Problème de Tri

Algorithme : TriTableau(**dr** T : tableau)
Données : T tableau
Résultat : T est modifié :

- T contient les mêmes élements qu'en entrée,
- T est croissant ($\forall 0 \leq i < j < taille(T), T[i] \leq T[j]$)

T	5	22	13	5	2	67	13	20

$\downarrow$

T	2	5	5	13	13	20	22	67

Beaucoup de problèmes sur les tableaux peuvent être résolus plus efficacement si les tableaux sont triés (Recherche d'un élément).

Spécifications du problème

- On étudie le problème du tri d'un tableau d'entiers mais les algorithmes sont applicables à d'autres types munis d'un ordre total
- Les valeurs à trier constituent une partie des *objets* à trier, leur clé
- Plusieurs éléments du tableau peuvent avoir la même valeur
- Pas d'hypothèse sur le domaine des valeurs à trier (domaine infini)
- Les algorithmes étudiés opèrent par comparaisons d'éléments (d'autres techniques existent si hypothèses sur le domaine (voir TD))
- Pour évaluer la complexité des algorithmes on compte le nombre de comparaisons entre éléments du tableau effectuées ; (on peut également compter le nombre d'échanges de place)
- La taille du problème est le nombre d'éléments du tableau

Algorithme

Algorithme : TriInsertion(**dr** T : tableau d'entiers)
Variables : $i, j, x \in \mathbb{N}$; **début**

```
pour i de 1 à taille(T)-1 faire
    x ← T[i]; /* Insertion de x dans le sous-tableau trié
       T[0..i-1]                                          */
fin pour
fin algorithme
```


Preuve

Arrêt

Évident car pour le `Tant que` $j + 1 \in \mathbb{N}$ et décroît strictement à chaque itération.

Invariants

Complexité.

début
  pour i `de 1 à N-1` **faire**
    $x \leftarrow T[i]$; $j \leftarrow \ldots$; **tant que** $\ldots$ `et` $T[j] > x$ **faire**
      | $\ldots$;
    fin tq
    $\ldots$;
  fin pour
fin algorithme

- on note N= taille(T)
- la seule comparaison est dans la condition du `Tant que`.
- Meilleur des cas : ; au total comparaisons
- Pire des cas : ;

Tri par Tas (HeapSort)

Principe

Utiliser une structure de données efficace.
Le **Tas** (tas binaire ou file de priorité) permet de représenter un ensemble muni d'une relation d'ordre optimisant les opérations suivantes :

- **CréerTas() :** renvoie un tas vide
- **InsérerTas(d e : entier, dr t : tas) :** insère l'élément `e` dans le tas `t`
- **ExtraireMax(dr t : tas, r e : entier) :** en résultat *e* est l'élément max du tas `t` ; de plus*e* est supprimé de `t`

Tri par Tas (HeapSort)

L'algorithme

Algorithme : TriParTas(**dr** $T[0..N-1]$: Tableau)
Variables : tas : Tas binaire ; **début**
  tas ←CreerTas() ; ; `/* Remplir le tas */`
  pour i `de 0 à` $N-1$ **faire**
    | InsérerTas(T[i],tas)
  fin pour
  ; `/* Vider le tas */`
  pour i `de` $N-1$ `bas 0 par pas de -1` **faire**
    | ExtraireMax(tas,T[i])
  fin pour
  ;
fin algorithme

Tri par Tas (HeapSort)

Complexité

- Nous allons voir que
  - L'opération `CréerTas` peut être réalisée en temps $\theta(1)$
  - Les opérations `InsérerTas(e,t)` et `ExtraireMax(t,e)` ont une complexité dans le pire des cas en $O(log_2 n)$ (n = nombre d'éléments du tas t)
- La complexité du tri par Tas est alors

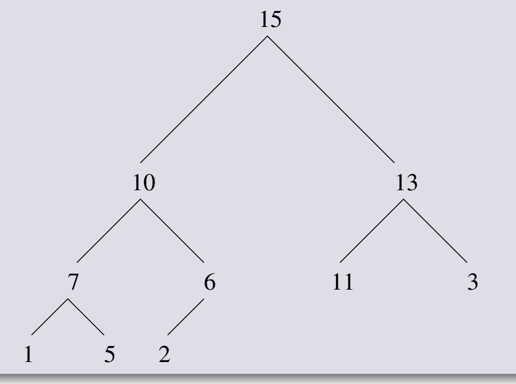
Réalisation d'un Tas

Définition

Un **Tas** est un arbre binaire vérifiant les 2 conditions suivantes :

- Condition sur les valeurs **CV** :
  tout noeud v de l'arbre, autre que la racine, vérifie $info(v) \leq info(pere(v))$
- Condition sur la forme de l'arbre **CF** :
  l'arbre est parfait
  - h étant la hauteur de l'arbre binaire, tous les niveaux de profondeur $p = 0, 1, \ldots, h-1$ sont complets : nombre de noeuds de profondeur $p = 2^p$
  - les feuilles de profondeur h sont regroupées *à gauche*

Exemple



Propriété

La hauteur d'un tas contenant n éléments est

Opérations sur les arbres utilisées pour les fonctions du Tas

- *CréerArbre()* : renvoie l'arbre vide
- *Racine(**d** A)* : renvoie le noeud racine de l'arbre *A*.
- *Père(**d** N)* : renvoie le noeud père du noeud *N*.
- *Filsd(**d** N)* : renvoie le noeud fils droit du noeud *N*.
- *Filsg(**d** N)* : renvoie le noeud fils gauche du noeud *N*.
- *Info(**d** N)* : renvoie l'information contenue dans le noeud *N*.
- *Feuille ?(**d** N)* : renvoie vrai ssi *N* est une feuille.
- *DernièreFeuille(**d** A)* : renvoie la feuille la plus à droite du dernier niveau de l'arbre *A*.
- *CréerFeuille(**dr** A, **d** e, **r** N)* : modifie l'arbre *A* en créant après la dernière feuille une nouvelle feuille *N* de contenu *e*.
- *SupprimerFeuille(**dr** A)* : Supprime la dernière feuille de l'arbre *A*.
- *EchangerContenu(**dr** N1,**dr** N2)* : échange les contenus des noeuds *N*1 et *N*2.
- *FilsMax(**d** N)* : Renvoie parmi les 2 fils du noeud *N* celui dont le contenu est le plus grand.

InsérerTas :

Principe

Satisfaire la condition sur la forme d'un tas (**CF**) puis échanger les contenus des noeuds pour satisfaire la condition sur les valeurs (**CV**).

Algorithme : InsérerTas(**d** e : entier,**dr** t : tas)
Données : *t* un tas et *e* en entier
Résultat : *t* est un tas contenant les éléments du tas initial plus *e*
Variables : *q* un noeud ; **début**
$CreerFeuille(t, e, q)$; **tant que** **faire**
fin tq
fin algorithme

Preuve

- **Arrêt :**
- **Invariant :**
- **Complexité :**

ExtraireMax

Principe de l'algorithme

Comme pour l'insertion : satisfaire la condition **CF** puis échanger les contenus des noeuds pour satisfaire la condition **CV**.

Algorithme : ExtraireMax(**dr** t : tas, **r** max : entier)
Données : *t* un tas non vide
Résultat : *max* est le plus grand élément de *t* et *max* est supprimé de *t*.
Variables : *q*, *f* 2 noeuds ; **début**
$max \leftarrow Info(racine(t))$; $EchangerContenu(racine(t), dernierefeuille(t))$;
$SupprimerFeuille(t)$; **si** $nonarbrevide(t)$ **alors**
$q \leftarrow racine(t)$; **tant que** **faire**
fin tq
fin si
fin algorithme

Preuve et Invariant

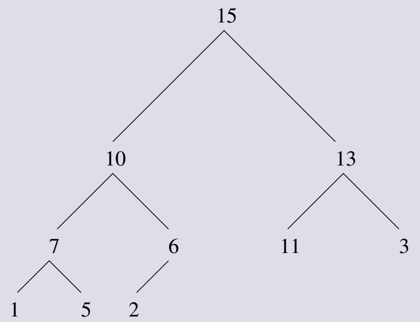
- **Arrêt :**
- **Invariant :**
- **Complexité :**

Complexité en espace

Cet algorithme de Tri par Tas utilise une structure de Données supplémentaire (le Tas), sa complexité en place est $O(n)$.
En fait le Tas peut être représenté à l'aide du tableau que l'on trie (voir TD).
$\Rightarrow$ Pas besoin d'espace supplémentaire.

Représentation d'un Tas par un tableau

Parcours en largeur de l'arbre



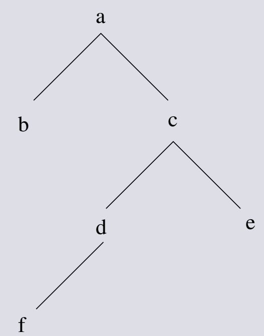
15	10	13	7	6	11	3	1	5	2

Cette représentation est compacte (taille du tableau = nombre de noeuds de l'arbre) et permet d'implanter chaque opération avec une complexité en O(1).

Représentation d'un Tas par un tableau

Codage pas toujours compact

Cette représentation par parcours en largeur est compacte pour les arbres parfaits . Ce n'est pas le cas pour tous les arbres :



a	b	c			d	e					f

« Diviser pour Résoudre »

Principe Général

Résoudre un problème :

- Décomposer le problème en sous-problèmes
- Résoudre indépendamment chaque sous-problème

Application au Tri

- Diviser le tableau en 2 sous–tableaux
- Trier chacun des 2 sous-tableaux

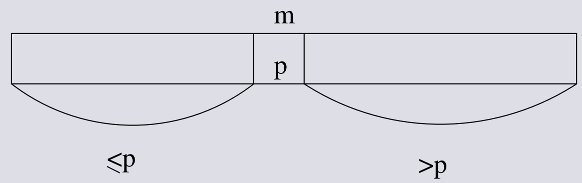
Ce principe général peut être appliqué de plusieurs façons :

- Tri Fusion
  1. Diviser le tableau en 2 sous–tableaux de taille identique
  2. Trier les 2 sous-tableaux
  3. Fusionner les 2 sous-tableaux triés

Tri Rapide (QuickSort)

Tri Rapide :

- Diviser le tableau en 2 sous–tableaux, de sorte que le tableau trié final s'obtienne directement à partir des 2 sous-tableaux triés.
  La division du tableau se fait par rapport à une valeur pivot :



Exemple

5	22	13	5	2	67	13	20

choix valeur Pivot = 13

5	22	13	5	2	67	13	20

division du tableau

5	13	2	5	13	67	22	20

tri du premier sous-tableau

2	5	5	13	13	67	22	20

tri du second sous-tableau

2	5	5	13	13	20	22	67

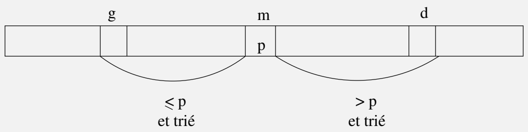
Algorithme : TriRapide(**dr**T : Tableau ,**d** g : indice,**d** d : indice)
Données : $T[g \dots d]$; $g \leq d+1$
Résultat : Permute les éléments du sous-tableau $T[g \dots d]$ pour que $T[g \dots d]$ soit trié
Variables : $m \in \mathbb{N}$; **début**
  si $g < d$ **alors**
  fin si
fin algorithme

Spécifications de l'algorithme `Pivot`

Algorithme : Pivot(**dr** T : tableau,**d** g : indice,**d** d : indice,**r** m : indice)
Données : $g < d$, $T[g \dots d]$
Résultat : en résultat $m \in [g \dots d]$ et $T[g \dots d]$ tels que et
$\forall i \in [g \dots m-1], T[i] \leq T[m], \forall i \in [m+1 \dots d], T[m] < T[i]$

Preuve du tri rapide

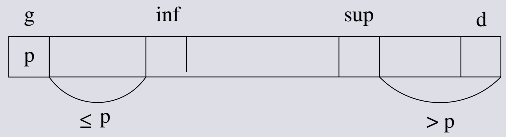
- Arrêt : la taille du sous-tableau à trier $(d - g + 1)$
  - est un entier naturel
  - décroît strictement à chaque appel récursif
- Preuve du résultat par induction sur la taille du problème .
  P(n) : TriRapide est correct pour tout tableau de taille n
  - $P(0), P(1)$ sont vérifiés car un tableau à 0 ou 1 élément est trié
  - soit $n > 1$ et supposons que $\forall n' < n, P(n')$
    soit $T[g \dots d]$ un tableau de taille n. Après le calcul du pivot on a $g \leq m \leq d$, donc $T[g \dots m-1]$ et $T[m+1 \dots d]$ ont une taille inférieure strictement à n.
    Par hypothèse d'induction les 2 appels récursifs trient correctement $T[g \dots m-1]$ et $T[m+1 \dots d]$
    Donc après le deuxième appel récursif on a :



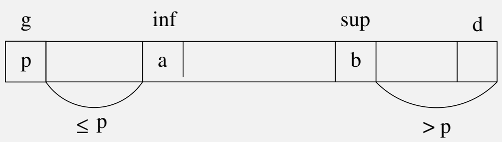
    Donc $T[g \dots d]$ est trié , donc $P(n)$ est vérifié

Calcul du Pivot

- Choix de la valeur pivot le premier élément : $p = T[g]$;
- On parcourt les éléments du tableau en permutant les éléments mal placés par rapport au pivot en maintenant l'invariant suivant :



itération du calcul de pivot



- Si $T[inf] \leq p$
- Sinon

Arrêt

- On s'arrête quand
- On place la valeur pivot à l'indice

L'algorithme

Algorithme : Pivot(**dr** T : tableau,**d** g : indice,**d** d : indice,**r** m : indice)
Données : $g < d$, $T[g \dots d]$
Résultat : en résultat $m \in [g \dots d]$ et $T[g \dots d]$ tels que et
$\forall i \in [g \dots m-1], T[i] \leq T[m], \forall i \in [m+1 \dots d], T[m] < T[i]$
début
  $p \leftarrow T[g]$; $inf \leftarrow g+1$; $sup \leftarrow d$; **tant que** $inf \leq sup$ **faire**
    si $T[inf] \leq p$ **alors**
    sinon
    fin si
  fin tq
fin algorithme

Complexité du Tri Rapide

Complexité de Pivot(T,g,d,m)

Le nombre de comparaisons effectuées est dans tous les cas exactement $d-g$, nombre d'éléments du sous–tableau `T[g..d]` - 1 .

Complexité de l'algorithme récursif Tri Rapide

Soit

- $t(n)$ le nombre de comparaisons effectuées par le tri rapide pour un tableau de n éléments
- n_1 la taille du premier sous-tableau $(m-g)$
- n_2 celle du second $(d-m)$.

t vérifie la récurrence (avec $n_1 + n_2 = n-1$) :

$$\begin{aligned} t(n) &= t(n_1) + t(n_2) + n - 1 \ si \ n > 1 \\ t(n) &= 0 \ si \ n \leq 1 \end{aligned}$$

Pire des cas

est atteint lorsque
l'un des sous–tableaux est vide : $n_1 = 0$ ou $n_2 = 0$.

$$\begin{aligned} t(n) &= t(n-1) + n - 1 \ si \ n > 1 \\ t(n) &= 0 \ si \ n \leq 1 \end{aligned}$$

$$t(n) = \sum_{i=1}^{n-1} i = n.(n-1)/2 = O(n^2)$$

Meilleur des cas

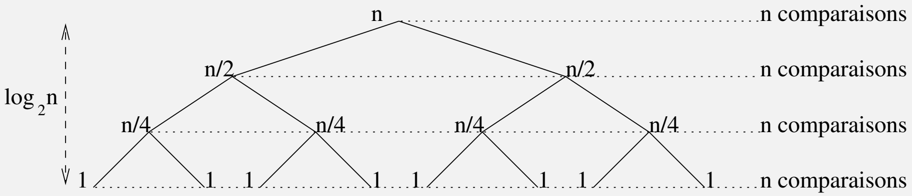
est atteint
lorsque les 2 sous-tableaux sont de même taille.

$$\begin{aligned} t(n) &= t(\lfloor n/2 \rfloor) + t(n - \lfloor n/2 \rfloor - 1) + n - 1 \ si \ n > 1 \\ t(n) &= 0 \ si \ n \leq 1 \end{aligned}$$

Pour simplifier les calculs on résout la récurrence suivante qui majore t

$$\begin{aligned} t'(n) &= 2.t'(n/2) + n \ si \ n > 1 \\ t'(n) &= 0 \ si \ n \leq 1 \end{aligned}$$

Intuition



$t'(n) = O(nlog_2(n))$

Preuve de $t'(n) \in O(n.log_2 n)$

$$
\begin{aligned}
t'(n) &= 2.t'(n/2) + n \; si \; n > 1 \\
t'(n) &= 0 \; si \; n \leq 1
\end{aligned}
$$

On montre que $\exists n_0, \exists c, \forall n > n_0, t'(n) \leq c.n.log_2(n)$ par récurrence sur n :

- $n = 1$ OK
- HR : $\forall n' < n, t'(n') \leq c.n'.log_2(n')$

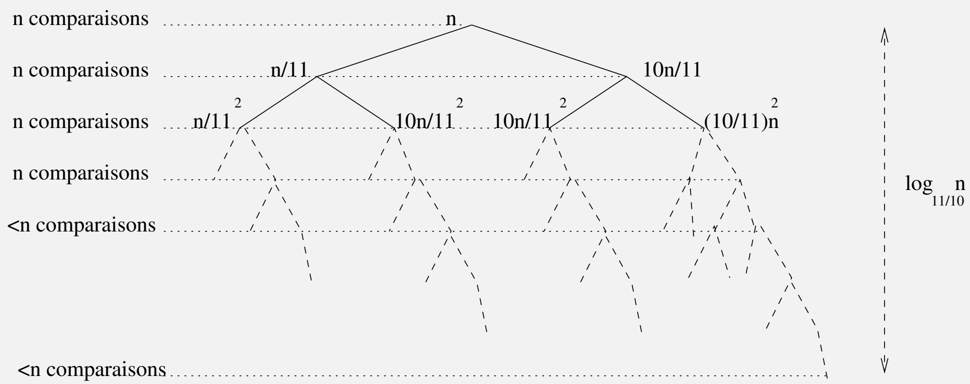
$$
\begin{aligned}
t'(n) &= 2.t'(n/2) + n \\
&\leq 2c.n/2.log_2(n/2) + n \\
&\leq c.n.(log_2(n) - 1) + n \\
&\leq c.n.log_2(n) + n(1 - c) \\
&\leq c.n.log_2(n)
\end{aligned}
$$

Vérifié si $c \geq 1$ (n_0 quelconque)

Tri Rapide ?

- La complexité dans le pire des cas du Tri rapide est pire que celle du Tri par Tas et identique à celle du Tri par insertion
- La complexité dans le meilleur des cas du tri rapide est pire que celle du Tri par insertion
- Mais le meilleur des cas est fréquent : même si à chaque étape le tableau n'est pas divisé en 2 sous-tableaux de même taille, la complexité peut être $O(n.ln(n))$.

Tri Rapide ?



- Par exemple si à chaque étape un sous-tableau est 10 fois plus grand que l'autre, la complexité reste $O(n.log_{\frac{11}{10}}(n)) = O(n.ln(n))$.
- on peut montrer que la complexité en moyenne est $t_{moy}(n) = O(n.ln(n))$

Conclusions Tri Rapide

Remarques

- limite de la complexité dans le pire des cas
- L'algorithme de calcul de Pivot ne minimise pas le nombre de déplacements d'éléments du tableau ; il peut être amélioré

Complexité minimum du problème de Tri par comparaisons

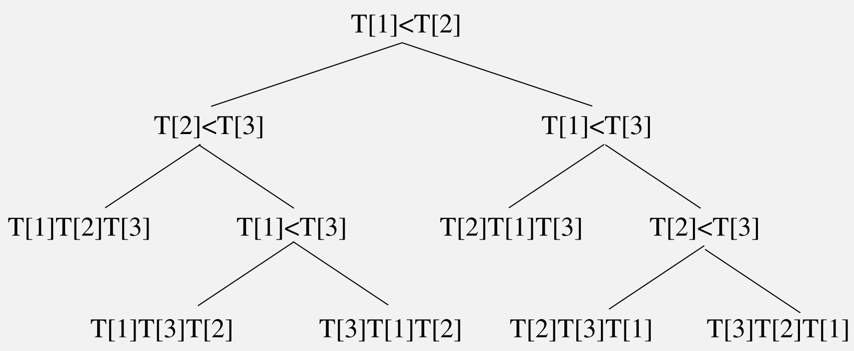
Peut-on trouver un algorithme de tri exécutant moins de O(n.ln(n)) comparaisons dans le pire des cas ?

- Un algorithme de tri par comparaisons exécute une séquence de comparaisons.
- La comparaison suivante dépend du résultat des comparaisons précédentes.
- On peut représenter l'ensemble des exécutions possibles d'un algorithme par un arbre binaire dont les étiquettes sont des comparaisons entre 2 éléments de tableau
- un noeud d'étiquette $T[i] < T[j]$ a pour sous–arbre gauche (respectivement droit) l'arbre représentant les comparaisons réalisées par l'algorithme lorsque $T[i] < T[j]$ (respectivement $T[i] \geqslant T[j]$)

Exemple

Algorithme : tri1(T[1..3])
début
  si `T[1]<T[2]` **alors**
    si `T[2]<T[3]` **alors**
      renvoyer `T[1]T[2]T[3]`
    sinon si `T[1]<T[3]` **alors**
      renvoyer `T[1]T[3]T[2]`
    sinon
      renvoyer `T[3]T[1]T[2]`
    fin si
  sinon si `T[1]<T[3]` **alors**
    renvoyer `T[2]T[1]T[3]`
  sinon si `T[2]<T[3]` **alors**
    renvoyer `T[2]T[3]T[1]`
  sinon
    renvoyer `T[3]T[2]T[1]`
  fin si
fin algorithme

Exemple



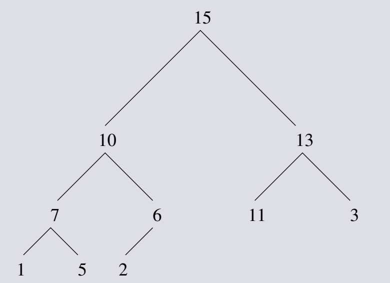
- Chaque exécution correspond à une branche de l'arbre. Le nombre de comparaisons qu'elle exécute est la longueur de la branche
- Le pire des cas correspond à la hauteur de l'arbre
- Tout algorithme de Tri doit différencier les $n!$ permutations. L'arbre a donc au moins $n!$ feuilles
- La hauteur minimum d'un arbre binaire possédant $n!$ feuilles est $log_2(n!) = O(n.log_2 n)$

Propriété

La complexité dans le pire des cas d'un algorithme triant par comparaison un tableau de n éléménts est $O(n.log_2 n)$

Arbre Binaire de Recherche

Retour sur le Tas



Opérations efficaces avec un Tas

- Ajouter un élément
- Supprimer l'élément de valeur maximum

Rechercher un élément dans un Tas de n éléments ?

Arbre Binaire de Recherche

Principe

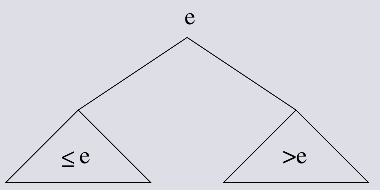
Comme le *Tas Binaire*, **l'Arbre Binaire de Recherche** est une représentation d'un ensemble muni d'un ordre total.
On prend comme exemple les entiers naturels.

Définition

Un **Arbre Binaire de Recherche (ABR)** est un arbre binaire dont tous les noeuds N vérifient :
Pour tout noeud N_1 du sous-arbre gauche de N
Pour tout noeud N_2 du sous-arbre droit de N on a :

$$\text{information de } N_1 \leq \text{information de } N < \text{information de } N_2.$$

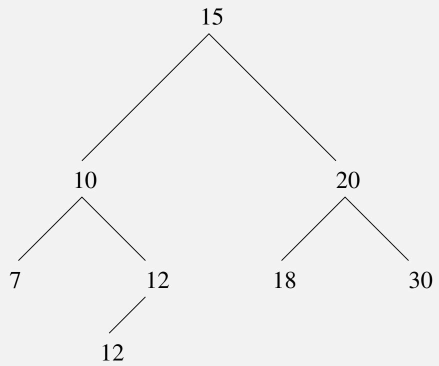
Arbre Binaire de Recherche



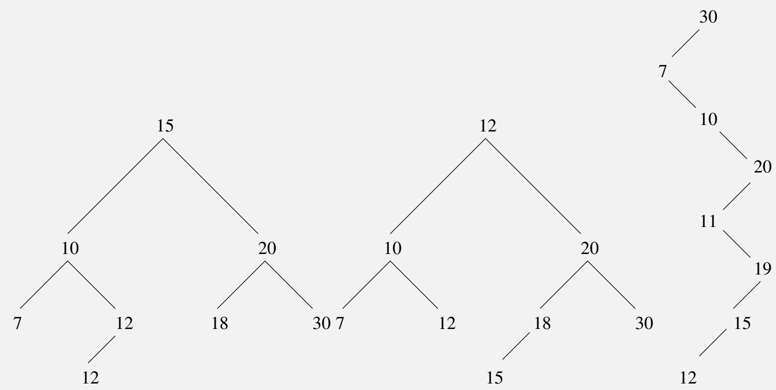
Remarque

Condition sur les étiquettes
Pas de condition sur la forme de l'arbre.

Exemple d'Arbre Binaire de Recherche



Plusieurs ABR peuvent représenter un même ensemble



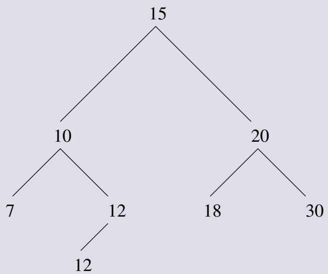
La hauteur d'un ABR représentant un ensemble à *n* éléments peut varier de

Tri et Arbre Binaire de Recherche

Pour obtenir la liste triée des étiquettes d'un ABR

.

Exemple



Opérations sur les ABR

Les opérations de base pour manipuler un ensemble :

- Rechercher un élément
- Ajouter un élément
- Supprimer un élément

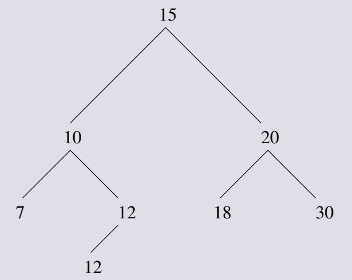
peuvent être réalisées sur un ABR dans un temps proportionnel à la hauteur de l'ABR.

Recherche d'une étiquette dans un ABR

Principe

Si l'élément recherché est différent de l'étiquette de la racine de l'ABR, en les comparant , on sait quel sous–arbre peut contenir l'élément recherché.

Exemple



Recherche de 13 :

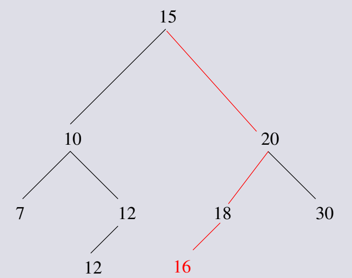
Algorithme : Recherche(**d** A : ABR, **d** x :entier)
Données : *A* 1 ABR, *x* un entier
Résultat : Renvoie `NULL` si *A* n'a pas de noeud d'étiquette *x*
Sinon renvoie un noeud de *A* ayant *x* pour étiquette
début
  si $A = NULL$ `ou` $A \uparrow info = x$ **alors**
    renvoyer *A*
  sinon
    si $x < A \uparrow info$ **alors**
      renvoyer
    sinon
      renvoyer
    fin si
  fin si
fin algorithme

Complexité :

Insertion dans un ABR

Exemple

Ajout du nouveau noeud comme feuille de l'arbre



Ajout de 16

Algorithme : Insertion(**dr** A : ABR,**d** x : entier)
Données : *A* 1 ABR, *x* un entier
Résultat : Modifie *A* en y ajoutant un noeud d'étiquette *x*
début
  si $A = NULL$ **alors**
    $A \leftarrow$
  sinon si $x \leq A \uparrow info$ **alors**

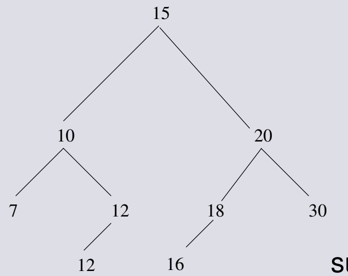
  sinon

fin algorithme

Complexité :

Suppression dans un ABR

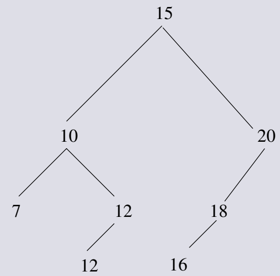
Cas 1 : Si le noeud à supprimer est une feuille



suppression de 30

Suppression dans un ABR

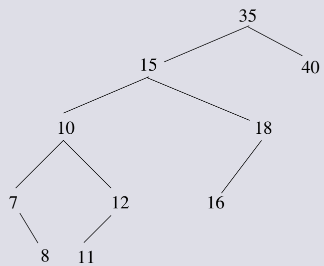
Cas 2 : Si le noeud à supprimer a l'un de ses 2 sous-arbres vide



suppression de 20

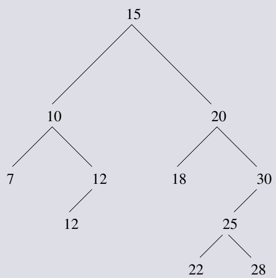
Suppression dans un ABR

Cas 3 : Si le noeud à supprimer n'a pas de sous-arbre vide



suppression de 15

Suppression du noeud d'étiquette max dans un ABR



Suppression de l'étiquette max d'un ABR

Algorithme : SupprimerMax(**dr** A : ABR, **r** max : entier)
Données : *A* 1 ABR non vide
Résultat : max est la plus grande étiquette de *A* ; supprime l'étiquette max de l'arbre *A*
début
    si $A \uparrow sad = NULL$ **alors**
        $max \leftarrow A \uparrow info$; $A \leftarrow A \uparrow sag$;
    sinon

    fin si
fin algorithme

Complexité :

Suppression dans un ABR

Algorithme : Suppression(**dr** A : ABR, **d** x : entier)
Données : *A* 1 ABR non vide contenant l'étiqtette *x*
Résultat : Supprime de *A* un noeud d'étiquette *x*
début
    si $x < A \uparrow info$ **alors**
        Suppression($A \uparrow sag, x$)
    sinon si $A \uparrow info > x$ **alors**
        Suppression($A \uparrow sad, x$)
    sinon
        /* $x = A \uparrow info$ */
        si $A \uparrow sag = NULL$ **alors**
            $A \leftarrow A \uparrow sad$
        sinon
            SupprimerMax($A \uparrow sag, max$) ; $A \uparrow info \leftarrow max$
        fin si
fin algorithme

Complexité :

Conclusion

Dans un ABR Rechercher, ajouter et supprimer un élément peuvent être réalisés en `O(hauteur)`. Mais la hauteur d'un ABR peut être égale au nombre d'éléments de l'ABR.

Si dans le pire des cas la hauteur d'un arbre est de l'ordre du nombre de ses noeuds, en moyenne elle est de l'ordre du logarithme du nombre de noeuds.
Il existe des Structures de Données qui

- vérifient la même condition sur les valeurs que les ABR
- vérifient une condition sur la Forme de l'arbre, garantissant une hauteur en `O(log(nombre d'éléments))`
- permettent de réaliser les 3 opérations avec la même complexité (en `O(hauteur)`)

`AVL, Arbre Rouge et Noir`