Chapitre 6 : Compléments sur les structures de données

Table des matières

1 Arbres

1.1 Définition, vocabulaire

1.1.1 Définition (*arbre*)

Un arbre est un ensemble fini A muni d'une relation $<$ telle que, si $A \neq \varnothing$:

(1) $\exists r_A \in A, \mid \forall x \in A,\ r_A \not< x$

(2) $\forall x \in A \setminus \{r_A\},\ \exists! y \in A \mid x < y$

(3) $\forall x \in A \setminus \{r_A\},\ \exists n \in \mathbb{N},\ \exists (x_k)_{k \in [\![0\ ;\ n]\!]} \subset A \mid x = x_0 < x_1 < \ldots < x_n < r_A$

1.1.2 Proposition

Soit $(A,\ <)$ un arbre non vide.
r_A est unique et $\forall x \in A \setminus \{r_A\}$, on a l'unicité de $n, x_0 \ldots x_n$ dans le point (3)

□ Démonstration :
• S'il existe $r \neq r_A$ vérifiant les mêmes propriétés, alors par (2), $\exists x \in A \mid r < x$.
Ceci contredit (1) pour r.

• Soient $\left| \begin{array}{l} x \in A \setminus \{r_A\} \\ n, m \in \mathbb{N} \\ (x_k)_{k \in [\![0\ ;\ n]\!]}, (y_k)_{k \in [\![1\ ;\ m]\!]} \subset A \end{array} \right.$ tel que $\begin{cases} x = x_0 < x_1 < \ldots < x_n < r_A \\ x = y_0 < y_1 < \ldots < y_m < r_A \end{cases}$

On suppose sans perte de généralité que $n \leqslant m$.
Supposons $\exists i \in [\![0\ ;\ n]\!] \mid x_i \neq y_i$. On choisit i minimal.
$i \neq 0$ car $x_0 = x = y_0$.

On a donc $\begin{cases} x_{i-1} < x_i \\ x_{i-1} = y_{i-1} < y_i \\ x_i \neq y_i \end{cases}$ qui contredit le point (2), donc $\forall i \in [\![0\ ;\ n]\!],\ x_i = y_i$.

Si $n < m$, $\begin{cases} x_n < r_A \\ x_n = y_n < y_{n+1} \\ r_A \neq y_{n+1} \text{ car } n+1 \leqslant m \text{ et } \forall j \in [\![1\ ;\ m]\!],\ r_A \neq y_i \text{ (contredit (1))} \end{cases}$

Cela contredit (2)
Donc $n = m$ ■

1.1.3 Vocabulaire

Soit $(A,\ <)$ un arbre non vide.
– Les éléments de A sont appelés *nœuds* ;
– r_A est appelé la *racine* de A ;
– $\forall x, y \in A \mid x < y$, y est le *père* de x et x est un *fils* de y ;
– Une *feuille* est un nœud sans fils. On parle aussi de nœud externe ;

– Un nœud qui n'est pas une feuille est dit *interne* ;
– L'*arité* d'un nœud est le nombre de ses fils ;
– Profondeur d'un nœud :
  * La profondeur de r_A est 0 ;
  * La profondeur de $x \neq r_A$ est l'entier $n+1$ où n est défini au point (3) de la définition 1.1.1.

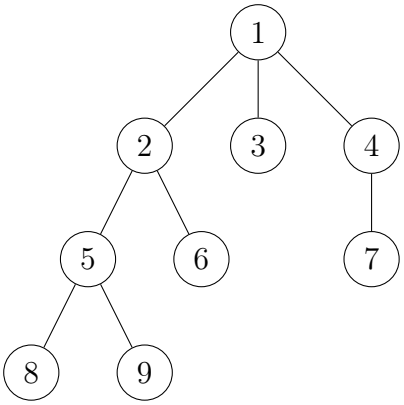
  C'est la longeur du chemin qui mène du nœud à la racine.

– Hauteur de A : c'est la profondeur maximale d'un nœud. Par convention, la hauteur de l'arbre vide est -1.

Exo Mq c'est aussi la profondeur maximale d'une feuille.

1.1.4 Représentation graphique, exemple

$A = [\![1\ ;\ 9]\!]$ avec $<= \{(2,1),(3,1),(4,1),(7,4),(5,2),(6,2),(8,5),(9,5)\}$

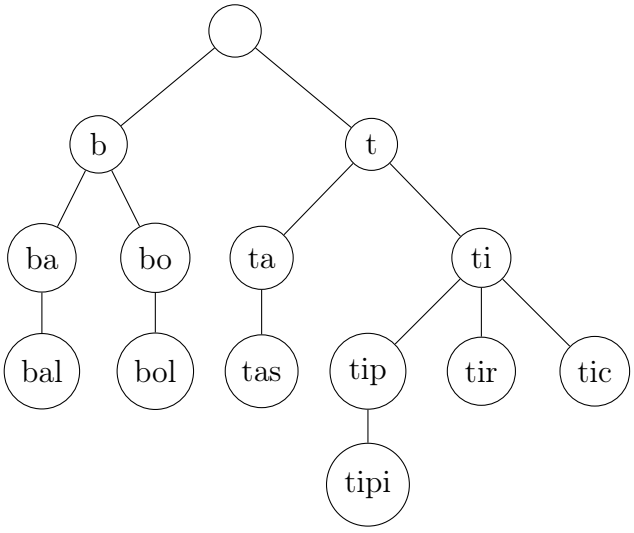


Racine : 1
Feuilles : 8, 9, 6, 3, 7
Arité de 1 : 3
Profondeur de 5 : 2
Hauteur : 3

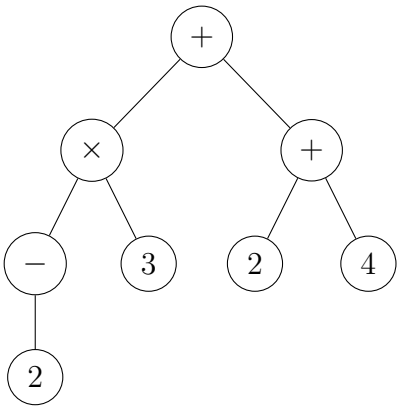
1.1.5 Remarque

On peut aussi associer des données aux nœuds afin d'obtenir une structure de données où l'information est hiérarchisée. On appelle étiquette d'un nœud l'information qui lui est associee.

Exemple : les tries, ou *arbres préfixes*, qui implémentent la structure de dictionnaire



Autre exemple : les arbres de syntaxe, par exemple les expressions arithmétiques



$((-2) \times 3) + (2 + 4)$ (notation infixe)
En préfixe (notation polonaise) : $+ \times -2\ 3 + 2\ 4$
Quand on observe un nœud et tous ces descendants, on retrouve un arbre valide.

1.1.6 Définition (*sous-arbre*)

Soit $(A, <)$ un arbre et $x \in A$.
On appelle *sous-arbre* de A enraciné en x l'arbre $(A_x, < \cap (A_x \times A_x))$ où A_x est le sous ensemble de A constitué de x et de ses descendants.

1.1.7 Remarque

Cela donne l'intuition d'une définition inductive des arbres :

$$\frac{}{\text{Vide est un arbre}} \quad \text{et} \quad \frac{A_1 \text{ est un arbre } \ldots\ A_n \text{ est un arbre}}{\text{Nœud}(A_1, \ldots, A_n) \text{ est un arbre}}$$

Dans ce contexte, une feuille est un nœud dont les sous-arbres sont vides.
On en déduit une implémentation du type des arbres en OCaml :

```
type arbre =
    | Vide
    | Noeud of arbre list
```

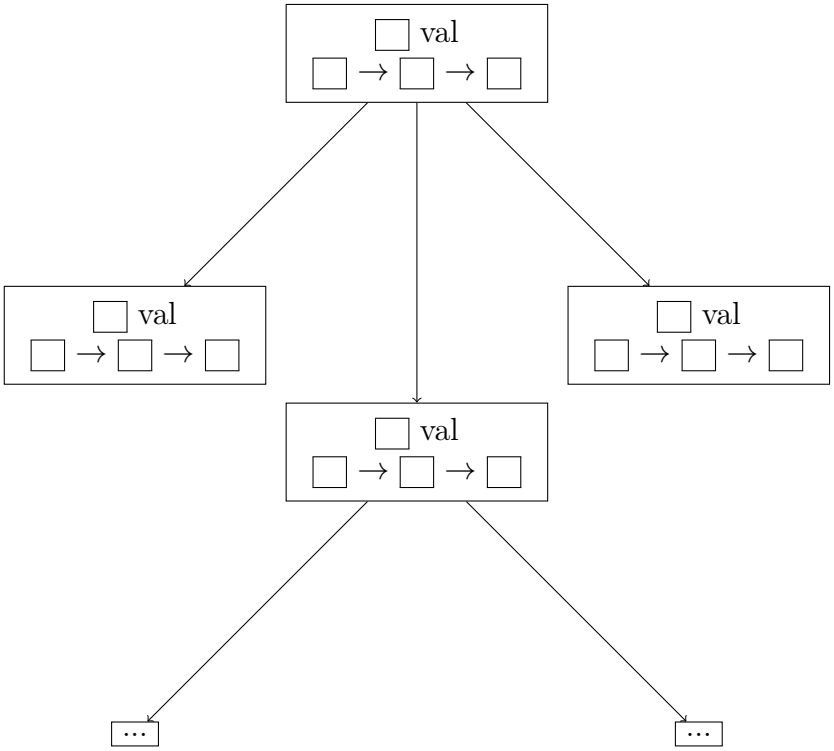
Ou avec des étiquettes :

```
type 'a arbre =
    | Vide
    | Noeud of 'a * ('a arbre list)
```

En C :

```
struct sarbre;
struct sliste;

struct sarbre {
    int val; //tag
```

```
6       struct sliste* fils;
7   };
8
9   struct sliste {
10      struct sabre* noeud;
11      struct sliste* suivant;
12  };
13
14  typedef struct sliste* listeAbre;
15  typedef struct sabre* arbre;
```



1.1.8 Définition (*arbre binaire*)

Un *arbre binaire* est un arbre dont tous les nœuds sont d'arité au plus 2.

Un arbre binaire est dit strict si tous les nœuds internes sont d'arité 2.

1.1.9 Exemple

L'arbre de syntaxe 1.1.5 est binaire mais pas strict. Si on remplace -2 par $0-2$, il devient strict.

1.1.10 Proposition

Soit A un arbre binaire dont on note n le nombre de nœuds internes.

Alors :

(1) A a au plus $n+1$ feuilles ;
(2) Si A est strict, alors A a exactement $n+1$ feuilles ;
(3) Si A est de hauteur h, alors A a au plus 2^h feuilles ;
(4) Si A a k feuilles, alors A est de hauteur au moins $\lceil \log_2(k) \rceil$

□ Démonstration :
(1) et (2) : par récurrence sur n.
(3) par induction structurelle sur A
(4) corollaire de (3)
■

1.1.11 Implémentation

On remplace la liste des fils par un fils gauche et un fils droit.
OCaml :

```
type 'a bin_tree =
    | Empty
    | Node of 'a * 'a bin_tree * 'a bin_tree
```

En interdisant l'arbre vide, on obtient un type pour les arbres binaires stricts non vides :

```
type 'a strict_bin_tree =
    | Leaf of 'a
    | Node of 'a * 'a strict_bin_tree * 'a strict_bin_tree

```

C :

```
struct noeud {
    int val;
    struct noeud* gauche;
    struct noeud* droite;
};

typedef struct noeud* arbre_binaire;

```

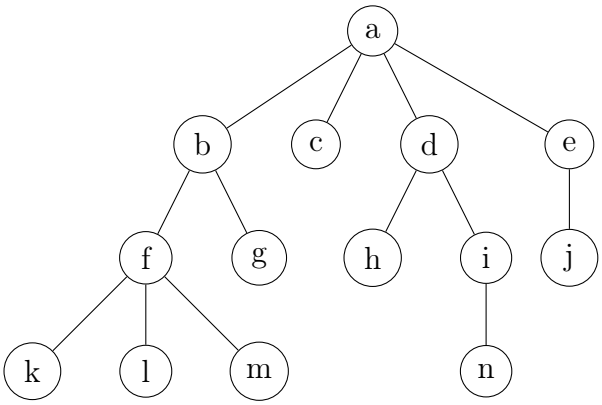
1.1.12 Conversion en arbre binaire

On utilise la représentation "*fils gauche-frère droit*", où l'arbre binaire comporte autant de nœuds, avec les mêmes étiquettes que l'arbre original et est organisté comme suit :

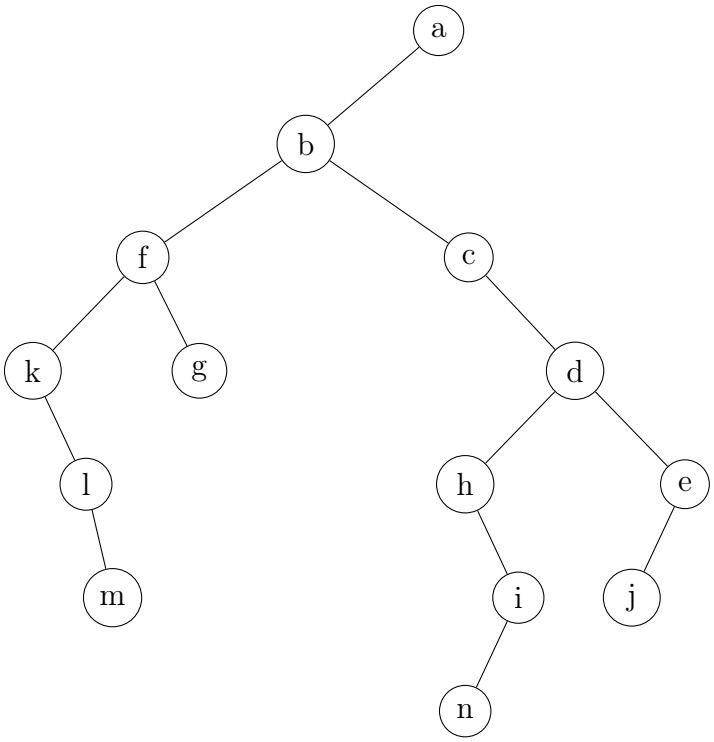
– Le fils gauche d'un nœud N, est le « premier » fils de N dans l'arbre initial (vide s'il n'a pas de fils) ;

– le fils droit d'un nœud N est le « frère suivant » de N dans l'arbre initial, c'est-à-dire un nœud de même père qui n'a pas encore été inséré dans l'arbre binaire (vide s'il n'y a pas d'autre frère).

Exemple :



donne :



Nombre de nœuds `#noeuds` d'un arbre d'arité quelconque :

```
let rec nb_noeuds (t : arbre) : int =
    match t with
    | Vide -> 0
    | Noeud l -> List.fold_left (fun a t' -> a + nb_noeuds t') 1 l
```

`List.fold_left : ('a -> 'b - > 'a) -> 'a -> 'b list -> 'a`

`List.fold_left f a` $[x_1 \dots x_n]$ `= f( ... f(f(a,` x_1`),` x_2`), ...,` x_n`)`

1.2 Parcours d'arbres

1.2.1 Introduction

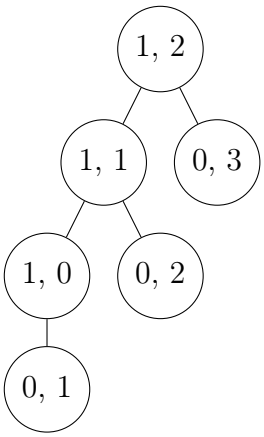
On est souvent amené à parcourir l'ensemble des nœuds d'un arbre, par exemple pour vérifier la présence d'une étiquette ou bien pour appliquer une fonction à chaque étiquette.

Par exemple, lorsqu'on étudie une fonction récursive, l'ensemble des appels récursifs est hiérarchisée selon une structure arborescente, où la racine est l'appel de fonction initial et les fils d'un nœud, *i.e* d'un appel de la fonction, sont les appels récursifs réalisés lors de cet appel.

Exemple : la fonction d'Ackermann :

```
int ack(int n, int m) {
    if (n == 0)
        return m + 1;
    if (m == 0)
        return ack(n - 1, 1);
    return ack(n - 1, ack(n, m - 1));
}
```

`ack(1, 2)` :



L'ordre d'exécution des appels récursifs correspond à l'ordre d'empilement des blocs d'activation de la fonction sur la pile d'exécution. Cet ordre d'empilement correspond à un parcours des nœuds de l'arbre des appels récursifs dans un arbre bien précis, appelé *parcours en profondeur*.

1.2.2 Parcours en profondeur

Dans un parcours en profondeur, ou DFS (*Deep First Search*), on parcours intégralement le sous-arbre enraciné en un fils d'un nœud avant de passer au fils suivant.

Rq : cela correspond à l'enchaînement des appels récursifs, puisqu'il faut attendre la fin d'un appel avant de passer au suivant.

Le moment où l'on applique une fonction à la racine d'un sous-arbre définit le type de parcours en profondeur qui est réalisé :

– Si on applique la fonction au nœud avant de parcourir ses fils, on parle de parcours *préfixe* ;

– Si on applique la fonction au nœud après le parcours de ses fils, on parle de parcours *postfixe*.

Exemple avec l'arbre des appels récursif de `ack(1, 2)` (1.2.1) :

– Préfixe : (1, 2), (1, 1), (1, 0), (0, 1), (0, 2), (0, 3)

– Postfixe : (0, 1), (1, 0), (0, 2), (1, 1), (0, 3), (1, 2)

Code en C pour afficher les étiquettes (entières) d'un arbre dont les fils sont donné dans une liste (*cf* 1.1.7 pour la structure) :

```c
void dfs(arbre t) {
    if (t != NULL) {
        printf("%d\n", t->val); //Parcours prefixe
        listeArbre l = t->fils;

        while (l != NULL) {
            dfs(l->noeud);
            l = l->suivant;
        }

        //printf ici pour le parcours postfixe.
    }
}
```

Rq : les parcours préfixes et postfixe de l'arbre représentant une expression arithmétique donnent respectivement les représentations polonaise et polonaise inverse de l'expression (*cf* TD_{17})

Dans le cas des arbres binaires (stricts), on distingue un troisième type de parcours en profondeur, le parcours *infixe*, où l'on traite l'étiquette d'un nœud entre les parcours de ses fils gauche et droite (le parcours infixe d'un arbre binaire représantant une expression arithmétique donne la représentation infixe).

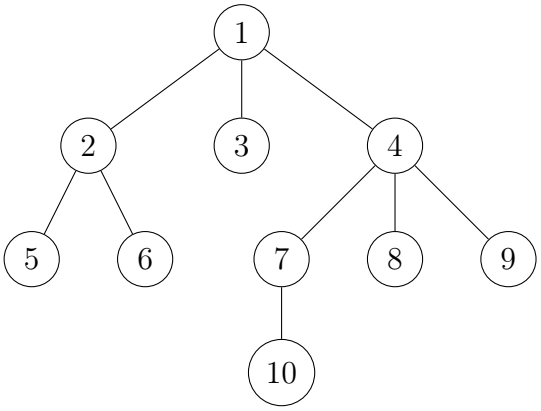
Code en C (*cf* 1.1.11 pour le type) :

```c
void infix_dfs(arbre_binaire t) {
    if (t != NULL) {
        infix_dfs(t->gauche);
        printf("%d\n", t->val);
        infix_dfs(t->droite);
    }
}
```

1.2.3 Parcours en largeur

Dans un parcours en largeur, ou BFS (*Breadth First Search*), on traite tous les nœuds à une même profondeur avant de passer à la profondeur suivante.

Exemple :



Comment implémenter un tel parcours efficacement ?

On ne veut pas avoir à remonter jusqu'à la racine pour passer d'un nœud à un autre de même profondeur.

Idée : on conserve un ensemble de sommets en attente d'être traités. On commence par la racine et quand on traite un nœud, on ajoute ses fils à l'ensemble des sommets en attente. Cela fonctionne si on peut extraire un nœud de profondeur minimale de cet ensemble. On peut utiliser pour cela une structure appelée *file de priorité*, où la priorité d'un nœud est sa profondeur. Dans ce cas particulier, on n'a pas besoin des files de priorité dans leur généralité, mais seulement d'une file.

Code en OCaml (*cf* 1.1.7 pour le type) :

```
let bfs (t : int arbre) : unit =
    let f = Queue.create() in
    Queue.add t f;

    while not (Queue.is_empty f) do
        match Queue.take f with
        | Vide -> ()
        | Noeud(n, l) ->
            print_int n;
            List.iter (fun t -> Queue.add t f) l
    done
```

Correction : on doit montrer que les nœuds sont traités par ordre croissant de profondeur.

On note $p(n)$ la profondeur du nœud n et on montre l'invariant $I(f)$ = «si f s'écrit $n_1, \dots, n_k$ (où n_1 est le premier nœud qui sera extrait) alors $p(n_1) \leqslant p(n_2) \leqslant \cdots \leqslant p(n_k) \leqslant p(n_1) + 1$»

• Avant d'entrer dans la boucle, f ne contient que la racine t et $p(t) \leqslant p(t) + 1$

• Si $I(f)$ est vrai au début d'une itération : comme f est non vide, f s'écrit $n_1, \dots, n_k$ avec $k \geqslant 1$.

On extrait donc n_1

– Si $n_1 =$ `Vide`, à la fin de l'itération (si $k \neq 1$ (cas $k = 1$ trivial)) f s'écrit $n_2, \dots, n_k$ et $p(n_2) \leqslant p(n_3) \leqslant \cdots \leqslant p(n_k) \leqslant p(n_1)+1 \leqslant p(n_2)+1$ car $p(n_1) \leqslant p(n_2)$.

– Si n_1 est un nœud de fils $n_{k+1}, \dots, n_{k+l}$, à la fin de l'itération f s'écrit $n_2, \dots, n_k, n_{k+1}, \dots, n_{k+l}$
et $p(n_2) \leqslant p(n_3) \leqslant \cdots \leqslant p(n_k) \leqslant p(n_1)+1 = p(n_{k+1}) = \cdots = p(n_{k+l}) \leqslant p(n_{k+1})+1$
donc les nœuds sont traités par ordre croissant de profondeur.

• Et si on avait utilisé une pile ? On retrouve un parcours en profondeur.

□ Démonstation :

On démontre par induction structurelle sur n que si le nœud n est extrait de la pile, alors la suite de l'algorithme réalise un parcours en profondeur du sous-arbre enraciné en n et laisse la pile dans le même état que juste après l'extraction de n.

– Si $n =$ `Vide` : pas de problème.

– Si n est un nœud de fils $n_1, \dots, n_k$ et p est la pile après l'extraction de n, la pile s'écrit $n_k, n_{k-1}, \dots, n_1, p$ après insertion des fils. Ensuite, n_k est extrait par hypothèse d'induction on réalise un parcours en profondeur du sous-arbre enraciné en n_k, qui laisse la pile dans l'état $n_{k-1}, n_{k-2}, \dots, n_1, p$
De même pour $n_{k-1}, \dots, n_1$ (récurrence finie sur k) on a bien réalisé un parcours en profondeur du sous-arbre enraciné en n qui laisse la pile dans l'état ↑. ■

Rq : dans le cas des arbres binaires, on préfère insérer d'abord le fils droit, puis le fils gauche pour les parcourir dans le bon ordre.

```
let dfs (t : 'a bin_tree) : unit =
    let p = Stack.create() in
    Stack.push t p;

    while not (Stack.is_empty p) do
        match Stack.pop p with
        | Empty -> ()
        | Node(_, l, r) ->
            Stack.push r p;
            Stack.push l p
    done
```

1.2.4 Complexité des parcours

Chaque nœud est parcouru une seule fois donc on a une complexité en

$$\mathcal{O}(n \times \text{complexité du traitement appliqué au nœud})$$

Dans le cas où ce traitement est de complexité constante on obtient un parcours en temps linéaire.

2 Arbres binaires de recherche (ABR)

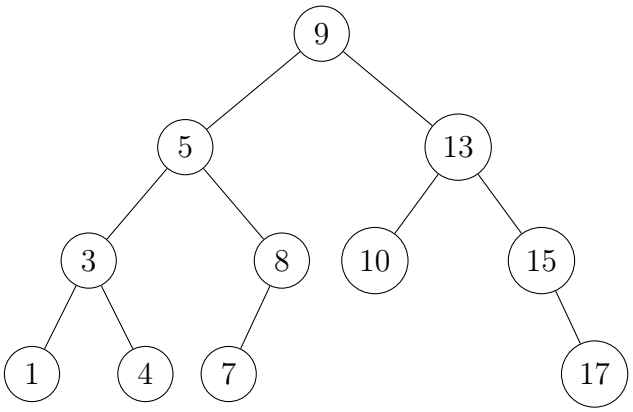
2.1 Définition et opérations

2.1.1 Définition ABR

Soit $(E, \leqslant)$ un ensemble totalement ordonné.

Un ABR étiqueté par des éléments de E est tel que pour tout nœud, l'étiquette de ce nœud est supérieure à toutes celles de son sous-arbre gauche et inférieure à toutes celles de son sous-arbre droit.

2.1.2 Exemple



2.1.3 Proposition

Soit $(E, \leqslant)$ un ensemble totalement ordonné et A un arbre binaire étiqueté par des éléments de E.

A est ABR $\Leftrightarrow$ le parcours en profondeur infixe de A est croissant.

□ Démonstration (par induction structurelle sur A) :

– Si A est vide : trivial

– Si A est un nœud d'étiquette x et de sous-arbre gauche et droit g et d

$\Rightarrow$ le parcours infixe de A s'écrit $P_g\ x\ P_d$ où P_g et P_d sont les parcours infixes de g et d.

g et d sont des ABR (par restriction de la quantification universelle) donc d'après l'hypothèse d'induction P_g et P_d sont croissants.

Or $\max P_g \leqslant x \leqslant \min P_d$, donc $P_g\ x\ P_d$ croissant.

$\Leftarrow$ Le parcours $P_g\ x\ P_d$ est croissant.

Donc P_g et P_d sont croissants et par hypothèse d'induction, g et d sont des ABR.

Pour montrer que A est un ABR, il suffit donc de remarquer que $\max P_g \leqslant x \leqslant \min P_d$ (vrai par croissance de $P_g\ x\ P_d$) ■

2.1.4 Remarque

– La définition donne l'intuition d'une définition inductive des ABR :

$$\frac{}{\text{Vide est un ABR}} \qquad \frac{}{\text{Nœud}(x, \text{Vide}, \text{Vide}) \text{ est un ABR}}$$

$$\frac{g \text{ est un ABR} \quad \max g \leqslant x}{\text{Nœud}(x, g, \text{Vide}) \text{ est un ABR}} \qquad \frac{d \text{ est un ABR} \quad x \leqslant \min d}{\text{Nœud}(x, \text{Vide}, d) \text{ est un ABR}}$$

$$\frac{g \text{ est un ABR} \quad d \text{ est un ABR} \quad \max g \leqslant x \leqslant \min d}{\text{Nœud}(x, g, d) \text{ est un ABR}}$$

– On peut trier des données en parcourant un ABR qui les contient.

2.1.5 Calcul du minimum / maximum des étiquettes d'un ABR

Il suffit de descendre dans le nœud le plus à gauche / droite de l'arbre.

☐ Par induction structurelle : pour le minimum (cas du maximum symétrique)

– Vide : cas non considéré.

– Nœud(x, g, d) : 2 cas

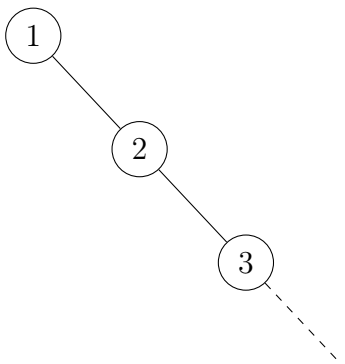
• g est vide : on est dans le nœud le plus à gauche et x est inférieur à toutes les étiquettes de d donc est le minimum ;

• g est non vide : le minimum est dans g, dans le nœud le plus à gauche de g par hypothèse d'induction.

Ce nœud est bien le nœud le plus à gauche de l'arbre ■

Exo Code.

Complexité : $\mathcal{O}(h)$ (à chaque étape, on incrémente la profondeur du nœud considéré)
Donc $\mathcal{O}(n)$ dans le pire cas



2.1.6 Remarque

On peut utiliser les ABR pour implémenter la structure de tableau associatif : on stocke les couples (clé, valeur) ordonnés selon les clés → ne fonctionne que si l'ensemble des clés est totalement ordonné.

2.1.7 Recherche d'étiquette

On procède récursivement en comparant l'étiquette recherchée à celle du nœud considéré.

Si elle est inférieure (resp. supérieure) on fait un appel récursif sur le sous-arbre gauche (resp. droit).

Cas de base : arbre vide et cas d'égalité.

Code :

```
let rec mem (x : 'a) (t : 'a bin_tree) : bool =
    match t with
    | Empty -> false
    | Node (y, _, _) when y = x -> true
    | Node (y, g, _) when y > x -> mem x g
    | Node (_, _, d) -> mem x d
```

Complexité : $\mathcal{O}(h)$

Exo : preuve de correction.

2.1.8 Ajout d'une étiquette

Idée : on descend dans l'arbre comme pour la recherche et lorsqu'on atteint un nœud `Vide`, on crée un nœud avec la nouvelle étiquette.

En cas d'égalité avec une étiquette, on peut laisser l'arbre inchangé, lever une exception, insérer à gauche ou à droite de manière déterministe ou aléatoire.

Code :

```
arbre_binaire new_node(int x) {
    arbre_binaire t = (arbre_binaire) malloc(sizeof(struct noeud))
    ;
    t->val = x;
    t->gauche = NULL;
    t->droite = NULL;
    return t;
}

void add(int x, arbre_binaire* t) {
    if (*t == NULL)
        *t = new_node(x);
    else if (x <= (*t)->val) {
        add(x, &((*t)->gauche));
    }
    else //x > (*t)->val
        add(x, &((*t)->droite));
}
```

Complexité : $\mathcal{O}(h)$

Exo : correction.

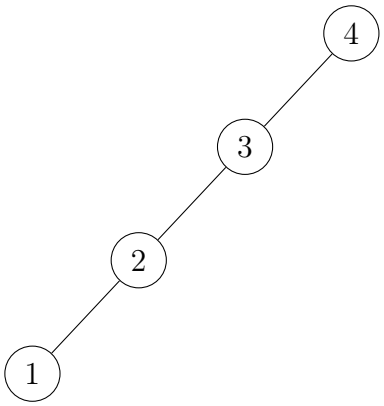
2.1.9 Tri à l'aide d'un ABR

On construit un ABR contentant les valeurs à trier par insertion successives et on réalise un parcours en profondeur infixe de l'ABR.

Code :

```
let rec abr_of_list (l : 'a list) : 'a bin_tree =
    match l with
    | [] -> Empty
    | t::q -> add t (abr_of_list q)
(*si add : 'a -> a' bin_tree -> 'a bin_tree*)

let abr_sort (l : 'a list) : 'a list = infix_dfs (abr_of_list l)
(*si infix_dfs : 'a bin_tree -> 'a list (Exo)*)
```

Complexité : $\mathcal{O}(n^2)$ dans le pire cas à cause du peigne : `l = [1 ; 2 ; 3 ; 4]` donne :



Remarque : d'après 1.1.10, le mieux qu'on puisse espérer est une hauteur en $\mathcal{O}(\log_2(n))$, donc une complexité en $\mathcal{O}(n\log_2(n))$ pour le tri.

Pour atteindre cette complexité, on peut rééquilibrer l'arbre en cours de construction, par exemple en utilisant une structure adaptée type arbre tricolore (*cf* 2.2)

2.1.10 Supression d'une étiquette

• Première idée : on remplace l'étiquette du nœud supprimé par le maximum du sous-arbre gauche.

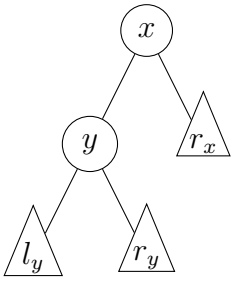
Code :

```
let rec take_max (t : 'a bin_tree) : 'a * 'a bin_tree =
    match t with
    | Empty -> failwith "Empty tree"
    | Node(x, l, Empty) -> (x, l)
    | Node (x, l, r) ->
        let (y, r') = take_max r in
        (y, Noeud(x, l, r'));;
```

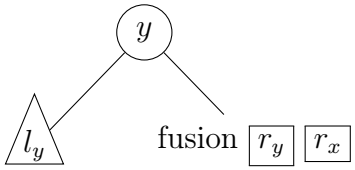
```
let rec del (x : 'a) (t : 'a bin_tree) : 'a bin_tree =
    match t with
    | Empty -> Empty
    | Node(y, l, r) when x < y -> Node(y, del x l, r)
    | Node(y, l, r) when y < x -> Node(y, l, del x r)
    | Node(_, l, r) ->
        match l with
        | Empty -> r
        | _ -> let (y, l') = take_max l in Node(y, l', r)
```

Complexité : $\mathcal{O}(h)$ pour `take_max`, et $\mathcal{O}(h)$ pour `del`.

• Seconde idée : on fusionne les sous-arbres du nœud supprimé.



Donne



Code :

```
let rec merge (l : 'a bin_tree) (r : 'a bin_tree) : 'a bin_tree =
    (*precondition : etiquettes de l inferieures a celles de r*)
    match l with
    | Empty -> r
    | Node(y, ly, r_y) -> Node(y, ly, merge ry r)

let rec del (x : 'a) (t : 'a bin_tree) : 'a bin_tree =
    match t with
    | Empty -> Empty
    | Node(y, l, r) when x < y -> Node(y, del x l, r)
    | Node(y, l, r) when y < x -> Node(y, l, del x r)
    | Node(_, l, r) -> merge l r

```

Complexité de `merge` : $\mathcal{O}(h)$
Complexité de `del` : $\mathcal{O}(h)$

2.2 Arbres bicolores

2.2.1 Introduction

Pour garantir une hauteur logarithmique pour les ABR (afin d'obtenir une complexité optimale pour les opérations sur les ABR), on ajoute des couleurs sur avec des nœuds

et on impose des contraintes sur le choix des couleurs qui vont garantir l'équilibre de l'arbre. Comme on utilise seulement deux couleurs, le surcoût en espace est faible : un bit d'information supplémentaire par nœud.

2.2.2 Définition (*arbre bicolore*)

Un *arbre bicolore* (ou *arbre rougre-noir*), est un ABR auquel on a ajouté des feuilles vides pour le rendre binaire strict (les feuilles de l'ABR deviennent des nœuds internes ayant deux feuilles vides comme filles) et tel que chaque nœud est coloré, soit en rouge, soit en noir, et en respectant les contraintes suivantes :

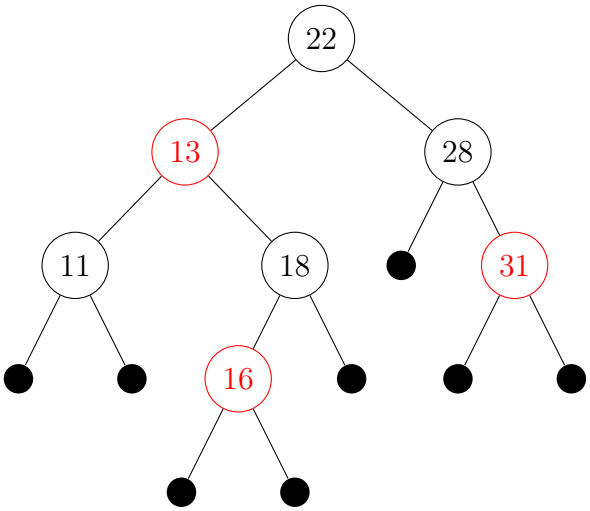
(1) Toutes les feuilles sont noires ;

(2) La racine est noire ;

(3) Les fils d'un nœud rouge sont noirs ;

(4) Tous les chemins de la racine vers les feuilles ont le même nombre de nœuds noirs.

2.2.3 Exemple



2.2.4 Définition (*hauteur noire*)

Soit A un arbre bicolore et x un nœud de A.

On appelle *hauteur noire* de x le nombre $h_N(x)$ de nœud noirs sur les chemins du nœud x vers les feuilles du sous-arbre enraciné en x.

La hauteur noire $h_N(A)$ de A est la hauteur noire de sa racine.

2.2.5 Proposition

Soit A un arbre bicolore dont on note n le nombre de nœuds internes.

Alors la hauteur noire de A est au plus $2\log_2(n+1)$

☐ Démonstration

Lemme : $\forall x$ nœud de A, le sous-arbre enraciné en x a au moins $2^{h_N(x)}-1$ nœuds internes.

☐ Démonstration (par induction structurelle) :

– Si x est une feuille, $h_N(x)=0$ et $2^{h_N(x)}-1=0$ est bien le nombre de nœuds internes du sous-arbre enraciné en x.

– Si x est un nœud interne de fils gauche g et droit d, les hauteurs noires de g et de d valent $h_N(x)-1$ selon la coloration.

Donc par hypothèse d'induction, les deux sous-arbres contiennent au moins $2^{h_N(x)-1}-1$ nœuds internes.

Le sous-arbre enraciné en x a donc au moins $2\left(2^{h_N(x)-1}-1\right)+1=2^{h_N(x)}-1$ ■

Par le point (3) de la définition 2.2.2, il y a au plus 1 nœud rouge sur 2 sur les chemins de la racine vers les feuilles, donc $h \leqslant 2h_N(A)$.

Or, par le lemme, $n \geqslant 2^{h_N(A)}-1 \geqslant 2^{\frac{h}{2}}-1$ donc $h \leqslant 2\log_2(n+1)$ ■

2.2.6 Remarque

Cette propriété garantit que les opérations des ABR sont de complexité logarithmique sur les arbres bicolores. Cependant, l'ajout et la suppression de nœuds peut enfreindre les règles de coloration. Il faut donc la rétablir, et nous verrons que cela ne change pas la complexité globale.

2.2.7 Implémentation des arbres bicolores

– En OCaml :

```
type color = Red | Black
type 'a tree =
    | Empty
    | Node of color * 'a * 'a tree * 'a tree
```

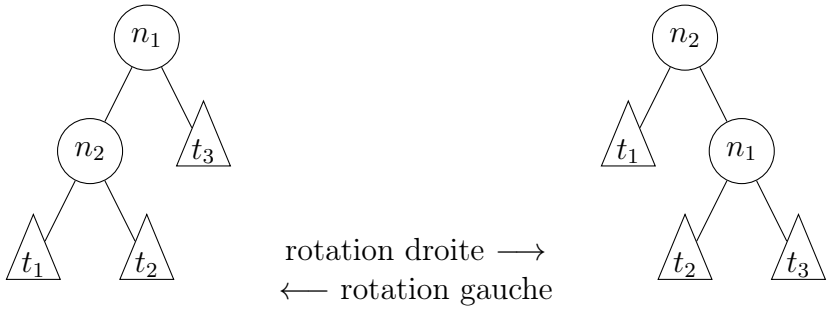
– En C :

```
struct node {
    int val;
    char color; //'R' or 'B'
    struct node* left;
    struct node* right;
    struct node* father;
};

typedef struct node* tree;
```

Le pointeur vers le père est utile pour une implémentation impérative des arbres rouges-noirs pour implémenter une opération appelée rotation.

2.2.8 Rotations

Une rotation est une modification locale d'un sous-arbre pour inverser des liens de parenté entre nœuds visant à réduire le déséquilibre de hauteur entre les fils gauche et droit, tout en conservant les propriétés des ABR.



Code en C :

```c
void right_rotation(tree n1, tree* root) {
    tree n2 = n1->left;
    n1->left = n2->right;

    if (n2->right != NULL)
        n2->right->father = n1;

    n2->father = n1->father;

    if (n2->father == NULL)
        *root = n2;
    else if (n1->father->left == n1)
        n1->father->left = n2;
    else
        n1->father->right = n2;

    n2->right = n1;
    n1->father = n2;
}
```

2.2.9 Insertion d'un nœud

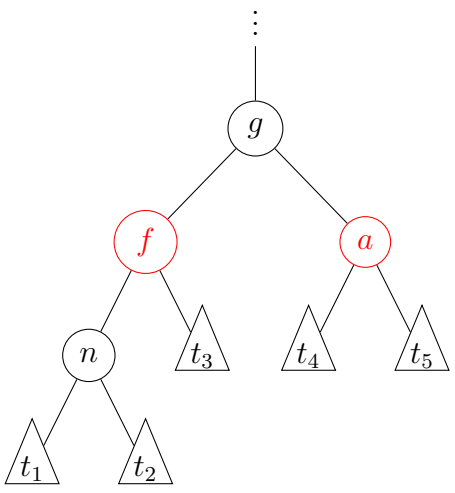
Idée : on procède comme dans les ABR (on descend jusqu'à une feuille qu'on remplace par un nouveau nœud) et on crée un nœud rouge par défaut : cela ne modifie pas la hauteur noire donc le résultat est un arbre bicolore valide sauf si le nouveau nœud est la racine ou si le père de ce nouveau nœud est rouge.

Code :

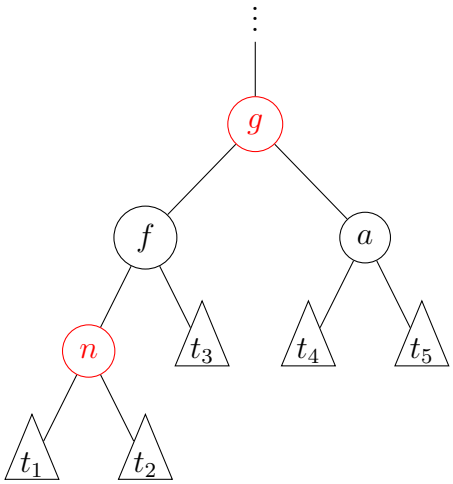
```c
void insert(int m, tree* root) {
    tree f = NULL, n = *root;

    while (n != NULL) {
        f = n;

        if (m < n->val)
            n = n->left;
        else
            n = n->right;
    }
    n = (tree) malloc(sizeof(struct node));
    n->val = m;
    n->father = f;
    n->left = NULL;
    n->right = NULL;
    n->color = 'R';

    if (f == NULL) {
        n->color = 'B';
        *root = n;
        return;
    }

    if (m < f->val)
        f->left = n;
    else
        f->right = n;

    fix_insert(n, root);
}
```

On distingue 4 cas pour la fonction `fix_insert` :

- Le père de n est noir : il n'y a rien à corriger ;
- Le père de n est rouge et son oncle est aussi rouge. À inversion des fils gauches / droit près, on est dans la situation :



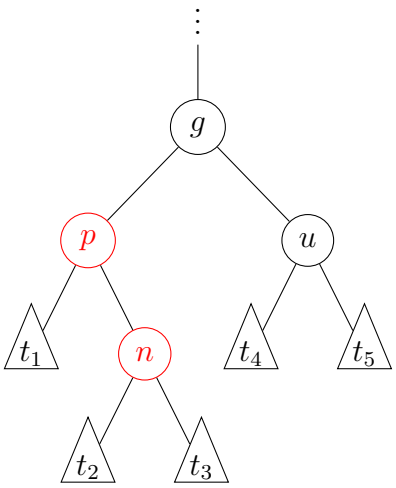
Changement de coloration ↓



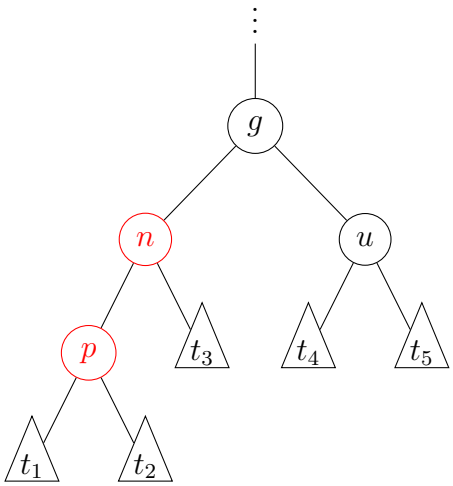
Puis appel récursif sur g.

– À cause d'un appel récursif, on obtient une racine rouge : on la colore en noir.

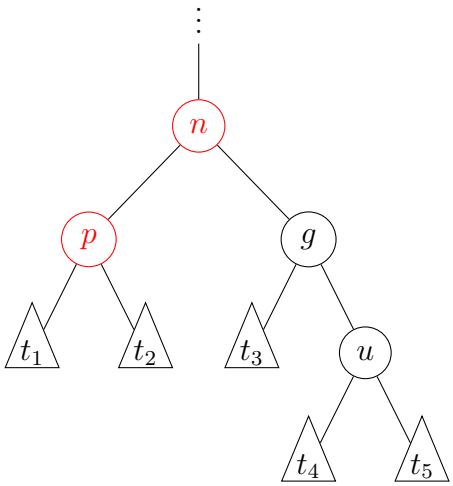
– Le père de n est rouge et son oncle est noir : on effectue si besoin une rotation du père de n pour obtenir le nœud rouge le plus bas comme petit-fils gauche-gauche ou droite-droite de son grand-père :



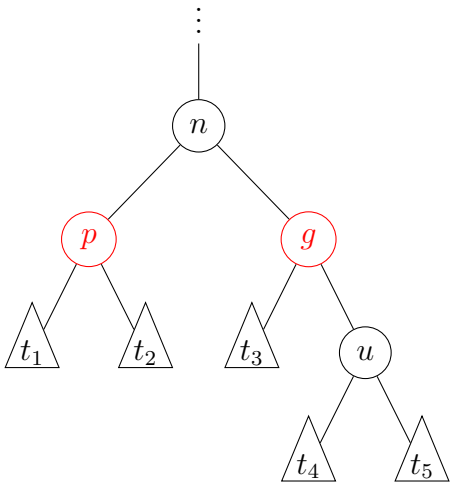
Rotation gauche de $p \downarrow$



On effectue ensuite une rotation grand-père et on rectifie la coloration.
Rotation droite de g :



Correction de coloration car $h_N(g) = h_N(p) + 1$:



pas d'appels récursifs.

Remarque : comme les appels récursifs éventuels font diminuer la profondeur de 2 unités, il y en a au plus $\frac{h}{2}$.

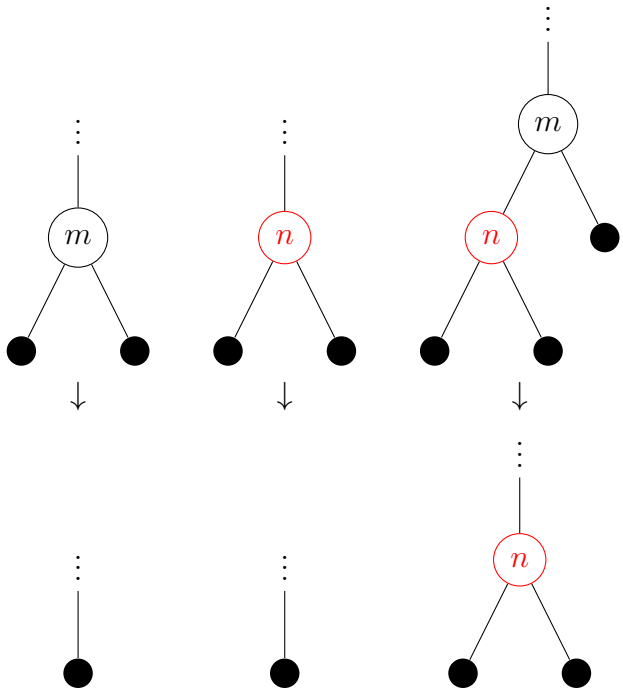
Donc la complexité de l'ajout de nœud reste logarithmique.

```
void fix_insert(tree n, tree* root) {
    if (n->father == NULL) {
        n->color = 'B';
        return;
    }

    if (n->father->color == 'B')
        return;

    if (uncle(n)->color == 'R') { //Exo : uncle implementation
        n->father->color = 'B';
        uncle(n)->color = 'B';
        n->father->father->color = 'R';
        fix_insert(n->father->father, root);
    }
    else {
        tree p = n->father, g = n->father->father;

        if (n == g->left->right) {
            left_rotation(p, root);
            swap(&n, &p); //Exo : swap implementation
        }
        else if (n == g->right->left) {
            right_rotation(p, root);
            swap(&n, &p);
        }

        if (n == p->left)
            right_rotation(g, root);
        else
            left_rotation(g, root);

        p->color = 'B';
        g->color = 'R';
    }
}

```

2.2.10 Supression d'un nœud

On ramène l'étude de la supression d'un nœud à celle de l'extraction du maximum grâce à la première idée de 2.1.10.

Le maximum est situé dans le nœud interne le plus à droite, donc son fils droit est une feuille vide. On a plusieurs cas possibles :



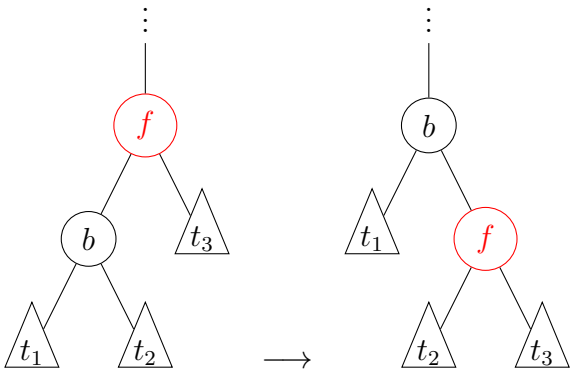
• Dans le troisième cas, on peut avoir un conflit rouge-rouge avec le père de m

• Dans les cas 1 et 3, on a fait diminuer la hauteur noire.

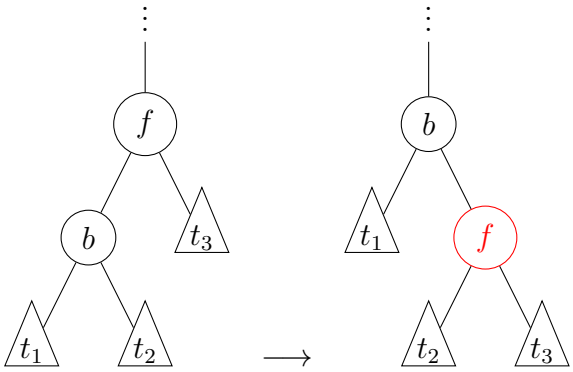
Si n était la racine, ce problème serait facilement résolu. On se place donc dans le cas d'un fils droit dont la hauteur noire est inférieure d'une unité à celle de son frère.

L'idée est de faire remonter un nœud noir du sous-arbre du frère pour rétablir l'étiquette des hauteurs noires.

• Si le père est rouge, alors le frère est noir et on peut utiliser une rotation :

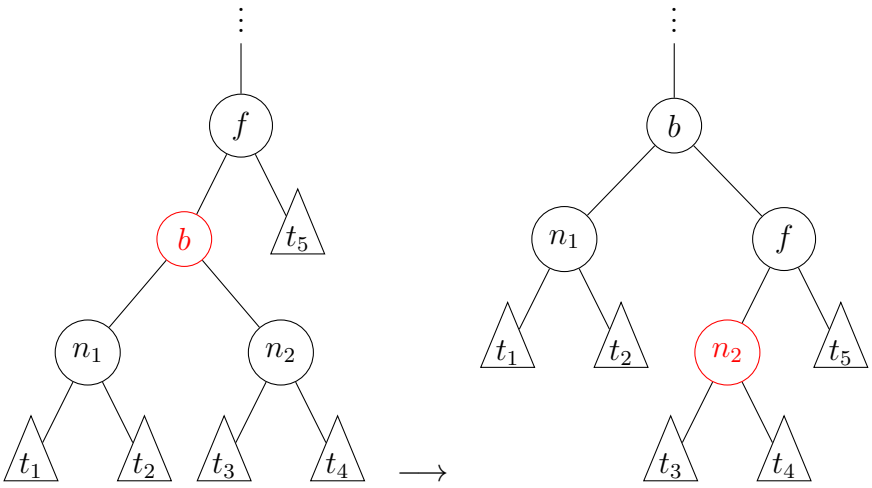


• Si le père et le frère sont noirs, on colore le père en rouge et on applique une rotation comme avant.



En plus de l'éventuel conflit rouge-rouge, la hauteur noire de ce sous-arbre a diminué et il faut appliquer la correction récursivement.

• Le père est noir, et le frère est rouge :



Il peut y avoir un conflit rouge-rouge mais la hauteur noire est bien corrigée

Remarque : comme pour l'insertion, on effectue un nombre logarithmique de corrections, donc la complexité attendue est préservée.

3 Files de priorité

3.1 Définition, premières implémentations

3.1.1 Introduction

La notion de *file de priorité* généralise celles de pile et de file : on extrait les éléments en fonction de leur priorité qui n'est pas nécessairement fonction de l'ordre d'insertion.

3.1.2 Définition (*file de priorité*)

Une *file de priorité* est une structure de données associant des valeurs prises dans un ensemble E à celles prises dans un ensemble totalement ordonné $(K, \leqslant)$ et dont les opérations sont les suivantes :

- Création d'une file de priorité vide ;
- Test de vacuité ;
- Insertion d'un élément de E associé à une priorité $k \in K$;
- Modification de la priorité d'un élément ;
- Extraction de l'élément de priorité maximale.

On parle de file de priorité max, par opposition aux files de priorité min où l'on extrait l'élément de priorité minimale.

3.1.3 Implémentation par liste triée

On stocke les couples (éléments, priorité) triés par priorité décroissante.

- Création / test de vacuité : $\mathcal{O}(1)$
- Insertion / modification de priorité : $\mathcal{O}(n)$
- Extraction du max : $\mathcal{O}(1)$

3.1.4 Implémentation par ABR équilibrés

- Création / test de vacuité : $\mathcal{O}(1)$
- Insertion / extraction du max : $\mathcal{O}(\log n)$
- Modification de priorité : $\mathcal{O}(\log n)$ (supression + insertion)

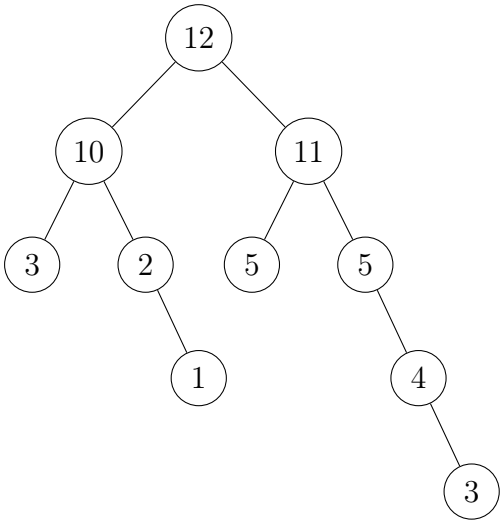
3.1.5 Arbres tournois

Définition :

Soit $(K, \leqslant)$ un ensemble totalement ordonné.

Un *arbre tournoi* est un arbre binaire étiqueté par des éléments de K tel que pour tout nœud, l'étiquette de ce nœud est supérieure à toutes celles de ses descendants.

Exemple :

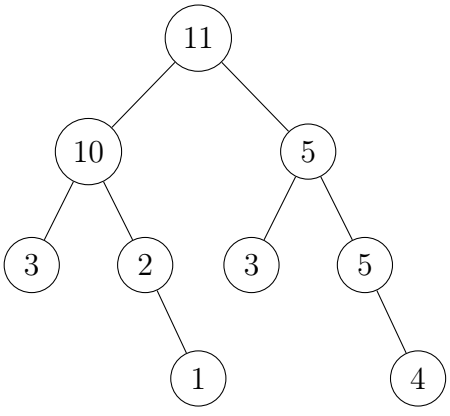


3.1.6 Implémentation par arbre tournoi

– Création / test de vacuité : $\mathcal{O}(1)$

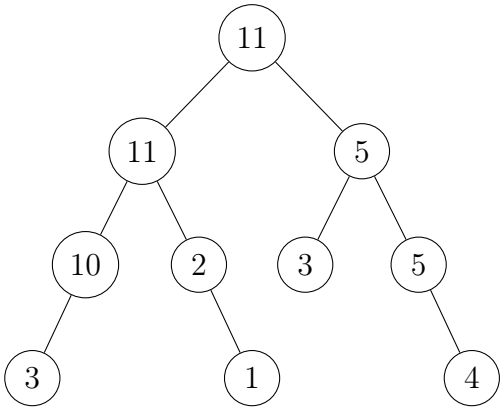
– Extraction du max : $\mathcal{O}(h)$ en fusionnant les deux sous-arbres ou en plaçant l'une des feuilles à la place de la racine et en le faisant redescendre (on parle de *percolation descendante*).

Exemple avec l'arbre précédant :



– Insertion : $\mathcal{O}(h)$ en plaçant une nouvelle feuille que l'on fait remonter (*percolation ascendante*)

Exemple avec l'arbre ci-dessus :



– Modification de priorité : percolation selon la nouvelle priorité : $\mathcal{O}(h)$

Besoins :

– Arbre équilibré ;
– Trouver le père d'un nœud donné ;
– Accès rapide aux feuilles
– Accès en temps constant à un nœud donné.

3.2 Tas binaire

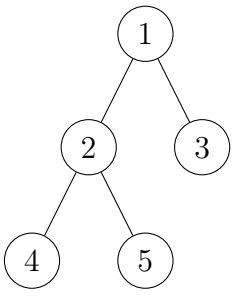
3.2.1 Définition (*arbre parfait*)

Un *arbre parfait* est un arbre binaire strict tel que :

(1) Toutes les feuilles sont de profondeur h ou $h-1$, où h est la hauteur ;
(2) $\forall p < h$, il y a exactement 2^p nœuds de profondeur p ;
(3) Les feuilles de profondeur h sont "tassées" à gauche.

Attention : certains parlent d'arbre complet.

3.2.2 Exemple



3.2.3 Proposition

Un arbre parfait à n nœuds est de hauteur $h = \lfloor \log_2 n \rfloor$

☐ Démonstration :

Le nombre de nœuds est compris entre le nombre d'un arbre complet de hauteur $h-1$ et celui d'un arbre complet de hauteur h.

Or le nombre de nœuds d'un arbre complet de hauteur l est

$$\sum_{p=0}^{l} 2^p = 2^{l+1} - 1 \qquad \boxed{\text{Exo}}$$

Donc $2^h - 1 < n \leqslant 2^{h+1} - 1$

donc $2^h \leqslant n < 2^{h+1}$, d'où $h \leqslant \log_2 n < h + 1$ ■

3.2.4 Représentation par tableaux

On place la racine dans la case d'indice 0 et pour tout nœud d'indice i, on place ses fils gauche et droit dans les cases d'indices $2i+1$ et $2i+2$ s'ils existent.

Exemple pour l'arbre en 3.2.2 :

1	2	3	4	5

C'est un parcours en largeur.

Remarque : cela répond aux besoins exprimés en 3.1.6 : feuilles à partir de l'indice $\left\lfloor \frac{n}{2} \right\rfloor$, père du nœud i à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$

3.2.5 Définition (*tas*)

Un *tas* est un arbre tournoi parfait.

On parle parfois de *tas max*.

3.2.6 Implémentation

On peut choisir d'utiliser un tableau statique ou un tableau dynamique.

on utilisera le type suivant :

```
type 'a heap = {t : 'a array; mutable n : int}
```

3.2.7 Percolation descendante

sift = tamis.

```
let rec sift_down (hp : 'a heap) (i : int) : unit =
    let j = ref i in
    if 2*i + 1 < hp.n && hp.t.(2*i + 1) > hp.t.(i) then
        j := 2*i + 1;
    if 2*i + 2 < hp.n && hp.t.(2*i + 2) > hp.t.(!j) then
        j := 2*i + 2;
```

```
    if !j <> i then begin
        swap hp.t i !j; (*Exo : coder swap*)
        sift_down hp !j
    end
```

3.2.8 Percolation ascendante

Exo

3.2.9 Extraction du max

```
let take_max (hp : 'a heap) : 'a =
    swap hp.t 0 (hp.n - 1);
    hp.n <- hp.n - 1;
    sift_down hp 0 ;
    hp.t.(hp.n)
```

3.2.10 Insertion

```
let insert (x : 'a) (hp : 'a heap) : unit =
    hp.t.(hp.n) <- x;
    sift_up hp hp.n;
    hp.n <- hp.n + 1
```

3.2.11 Conversion d'un tableau en tas

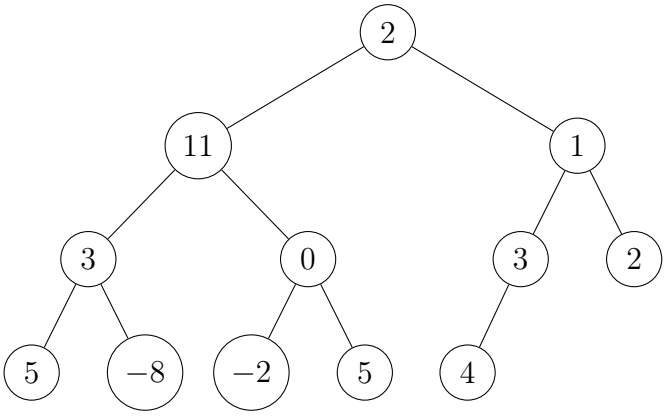
$\mathcal{O}(n \log n)$ par insertion successives car chaque feuille insérée peut percoler jusqu'à la racine.

Construction par forêt : on voit les éléments du tableau comme des tas séparés que l'on fusionne :

```
let array_to_heap (t : 'a array) : 'a heap =
    let hp = {t = Array.copy t ; n = Array.length t} in
    for i = hp.n / 2 - 1 downto 0 do
        sift_down hp i
    done;
    hp
```

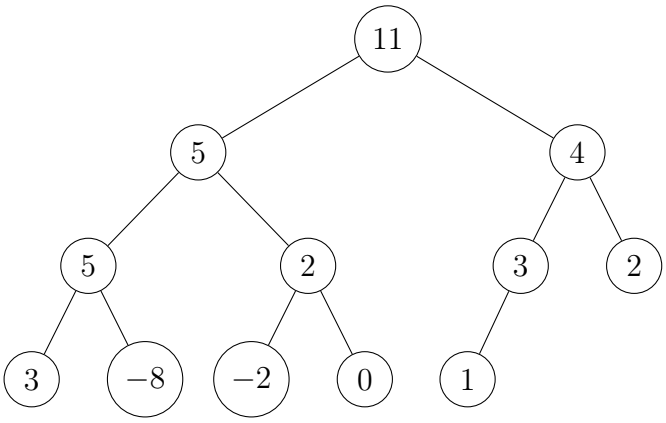
Exemple :

2	11	1	3	0	3	2	5	-8	-2	5	4



Donne :

11	5	4	5	2	3	2	3	−8	−2	0	1



Correction : invariant $I(i) : \forall j \geqslant i$, l'arbre de racine j est un tas.

Complexité : chaque nœud de profondeur p ne peut descendre que de $h - p$ niveaux.

$$\sum_{p=0}^{h-1}(h-p)2^p = \sum_{p=1}^{h} p2^{h-p} = 2^h \sum_{p=1}^{h} p2^{-p} \leqslant 2^h \underbrace{\sum_{p=1}^{\infty} p2^{-p}}_{\text{constant}}$$

donc $\mathcal{O}(2^h)$, *i.e* $\mathcal{O}(n)$.

3.2.12 Tri par tas

```
let heap_sort (a : 'a array) : unit =
    let hp = array_to_heap a
    and n = Array.length a in
    for i = n - 1 downto 0 do
        a.(i) <- take_max hp
    done
```

Complexité : conversion en $\mathcal{O}(n)$ + n extractions de max en $\mathcal{O}(\log n)$ donc en $\mathcal{O}(n \log n)$.

Remarque : le type `'a heap` n'est pas nécessaire, on peut tout faire en place sur le tableau $\rightarrow \mathcal{O}(1)$ en espace.