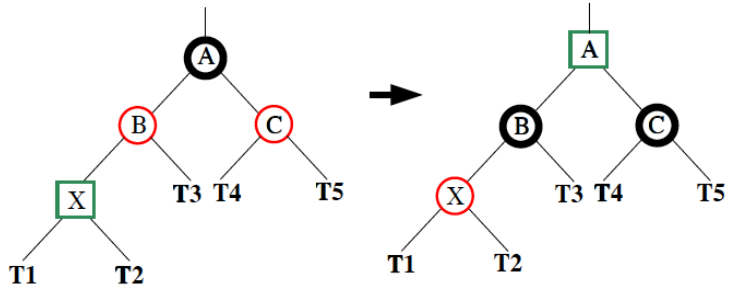


Arbres rouge-noirs : AJOUTER

UMLV ©

Cas 3: le parent de X est rouge ⇨ le parent B de X ne peut pas être la racine ⇨ il existe un grand-parent A de X

Cas 3.1: un oncle C de X existe et il est rouge ⇨ colorier X en rouge, changer B et C en noir, changer A en vert



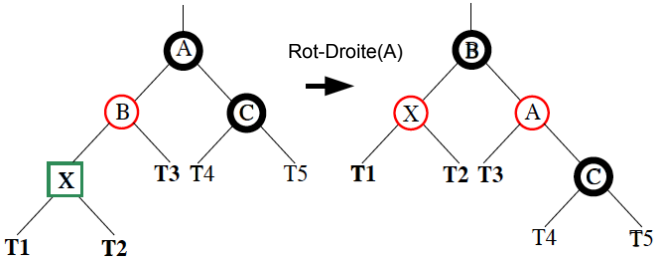
Arbres rouge-noirs : AJOUTER

UMLV ©

Cas 3: le parent de X est rouge ⇨ le parent B de X ne peut pas être la racine ⇨ il existe un grand-parent A de X

Cas 3.2: un oncle C de X est noir ou bien il n'existe pas

Cas 3.2.1: X est un fils gauche de son parent ⇨ effectuer une rotation droite et recoloriage :



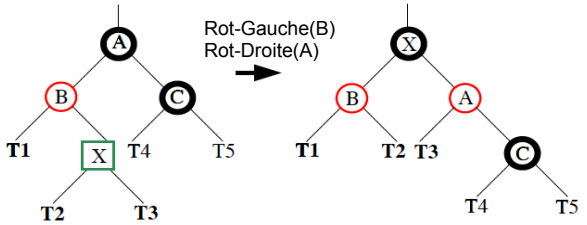
Arbres rouge-noirs : AJOUTER

UMLV ©

Cas 3: le parent de X est rouge ⇨ le parent B de X ne peut pas être la racine ⇨ il existe un grand-parent A de X

Cas 3.2: un oncle C de X est noir ou bien il n'existe pas

Cas 3.2.2: X est un fils droite de son parent ⇨ effectuer une rotation droite et recoloriage : effectuer une rotation gauche, puis rotation droite, puis recoloriage



Arbres rouge-noirs : AJOUTER

UMLV ©

Conclusions:
• Cas 1, 2, 3.2.1, 3.2.2 achèvent l'insertion. Cas 3.1 propage le nœud vert vers la racine
• L'algorithme est correcte car chaque cas préserve les propriétés de l'arbre rouge-noir (mise à part le nœud vert)
• Le traitement de chaque cas prend temps O(1)
• L'ajout prend temps O(log *n*)

Arbres rouge-noirs : ENLEVER

UMLV ©

L'algorithme est similaire à celui de AJOUTER. Sa complexité est O(log *n*).

Conclusion finale

Arbres rouge-noirs permettent de réaliser les opérations TROUVER, AJOUTER, ENLEVER, SUCC, PRED, MAX, MIN en temps O(log *n*)