Ensembles

Type « dictionnaire »

sous-ensembles finis de *E* (ordonné) avec les opérations :

Ens_vide : $\rightarrow$ Ens

Ajouter : Ens x Elément $\rightarrow$ Ens

Enlever : Ens x Elément $\rightarrow$ Ens

Elément : Ens x Elément $\rightarrow$ Booléen

Vide : Ens $\rightarrow$ Booléen

Min, Max : Ens $\rightarrow$ Elément

Elt : Adresse $\rightarrow$ Elément

Place : Ens x Elément $\rightarrow$ adresse

Implémentations possibles

Tables (triées ou non), Listes chaînées,

Tables hachées, Arbres (équilibrés ou non)

Orthographe


```
fonction SPELL (t texte, d dictionnaire) : liste de noms ;
début
    E ← Ens_vide ; x ← lire_mot (t) ;
    tant que x ≠ FF faire {
        si x ≠ FL alors E ← Ajouter (E, x) ;
        x ← lire_mot (t) ;
    }
    y ← lire_mot (d) ;
    tant que y ≠ FF faire {
        si y ≠ FL alors E ← Enlever (E, y) ;
        y ← lire_mot (d) ;
    }
    retour (E) ;
fin
```

Texte = suite de lignes, suivie de FF
Ligne = suite de mots, suivie de FL

Lire_mot(texte) = { mot suivant ou FL ou FF

Temps d'exécution

opérations sur ensemble

implémentation

	Ens_vide	Ajouter Enlever	Élément	Min
table	cst	*O*(1)*	*O*(*n*)	*O*(*n*)
table triée	cst	*O*(*n*)	*O*(log *n*)	*O*(1)
liste chaînée	cst	*O*(1)*	*O*(*n*)	*O*(*n*)
arbre équilibré	cst	*O*(log *n*)	*O*(log *n*)	*O*(log *n*)
arbre	cst	*O*(log *n*)	*O*(log *n*)	*O*(log *n*)
table hachée	*O*(*B*)	cst	cst	*O*(*B*)

en moyenne

n nombre d 'éléments, *B* > *n*

**sans le test d 'appartenance*

Arbre binaire de recherche

A arbre binaire
noeuds étiquetés par des éléments

A est un arbre binaire de recherche

ssi l'ordre symétrique donne une liste croissante des éléments

ssi en tout noeud *p* de A

Elt(G(*p*)) < Elt(*p*) < Elt(D(*p*))

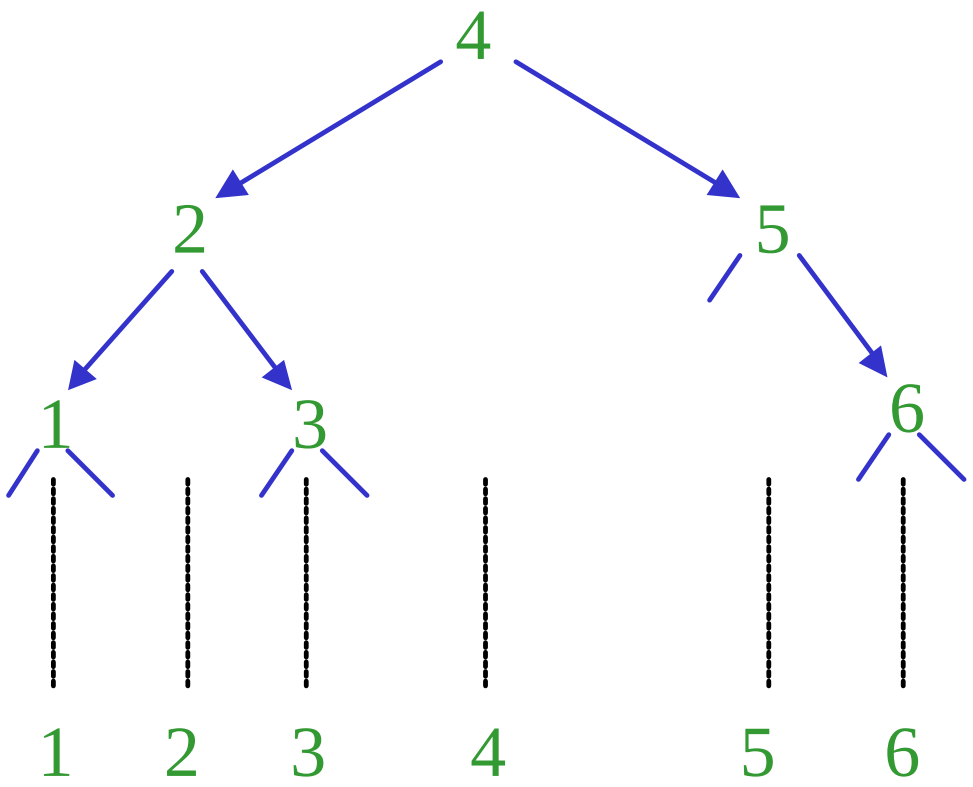
ssi A = Arbre_vide ou

A = (*r*, *G*, *D*) avec

. *G*, *D* arbres binaires de recherche et

. Elt(*G*) < Elt (*r*) < Elt(*D*)

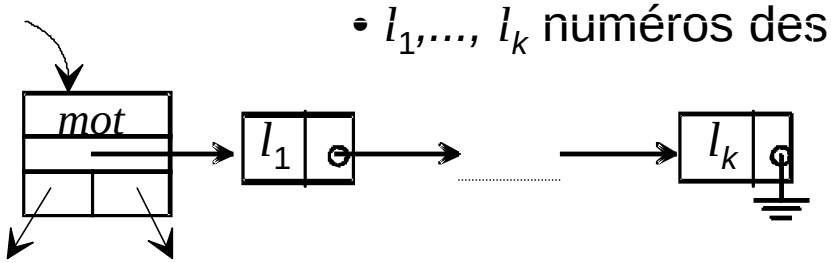
Exemple



Index

Index (texte) = liste de (*mot*, l_1,..., l_k)

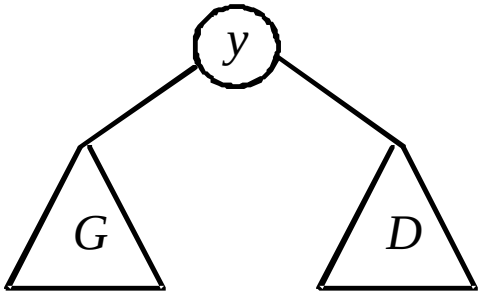
- en ordre croissant des mots
- l_1,..., l_k numéros des lignes où apparaît *mot*




```
fonction INDEX (t texte) : liste de noms ;
début
   A ¬ Ens_vide ; x ¬ lire_mot (t) ;  l ¬ 1  ;
   tant que  x ≠ FF  faire
   {  si  x = FL  alors  l ¬ l + 1 ;
      sinon A ¬ Ajouter (A, x, l) ;
      x ¬ lire_mot (t) ;
   }
   retour (SYM (A)) ;
fin .
```

Elément ?

Elément (A, x) = vrai ssi x est étiquette d'un noeud de A (abr)

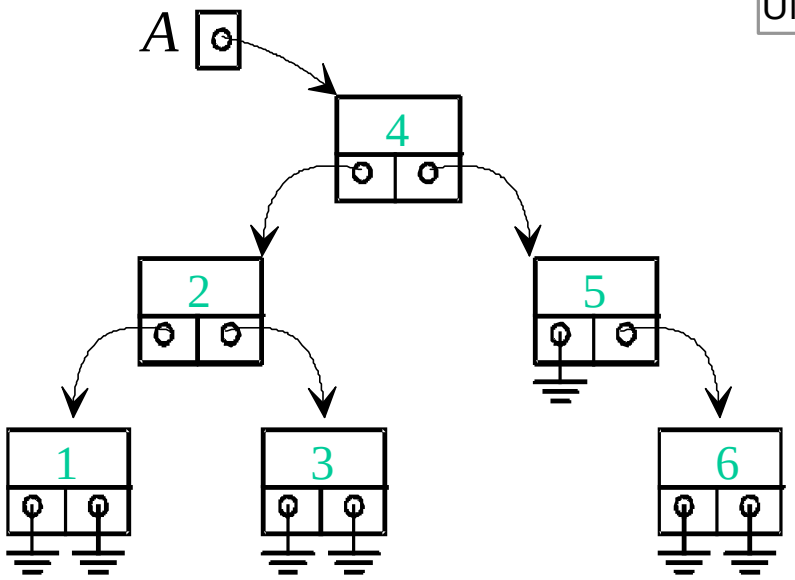


Élément (A, x) =

faux	si A vide
vrai	si $x = y$
Elément (G(A), x)	si $x < y$
Elément (D(A), x)	si $x > y$

Calcul en O(Hauteur(A))

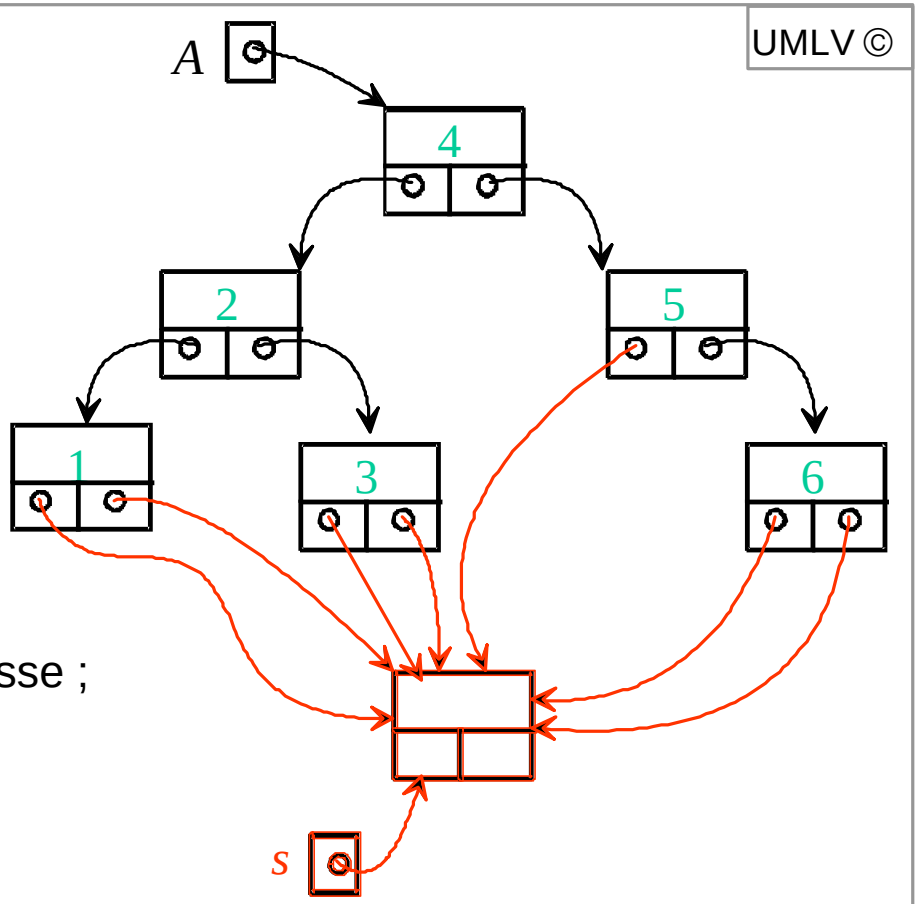
Recherche itérative



fonction Place (*A* abr, *x* élément) : adresse ;

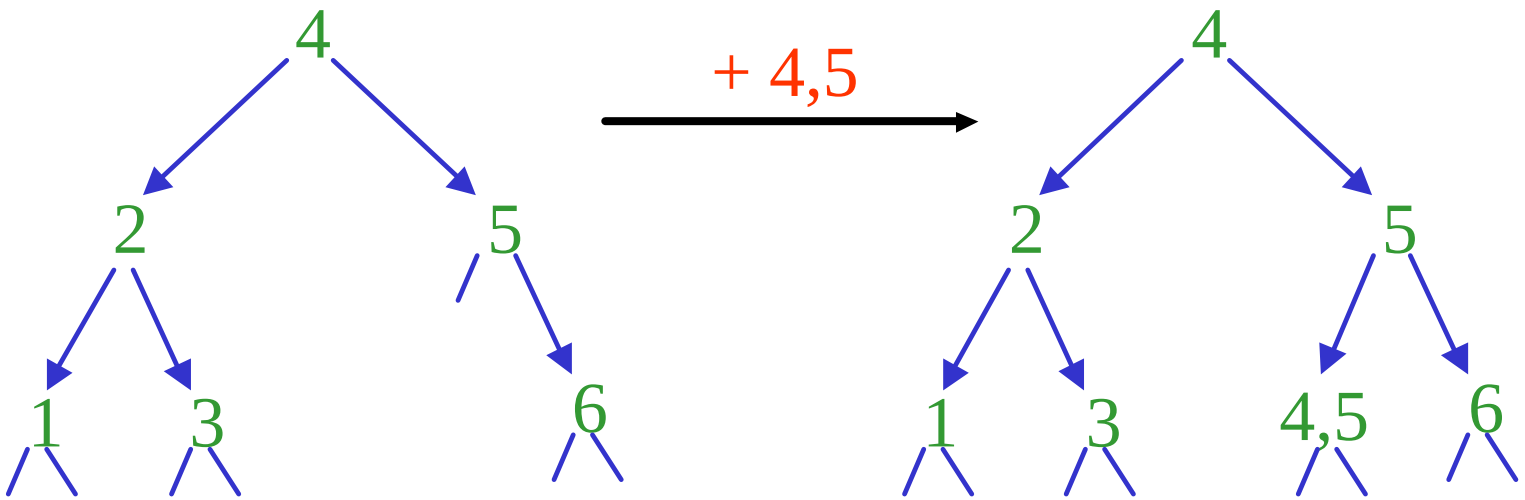
début
  p ← Racine (*A*) ;
  tant que *p* ≠ nil **et** *x* ≠ Elt (*p*) **faire**
      si *x* < Elt (*p*) **alors** *p* ← *p*→*g* ;
                sinon *p* ← *p*→*d* ;
  retour (*p*) ;
fin

avec sentinelle



fonction Place (*A* abr, *x* élément) : adresse ;
/* version itérative, avec sentinelle *s* */
début
    p ← Racine(*A*) ; *s*←Elt ← *x* ;
    tant que *x* ≠ Elt (*p*) **faire**
        si *x* < Elt (*p*) **alors** *p* ← *p*→*g* ;
                sinon *p* ← *p*→*d* ;
    si *p* = *s* **alors** **retour** (nil)
            sinon **retour** (*p*)
fin

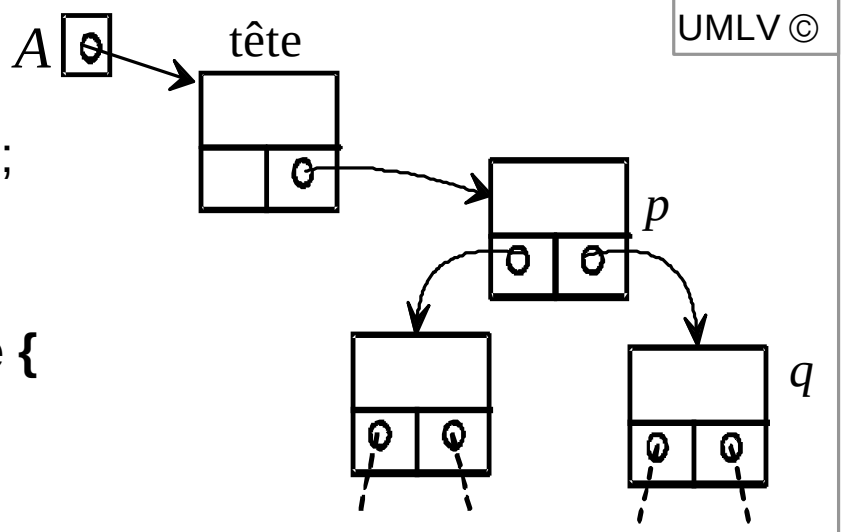
Ajout



$$A + \{x\} = \begin{cases} (x, \wedge, \wedge) & \text{si } A = \wedge \text{ , arbre vide} \\ (r, G + \{x\}, D) & \text{si } x < \text{Elt } (r) \\ (r, G, D + \{x\}) & \text{si } x > \text{Elt } (r) \\ A & \text{sinon} \end{cases}$$

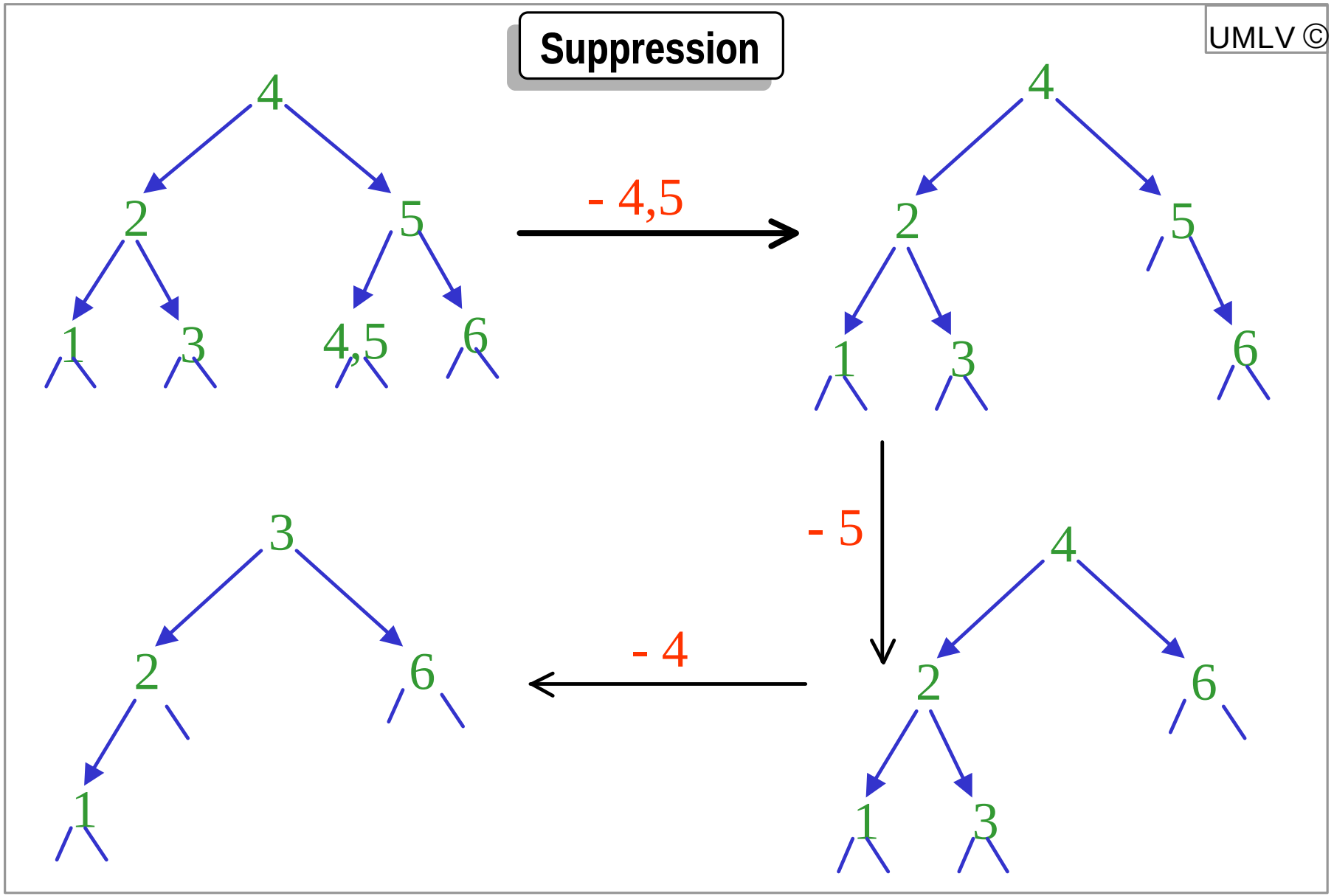
```
fonction Ajouter (A abr, x élément) : abr ;
début
    si Vide (A) alors
        retour (arbre à 1 seul nœud d'étiquette x) ;
    sinon si x < Elt (Racine (A)) alors
        retour (Racine (A), Ajouter(G(A), x), D(A)) ;
    sinon si x > Elt (Racine (A)) alors
        retour (Racine (A), G(A), Ajouter (D(A), x)) ;
    sinon retour (A) ;     /* x = Elt (Racine (A)), rien à faire */
fin
```

avec Tête



Racine (A)
=
$A \rightarrow d$

```
fonction Ajouter (A abr, x élément) : abr ;
début
        p ¬ A ;   q ¬ p→d ;   dir ¬ "d" ;
        tant que  q ≠ nil et x ≠ Elt (q) faire {
            p ¬ q ;
           si  x < Elt (q)  alors
           { q ¬ p→g ;  dir ¬ "g" ; }
           sinon
           { q ¬ p→d ;  dir ¬ "d" ; }
        }
        si q = nil alors {
           créer une feuille q d'étiquette x ;
           si dir = "g"  alors p→g ¬ q  sinon p→d ¬ q ;
        }
        retour (A) ;
fin
```


Suppression
UMLV ©
4
2
5
1
3
4,5
6
- 4,5
4
2
5
1
3
6
- 5
4
2
6
1
3
- 4
3
2
6
1

Suppression

$A = (r, G, D)$

$$A - \{x\} = \begin{cases} (r, G - \{ x \}, D) & \text{si } x < \text{Elt}(r) \\ (r, G, D - \{ x \}) & \text{si } x > \text{Elt}(r) \\ D & \text{si } x = \text{Elt}(r) \text{ et } G \text{ vide} \\ G & \text{si } x = \text{Elt}(r) \text{ et } D \text{ vide} \\ (r, G - \{\text{MAX}(G)\}, D) & \text{sinon} \end{cases}$$

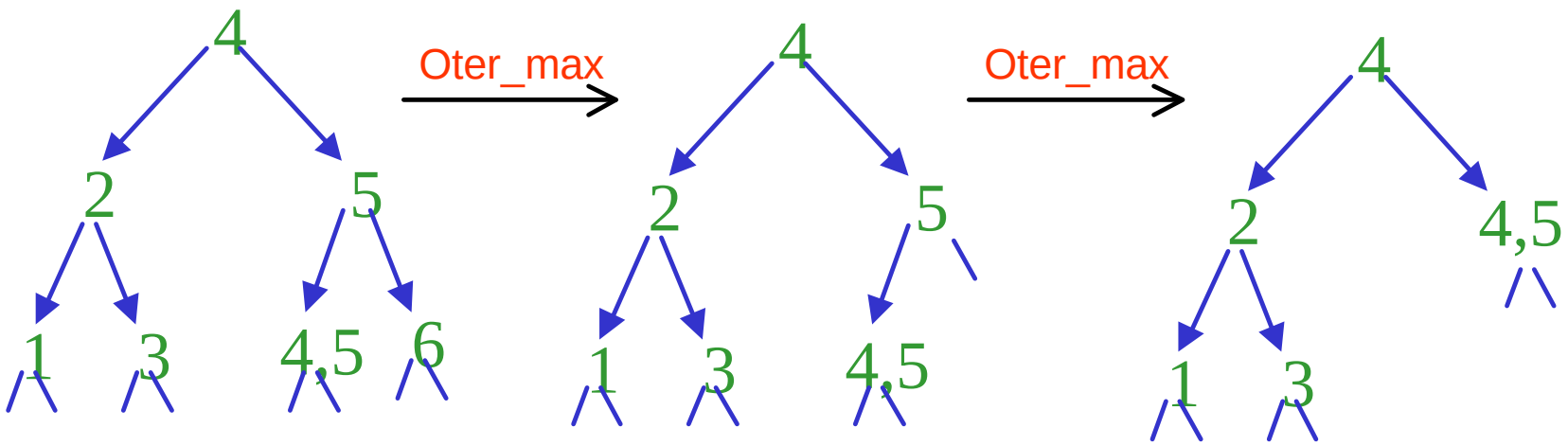
avec Elt (r) = MAX (G)

UMLV ©

```
fonction  Enlever (A abr, x élément) : abr ;
début
   si Vide (A) alors
      retour (A) ;
  sinon si  x < Elt (Racine (A)) alors
      retour (Racine(A), Enlever (G(A), x), D(A))
  sinon si  x >  Elt (Racine (A) ) alors
      retour (Racine (A), G(A), Enlever (D(A),x) )
  sinon si  Vide (G(A)) alors
     retour (D(A))
  sinon si  Vide (D(A)) alors
     retour  (G(A))
  sinon
     {  Elt (Racine (A)) ← MAX (G(A))  ;
        retour ( Racine (A), Oter_max ( G(A) ), D(A) ) ;
     }
fin
```

UMLV ©

Oter_max

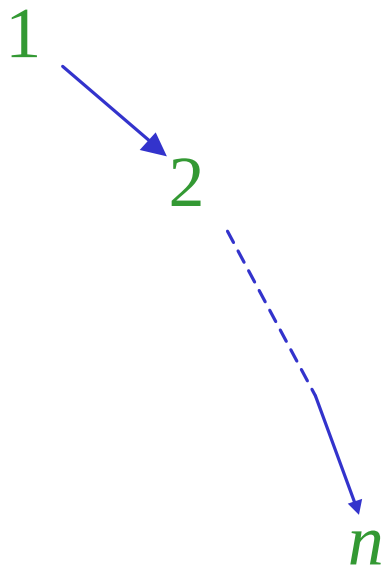


fonction Oter_max (A abr) : abr :
début
    si vide (D (A)) **alors**
        retour (G (A))
    sinon
        retour (Racine(A), G (A), Oter_max (D (A))) ;
fin

Temps maximal

Insertions successives de 1, 2, ..., *n*

dans l'arbre vide



Nombre de comparaisons de clés :

$$0 + 1 + \ldots + n\text{-}1 \quad = \frac{n(n-1)}{2}$$

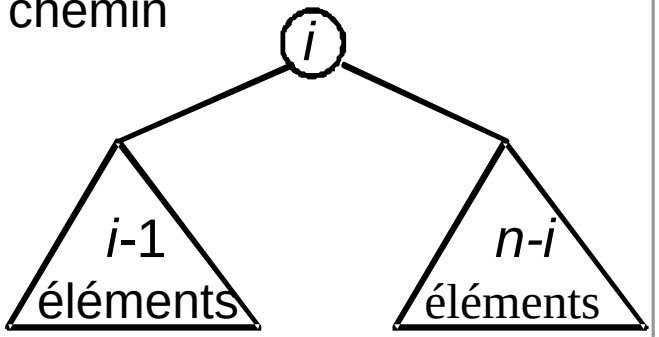
Nombre moyen de comparaisons par insertion : $O(n/2)$

Temps moyen

Insertions successives de *i, j*, … permutation de (1, 2, ..., *n*)

Toutes permutations de (1, 2, ..., *n*) équiprobables.

P(*n*) = moyenne du nombre de noeuds par chemin
= moyenne des 1 + niveau (*i*)



$$P(n) = \frac{1}{n}\sum_{i=1}^{n}\left[\frac{1}{n} + \frac{i-1}{n}\left(P(i-1)+1\right) + \frac{n-i}{n}\left(P(n-i)+1\right)\right]$$

$$= 1 + \frac{1}{n^2}\sum_{i=1}^{n}\left[(i-1)P(i-1) + (n-i)P(n-i)\right]$$

$$= 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} iP(i)$$

$$P(n) \leq 1{,}39 \log_2 n$$

Classement par arbre

fonction TRI (*L* liste) : liste ;

début A ¬ abr_vide ;

pour *x* ¬ premier au dernier élément de *L* **faire**

A ¬ Ajouter (A, *x*) ;

retour (SYM (A)) ;

fin.

Tri par double changement de représentation

Temps :

maximal $0(|L| \log |L|)$ avec abr équilibré

moyen $0(|L| \log |L|)$ avec abr ordinaire

Arbres équilibrés

AVL : abr équilibrés en hauteur

- pour tout nœud p, $-1 \leq |\, h(D(p)) - h(G(p)) \,| \leq 1$

B-arbres

- toutes les feuilles sont au même niveau
- la racine est une feuille ou possède ≥ 2 enfants
- tout autre nœud interne a entre a et b enfants
- et $2 \leq a < b \leq 2a - 1$

B-arbre ordinaire $\rightarrow$ $b = 2a - 1$
2.3-arbre
2.3.4-arbre ou arbre bicolore ou arbre rouge-noir

Queue de priorité

Type abstrait

ensemble d ’éléments comparables

Opérations principales

AJOUTER(*E, x*)

OTER_MIN(*E*)

Implémentations

Liste triées ou non

Arbres partiellement ordonnés

(tournois, tas, *heap*)

Arbres partiellement ordonnés, tas

Implémentation d'ensembles avec

AJOUTER (A, x)
OTERMIN (A)
} $O(\log |A|)$

A arbre binaire

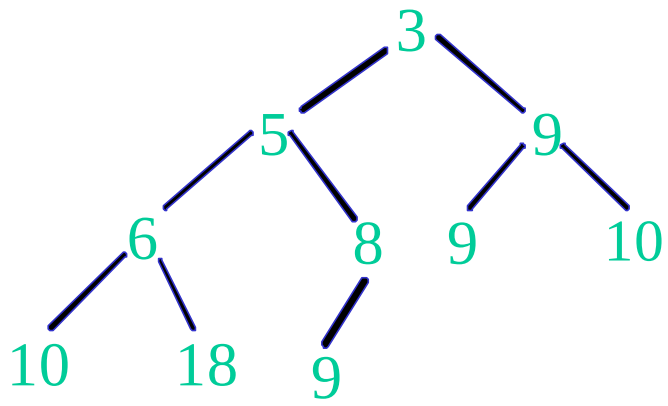
A arbre partiellement ordonné ssi

1 - A est parfait (complet, sauf peut-être au dernier niveau) et

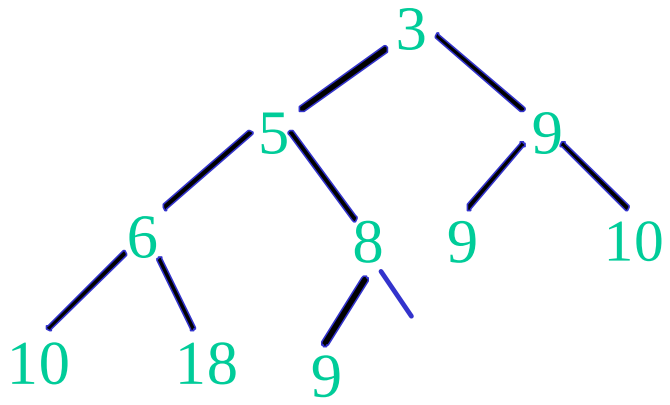
2 - pour tout noeud $p \neq$ racine

Elt (p) ≥ Elt (Parent (p))

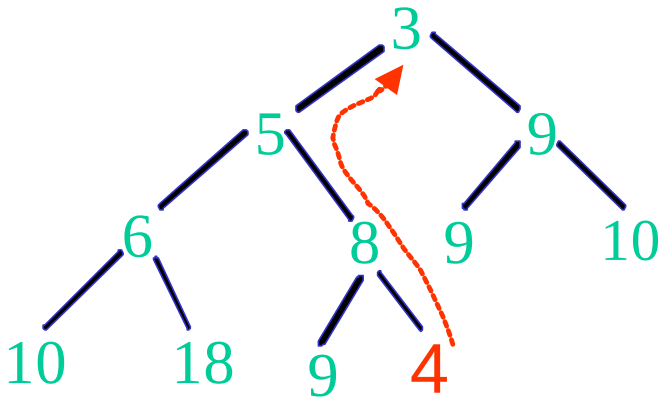
Hauteur(A) = $\lfloor \log_2 | A | \rfloor$

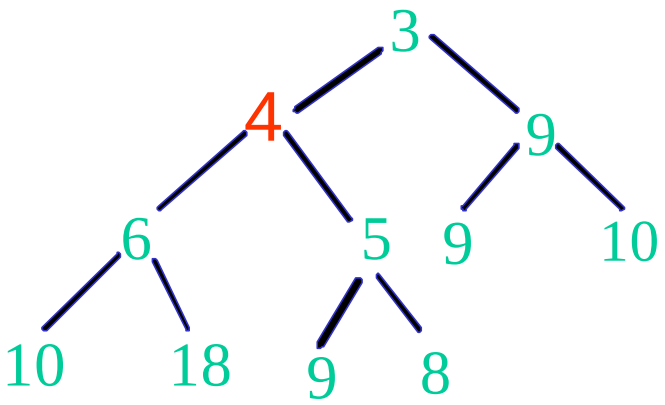


Ajout

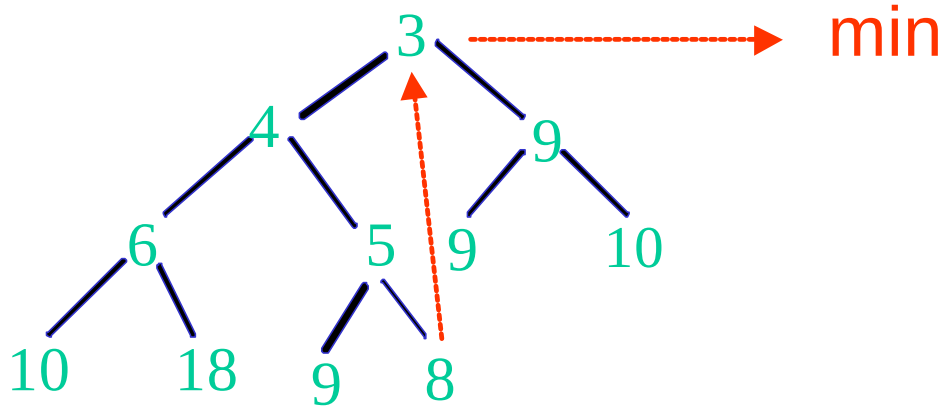


AJOUTER (A, 4)

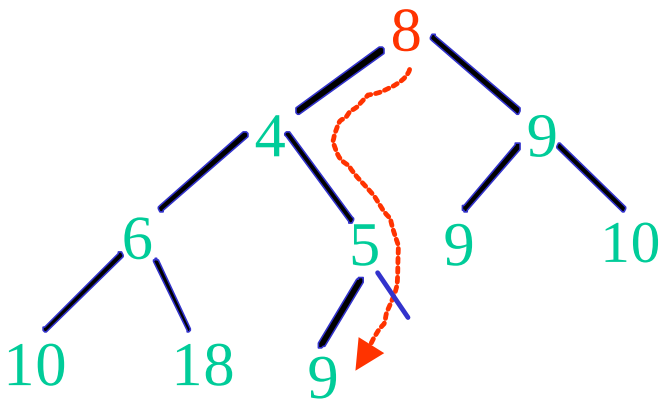


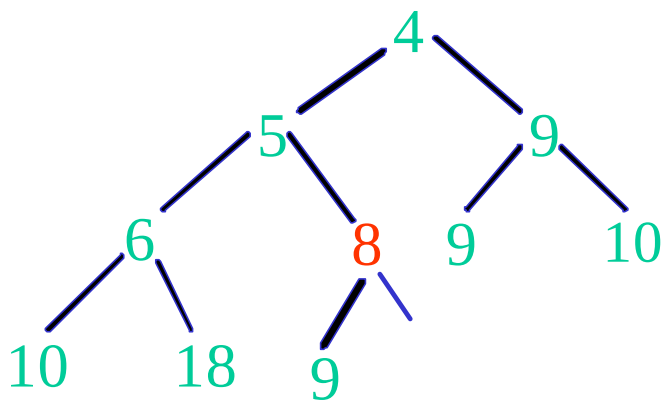


Suppression du minimum

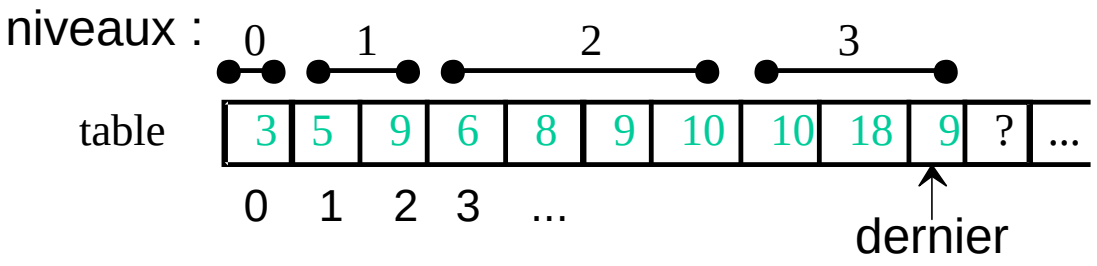


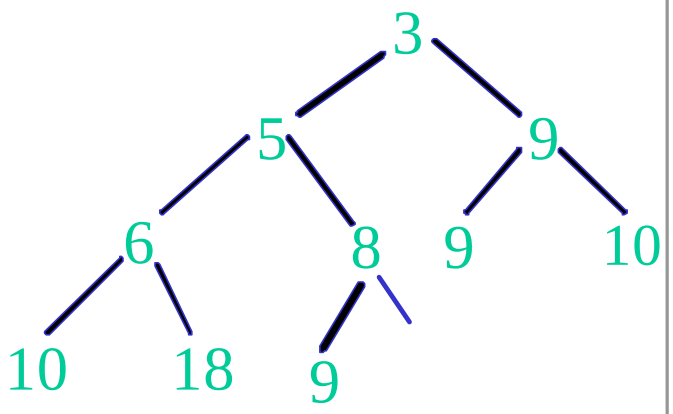
OTER_MIN (A)





Implémentation en table





parent (i) = $\lfloor (i-1)/2 \rfloor$ si $i > 0$

enfantgauche (i) = $2i+1$ si $2i+1 \leq$ dernier

enfantdroit (i) = $2i+2$ si $2i+2 \leq$ dernier

```
typedef struct {
     element table[MAX];
     int dernier;
} Queuedepriorité;
```

UMLV ©

```
fonction Ajouter (t tas, x élément) : tas ;
début
    t.dernier ¬ i ¬ t.dernier + 1 ;  t[i] ¬ x ;
    tant que i > 1 et t.table[i] < t.table[⌊ (i-1)/2 ⌋] faire {
            échanger t.table[i] et t.table[⌊ (i-1)/2 ⌋] ;  i ¬ ⌊ (i-1)/2 ⌋ ;
    }
    retour (t) ;
fin
fonction Oter_min (t tas non vide) : tas ;
début
    t.dernier ¬ d ¬ t.dernier - 1 ;  t.table[0] ¬ t.table [d+1] ;  i ¬ 0 ;
    répéter
            fin ¬ vrai ;
            si 2i+2 ≤ d alors {
                    si t.table [2i+1] < t.table [2i+2] alors k ¬ 2i+1 sinon k ¬ 2i+2 ;
                    si t[i] > t.table[k] alors {
                        échanger t.table[i] et t.table[k] ; i ¬ k ; fin ¬ faux ;
                    } sinon si 2i+1= d et t.table[i] > t.table[d] alors {
                    échanger t.table[i] et t.table[d] ;
            }
    jusqu'à fin ; retour (t) ;
fin
```

Tri par tas

fonction TRI (*L* liste) : liste ;
début $A \leftarrow$ tas_vide ;
  pour $x \leftarrow$ premier au dernier élément de *L* **faire**
    $A \leftarrow$ Ajouter (A, x) ;
  $L' \leftarrow$ liste_vide ;
  tant que non Vide (A) **faire {**
    $L' \leftarrow L' \cup (\mathrm{MIN}(A))$; $A \leftarrow$ Oter_min(A) ;
  }
  retour (L') ;
fin

Temps maximum $O(|L| \log |L|)$

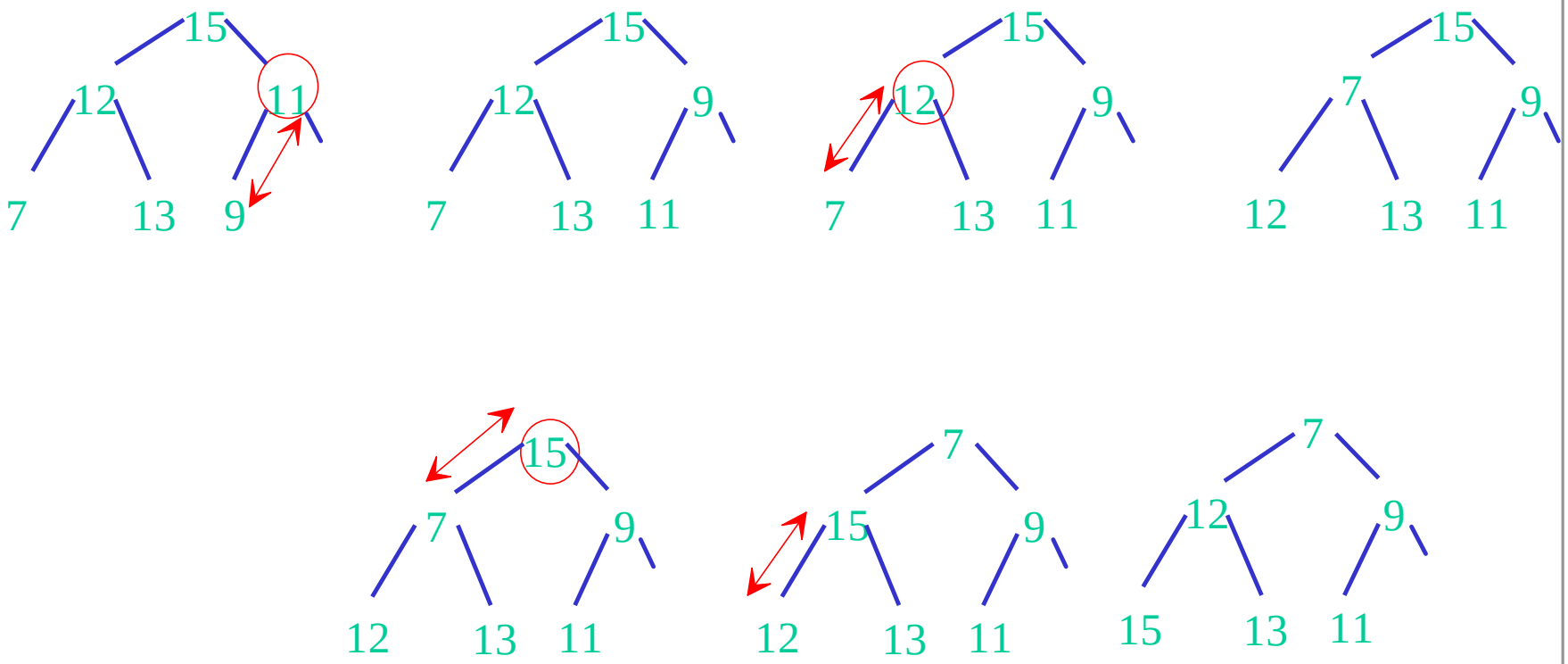
Temps du "pour"

$\Theta(|L| \log |L|)$ avec $L = (n, n\text{-}1, ..., 3, 2, 1)$

Mise en tas globale

Création d'un tas avec *n* éléments déjà dans la table

Transformation d ’un arbre binaire parfait en tas



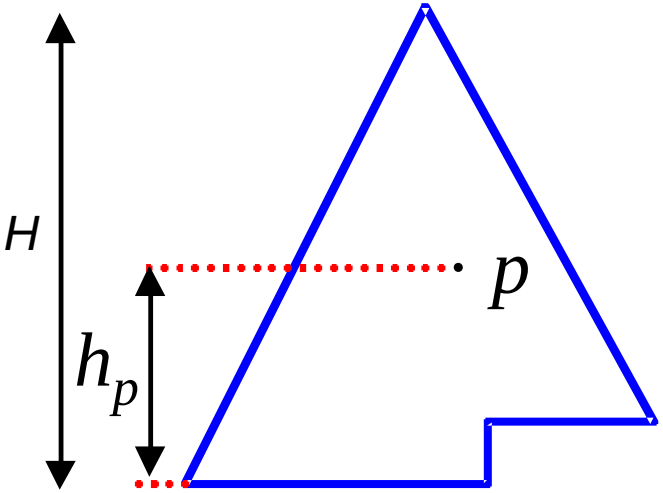
UMLV ©

```
fonction Mise_en_tas (tab [0 .. n-1] table ) : tas ;
début
    pour i ¬ ⌊(n-2)/2⌋ à 0 pas -1 faire
        Entas (tab, i) ;
    retour (tab,n-1) ;
fin

procédure Entas (tab [0 .. n-1] table, i indice) ;
/* les sous-arbres de racines k sont des tas, pour i< k ≤ n-1 */
début
    si 2i+2 = n ou tab[2i+1] ≤ tab[2i+2] alors k ¬ 2i+1 sinon k ¬ 2i+2
    si tab[i ] > tab[k] alors {
        échanger tab[i ] et tab[k ] ;
        si k ≤ ⌊ (n-2)/2⌋ alors Entas (tab, k) ;
    }
fin
```

Temps maximum de Entas(t, i) = $O(\text{Hauteur}_A(i))$

Temps linéaire



Temps (Entas, p) $\leq \alpha\, h_p$

nombre d 'échanges $\leq h_p$

nombre de comparaisons $\leq 2h_p$

$2^H \leq n \leq 2^{H+1}\text{-}1 \qquad H = \lfloor \log_2 n \rfloor$

L'algorithme peut passer beaucoup de temps sur les noeuds proches de la racine qui sont peu nombreux

mais il passe peu de temps sur les éléments profonds dans l'arbre qui sont très nombreux

Temps (Mise_en_tas, *n*)

$$\leq a\,H1 + a\,(H-1)2 + \ldots + a\,(H-i)2^i + \ldots + a\,1\,2^{H-1}$$

$$\leq a\sum_{i=0}^{H-1}(H-i)2^i = a\,2^H\sum_{i=0}^{H-1}\frac{H-i}{2^{H-i}} \leq a\,n\sum_{i=0}^{H-1}\frac{H-i}{2^{H-i}} = a\,n\sum_{k=1}^{H}\frac{k}{2^k} \leq 2a\,n$$

$$\sum_{k=1}^{\infty}\frac{k}{2^k} = \frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\ldots = 2$$

$$= \frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\ldots = 1$$

$$+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\ldots = \frac{1}{2}$$

$$+\frac{1}{8}+\frac{1}{16}+\ldots = \frac{1}{4}$$

$$+\ldots = \ldots$$

Théorème : l'algorithme de mise en tas fonctionne en temps $O(n)$