1 Généralités sur les arbres

1.1 Graphes et arbres

Définition 1.1. *On appelle graphe un couple $G = (V, E)$ d'un ensemble fini V (les sommets ou noeuds) et d'une partie E de $V \times V$ (les arêtes). Si $x, y \in V$, on note $x \to y$ pour $(x, y) \in E$. On dit que le graphe est non orienté si*

$$\forall x, y \in V, \quad x \to y \iff y \to x$$

Définition 1.2. *Soit G un graphe non orienté. Une chaîne dans un graphe est une suite de sommets reliés par des arêtes. La longueur d'une chaîne est le nombre d'arêtes utilisées, ou, ce qui revient au même, le nombre de sommets utilisés moins un. Une chaîne est dite simple si elle ne visite pas deux fois le même sommet (sauf éventuellement en ses extrémités).*

Définition 1.3. *Soit G un graphe non orienté. On appelle cycle une chaîne simple dont les extrémités coïncident, de longueur différente de 2. On ne rencontre pas deux fois le même sommet, sauf celui choisi comme sommet de départ et d'arrivée.*

Définition 1.4. *Soit G un graphe non orienté. La relation $\mathcal{R}$ définie par $a\mathcal{R}b$ signifie "il existe une chaîne d'origine a et d'extrémité b" est une relation d'équivalence sur l'ensemble V. Les classes d'équivalence sont appelées les composantes connexes du graphe. on dit que le graphe est connexe s'il possède une seule classe d'équivalence.*

Théorème 1.1. *Pour tout graphe ayant m arêtes, n sommets et p composantes connexes, on a $m - n + p \geq 0$. De plus, on a $n = m + p$ si et seulement si G est sans cycle.*

Démonstration. Par récurrence sur le nombre n d'arêtes. Si $n = 0$, G comporte exactement n composantes connexes, donc $m = p$ et le résultat est vérifié. Soit G un graphe ayant m arêtes, n sommets et p composantes connexes et supprimons une arête $a \leftrightarrow b$. Le nouveau graphe G_1 comporte $m_1 = m - 1$ arêtes, $n_1 = n$ sommets et p ou $p + 1$ composantes connexes (la composante connexe commune de a et b a pu exploser en 2). Par hypothèse de récurrence, on a $m_1 + p_1 \geq n_1$ soit encore $m - 1 + p_1 \geq n$, soit encore $m + (p_1 - 1) \geq n$. Mais p_1 est soit égal à p, auquel cas $m + p - 1 \geq n$ et a fortiori $m + p \geq n$, soit à $p + 1$ auquel cas $m + p \geq n$, ce qui achève la récurrence.

. Supposons maintenant que l'on a égalité. On se trouve forcément dans le second cas, ce qui montre que, dès que l'on retire une arête $a \leftrightarrow b$, le nombre de composantes connexes augmente de 1, autrement dit aucune arête ne peut faire partie d'un cycle, donc le graphe est sans cycle.

Inversement, supposons le graphe sans cycle. Alors le graphe G_1 est sans cycle et par récurrence vérifie $m_1 + p_1 = n_1$. Mais le graphe, G ne comportant pas de cycle, les composantes connexes de a et b dans G_1 sont disctinctes, donc $p_1 = p + 1$. On a donc $m + p = n$.

Le théorème précédent justifie l'équivalence des propriétés suivantes qui caractérisent un arbre :

Définition 1.5. *On dit qu'un graphe non orienté $G = (V, E)$ est un arbre s'il vérifie les propriétés équivalentes*
(i) G est sans cycle et connexe.
(ii) G est sans cycle et $|V| = |E| + 1$
(iii) G est connexe et $|V| = |E| + 1$

Définition 1.6. *On appelle forêt un graphe non orienté sans cycle. Dans ce cas, ses composantes connexes sont des arbres.*

Remarque 1.1. Dans un arbre, il existe un unique chemin reliant un sommet a à un sommet b.

1.2 Arbres enracinés

Définition 1.7. *On appelle arbre enraciné tout couple (r, G) d'un arbre $G = (V, E)$ et d'un sommet $r \in V$ (appelé la racine de l'arbre).*

Définition 1.8. *Soit (r, V, E) un arbre enraciné, $x, y \in V$ distincts. On dit que y est un descendant de x (ou que x est un ascendant de y) s'il existe un chemin (nécessairement unique) de r à y passant par x.*

Remarque 1.2. La relation "x est un ascendant de y" est visiblement une relation d'ordre strict partielle sur V.

Définition 1.9. *On dit que x est père de y (ou que y est fils de x) si x est un ascendant de y tel que $x \leftrightarrow y$.*

Remarque 1.3. Tout élément x de V distinct de r possède un unique père. C'est le dernier sommet visité dans l'unique chemin reliant la racine à x.

Définition 1.10. *On appelle feuille de l'arbre (ou encore noeud externe), un noeud sans descendant. On appelle noeud interne de l'arbre, un noeud qui possède des descendants.*

Remarque 1.4. Considérons $G = (r, V, E)$ un arbre enraciné. Soit $V_1 = V \setminus \{r\}$ et $E_1 = E \setminus \{r \leftrightarrow x \mid r \leftrightarrow x \in E\}$. Alors (V_1, E_1) est une forêt, chacun des arbres qui la compose comportant un et un seul fils de r. On peut donc considérer que cette forêt est une forêt d'arbres enracinés par les fils de r. Ceci nous conduit à la définition inductive suivante :

Définition 1.11. *On appelle ensemble des arbres enracinés l'ensemble défini inductivement par*
(i) Tout singleton est un arbre enraciné (réduit à une feuille)
(ii) Toute réunion disjointe d'arbres enracinés est une forêt enracinée
(iii) Tout couple (r, F) d'un élément r et d'une forêt enracinée F est un arbre enraciné.

Définition 1.12. *Soit $A = (r, V, E)$ un arbre enraciné, V_i l'ensemble de ses noeuds internes et V_e l'ensemble de ses feuilles ou noeuds externes. On appelle étiquetage de A tout couple (f, g) d'une application f de V_i dans un ensemble N et d'une application g de V_e dans un ensemble F. On dit que l'étiquetage est homogène si $N = F$, hétérogène si $N \neq F$.*

Définition 1.13. *On dit qu'un arbre enraciné est ordonné si, pour tout noeud interne, l'ensemble de ses fils est totalement ordonné (de l'aîné au cadet).*

Définition 1.14. *Dans toute la suite de ce chapitre, on désignera par arbre hétérogène (resp. homogène) un arbre enraciné ordonné muni d'un étiquetage hétérogène (resp. homogène).*

1.3 Représentation graphique d'un arbre

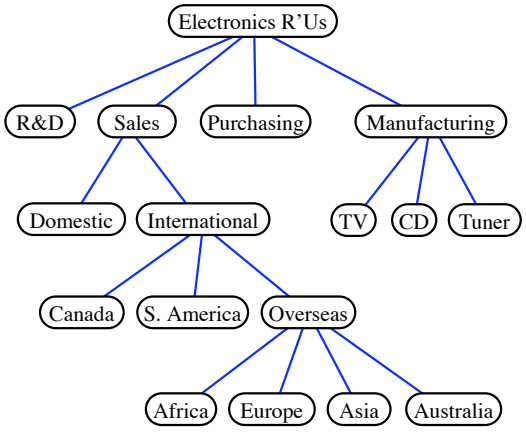
Il est d'usage de ne pas faire figurer dans les arbres les noms des éléments, mais au contraire d'y faire figurer leur étiquette (qui peut être de nature différente suivant que les noeuds de l'arbre sont externes ou internes).

On adopte une représentation *généalogique* de l'arbre avec la racine en haut, ses fils en dessous ordonnés de gauche à droite, les fils des fils en dessous, et ainsi de suite.

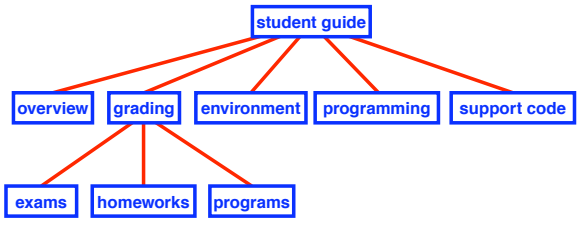
1.4 Exemples d'arbres

L'organigramme d'une société :

- organigramme d'une société

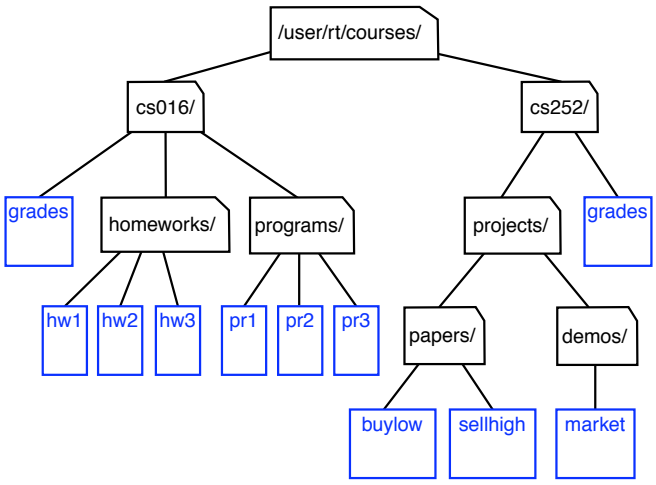


- table des matières d'un livre

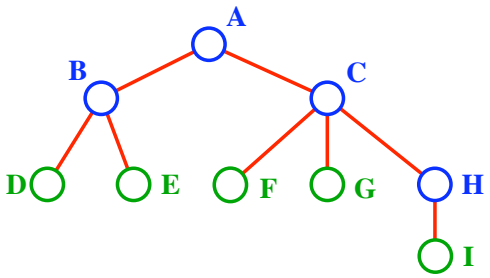


La table des matières d'un livre :
Le contenu d'un disque dur

• Système de fichiers Unix ou DOS/Windows



1.5 Terminologie



- A est la racine de l'arbre
- B est le père de D et E.
- C est le frère de B
- D et E sont les fils de B.
- D, E, F, G, I sont des noeuds externes ou des feuilles
- A, B, C, H sont des noeuds internes.
- La profondeur du noeud E est2
- La hauteur de l'arbre est 3.
- Le degré du noeud B est2.

1.6 Typages des arbres hétérogènes en Caml

Les différents typages diffèrent essentiellement en la manière de typer les forêts comme ensembles d'arbres :
- totalement dynamique : sous forme de liste d'arbres
- semi-dynamique : sous forme de tableaux d'arbres
- statique : sous forme de n-uplet (sous-entend que le degré des noeuds internes est fixe ou au moins majoré)

Caml

```
(*définition d'un arbre hétérogène à l'aide de listes *)
type ('f,'n) arbre =
    | Feuille of 'f
    | Noeud of 'n*(('f,'n) foret)
and ('f,'n) foret =
     Arbres of ('f,'n) arbre list;;
(*définition d'un arbre hétérogène à l'aide de tableaux *)
type ('f,'n) arbre =
    | Feuille of 'f
```

```
    | Noeud of 'n*(('f,'n) foret)
and ('f,'n) foret =
     Arbres of ('f,'n) arbre vect;;
```

Enfin, si p est un entier fixé et si tous les noeuds de l'arbre ont le même degré p, il peut être intéressant de représenter les ensembles d'arbres à p éléments sous formes de p-uplets.

Définition 1.15. *On appelle arbre binaire (resp. ternaire) un arbre dont tous les noeuds ont pour degré 2 (resp. 3).*

On pourra implémenter les arbres binaires hétérogènes en Caml, en choisissant de mettre un fils à gauche et un fils à droite de l'étiquette, de la façon suivante :

Caml

```
#type ('f,'n) arbre_bin =
                    Feuille of 'f
                  | Noeud of (('f,'n) arbre_bin) * 'n * (('f,'n) arbre_bin);;
Type arbre_bin defined.
```

Dans le cas où l'on souhaite un arbre dans lequel les feuilles, les noeuds et les branches sont modifiables en place, on peut préférer des types enregistrement modifiables :

Caml

```
#type  ('f,'n) arbre   =
                  | Feuille of 'f t_feuille
                  | Noeud of ('f,'n) t_noeud
and    'f t_feuille   = { mutable etiq_feuille :'f}
and   ('f,'n) t_noeud = { mutable etiq_noeud : 'n ;
                          mutable branches : ('f,'n) arbre vect};;
```

1.7 Noeuds, feuilles, arêtes

Théorème 1.2. *Tout arbre à n noeuds (internes ou externes) possède $n-1$ arêtes.*

Démonstration. par induction structurelle. Si l'arbre A est réduit à une feuille, il a 1 sommet et 0 arêtes et le résultat est vérifié. Sinon, soit n le nombre de sommets de A, a_0 la racine de A, $A_1, \ldots, A_p$ les branches issues de A ayant pour nombre de sommets respectifs $n_1, \ldots, n_p$. On a $n = n_1 + \ldots + n_p + 1$. Par hypothèse d'induction, chaque arbre A_i possède $n_i - 1$ arêtes. Or les arêtes de A sont d'une part les arêtes des A_i, d'autre part les p arêtes reliant a_0 aux racines des A_i. En conséquence, A possède $p + (n_1 - 1) + \ldots + (n_p - 1) = n_1 + \ldots + n_p = n - 1$ arêtes, ce qui achève la démonstration.

Théorème 1.3. *Soit A un arbre binaire hétérogène. Si A possède p noeuds internes, il possède $p+1$ feuilles.*

Démonstration. par induction structurelle. Si l'arbre est réduit à une feuille, il possède 0 noeuds et 1 feuille et la propriété est vérifiée. Sinon, soit a_0 la racine de A, n le nombre de sommets de A, A_1 et A_2 les deux branches partant de a_0, n_1 et n_2 le nombre de leurs noeuds, si bien que $n = n_1 + n_2 + 1$. Par hypothèse d'induction, A_1 possède $n_1 + 1$ feuilles et A_2 possède $n_2 + 1$ feuilles, donc A possède $n_1 + n_2 + 2 = n + 1$ feuilles, ce que l'on voulait démontrer.

1.8 Implémentation des opérations élémentaires sur les arbres hétérogènes

Une première opération souvent nécessaire est le calcul du nombre de noeuds d'un arbre. Prenons tout d'abord le cas d'un arbre hétérogène dont les noeuds sont étiquetés par N et les feuilles étiquetées par F, défini par :

- si x est un élément de F, alors x est un arbre (réduit à une feuille)
- toute famille d'arbres est une forêt
- tout couple formé d'un élément y de N et d'une forêt est un arbre

Le calcul du nombre de noeuds peut se faire par induction structurelle :

- si l'arbre est réduit à une feuille, il possède 0 noeuds
- le nombre de noeuds d'une forêt est la somme du nombre de noeuds des arbres qui la composent
- le nombre de noeuds d'un arbre non réduit à une feuille est le nombre de noeuds de la forêt des branches de sa racine augmenté de 1 (le noeud racine)

Dans le cas où l'on implémente les forêts comme des listes d'arbres, ceci conduit à une fonction Caml doublement récursive, une première récursivité étant due à la structure d'arbre, la deuxième à la structure de liste de la forêt :

Caml

```
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre list);;

#let rec nb_noeuds_arbre = function
                |  Feuille _ -> 0
                |  Noeud ( _ , branches) -> 1 + (nb_noeuds_foret branches)
and nb_noeuds_foret = function
                |  [] -> 0
                |  t :: q -> (nb_noeuds_arbre t) + (nb_noeuds_foret  q);;
```

Dans le cas où l'on implémente les forêts comme des tableaux d'arbres, il est plus naturel d'introduire une référence pour calculer le nombre de noeuds d'une forêt :

Caml

```
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre vect);;

#let rec nb_noeuds_arbre = function
                  |  Feuille _ -> 0
                  |  Noeud ( _ , branches) -> 1 + (nb_noeuds_foret branches)
and nb_noeuds_foret f =
        let nb = ref 0 in (* on initialise à 0 le nombre de noeuds de la forêt *)
        for i = 0 to (vect_length f) -1 do
            (* on ajoute le nombre de noeuds *)
             nb := !nb + nb_noeuds_arbre (f.(i))
        done;
        !nb ;;
```

Enfin dans le cas d'un arbre binaire hétérogène, on peut se passer de la fonction calculant le nombre de noeuds d'une forêt et écrire directement

Caml

```
#type ('f,'n) arbre_bin =
                  | Feuille of 'f
                  | Noeud of (('f,'n) arbre_bin * 'n * ('f,'n) arbre_bin);;
#let rec nb_noeuds_arbre = function
                  |  Feuille _ -> 0
                  |  Noeud ( gauche, _ , droite) ->
                       1 + (nb_noeuds_arbre gauche)+(nb_noeuds_arbre droite) ;;
nb_noeuds_arbre : ('a, 'b) arbre_bin -> int = <fun>
```

Le calcul du nombre de feuilles se fait suivant une méthode tout à fait similaire

- si l'arbre est réduit à une feuille, il possède 1 feuille
- le nombre de feuilles d'une forêt est la somme du nombre de feuilles des arbres qui la composent
- le nombre de feuilles d'un arbre non réduit à une feuille est le nombre de feuilles de la forêt des branches de sa racine

Si les forêts sont implémentées comme des listes :

Caml

```
#type ('f,'n) arbre=
                    Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre list);;

#let rec nb_feuilles_arbre = function
                    Feuille _ -> 1
                |  Noeud ( _ , branches) -> (nb_feuilles_foret branches)
and nb_feuilles_foret = function
                    [] -> 0
                |  t :: q -> (nb_feuilles_arbre t) + (nb_feuilles_foret  q);;
```

Si les forêts sont implémentées comme des tableaux :

Caml

```
#type ('f,'n) arbre=
                    Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre vect);;

#let rec nb_feuilles_arbre = function
                     Feuille _ -> 1
                  |  Noeud ( _ , branches) -> (nb_feuilles_foret branches)
and nb_feuilles_foret f =
          let nb = ref 0 in
               (* on initialise à 0 le nombre de feuilles de la forêt *)
          for i = 0 to (vect_length f) -1 do
               (* on ajoute le nombre de feuilles *)
                nb := !nb + nb_feuilles_arbre (f.(i))
          done;
          !nb ;;
```

Dans le cas d'un arbre binaire :

Caml

```
#type ('f,'n) arbre_bin =
                  | Feuille of 'f
                  | Noeud of (('f,'n) arbre_bin * 'n * ('f,'n) arbre_bin);;

#let rec nb_feuilles_arbre = function
                  |  Feuille _ -> 1
                  | Noeud ( gauche, _ , droite) ->
                      (nb_feuilles_arbre gauche) + (nb_feuilles_arbre droite) ;;
```

Quant à la hauteur d'un arbre hétérogène, par induction structurelle on obtient
- si l'arbre est réduit à une feuille, il est de hauteur 0
- la hauteur d'une forêt est le maximum des hauteurs des arbres qui la composent
- la hauteur d'un arbre non réduit à une feuille est la hauteur de la forêt des branches de sa racine augmentée de 1

Si les forêts sont implémentées comme des listes :

Caml

```
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre list);;

#let rec hauteur_arbre = function
                |  Feuille _ -> 0
                |  Noeud ( _ , branches) -> 1 + (hauteur_foret branches)
and hauteur_foret = function
                |  [] -> 0
                |  t :: q -> max (hauteur_arbre t) (hauteur_foret  q);;
```

Si les forêts sont implémentées comme des tableaux, on utilise une référence :

Caml

```
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre vect);;

#let rec hauteur_arbre = function
                  |  Feuille _ -> 0
```

```
                  |  Noeud ( _ , branches) -> 1 + (hauteur_foret branches)
and hauteur_foret f =
        let nb = ref 0 in
           (* on initialise à 0 le nombre de feuilles de la forêt *)
        for i = 0 to (vect_length f) -1 do
                (* on ajoute le nombre de feuilles *)
                nb := max !nb  (hauteur_arbre f.(i))
        done;
        !nb ;;
```

Dans le cas d'un arbre binaire, on peut écrire :

Caml

```
#type ('f,'n) arbre_bin =
                  | Feuille of 'f
                  | Noeud of (('f,'n) arbre_bin * 'n * ('f,'n) arbre_bin);;

#let rec hauteur_arbre = function
                  |  Feuille _ -> 0
                  |  Noeud ( gauche, _ , droite) ->
                        1 + max (hauteur_arbre gauche) (hauteur_arbre droite);;
```

2 Arbres binaires

2.1 Arbres binaires hétérogènes

Un arbre binaire possède des noeuds et des feuilles. Certains disent plutôt noeuds internes et noeuds externes. On peut définir la structure d'arbre binaire de façon récursive. Si N (resp. F) désigne l'ensemble des valeurs des noeuds (resp. des valeurs des feuilles), l'ensemble A(N,F) des arbres binaires est défini par : - toute feuille est un arbre : $F \subset A(N,F)$; - si α et β sont deux arbres, et si $n \in N$, alors (n, α, β) est un arbre.

Le typage Caml correspondant est le suivant :

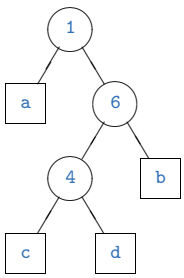
Caml

```
type ('n,'f) arbre_binaire =
      | Feuille of 'f
      | Noeud of 'n * ('n,'f) arbre_binaire * ('n,'f) arbre_binaire ;;
```

Un exemple d'arbre binaire du type `(int,string) arbre_binaire` est :

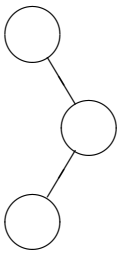
```
Noeud(1,Feuille "a", Noeud(6,Noeud(4,Feuille "c",
Feuille "d")  Feuille "b"))
```

qui correspond au dessin suivant, où on a choisi de représenter les noeuds par des ronds et les feuilles par des carrés.



2.2 Squelette d'un arbre

Le squelette d'un arbre définit sa géométrie : on l'obtient en supprimant toute information aux noeuds et feuilles, et en supprimant les feuilles. Voici le squelette de l'arbre précédent :



On type aisément les squelettes d'arbres binaires ainsi :

Caml

```
type squelette_binaire =
   | Rien
   | Jointure of squelette_binaire * squelette_binaire ;;
```

Quelques fonctions sur les squelettes d'arbres binaires :

Caml

```
let rec décharne = function
   | Feuille _ -> Rien
   | Noeud(_,g,d) -> Jointure(décharne g, décharne d) ;;

let rec symétrie = function
   | Rien -> Rien
   | Jointure(g,d) -> Jointure(symétrie d,symétrie g) ;;

let rec égalité_squelettes a b = match (a,b) with
   | (Rien,Rien) -> true
   | (Jointure(g,d),Jointure(g',d'))
                -> (égalité_squelettes g g') && (égalité_squelettes d d')
   | _ -> false ;;
```

2.3 Propriétés combinatoires

Vocabulaire Nous appellerons taille d'un arbre binaire le nombre de ses noeuds internes. C'est aussi la taille de son squelette. On peut donc écrire :

Caml

```
let rec taille = function
    | Feuille -> 0
   | Noeud(_,g,d) -> 1 + (taille g) + (taille d) ;;
```

Sa hauteur (on parle aussi de profondeur) est définie inductivement par : toute feuille est de hauteur nulle, la hauteur d'un arbre (n, g, d) est égale au maximum des hauteurs de ses fils g et d augmenté d'une unité : la hauteur mesure donc l'imbrication de la structure.

Caml

```
let rec hauteur = function
    | Feuille -> 0
    | Noeud(_,g,d) -> 1 + (max (hauteur g) (hauteur d)) ;;
```

Le nombre de feuilles se déduit facilement du nombre de noeuds, c'est-à-dire de la taille :

Théorème 2.1 (Feuilles et noeuds d'un arbre binaire)**.** *Le nombre de feuilles d'un arbre binaire de taille n est égal à n + 1.*

Démonstration. D'aucuns peuvent utiliser une preuve inductive. Je préfère dire que si f est le nombre de feuilles, $n + f$ compte le nombre de noeuds et feuilles qui sont, sauf la racine, fils d'un noeud interne : $n + f - 1 = 2n$. D'où le résultat : $f = n + 1$.

Taille et profondeur d'un arbre binaire sont étroitement liés.

Théorème 2.2 (Hauteur d'un arbre binaire)**.** *Soit h la hauteur d'un arbre binaire de taille n. On dispose de :* $1+\lceil \lg n \rceil \leq h \leq n$.

Démonstration. La hauteur est la longueur du plus long chemin de la racine à une feuille. Au lieu de compter les arêtes, on peut compter les noeuds (internes)de départ, et on a bien $h \leq n$. Un arbre binaire de hauteur h est dit complet si toutes ses feuilles sont à la profondeur h ou $h-1$. On vérifie facilement pour un tel arbre la relation $h = 1 + \lceil \lg n \rceil$.

A un arbre binaire non complet on peut ajouter suffisament de noeuds (et de feuilles) pour qu'il soit complet : la taille passe alors de n à $n' = 2^h - 1 \geq n$, donc $\lceil \lg n \rceil < h$ et ainsi $\lceil \lg n \rceil + 1 \leq h$.

On en déduit aussitôt le

Corollaire 2.1. *Soit h la hauteur d'un squelette binaire de taille n. On dispose de :* $\lceil \lg n \rceil \leq h \leq n-1$.

Ici encore les bornes sont optimales (c'est-à-dire qu'il existe des squelettes pour lesquels elles sont atteintes).

Une fonction Caml qui renvoie un squelette binaire complet de taille donnée.

Caml

```
let rec squelette_complet = function
    | 0 -> Rien
    | n -> let g = squelette_complet ((n - 1)/2)
             and d = squelette_complet (n - 1 - (n - 1)/2)
             in Jointure(g,d) ;;
```

Une fonction Caml qui décide si un squelette binaire est ou non complet.

Caml

```
let rec est_complet = function
    | Rien -> true
    | Jointure(g,d)
              -> let hg = hauteur g and hd = hauteur d
                  in if hg = hd then
                         (est_complet g) && (est_complet d)
                  else if hg = 1 + hd then
                        (est_complet g) && (est_vraiment_complet d)
                 else if hg + 1 = hd then
                        (est_vraiment_complet g) && (est_complet d)
                 else false
and est_vraiment_complet = function
     | Rien -> true
    | Jointure(g,d) -> ((hauteur g) = (hauteur d))
              && (est_vraiment_complet g) && (est_vraiment_complet d) ;;
```

En fait il est beaucoup plus facile (et efficace) d'écrire d'abord une fonction qui renvoie une liste des profondeurs des feuilles.

Caml

```
let liste a =
    let recaux n l = function
        | Rien -> n :: l
        | Jointure(g,d) -> aux (n + 1) (aux (n + 1) l d) g
    in
        aux 0 [] a ;;
let est_complet a =
     let recminmax = function
        | [] -> failwith "Liste vide"
        | [ t ] -> (t,t)
        | t :: q -> let (m,M) = minmax q
                      in  (min m t, max M t)
     in
        let (m,M) = minmax (liste a)
        in 1 >= (M - m) ;;
```

2.4 Dénombrement

Soit C_n le nombre de squelettes binaires de taille n : $C_0 = 0$, $C_1 = 1$,$C_2 = 2$/

On dispose de la récurrence : $\forall n \geq 1,\ C_n = \sum_{k=1}^{n-1} = C_k C n - 1 - k$, d'où pour la série génératrice $C(z) = \sum_{n=0}^{+\infty} C_n z^n$, la relation :

$$C(z) = C_0 + z \sum_{n=1}^{+\infty} \sum_{k=1}^{n-1} C_k C_{n-1-k} z^{n-1} = 1 + zC(z)^2$$

On résout cette équation (en tenant compte que $C(0) = C_0 = 1$) et on trouve

$$C(z) = \frac{1 - \sqrt{1-4z}}{2z}$$

et on en déduit le

Théorème 2.3 (Nombres de Catalan)**.** *Le nombre de squelettes binaires de taille n est égal à $\frac{1}{n+1} C_{2n}^n$. Il est équivalent à $\frac{4^n}{n\sqrt{\pi n}}$.*

2.5 Complexité moyenne d'accès à un noeud dans un arbre binaire

Soit A un arbre binaire pris au hasard possédant n noeuds. Nous allons estimer la longueur moyenne $L(n)$ d'un chemin allant de la racine à un noeud x pris au hasard. Soit a_0 la racine de l'arbre, A_1 et A_2 les deux branches issues de a_0. Supposons que A_1 possède i noeuds ; en conséquence A_2 possède $n-i-1$ noeuds. Il y a i chances que x soit un noeud de A_1 avec une longueur moyenne de chemin égale à $L(i)+1$, $n-i-1$ chances que x soit un noeud de A_2 avec une longueur moyenne de chemin égale à $L(n-i-1)+1$ et enfin 1 chance que x soit égale à a_0 avec une longueur de chemin égale à 0. On en déduit que la longueur moyenne du chemin dans le cas où A_1 possède i noeuds est égale à $\frac{1}{n}\Big(i\big(L(i)+1\big) + (n-i-1)\big(L(n-i-1)+1\big)\Big)$.

Maintenant, pour un arbre pris au hasard, toutes les valeurs de i de 0 à $n-1$ sont équiprobables et donc la longueur moyenne du chemin d'accès est égale à

$$\begin{aligned} L(n) &= \frac{1}{n}\sum_{i=0}^{n-1} \frac{1}{n}\Big(i\big(L(i)+1\big) + (n-i-1)\big(L(n-i-1)+1\big)\Big) \\ &= \frac{1}{n^2}\left(\sum_{i=0}^{n-1} iL(i) + \sum_{i=0}^{n-1}(n-i-1)L(n-i-1) + n(n-1)\right) \\ &= \frac{2}{n^2}\sum_{i=0}^{n-1} iL(i) + \frac{n-1}{n} \leq \frac{2}{n^2}\sum_{i=1}^{n-1} iL(i) + 1 \end{aligned}$$

puisque $\sum_{i=1}^{n-1} i = \sum_{i=1}^{n-1}(n-i-1) = \frac{n(n-1)}{2}$ et

$$\sum_{i=0}^{n-1} iL(i) = \sum_{i=0}^{n-1}(n-i-1)L(n-i-1)$$

(changer i en $n-1-i$).

Nous allons montrer par récurrence sur n que $L(n) \leq 4\log n$. C'est vrai pour $n = 1$ et pour $n = 2$. Supposons donc l'inégalité vérifiée pour $i = 1, \ldots, n-1$. On a alors

$$L(n) \leq \frac{8}{n^2}\sum_{i=1}^{n-1} i\log(i) + 1$$

Comme la fonction $x \mapsto x\log x$ est croissante sur $[1, +\infty[$, on a

$$\forall i \in \mathbb{N},\ i\log i \leq \int_i^{i+1} t\log t\ dt$$

d'où

$$L(n) \leq \frac{8}{n^2}\int_1^n t\log t\ dt + 1 = \frac{8}{n^2}\left(\frac{n^2}{2}\log n - \frac{n^2}{4} + \frac{1}{4}\right) + 1 \leq 4\log n$$

dès que $n \geq 3$, ce qui achève la récurrence. On obtient donc

Théorème 2.4. *La longueur moyenne du chemin allant de la racine à un noeud dans un arbre binaire pris au hasard est un* $O(\log n)$.

2.6 Arbres binaires homogènes

Nous désignerons encore par N l'ensemble des étiquettes des noeuds et F l'ensemble des étiquettes des feuilles, mais nous supposerons ici que $N = F$ (ce que nous appelerons un arbre homogène). Nous pouvons alors définir de manière récursive l'ensemble des arbres dont les noeuds et les feuilles sont étiquetés par F de la manière suivante :

- si x est un élément de F, alors x est un arbre (réduit à une feuille)
- tout famille finie d'arbres est une forêt
- tout couple formé d'un élément y de F et d'une forêt est un arbre

Dans les arbres homogènes, les feuilles ne se distinguent des noeuds que par le fait qu'elles n'ont pas de fils. Autrement dit, les feuilles sont simplement des noeuds de degré 0.

Il y a deux manières de traduire cela. La première consiste à autoriser une forêt à être vide : dans ce cas les feuilles sont les noeuds dont la forêt des branches est vide. La deuxième est d'autoriser un arbre lui même à être vide : dans ce cas, les feuilles sont les noeuds dont toutes les branches sont vides.

L'implémentation des branches partant d'un noeud sous forme de liste est tout particulièrement adaptée à la première traduction puisqu'une liste peut être vide. Ceci conduit à la définition suivante d'un arbre homogène en Caml :

Caml

```
#type 'f arbre = Noeud of 'f * ('f foret)
and  'f foret = Branches of 'f arbre list;;
Type arbre defined.
Type foret defined.
```

On peut aussi autoriser un arbre à être vide, ce qui conduit à l'implémentation suivante pour les arbres binaires :

Caml

```
#type 'f arbre_bin =
            Vide
        |   Noeud of ('f arbre_bin) * 'f * ('f arbre_bin);;
Type arbre_bin defined.
```

Autrement dit, nous identifions un arbre binaire homogène à un arbre binaire hétérogène dont les feuilles sont `Vide`. Par la suite, c'est à cette implémentation des arbres binaires homogènes que nous intéresserons plus particulièrement. Remarquons que dans ce cas, de tout noeud de l'arbre partent deux branches (une branche gauche et une branche droite), mais que, dans la mesure où certaines de ces branches peuvent être vides, un noeud peut avoir 2, 1 ou même 0 fils, ce dernier cas caractérisant les noeuds dits *terminaux* (pour éviter de parler de feuilles, ce qui pourrait prêter à confusion entre l'arbre homogène et l'arbre hétérogène associé). Les noeuds non terminaux seront dits *internes*.

3 Parcours d'arbres

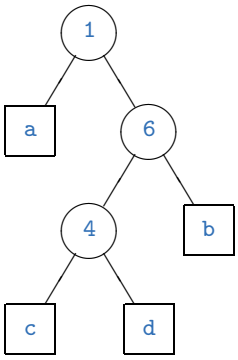
3.1 Principes

Prenons un arbre de type (F, N). Nous allons étudier les méthodes de parcours de l'arbre, le but étant de passer par chaque noeud et chaque feuille, tout en effectuant au passage certaines actions.

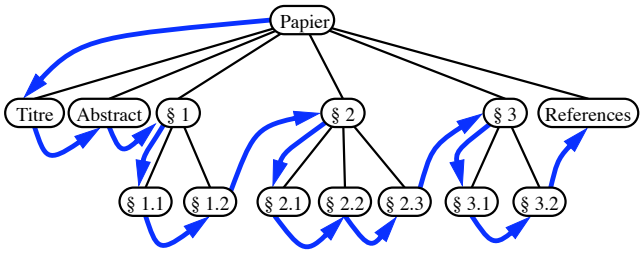
La programmation d'un tel parcours, en profondeur d'abord, de manière récursive est évidente : pour parcourir un arbre il suffit d'explorer la racine et (si l'arbre n'est pas réduit à une feuille) de parcourir chaque branche. Par contre, l'ordre de ces deux types d'opérations (exploration de la racine et parcours de chaque branche) n'est pas fixé. On dira que l'on parcourt l'arbre de manière préfixée si l'on explore la racine avant d'explorer les branches qui en sont issues, de manière postfixée si l'on explore la racine après avoir exploré les branches qui en sont issues, de manière infixée si l'on explore la racine entre le parcours de deux branches qui en sont issues.

Pour être complet, nous envisagerons également un parcours en largeur d'abord, encore appelé le parcours militaire : il consiste à lire profondeur par profondeur, de gauche à droite

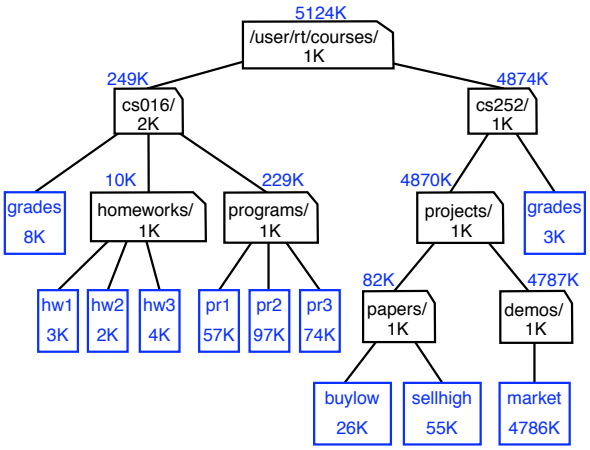
3.2 Exemples

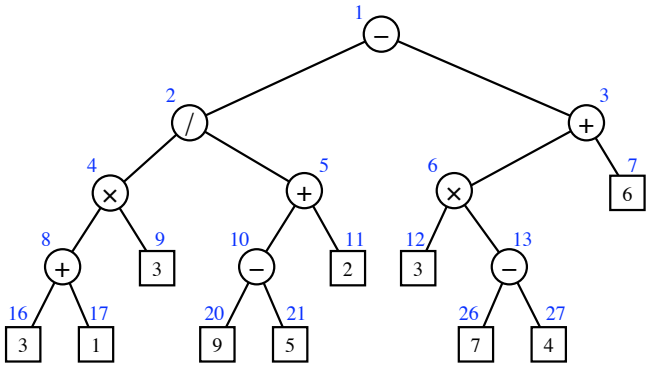


Voici ses parcours :
militaire : 1a64bcd ; préfixe : 1a64cdb ; infixe : a1c4d6b ; suffixe : acd4b61.

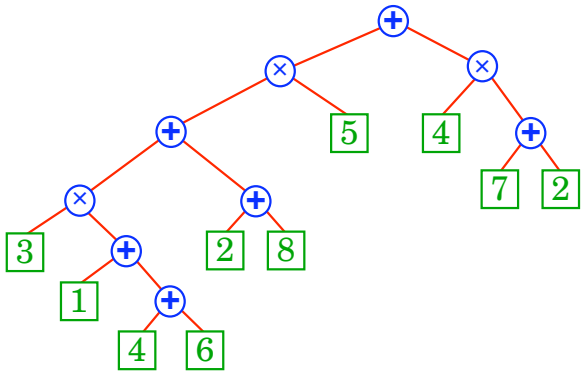


Parcours infixe : lecture d'un livre à partir de sa table des matières.
Parcours postfixe : calcul de la taille occupée par un répertoire sur un disque dur (commande `du` d'unix)





Parcours postfixe : évaluation d'une expression arithmétique



Parcours (presque) infixe : impression d'une expression arithmétique

La programmation de ces parcours est évidente. Nous passerons à la fonction de parcours deux procédures qui agissent sur les objets de type F et de type N, l'une appelée `explore_feuille` et l'autre `explore_noeud` de types respectifs `'f -> unit` et `'n -> unit`. On a alors une procédure de parcours préfixé dans le cas où les forêts sont implémentées commes des listes

Caml

```
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre list);;
#let parcours_prefixe explore_feuille explore_noeud =
        let rec prefixe_arbre = function
                  | Feuille f -> explore_feuille f
                  | Noeud (n,foret) -> explore_noeud n; prefixe_foret foret
        and prefixe_foret = function
                  | [] -> ()
                  | t::q -> prefixe_arbre t; prefixe_foret q
        in prefixe_arbre;;
```

Pour le parcours postfixé, il suffit d'échanger l'ordre de l'exploration de la racine et de l'exploration des branches

Caml

```
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre list);;

#let parcours_postfixe explore_feuille explore_noeud =
        let rec postfixe_arbre = function
                  | Feuille f -> explore_feuille f
                  | Noeud (n,foret) -> postfixe_foret foret; explore_noeud n
        and postfixe_foret = function
                  | [] -> ()
                  | t::q -> postfixe_arbre t; postfixe_foret q
        in postfixe_arbre;;
```

Pour le parcours infixé, il faut se prémunir contre l'exploration de la forêt vide et passer la valeur de l'étiquette de la racine à la fonction de parcours de la forêt

Caml

```
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre list);;

#let parcours_infixe explore_feuille explore_noeud =
        let rec infixe_arbre = function
                  | Feuille f -> explore_feuille f
```

```
                | Noeud (n,foret) -> infixe_foret n foret
        and infixe_foret n = function
                | [] -> failwith "arbre incorrect"
                | [branche] -> infixe_arbre branche
                | t::q -> infixe_arbre t; explore_noeud n; infixe_foret n q
        in infixe_arbre;;
```

3.3 Exemples d'impression

Voici un exemple d'impression d'arbre sous forme préfixée, postfixée et infixée dans le cas de l'arbre déjà rencontré

Caml

```
##let a = Noeud (`a`,[Noeud (`b`,[Noeud (`d`,[Feuille 2;Feuille 4]);Feuille 1]);
                      Feuille 5; Noeud (`c`,[Feuille 3])])
and explore_feuille = print_int and explore_noeud = print_char in
parcours_prefixe explore_feuille explore_noeud a; print_newline();
parcours_postfixe explore_feuille explore_noeud a; print_newline();
parcours_infixe explore_feuille explore_noeud a;;

#abd2415c3
24d1b53ca
2d4b1a5a3- : unit = ()
```

3.4 Parcours d'Euler

On peut aussi écrire une procédure générale de parcours d'arbre où le noeud est exploré une première fois avant les branches, puis entre chaque branche, puis une dernière fois après la dernière branche, avec trois procédures différentes d'exploration des noeuds :

Caml

```
#let parcours explore_feuille pre_explore_noeud in_explore_noeud post_explore_noeud=
        let rec parcours_arbre = function
                | Feuille f -> explore_feuille f
                | Noeud (n,foret) -> pre_explore_noeud n; parcours_foret n foret
        and parcours_foret n = function
                | [] -> failwith "arbre incorrect"
                | [branche] -> parcours_arbre branche; post_explore_noeud n
                | t::q -> parcours_arbre t; in_explore_noeud n; parcours_foret n q
        in parcours_arbre;;
```

Avec un exemple intéressant qui imprime la syntaxe concrète d'un arbre

Caml

```
#let a = Noeud (`a`,[Noeud (`b`,[Noeud (`d`,[Feuille 2;Feuille 4]);Feuille 1]);
                     Feuille 5; Noeud (`c`,[Feuille 3])])
and explore_feuille = print_int
and pre_explore_noeud n = print_char n; print_char `(`
and in_explore_noeud n = print_char `,`
and post_explore_noeud n = print_char `)` in
parcours explore_feuille pre_explore_noeud in_explore_noeud post_explore_noeud a;;

a(b(d(2,4),1),5,c(3))- : unit = ()
```

Un exemple amusant qui imprime la syntaxe Caml de l'arbre Caml (! !)

Caml

```
#let a = Noeud (`a`,[Noeud (`b`,[Noeud (`d`,[Feuille 2;Feuille 4]);Feuille 1]);
                    Feuille 5; Noeud (`c`,[Feuille 3])])
and explore_feuille f = print_string "Feuille ";print_int f
and pre_explore_noeud n = print_string "Noeud (`";print_char n; print_string "`,["
and in_explore_noeud n = print_char `;`
and post_explore_noeud n = print_string "])" in
parcours explore_feuille pre_explore_noeud in_explore_noeud post_explore_noeud a;;

Noeud (`a`,[Noeud (`b`,[Noeud (`d`,[Feuille 2;Feuille 4]);Feuille 1]);Feuille 5;
Noeud (`c`,[Feuille 3])]) - : unit = ()
```

3.5 Parcours d'un arbre binaire

Dans le cas d'un arbre binaire, on peut écrire de manière plus simple :

Caml

```
#type ('f,'n) arbre_bin =
                  | Feuille of 'f
                  | Noeud of (('f,'n) arbre_bin * 'n * ('f,'n) arbre_bin);;

let parcours_prefixe explore_feuille explore_noeud =
          let rec prefixe = function
                  |  Feuille f -> explore_feuille f
                  |  Noeud ( gauche, n , droite) ->
                        explore_noeud n; prefixe gauche; prefixe droite
          in prefixe;;
let parcours_postfixe explore_feuille explore_noeud =
          let rec postfixe = function
                  |  Feuille f -> explore_feuille f
                  |  Noeud ( gauche, n , droite) ->
                        postfixe gauche; postfixe droite; explore_noeud n
          in postfixe;;
let parcours_infixe explore_feuille explore_noeud =
          let rec infixe = function
                  |  Feuille f -> explore_feuille f
                  |  Noeud ( gauche, n , droite) ->
                        infixe gauche; explore_noeud n; infixe droite
          in infixe;;
```

3.6 Reconstitution d'un arbre à l'aide de son parcours préfixe

On suppose qu'on se donne un parcours d'un arbre par une liste d'objets du type `('n,'f) listing_d'arbre = F of 'f | N of`
On remarque que le parcours préfixe de la branche gauche est l'unique préfixe du parcours total qui soit un parcours préfixe syntaxiquement correct d'un arbre. On en déduit comment récupérer l'arbre depuis son parcours préfixe :

Caml

```
let recompose_préfixe l =
     let recaux = function
         | (F f) :: reste -> (Feuille f), reste
         | (N n) :: reste -> let g, reste' = aux reste in
                     let d, reste'' = aux reste' in
                              Noeud(n,g,d), reste''
         | [] -> failwith "Description préfixe incorrecte"
     in match aux l with
             | a,[] -> a
             | _ -> failwith "Description préfixe incorrecte" ;;
```

3.7 Reconstitution d'un arbre à l'aide de son parcours postfixe

On utilise une pile d'arbres. Lorsqu'on rencontre une feuille, on l'empile. Lorsqu'on recontre un noeud, on assemble à l'aide de ce noeud les deux arbres au sommet de la pile et on réempile le résultat. Voici ce que cela donne :

Caml

```
let recompose_suffixe l =
    let rec aux ss_arbres parcours = match (ss_arbres, parcours) with
        | pile, (F f) :: q -> aux (Feuille(f) :: pile) q
        | d :: g :: pile, (N n) :: q -> aux (Noeud(n,g,d) :: pile) q
        | [ arbre ],[] -> arbre
        | _ -> failwith "Description suffixe incorrecte"
     in
          aux [] l ;;
```

3.8 Reconstitution d'un arbre à l'aide de son parcours militaire

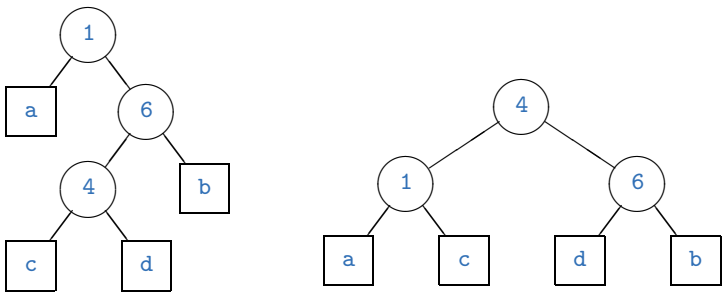
On utilise la même méthode que pour le parcours postfixe, mais en utilisant une file d'attente au lieu d'une pile. On se convaincra facilement qu'il faut en plus commencer par la fin du parcours miltaire, autrement dit applique la méthode au miroir du parcours. on obtient alors, avec une gestion rudimentaire des files d'attente sous forme de liste, le code suivant :

Caml

```
let recompose_militaire l =
    let rec aux ss_arbres parcours = match (ss_arbres, parcours) with
        | file, (F f) :: q -> aux (file @ [ Feuille f ]) q
        | d :: g :: file, (N n) :: q
                -> aux (file @ [ Noeud(n,g,d) ]) q
        | [ arbre ],[ ] -> arbre
        | _ -> failwith "Description militaire incorrecte"
    in
        aux [ ] (rev l) ;;
```

3.9 Reconstitution d'un arbre à l'aide de son parcours infixe

La reconstitution d'un arbre binaire à l'aide de son parcours infixe est malheureusement impossible, deux arbres distincts pouvant avoir le même parcours infixe :



4 Arbres binaires de recherche

4.1 Objet des arbres binaires de recherche

Le gestionnaire d'un hôtel dans une ville touristique est amené à gérer un fichier des clients. Bien entendu, chaque jour, certains clients quittent leur chambre, d'autres prolongent leur séjour, enfin de nouveaux clients arrivent. Nous recherchons une structure de données qui permette de gérer un tel fichier en évolution constante tout en autorisant une recherche aussi rapide que possible d'un client donné.

Une première solution est de gérer un tableau des clients. Ceci nécessite soit que nous choisissions dès le départ un tableau assez grand pour contenir le nombre maximum de clients que peut recevoir l'hôtel (attention au surbooking) en perdant une place

considérable dans notre fichier en basse-saison, soit que nous procédions à des recopies incessantes de tableau dans des tableaux plus grands ou plus petits dès que l'évolution du nombre de clients le nécessite. Dans un tel tableau, si nous voulons avoir une procédure efficace de recherche d'un client donné, nous avons vu dans le cours de Math Sup qu'il fallait trier le tableau des clients, par exemple par ordre alphabétique, et procéder ensuite à une recherche dichotomique. La complexité de la recherche est alors un $O(\log_2 n)$ si n est le nombre de clients.

Pour supprimer la fiche d'un client, il faut d'abord la rechercher dans le tableau (complexité en $O(\log_2 n)$), puis décaler tous les clients suivants d'un cran vers la gauche (complexité moyenne en $O(n)$: si l'élément figure à la p-ième place, il faut décaler $n-p$ éléments, soit $n-p$ opérations ; comme toutes les places p de 1 à n sont équiprobables, la complexité moyenne est $\frac{1}{n}\sum_{p=1}^{n}(n-p) = \frac{n-1}{2} = O(n)$, ce qui maintient le tableau trié.

Pour insérer un client, il faut d'abord trouver par une recherche dichotomique à quel endroit il faut l'insérer (complexité en $O(\log_2 n)$) puis décaler tous les clients suivants d'un cran vers la droite (complexité moyenne en $O(n)$) avant d'insérer l'élément (complexité en $O(1)$), ce qui maintient le tableau trié. La structure de tableau trié est donc une structure de donnée statique (figée dans le temps hors recopie) pour laquelle la recherche se fait avec une complexité en $O(\log_2 n)$ alors que l'insertion et la suppression se font avec une complexité moyenne en $O(n)$.

Une deuxième solution est de gérer une liste des clients. La structure peut alors croître et décroître suivant les besoins. Le fait que de toute manière on ne puisse accéder directement qu'à la tête d'une liste rend sans intérêt tout tri de la liste : la recherche dans une liste ne peut se faire que de manière linéaire avec une complexité moyenne en $O(n)$. L'insertion d'un élément dans une liste se fait en temps constant $O(1)$ puisqu'il suffit de le mettre en tête de la liste. Par contre la suppression d'un élément nécessite sa recherche (complexité en $O(n)$), puis la concaténation des deux listes obtenues en supprimant cet élément (complexité moyenne en $O(n)$). La structure de liste est donc une structure de données dynamique (évolutive avec le temps) pour laquelle l'insertion se fait avec une complexité en $O(1)$ alors que la recherche ou la suppression se font avec une complexité moyenne en $O(n)$.

Nous allons voir que les arbres de recherche permettent de gérer un tel fichier de manière dynamique, la recherche, l'insertion et la suppression se faisant avec une complexité moyenne en $O(\log_2 n)$ et une complexité dans le pire des cas en $O(n)$.

4.2 Arbres binaires de recherche

Définition 4.1. *Soit $(X, \leq)$ un ensemble ordonné. On appelle arbre binaire de recherche dans X tout arbre binaire homogène A étiqueté par X tel que*

- *pour tout noeud r de A étiqueté par $\varepsilon(r) \in X$, pour tout noeud g de la branche gauche de r, étiqueté par $\varepsilon(g) \in X$, on a $\varepsilon(g) \leq \varepsilon(r)$*
- *pour tout noeud r de A étiqueté par $\varepsilon(r) \in X$, pour tout noeud d de la branche droite de r, étiqueté par $\varepsilon(d) \in X$, on a $\varepsilon(r) < \varepsilon(d)$*

Autrement dit dans un arbre binaire de recherche, les éléments à gauche d'un noeud sont inférieurs ou égaux à l'étiquette de ce noeud et les éléments à droite sont strictement supérieurs à cette étiquette.

Par la suite, et uniquement pour la facilité d'écriture, nous supposerons que l'ensemble ordonné X est celui des entiers naturels, implémenté par le type Caml `int arbre_bin`.

Voici un exemple d'arbre binaire de recherche dans les entiers :

Par contre, si on remplace le 3 par un 5, on n'a plus un arbre binaire de recherche puisque le noeud étiqueté par 5 se trouve dans la branche gauche du noeud étiqueté par 4 alors que $5 > 4$.

4.3 Test d'un arbre binaire de recherche

Soit A un arbre binaire, par exemple étiqueté par $\mathbb{N}$; nous définirons $\min(A)$ comme la plus petite des étiquettes des noeuds de A et $\max(A)$ comme la plus grande de ces mêmes étiquettes. Pour un arbre vide, nous poserons $\min(A) = +\infty$ et $\max(A) = -\infty$.

Soit alors A un arbre binaire de racine r (étiquetée par $\varepsilon(r)$) ayant pour branches les deux arbres binaires A_1 et A_2. Alors A est un arbre binaire de recherche si et seulement si il vérifie

- A_1 et A_2 sont des arbres binaires de recherche
- $\max(A_1) \leq \varepsilon(r) < \min(A_2)$

Pour que cette définition récursive soit complète, il faut préciser qu'un arbre vide est un arbre binaire de recherche.

Nous pouvons alors définir une fonction qui à tout arbre A associe un triplet formé d'un booléen indiquant si A est un arbre binaire de recherche et de deux entiers $\min(A)$ et $\max(A)$ de la façon suivante :

Caml

```
#let grandInt = 1000000;; (* très grand *)
grandInt : int = 1000000
```

```
#let rec testminmax = function
             | Vide -> (true, grandInt, -grandInt)
             | Noeud (gauche , n , droite) ->
                   match (testminmax gauche,testminmax droite) with
                     ((testg,ming,maxg),(testd,mind,maxd)) ->
                         (testg && testd && (maxg <= n) && (n < mind),
                          min (min ming mind) n,
                          max (max maxg maxd) n);;
```

La complexité de ce calcul est en $O(n)$ si n est le nombre total de noeuds de l'arbre, puisque chaque noeud est examiné une et une seule fois. Il est alors facile de construire une fonction de test d'un arbre binaire en éliminant le minimum et le maximum qui sont sans intérêt pour l'arbre complet (alors qu'ils sont essentiels pour les sous-arbres) :

Caml

```
#let teste_arbre_rech a =
    match testminmax a with (test,_,_) -> test;;
teste_arbre_rech : int arbre_bin -> bool = <fun>
```

4.4 Recherche dans un arbre binaire

Nous allons nous intéresser à quatre opérations sur les arbres binaires de recherche : le test pour savoir si un élément figure comme étiquette dans un arbre donné, l'insertion d'un nouvel élément, la suppression d'un élément, et enfin le parcours des éléments dans l'ordre.

Le test de présence d'un élément x comme étiquette de l'arbre binaire de recherche A se programme par induction structurelle :

- si l'arbre est vide, l'élément x ne figure pas dans l'arbre
- si l'arbre possède r comme racine étiquetée par $\varepsilon(r)$, de branche gauche A_1 et de branche droite A_2 alors
  - si $x = \varepsilon(r)$, alors x figure dans l'arbre
  - si $x < \varepsilon(r)$ alors x figure dans A si et seulement si il figure dans A_1 (puisque A_1 contient tous les éléments de l'arbre inférieurs ou égaux à $\varepsilon(r)$)
  - si $x > \varepsilon(r)$ alors x figure dans A si et seulement si il figure dans A_2 (puisque A_2 contient tous les éléments de l'arbre supérieurs à $\varepsilon(r)$)

Ceci conduit à la fonction Caml suivante (si l'on s'autorise un filtrage avec surveillants) :

Caml

```
#let rec figure_dans_arbre x a =
      match a with
        | Vide -> false
        | Noeud (_ , n , _) when x = n -> true
        | Noeud (gauche , n , _) when x < n -> figure_dans_arbre x gauche
        | Noeud (_ , n , droite) (* x > n *) -> figure_dans_arbre x droite;;
figure_dans_arbre : 'a -> 'a arbre_bin -> bool = <fun>
```

Remarque 4.1. Nous n'avons pas mis de surveillant dans le dernier cas du filtrage pour éviter un avertissement de Caml disant que notre filtrage n'est pas exhaustif ; le compilateur émet ce message de manière à peu près systématique dans un filtrage avec surveillants car il n'a aucun moyen de savoir que les différents surveillants `when ...` couvrent tous les cas possibles. Nous avons simplement rappelé en commentaire quelle est la situation lorsque ce dernier cas du filtrage est examiné.

Dans tous les cas, on constate que le nombre de comparaisons effectuées avant de trouver (ou de ne pas trouver) l'élément est égal au nombre de noeuds explorés, et comme on suit un chemin descendant de la racine au dernier noeud exploré, ce nombre est majoré par la hauteur de l'arbre augmentée de 1. Si l'arbre possède n noeuds, ce nombre est donc majoré par n. De plus comme la hauteur moyenne d'un arbre à n noeuds est un $O(\log_2 n)$, le nombre moyen de comparaisons effectuées pour rechercher un élément dans un arbre binaire pris au hasard est aussi un $O(\log_2 n)$.

Théorème 4.1. *La recherche d'un élément dans un arbre binaire de recherche à n noeuds se fait en un temps moyen $O(\log_2 n)$ et en un temps $O(n)$ dans le pire des cas.*

4.5 Insertion dans un arbre binaire de recherche

L'insertion d'un élément x dans un arbre binaire de recherche A (tout en maintenant évidemment la structure ordonnée d'arbre binaire de recherche) peut se faire par induction structurelle en plaçant le nouvel élément comme noeud terminal de l'arbre :

- si l'arbre est vide, le résultat est l'arbre dont l'unique noeud (terminal) est x
- si l'arbre possède r comme racine étiquetée par $\varepsilon(r)$, de branche gauche A_1 et de branche droite A_2 alors
  - si $x \leq \varepsilon(r)$ alors le résultat est l'arbre de racine r de branche droite A_2 et de branche gauche le résultat de l'insertion de x dans A_1
  - si $x > \varepsilon(r)$ alors le résultat est l'arbre de racine r de branche gauche A_1 et de branche droite le résultat de l'insertion de x dans A_2.

Il est clair par induction structurelle que l'on maintient ainsi la structure d'arbre de recherche : c'est évident pour un arbre vide et si c'est vrai pour les branches A_1 et A_2, cela reste vrai pour A puisque toutes les étiquettes des noeuds de la branche gauche restent inférieures à $\varepsilon(r)$ et toutes les étiquettes de la branche droite restent strictement supérieures à $\varepsilon(r)$.

On construit ainsi une fonction Caml, en utilisant des surveillants :

Caml

```
#let rec insere_dans_arbre x a =
              match a with
                 | Vide -> Noeud (Vide , x , Vide)
                 | Noeud (gauche , n , droite) when x <= n  ->
                           Noeud( insere_dans_arbre x gauche, n, droite)
                 | Noeud (gauche , n , droite)  (* x > n *) ->
                           Noeud( gauche, n, insere_dans_arbre x droite);;
insere_dans_arbre : 'a -> 'a arbre_bin -> 'a arbre_bin = <fun>
```

On peut alors insérer tous les éléments d'une liste dans un arbre binaire de recherche :

Caml

```
#let rec liste_a_arbre = function
                    | [] -> Vide
                    | h::r -> insere_dans_arbre h (liste_a_arbre r);;
liste_a_arbre : 'a list -> 'a arbre_bin = <fun>
```

Nous donnons ci dessous quelques exemples d'exécutions, tout d'abord avec une liste triée par ordre croissant :

Caml

```
#liste_a_arbre [1;2;3;4;5;6];;
- : int arbre_bin =
 Noeud
  (Noeud
    (Noeud (Noeud (Noeud (Noeud (Vide, 1, Vide), 2, Vide), 3, Vide), 4, Vide),
     5, Vide),
   6, Vide)
```

puis avec une liste triée par ordre décroissant :

Caml

```
#liste_a_arbre [6;5;4;3;2;1];;
- : int arbre_bin =
 Noeud
  (Vide, 1,
   Noeud
    (Vide, 2,
     Noeud (Vide, 3, Noeud (Vide, 4, Noeud (Vide, 5, Noeud (Vide, 6, Vide))))))
```

puis avec une liste non triée :

Caml

```
#liste_a_arbre [4;2;3;6;5;1];;
- : int arbre_bin =
 Noeud
  (Vide, 1,
   Noeud
    (Noeud (Noeud (Vide, 2, Vide), 3, Noeud (Vide, 4, Vide)), 5,
     Noeud (Vide, 6, Vide)))
```

Testons les hauteurs des arbres de recherche obtenus :

Caml

```
#let rec hauteur a =
      match a with
        | Vide -> -1
        | Noeud (gauche , _, droite) -> 1+max (hauteur gauche) (hauteur droite);;
hauteur : 'a arbre_bin -> int = <fun>
#hauteur (liste_a_arbre [1;2;3;4;5;6]);;
- : int = 5
#hauteur (liste_a_arbre [6;5;4;3;2;1]);;
- : int = 5
#hauteur (liste_a_arbre [4;2;3;6;5;1]);;
- : int = 3
```

Nous constatons que dans les deux premiers cas, l'introduction de données préalablement ordonnées a donné lieu à des constructions d'arbres binaires complètement déséquilibrés (ils ont même une structure linéaire) et qui sont de hauteur maximum. Dans le troisième cas, l'introduction de données dans un ordre aléatoire a donné lieu à la construction d'un arbre beaucoup plus équilibré.

Les mêmes arguments que pour la recherche dans un arbre binaire conduisent à une conclusion similaire :

Théorème 4.2. *L'insertion d'un élément dans un arbre binaire de recherche à n noeuds se fait en un temps moyen $O(\log_2 n)$ et en un temps $O(n)$ dans le pire des cas.*

4.6 Suppression dans un arbre binaire de recherche

Soit A un arbre binaire de recherche sur X et $x \in X$. Nous cherchons à supprimer la première occurrence d'un noeud ayant l'étiquette x dans l'arbre A, l'arbre étant parcouru de haut en bas.

Pour savoir supprimer un élément d'un arbre binaire de recherche, il suffit, par induction structurelle, de savoir supprimer la racine d'un tel arbre avec le code évident :

Caml

```
let rec supprime_dans_arbre x a=
      match a with
          | Vide -> raise Not_found
          | Noeud (gauche , n , droite) when x = n -> supprime_racine a
          | Noeud (gauche , n , droite) when x < n ->
                 Noeud( supprime_dans_arbre x gauche, n, droite)
          | Noeud (gauche , n , droite) (* x > n *) ->
                Noeud( gauche, n, supprime_dans_arbre x droite)
where supprime_racine = ..... ;;
```

Il nous reste donc à savoir supprimer la racine d'un tel arbre. Soit donc A un arbre de racine r ayant une branche gauche A_1 et une branche droite A_2. Si A_1 est vide, le résultat de la suppression de la racine est bien évidemment A_2 ; de même si A_2 est vide, le résultat de la suppression de la racine est bien évidemment A_1. Nous supposerons donc que A_1 et A_2 sont non vides. Soit alors n le noeud de A_1 qui a l'étiquette la plus grande. On a donc $b = \varepsilon(n) \leq \varepsilon(r) = a$ et pour tout noeud n' de A_1 on a $\varepsilon(n') \leq \varepsilon(n)$. Le noeud n ne peut pas avoir de fils droit d, sinon on devrait avoir à la fois $\varepsilon(d) > \varepsilon(n)$ (car d est un fils droit de n) et $\varepsilon(d) \leq \varepsilon(r)$ (car d est un noeud de A_1), ce qui contredirait la définition de n. Autrement dit le noeud n est facile à supprimer de l'arbre. Il suffit alors de reporter son étiquette sur la racine r : on obtient un nouvel arbre binaire A', dont nous allons montrer qu'il s'agit bien d'un arbre binaire de recherche.

La branche droite de la racine r' n'a pas changé, donc c'est un arbre binaire de recherche. Dans la branche gauche de la racine, nous avons supprimé un noeud n'ayant pas de fils droit, et par conséquent nous avons encore un arbre binaire de recherche A'_1. Il nous suffit de vérifier que pour tout noeud p de A'_1 et pour tout noeud q et A_2, on a $\varepsilon(p) \leq b = \varepsilon(r') < \varepsilon(q)$. La première inégalité est claire par la définition de b comme l'étiquette la plus grande de A_1 ; la deuxième inégalité provient de ce que

$$\varepsilon(r') = b = \varepsilon(n) \leq \varepsilon(r) < \varepsilon(q)$$

puisque A était un arbre binaire de recherche.

Il nous reste donc à trouver le noeud dont l'étiquette est la plus grande dans l'arbre gauche A_1. Il nous suffit pour cela de rechercher le noeud le plus à droite de cet arbre c'est-à-dire de parcourir l'arbre en restant le plus à droite possible. Nous commencerons donc par définir une fonction qui parcourt un arbre vers la droite tout en éliminant le noeud le plus à droite ; au passage la fonction retournera l'étiquette de ce noeud :

Caml

```
#let rec suppr_a_droite = function
      | Vide -> raise Not_found
      | Noeud (gauche, n, Vide) -> (gauche , n)
      | Noeud (gauche, n, droite) ->
            let (nouveau_droite , plus_a_droite) = suppr_a_droite droite in
                  Noeud (gauche, n, nouveau_droite) , plus_a_droite;;
suppr_a_droite : 'a arbre_bin -> 'a arbre_bin * 'a = <fun>
```

On peut alors écrire notre procédure de suppression de la racine d'un arbre binaire de recherche :

Caml

```
#let supprime_racine = function
      | Vide -> raise Not_found
      | Noeud (gauche, _, Vide) -> gauche
      | Noeud (Vide, _, droite) -> droite
      | Noeud (gauche, r, droite) ->
           let (nouveau_gauche , nouvelle_racine) = suppr_a_droite gauche in
           Noeud(nouveau_gauche, nouvelle_racine , droite);;
supprime_racine : 'a arbre_bin -> 'a arbre_bin = <fun>
```

Nous avons donc la fonction générale de suppression d'un élément dans un arbre binaire de recherche :

Caml

```
#let rec supprime_dans_arbre x a =
      match a with
         | Vide -> raise Not_found
         | Noeud (gauche , n , droite) when x = n -> supprime_racine a
         | Noeud (gauche , n , droite) when x < n ->
               Noeud( supprime_dans_arbre x gauche, n, droite)
         | Noeud (gauche , n , droite) (* x > n *) ->
              Noeud( gauche, n, supprime_dans_arbre x droite);;
supprime_dans_arbre : 'a -> 'a arbre_bin -> 'a arbre_bin = <fun>
```

5 Files de priorité

5.1 files d'attente

Types courants de files d'attente :

1. file LIFO ou pile : dernier entré, premier sorti
2. file FIFO ou queue : premier entré, premier sorti
3. file de priorité : celui qui a la plus grande priorité est le premier sorti

Analogue à la file d'attente aux urgence d'un hopital : on commence par s'occuper du plus gravement atteint.
Le type de donnée *File de priorité* doit disposer des méthodes :

– test pour savoir si la liste est vide
– insérer un élément dans la file
– trouver l'élément de plus grande priorité
– retirer l'élément de plus grande priorité

Pour simplifier, nous supposerons par la suite que les objets stockés sont des entiers et que le plus grand a la plus grande priorité.

5.2 Files de priorité et tris

Toute file de priorité permet de trier des éléments : on insère au fur et à mesure les éléments à trier dans la file de priorité, puis on retire les éléments un par un du plus grand au plus petit.

5.3 Implémentations élémentaires

5.3.1 Dans une liste non triée

Dans une liste non triée
– test pour savoir si la liste est vide : évident, $O(1)$
– insérer un élément dans la file : en tête de la liste, $O(1)$
– trouver l'élément de plus grande priorité : on parcourt la liste, $O(n)$
– retirer l'élément de plus grande priorité : on parcourt la liste, $O(n)$

Tri associé : tri par sélection.

Caml

```
let new_priorite () =
    let file = ref [] in
    let rec max_liste = function
       | [] -> failwith "liste vide"
       | [x] -> x
       | t::q -> max t (max_liste q)
    and extrait_max = function
       | [] -> failwith "liste vide"
       | [x] -> x,[]
       | t::q -> let (x,reste) = extrait_max q in
                 if t >= x then (t,q) else (x,t::reste)
    in
    let est_vide () = !file = []
    and insere x = file := x::!file
    and plus_grand () = max_liste !file
    and depile () = let (x,reste) = extrait_max !file in file:=reste; x
    in (est_vide,insere,plus_grand,depile);;
```

5.3.2 Dans une liste triée

Dans une liste triée
– test pour savoir si la liste est vide : évident, $O(1)$
– insérer un élément dans la file : parcourir la liste, $O(n)$
– trouver l'élément de plus grande priorité : en tête de la liste, $O(1)$
– retirer l'élément de plus grande priorité : on supprime la tête de la liste, $O(1)$

Tri associé : tri par insertion.

Caml

```
let new_priorite () =
    let file = ref [] in
    let rec insere x = function
       | [] -> [x]
       | t::q -> if x >= t then x::t::q else t::(insere x q)
    in
    let est_vide () = !file = []
```

```
    and insere x = file := insere x !file
    and plus_grand () = hd !file
    and depile () = match !file with
           | [] -> failwith "liste vide"
           | t::q -> file := q; t
    in (est_vide,insere,plus_grand,depile);;
```

5.4 Arbre maximier

C'est un arbre binaire homogène dans lequel l'étiquette d'un noeud interne est toujours supérieure ou égale aux étiquettes de chacun de ses fils.

Test d'un arbre maximier :

Caml

```
type 'a arbre_bin  = Vide | Noeud of ('a arbre_bin * 'a * 'a arbre_bin);;

let rec est_maximier = function
        | Vide -> true
        | Noeud (Vide,_,Vide) -> true
        | Noeud (Noeud(_,y,_) as g, x , Vide ) -> (est_maximier g) && x >= y
        | Noeud (Vide, x , Noeud(_,z,_) as d)  -> (est_maximier d) && x >= z
        | Noeud (Noeud(_,y,_) as g, x ,Noeud(_,z,_) as d) ->
                          (est_maximier g) && (est_maximier d)
                          && (x >= y) && (x >= z);;
```

Fonctions évidentes :

Caml

```
let est_vide = function
         | Vide -> true
         | _ -> false
and maximum = function
         | Vide -> failwith "arbre vide"
         | Noeud (_,x,_) -> x;;
```

5.5 Insertion dans un arbre maximier

Par induction structurelle.

Insérer un élément x dans un arbre maximier A.

- Si l'arbre est vide, retourner l'arbre à un seul noeud étiqueté par x.
- Si l'arbre possède une racine r étiquetée par $\varepsilon(r)$
  - si $\varepsilon(r) \geq x$, insérer l'élément x dans l'une des branches g issue de r ; cette branche deviendra un arbre maximier g' dont la racine sera soit étiquetée par $x \leq \varepsilon(r)$, soit étiquetée par l'une des étiquettes d'un noeud de g, donc inférieure ou égale à $\varepsilon(r)$; comme l'autre branche n'aura pas été modifiée, l'arbre obtenu sera bien un arbre maximier
  - si $\varepsilon(r) < x$, remplacer l'étiquette de la racine par x puis insérer l'élément $\varepsilon(r)$ dans l'une des branches d issue de r ; cette branche deviendra un arbre maximier d' dont la racine sera soit étiquetée par $\varepsilon(r) < x$, soit par l'étiquette d'un noeud de d donc inférieure à $\varepsilon(r) < x$ (en fait, il est clair que ce dernier cas ne peut pas se produire) ; quant à l'autre branche g issue de r, elle n'a pas été modifiée, c'est donc encore une arbre maximier et l'étiquette de sa racine est inférieure ou égale à $\varepsilon(r)$ et a fortiori à x.

Dans tous les cas, l'arbre obtenu sera un arbre maximier. Ceci peut se traduire en Caml par la fonction :

Caml

```
#let rec insere x a = match a with
        |  Vide -> Noeud(Vide,x,Vide)
        |  Noeud(gauche,n,droite) when x<n ->
                                              Noeud((insere x gauche),n,droite)
        |  Noeud(gauche,n,droite) -> Noeud(gauche,x,(insere n droite));;
insere : 'a -> 'a arbre_bin -> 'a arbre_bin = <fun>
```

Pourquoi gauche-droite ? ?

5.6 Depilement récursif dans un arbre maximier

La suppression de la racine d'un arbre maximier peut se faire de manière récursive : on choisit la branche ayant la racine de plus grande priorité, on extrait cette racine et on en fait la nouvelle racine de l'arbre.

On aboutit à l'algorithme suivant :

Caml

```
let rec depile_racine =
    (* retourne la racine d'un maximier et l'arbre privé de sa racine *)
     let rec aux = function (* supprime la racine d'un maximier *)
        |  Vide -> failwith "arbre vide"
        |  Noeud (Vide,_,Vide)-> Vide
        |  Noeud (Vide,_,d) -> d
        |  Noeud (g,_,Vide) -> g
        |  Noeud (Noeud(_,y,_) as g, x ,Noeud(_,z,_) as d) ->
                if y >= z then Noeud (aux g,y,d) else Noeud (g,z,aux d)
     in function
        |  Vide -> failwith "arbre vide"
        |  Noeud (g,x,d) as a -> x,aux a;;
```

Inconvénient : on ne maîtrise absolument pas la géométrie de l'arbre.

5.7 Dépilement dans un arbre maximier par percolation

Il est facile de supprimer sont les noeuds terminaux d'un arbre maximier.

La suppression de la racine peut alors se faire en trois étapes.

– échanger la racine avec un noeud terminal de l'arbre ;
– supprimer ce noeud terminal ;
– rétablir ensuite la structure de file de priorité : la percolation.

La percolation va consister à faire redescendre l'étiquette de la racine tout au long de l'arbre en l'échangeant récursivement avec la plus grande des étiquettes de ses fils, jusqu'à ce que son étiquette soit plus grande que toutes les étiquettes de ses fils.

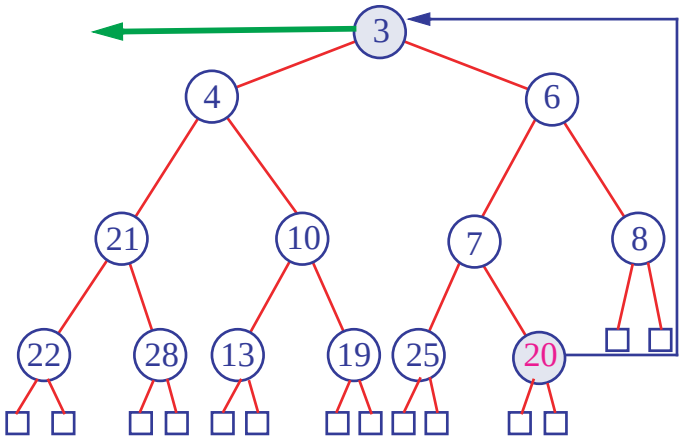


FIG. 1 – Retrait de 3, à remplacer par 20

5.8 Percolation de la racine à travers un arbre maximier

Caml

```
let rec percole = function
      | Noeud(Noeud (ssgauche,m,ssdroite),n,Vide) when m>n ->
                Noeud(percole (Noeud(ssgauche,n,ssdroite)),m,Vide)
     | Noeud(Vide,n,Noeud (ssgauche,m,ssdroite)) when m>n ->
                Noeud(Vide,m,percole (Noeud(ssgauche,n,ssdroite)))
     | Noeud(Noeud(gg,mg,dg),n,(Noeud(gd,md,dd) as droite))
```

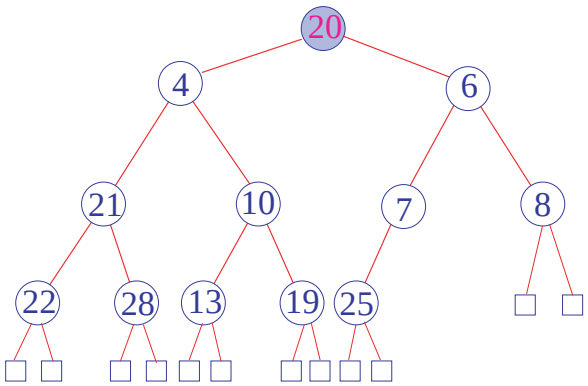


FIG. 2 – Arbre à corriger

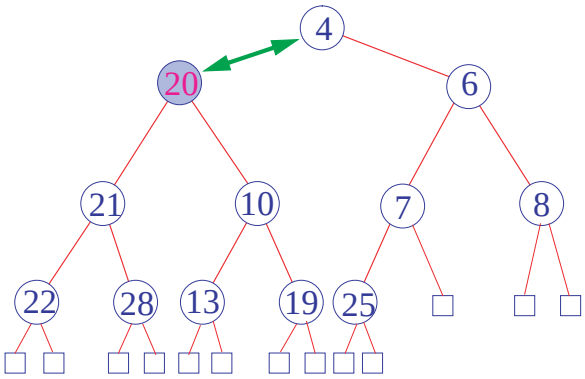


FIG. 3 – Echanger 20 avec le plus petit de ses fils

```
            when (mg>n && mg>=md) ->
              Noeud(percole (Noeud(gg,n,dg)),mg,droite)
    | Noeud((Noeud(gg,mg,dg) as gauche),n,Noeud(gd,md,dd))
            when md>n ->
              Noeud(gauche,md,percole (Noeud(gd,n,dd)))
     | a -> a;;
percole : 'a arbre_bin -> 'a arbre_bin = <fun>
```

5.9 Extraction de la racine d'un arbre maximier

Caml

```
let supprime_racine =
    let rec supprime_terminal = function
     | Vide -> invalid_arg "arbre vide"
     | Noeud(Vide,n,Vide) -> n,Vide
     | Noeud(Vide,n,droite) -> let (x,d) = supprime_terminal droite
                                          in (x,Noeud(Vide,n,d))
      | Noeud(gauche,n,droite) -> let (x,g) = supprime_terminal gauche
                              in (x,Noeud(g,n,droite))
    in function
     | Vide -> invalid_arg "arbre vide"
     | Noeud(Vide,n,Vide) -> Vide
     | a -> begin
          match supprime_terminal a with
           | (term , Noeud (gauche,r,droite)) ->
                 (r , percole (Noeud(gauche,term,droite)))
           | _ -> invalid_arg "erreur inconnue"
         end;;
```

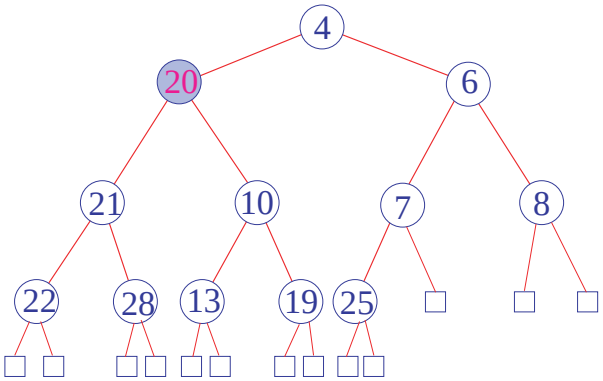


FIG. 4 – Arbre à corriger

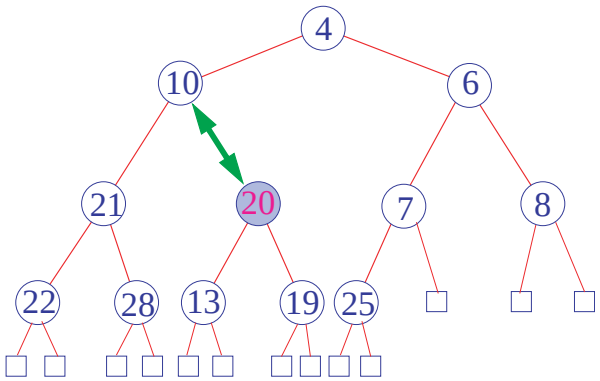


FIG. 5 – Echanger 20 avec le plus petit de ses fils

5.10 La structure de tas

Procédures de suppression de la racine ou d'insertion d'un élément dans un arbre maximier : complexité en $O(h)$ où h est la hauteur de l'arbre.

On peut espérer une complexité moyenne en $O(\log_2 n)$, si n est le nombre de noeuds.

Mais ces opérations répétées ont tendance à déséquilibrer les arbres du fait que l'on a à tout moment la liberté de choix entre la *droite* et la *gauche*, et que la méthode la plus naturelle consiste à faire ce choix de manière systématique.

Or pour des arbres déséquilibrés, la complexité passe en $O(n)$.

Solution : forcer les arbres à être équilibrés. Intervention sur la géométrie de l'arbre.

Exemple : dans la suppression par percolation, supprimer un des noeuds terminaux de profondeur maximale.

Considérons un arbre binaire homogène de hauteur h. Si $0 \leq d \leq h$, on appellera *niveau* d de l'arbre l'ensemble des noeuds situés à une distance d de la racine.

Le niveau d de l'arbre a au plus 2^d éléments.

Définition 5.1. *On dit qu'un arbre binaire homogène de hauteur h est complet si pour tout $d \in [0, h-1]$, le niveau d de l'arbre possède 2^d éléments et si les noeuds du niveau h sont les plus à gauche possible.*

Définition 5.2. *On appelle tas (en anglais heap) un arbre binaire homogène qui est un arbre maximier.*

On numérote alors les noeuds de haut en bas et de gauche à droite, depuis 0 jusqu'à $n-1$, suivant le schéma suivant :

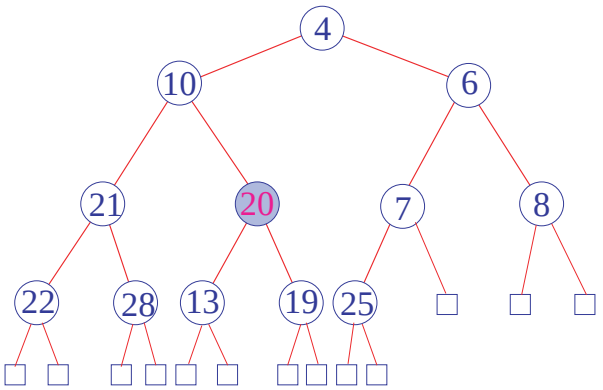


FIG. 6 – Arbre à corriger

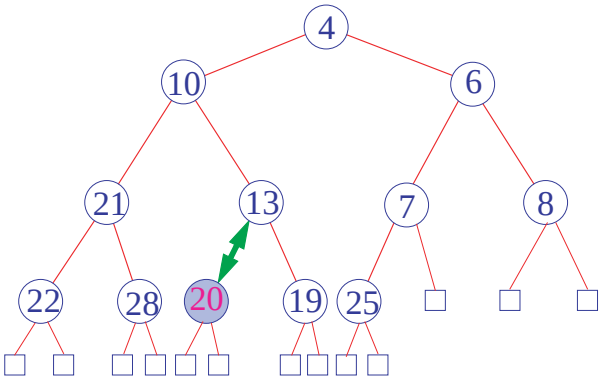


FIG. 7 – Echanger 20 avec le plus petit de ses fils

Cette numérotation permet de stocker les éléments de l'arbre dans un tableau de longueur n (dont les indices varieront de 0 à $n-1$, attention au décalage).

Les fils de l'élément numéroté i sont l'élément numéroté $2i+1$ pour le fils gauche et $2i+2$ pour le fils droit.

Tableau de grande taille dont seuls les n premiers éléments seront utilisés : type enregistrement pour conserver à la fois la vraie longueur n du tableau et le tableau lui même.

Caml

```
#type tas={mutable nombre:int; contenu:int vect};;
Type tas defined.
#let nouveau_tas n={nombre = 0; contenu = make_vect n 0};;
nouveau_tas : int -> tas = <fun>
```

Transformer un tas en un arbre :

Caml

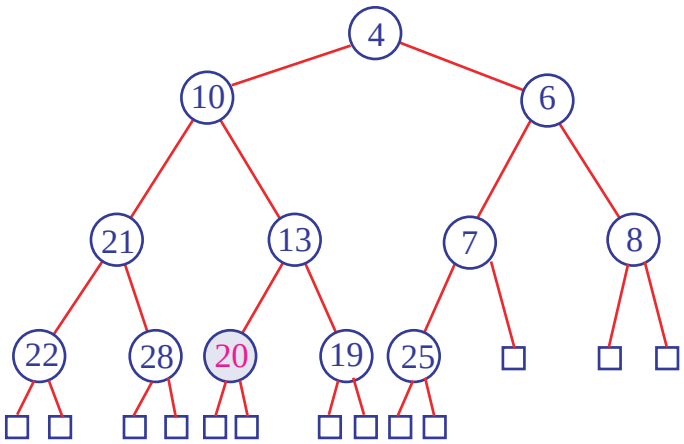


FIG. 8 – Arbre corrigé

```
#let tas_en_arbre h = traite_noeud 0
   where rec traite_noeud i=
              if i >= h.nombre then Vide
              else Noeud(
                         (traite_noeud (2*i+1)),h.contenu.(i),
                         (traite_noeud (2*i+2))
                        );;
tas_en_arbre : tas -> int arbre_bin = <fun>
```

Transformer un arbre (supposé complet) en tas :

Caml

```
#let arbre_en_tas a h = traite_noeud 0 a
     where rec traite_noeud i = function
               Vide -> ()
             | Noeud(gauche,n,droite) ->
                     begin
                        if i>= h.nombre then h.nombre <- i+1;
                        h.contenu.(i) <- n;
                        traite_noeud (2*i+1) gauche;
                        traite_noeud (2*i+2) droite
                     end;;
arbre_en_tas : int arbre_bin -> tas -> unit = <fun>
```

5.11 Suppression de la racine dans un tas

Adapter l'algorithme de suppression de la racine d'un arbre maximier : échanger l'étiquette de la racine avec l'étiquette d'un noeud terminal, puis supprimer ce noeud terminal, et enfin effectuer une percolation.

Supprimer le noeud le plus à droite du niveau maximal, c'est-à-dire dans la structure de tas, le dernier élément numéroté.

Caml

```
let supprime_racine h =
           let n = h.nombre and racine = h.contenu.(0) in
           h.contenu.(0) <- h.contenu.(n-1);
           h.nombre <- n-1;
           percole h;
           racine
where percole h=...
```

Percolation : faire redescendre l'étiquette de la racine tout au long du tas en l'échangeant récursivement avec la plus grande des étiquettes de ses fils, jusqu'à ce que son étiquette soit plus grande que toutes les étiquettes de ses fils.

Caml

```
#let percole h =
        let n = h.nombre and v = h.contenu in
        let rec traite_noeud i = (* échange l'élément i avec le plus grand
                                    de ses fils si nécessaire et recommence *)
             if 2*i+1 < n then (* le noeud a au moins un fils *)
               let j=2*i+1 in (* le fils gauche *)
               let fils = (* sélectionne le plus grand des "deux" fils *)
                 if (j+1<n && v.(j)<v.(j+1)) then j+1 else j
               in
                 if v.(fils) > v.(i) then
                    begin
                      echange i fils v;
                      traite_noeud fils
                    end
       in traite_noeud 0;;
percole : tas -> unit = <fun>
```

Caml

```
#let echange i j v = (* échange les éléments de numéros i et j du tableau v*)
             let temp = v.(i) in v.(i) <- v.(j); v.(j) <- temp;;
```

Amélioration : les échanges concernent toujours un même élément, la racine.
Pas nécessaire d'effectuer complètement chaque échange :
– affecter au père l'étiquette de son fils lorsqu'un échange est nécessaire
– lorsque le processus s'arrête, d'affecter au noeud l'étiquette de la racine.

Caml

```
let percole h =
        let n = h.longueur and v = h.contenu and racine = h.contenu.(1) in
        let rec traite_noeud i = (* échange l'élément i avec le plus petit
                                    de ses fils si nécessaire et recommence *)
             if 2*i+1 < n then (* le noeud a au moins un fils *)
                begin
                  let j = 2*i+1 in (* le fils gauche *)
                  let fils = (* sélectionne le plus petit des "deux" fils *)
                     if (j+1<n && v.(j)<v.(j+1)) then j+1 else j
                  in
                     if v.(fils) > racine then
                       begin
                         v.(i) <- v.(fils); (* on remonte le fils *)
                         traite_noeud fils  (* on traite la branche *)
                       end
                     else
                       v.(i) <- racine (* on stocke l'élément *)
               end
            else (* le noeud est terminal *)
               v.(i) <- racine (* on stocke l'élément *)
       in traite_noeud 1;;
percole : tas -> unit = <fun>
```

Complexité de cet algorithme est dominée par la hauteur de l'arbre, c'est donc un $O(\log_2 n)$ si n est le nombre de noeuds.

5.12 Insertion dans un tas

La technique d'insertion dans un arbre maximier par induction structurelle est totalement inadaptée à la structure de tas.
Cette méthode consistait
– dans un cas à insérer l'élément dans la branche gauche ou la branche droite de l'arbre

– dans l'autre cas à remplacer l'étiquette de la racine par le nouvel élément puis à insérer l'ancienne racine dans la branche gauche ou la branche droite de l'arbre

Détruit la structure d'arbre complet : un seul chemin d'insertion dans l'arbre qui permette de maintenir cette structure, mais il faudrait pratiquement être devin pour le trouver.

Privilégier la structure d'arbre complet en insérant le nouvel élément à la première place libre. On détruit la structure d'arbre maximier.

Rétablir la structure de tas par une opération inverse de la percolation : remonter l'élément de proche en proche en l'échangeant avec l'étiquette de son père tant que cette étiquette est inférieure.

Caml

```
#let insere x h =
            if h.nombre >= vect_length h.contenu then
                  failwith "débordement du tas";
            h.contenu.(h.nombre) <- x;
            h.nombre <- h.nombre +1;
            retablis h
where retablis h =
            let n = ref (h.nombre-1) in (* dernier élément *)
            let p = ref ((!n-1)/2) (* son père *)
            and v = h.contenu in
              while (!n>=1 && v.(!p) < v.(!n)) do
                    echange !n !p  v;
                    n := !p; (* nouveau fils *)
                    p := (!n-1)/2 (* nouveau père *)
              done;;
insere : int -> tas -> unit = <fun>
```

Validité de cet algorithme : la procédure `retablis` appliquée à un tableau qui est un tas, sauf peut-être en ce qui concerne son dernier élément, transforme ce tableau en un tas.

Cette procédure ne modifiant pas la structure de l'arbre conserve la propriété d'être un arbre complet. Il suffit donc de montrer qu'elle transforme l'arbre étiqueté en un arbre maximier.

Terminaison de la boucle : stricte décroissance du contenu de la référence n (qui est divisée par 2 à chaque itération) et le contenu de n est supérieure ou égale à 1 (ce qui garantit que n n'est pas la racine de l'arbre).

Invariant de la boucle :

– au début de l'itération, p est père de n, les deux branches issues de p sont des files de priorité

– l'étiquette de p est supérieure ou égale à celle de son autre fils éventuel n'

La complexité de cette insertion est manifestement majorée par la hauteur de l'arbre, elle est donc un $O(\log_2 n)$ où n est le nombre de noeuds.

5.13 Tri en tas

Le principe général : on insère les éléments du tableau successivement dans un tas initialement vide puis on extrait itérativement la racine de ce tas (qui en est toujours le plus petit élément) jusqu'à ce que ce tas soit vide.

Caml

```
#let tri_en_tas v =
         let n = vect_length v in
         let h = nouveau_tas n in
         for i=0 to n-1 do
             insere v.(i) h
         done;
         for i=0 to n-1 do
             v.(i) <- supprime_racine h
         done;;
tri_en_tas : int vect -> unit = <fun>
```

Comme l'insertion et la suppression de la racine ont une complexité en $O(\log_2 n)$, le tri en tas a une complexité en $O(n \log_2 n)$ dans le pire des cas.

Eviter le recours à un tas auxiliaire : économie de temps et de mémoire.

Propager la structure de tas à partir des sous-arbres de l'arbre complet obtenu à partir du tableau, en remontant la structure de tas depuis les noeuds terminaux jusqu'à la racine de l'arbre.

Les sous-arbres réduits aux noeuds terminaux sont évidemment des tas.

Soit n un noeud non terminal et supposons que la ou les branches issues de ce noeud soient des tas. Si le sous-arbre de racine n n'est pas un tas, il le deviendra en effectuant une percolation de l'étiquette de n à travers ce sous arbre. Après l'exécution complète de cette procédure, le tableau aura été transformé en un tas sans avoir utilisé d'insertions dans un autre tas.

Caml

```
#let vect_en_tas v=
        let n = (vect_length v)-1 in
        let rec traite_noeud i a_inserer=
                                (* échange l'élément i avec le plus grand
                                   de ses fils si nécessaire et recommence;
                                   au final insère l'élément a_inserer *)
             if 2*i+1 <= n then (* le noeud a au moins un fils *)
                begin
                  let j = 2*i+1 in (* le fils gauche *)
                  let fils = (* sélectionne le plus grand des "deux" fils *)
                     if (j<n && v.(j)<v.(j+1)) then j+1 else j
                  in
                     if v.(fils) > a_inserer then
                       begin
                         v.(i) <- v.(fils); (* on remonte le fils *)
                         traite_noeud fils a_inserer (* on traite la branche *)
                       end
                     else
                       v.(i) <- a_inserer (* on insère l'élément *)
               end
```

Caml

```
             else (* le noeud est terminal *)
                v.(i) <- a_inserer (* on insère l'élément *)
       in
             for i = (n+1)/2 downto 0 do
                    traite_noeud i v.(i)
             done;;
vect_en_tas : 'a vect -> unit = <fun>
```

Invariant de la dernière boucle indexée par i :

– après exécution de l'itération d'indice i, tous les sous-arbres ayant pour racines $v.(i), v.(i+1) \ldots, v.(n)$ sont des tas

La terminaison de la boucle est évidente puisqu'il s'agit d'une boucle indexée. Après l'exécution de l'itération d'indice 0, le tableau tout entier a été transformé en tas.

L'extraction successive des racines peut alors se faire directement en place : pour extraire la racine d'un tas (c'est à dire son plus grand élément), nous l'avons échangée avec le dernier élément du tas (ce qui la place donc en dernière position), puis nous l'avons supprimée (ce qui revient à diminuer de 1 la longueur de travail) et enfin nous avons effectué une percolation de l'étiquette du premier élément du tableau. En itérant cette méthode, nous allons donc obtenir un tableau dans lequel le plus grand élément sera en dernière position, puis l'élément juste un peu plus petit sera en avant-dernière position, et ainsi de suite. Autrement dit, notre tableau sera trié en ordre croissant.

Ceci nous conduit à la procédure suivant de tri par ordre croissant d'un tas :

Caml

```
#let tri_tas v =
       let rec traite_noeud i a_inserer n =
                        (* échange l'élément i avec le plus petit
                           de ses fils si nécessaire et recommence;
                           au final insère l'élément a_inserer;
                           ne traite que les éléments du tableau d'indice 0 à n *)
             if 2*i+1 <= n then (* le noeud a au moins un fils *)
```

```
                begin
                  let j = 2*i+1 in (* le fils gauche *)
                  let fils = (* sélectionne le plus grand des "deux" fils *)
                     if (j<n && v.(j)<v.(j+1)) then j+1 else j
                  in
                     if v.(fils) > a_inserer then
                       begin
                         v.(i) <- v.(fils); (* on remonte le fils *)
                         traite_noeud fils a_inserer n (* on traite la branche *)
                       end
                     else
                       v.(i) <- a_inserer (* on insère l'élément *)
               end
```

Caml

```
            else (* le noeud est terminal *)
               v.(i) <- a_inserer (* on insère l'élément *)
       in
         for n = (vect_length v)-1 downto 1 do
echange 0 n v;
           traite_noeud 0 v.(0) (n-1)
         done;;
tri_tas : 'a vect -> unit = <fun>
```

Invariant de la boucle finale :

après l'itération d'indice n les éléments $v.(0), \ldots, v.(n-1)$ forment un tas et on a

$$v.(0) \geq v.(n) \geq v.(n+1) \geq \cdots \geq v.(N-1)$$

si N désigne la longueur du tableau.

Tri en tas d'un tableau :

Caml

```
#let tri_en_tas v= (* trie les éléments du tableau v *)
let rec traite_noeud i a_inserer n=
                      (* échange l'élément i avec le plus petit
                         de ses fils si nécessaire et recommence;
                         au final insère l'élément a_inserer;
                         ne traite que les éléments du tableau
                         d'indice 0 à n *)
             if 2*i+1 <= n then (* le noeud a au moins un fils *)
                begin
                  let j = 2*i+1 in (* le fils gauche *)
                  let fils = (* sélectionne le plus petit des "deux" fils *)
                     if (j<n && v.(j)<v.(j+1)) then j+1 else j
                  in
                     if v.(fils) > a_inserer then
                       begin
                         v.(i) <- v.(fils); (* on remonte le fils *)
                         traite_noeud fils a_inserer n(* on traite la branche *)
                       end
                     else
                       v.(i) <- a_inserer (* on insère l'élément *)
                end
             else (* le noeud est terminal *)
                v.(i) <- a_inserer (* on insère l'élément *)
```

Caml

```
       in
        let dernier = (vect_length v) -1 in
            for i=(dernier+1)/2 downto 0 do
                   traite_noeud i v.(i) dernier
            done;
            for n=dernier downto 1 do
echange 0 n v;
                   traite_noeud 0 v.(0) (n-1)
            done;;
tri_en_tas : 'a vect -> unit = <fun>
```

Caml

```
#let v= [|8;1;2;3;4;3;8;5;1;8;2;4;12;14;1;3;5|];;
v : int vect = [|8; 1; 2; 3; 4; 3; 8; 5; 1; 8; 2; 4; 12; 14; 1; 3; 5|]
#tri_en_tas v; v;;
- : int vect = [|1; 1; 1; 2; 2; 3; 3; 3; 4; 4; 5; 5; 8; 8; 8; 12; 14|]
```

Comme la procédure `traite_noeud` a une complexité en $O(\log_2 n)$ et que les deux boucles sont au plus de longueur n, la complexité de la procédure de tri en tas est en $O(n \log_2 n)$ dans le pire des cas.