**Chapitre 1. Arbres binaires de recherche. Tas.**

Nous allons nous intéresser dans ce chapitre à des arbres binaires particuliers qui permettent d'implémenter concrètement certaines structures de données abstraites.

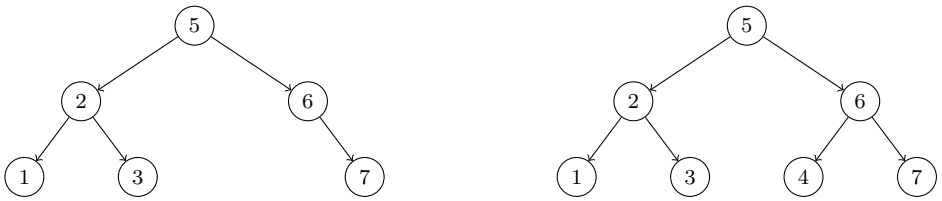
# 1 Arbres binaires de recherche

On considère des arbres binaires étiquetés par des éléments d'un ensemble totalement ordonné $E$ (par exemple les entiers naturels ou les mots classés par ordre alphabétique).

## 1.1 Définitions

**Définition 1.** *On appelle* arbre binaire de recherche *un arbre binaire tel que pour chaque nœud de cet arbre, toutes les étiquettes des nœuds de son fils gauche sont strictement inférieures à celles de ce nœud et toutes les étiquettes des nœuds de son fils droit sont strictement supérieures à celle de ce nœud.*

**Remarque :** Autrement dit, un arbre binaire est de recherche si le parcours en profondeur infixe de cet arbre retourne les étiquettes de ses nœuds dans l'ordre croissant.
**Exemples :** L'arbre de gauche ci-dessous est de recherche mais pas celui de droite




**Remarque :** La structure d'arbre binaire de recherche correspond à une réalisation concrète (et persistante) de la structure de données abstraite Dictionnaire. Il suffit pour cela d'étiqueter les nœuds avec des couples (clé,élément) et de faire en sorte que la structure d'arbre binaire de recherche soit adaptée à l'ordre sur les clés.

**Implémentation :** Dans la suite nous allons considérer un certain nombre d'opérations sur les arbres binaires et nous les coderons en implémentant les arbres grâce au type suivant :

```
type 'a arbre =
    |Vide
    |N of 'a * 'a arbre * 'a arbre;;
```

Vide permet de traiter les feuilles et les nœuds internes d'arité 1.

## 1.2 Recherche d'un élément

Grâce à la structure d'arbre binaire de recherche, la recherche d'un élément dans un tel arbre peut se faire de manière dichotomique, en cherchant dans la moitié utile de l'arbre.

```
let rec chercher_abr a x = match a with
    |Vide              -> false
    |N(y,_,_) when y=x -> true
    |N(y,g,_) when x<y -> chercher_abr g x
    |N(y,_,d)          -> chercher_abr d x;;
```

**Analyse :** Établissons la terminaison, la correction et évaluons la complexité de la fonction précédente, en distinguant deux cas selon que l'élément cherché est ou non dans l'arbre.

— Si `x` n'est pas l'étiquette d'un nœud de l'arbre `a`, chaque appel de la fonction avec un sous-arbre non vide fera un appel à la fonction avec un de ses deux fils. Ainsi la hauteur de l'arbre en paramètre diminue strictement à chaque appel. Comme la hauteur ne peut être inférieure à $-1$, le nombre d'appel est fini et on aboutit à `chercher_abr x Vide` qui retourne la bonne réponse. De plus, si $h$ est la hauteur de l'arbre, la fonction est appelée au plus $h+1$ fois. Comme on fait deux comparaisons à chaque appel sauf au dernier, on effectue au maximum $2h$ comparaisons.

— On traite le cas où `x` est l'étiquette d'un nœud de l'arbre par récurrence en posant :
$P(h)$ Pour tout arbre `a` de hauteur $h$ dans lequel `x` apparaît, `chercher_abr a x` termine et retourne `true` en effectuant au plus $2h+1$ comparaisons
– $P(0)$ est vraie car si `a` est de hauteur 0 et contient `x`, c'est nécessairement `N(x,Vide,Vide)` et dans ce cas, la fonction termine, retourne `true` et effectue 1 comparaison.
– Soit $h \geqslant 1$ tel que $P(m)$ soit vraie pour tout $m < h$. Soit `a=N(y,g,d)` un arbre de hauteur $h$ contenant `x`.

i. Si `y=x`, l'appel `chercher_abr a x` retourne `true` avec 1 comparaison.
ii. Si `x<y` alors `d` ne contient pas `x` donc `g` contient `x` et est de hauteur au plus $h-1$.
`chercher_abr a x` effectue 2 comparaisons puis appelle `chercher_abr g x` qui, d'après l'hypothèse de récurrence, renvoie `true` après au plus $2(h-1)+1$ comparaisons au plus. Ainsi, `chercher_abr a x` renvoie la bonne réponse avec au maximum $2+2(h-1)+1 = 2h+1$ comparaisons.
iii. On raisonne de même si `x>y`.

Conclusion : on a montré la terminaison, la correction de la fonction et sa complexité est majorée par $2h+1$ donc est $O(h)$.

**Remarque :** Dans le cas d'une arbre binaire de recherche représentant un dictionnaire, une modification simple de la fonction précédente permet d'obtenir le contenu associé à une clé présente dans le dictionnaire.

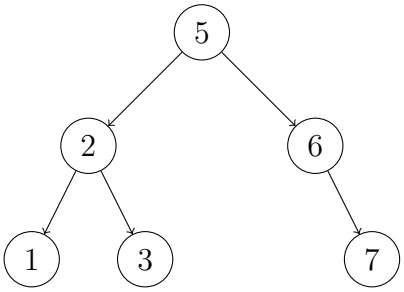
```
let rec lecture_abr a k = match a with
    |Vide                    -> failwith ("clé non présente")
    |N(y,_,_) when (fst y)=k -> snd y
    |N(y,g,_) when k<(fst y) -> lecture_abr g k
    |N(y,_,d)                -> lecture_abr d k;;
```

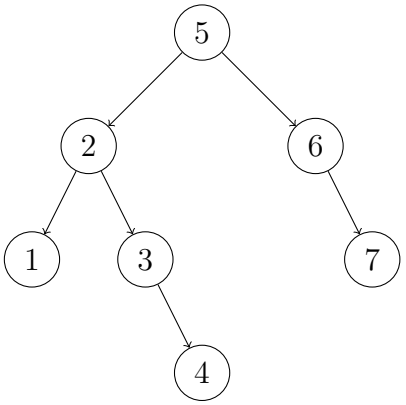
Voici un exemple d'utilisation :

```
# let dico = N((3,"eve"),N((1,"simon,"),Vide,Vide),N((4,"martin"),Vide,Vide));;
val dico : (int * string) arbre =
  N ((3, "eve"), N ((1, "simon,"), Vide, Vide),
   N ((4, "martin"), Vide, Vide))
# lecture_abr dico 3;;
- : string = "eve"
```

## 1.3 Insertion aux feuilles

On peut insérer un élément à un arbre binaire de recherche en l'ajoutant à une feuille de l'arbre : on recherche l'élément et une recherche infructueuse conduit à un arbre Vide. On peut alors y placer l'élément.

**Exemple :** Ajouter 4 à 

conduit à 


---

```
let rec ins_f_abr a x = match a with
    |Vide              -> N(x,Vide,Vide)
    |N(y,g,d) when x<y -> N(y, ins_f_abr g x, d)
    |N(y,g,d) when x>y -> N(y,g, ins_f_abr d x)
    | _                -> a;;
```

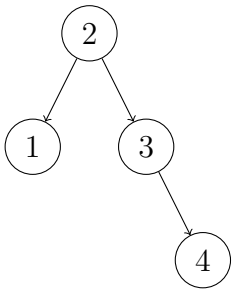
---

Ici encore, la complexité est proportionnelle au nombre d'appels récursifs effectués donc est $O(h)$ où $h$ est la hauteur de l'arbre.
On peut utiliser cette fonction pour créer un arbre binaire de recherche dont les étiquettes sont les éléments d'une liste :

---

```
let rec creation_abr l = match l with
    | [] -> Vide
    |t::q -> ins_f_abr (creation_abr q) t;;
```

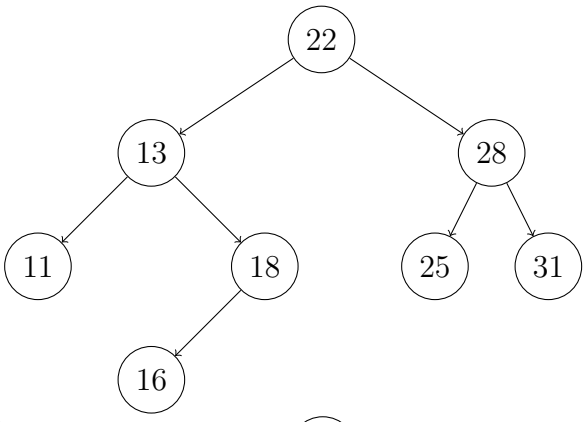
---

Par exemple `creation_abr [1;4;3;2]` conduit à 

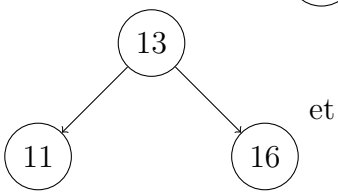
## 1.4 Insertion à la racine

On peut également choisir d'insérer l'élément à la racine. Pour cela, on découpe l'arbre en deux parties regroupant d'une part les nœuds d'étiquettes inférieures à l'élément qu'on veut rajouter, et d'autre part les nœuds d'étiquettes supérieures à l'élément à insérer.

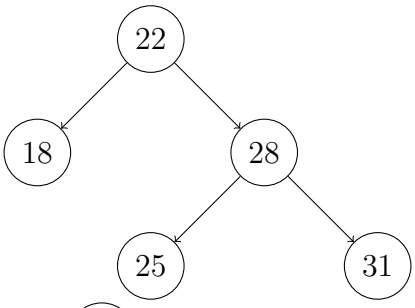
**Exemple :** Pour insérer 17 dans l'arbre



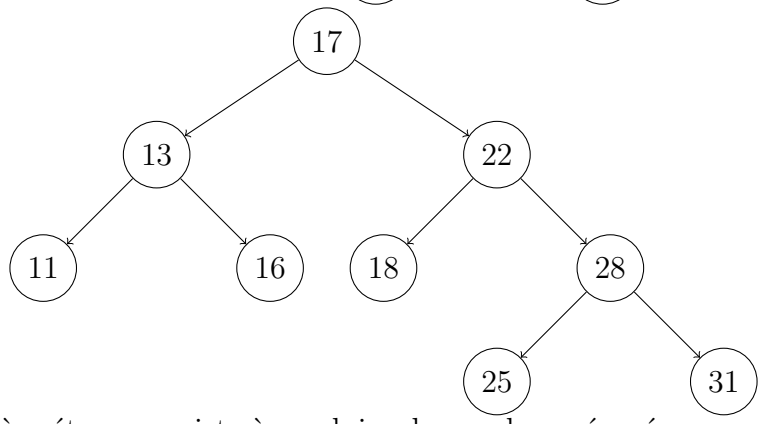

on reconstruit deux arbres




et



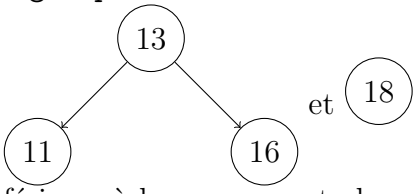

on peut alors rajouter 17 à la racine :




Pour coder cette méthode, la première étape consiste à produire deux arbres séparés par la valeur du nœud qu'on veut ajouter. Le découpage et la reconstitution se font en même temps, récursivement.
On suppose par exemple que l'étiquette de coupure est strictement inférieure à l'étiquette de la racine.

- On découpe le fils gauche avec la même étiquette de coupure. On obtient deux arbres : `g_inf`, formé des nœuds du fils gauche dont l'étiquette est inférieure à la clé de coupure et `g_sup`

  Sur notre exemple :

  et

- l'arbre des nœuds inférieurs à la coupure est alors `g_inf`
- l'arbre des nœuds supérieurs à la coupure est obtenu en remplaçant le fils gauche initial par `g_sup`

On procède de manière symétrique si la clé de coupure est strictement supérieure à l'étiquette de la racine. En cas d'égalité , on renvoie les deux fils de la racine.

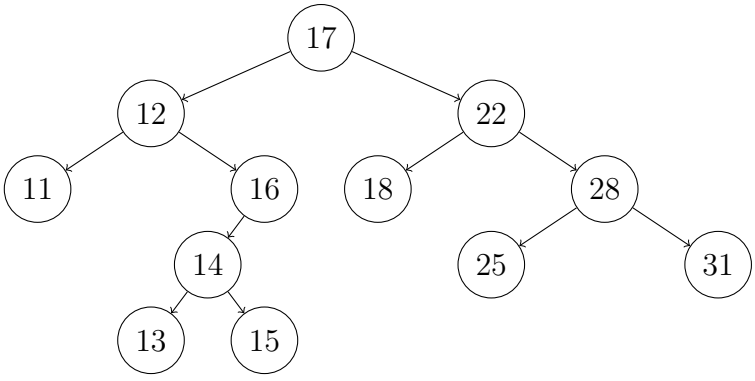
```
let rec decoupage a k = match a with
    |Vide -> Vide,Vide
    |N(x,g,d) when k=x -> g,d
    |N(x,g,d) when k<x -> let gg,gd = decoupage g k
                          in gg, N(x,gd,d)
    |N(x,g,d)          -> let dg,dd = decoupage d k
                          in  N(x,g,dg),dd;;

let ins_r_abr a k = let g,d = decoupage a k in N(k,g,d)
```

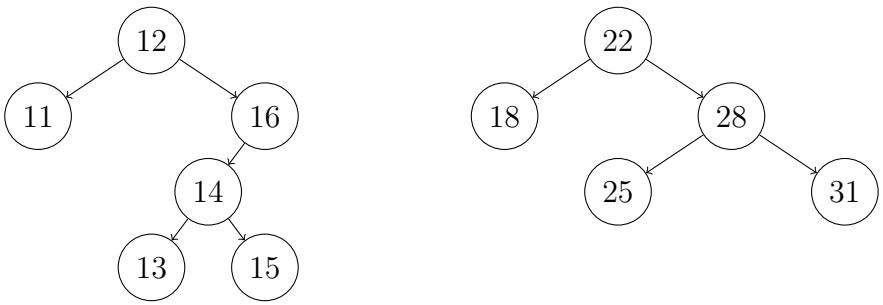
## 1.5 Suppression

Nous allons voir comment supprimer un nœud d'un arbres binaire de recherche en conservant la structure ordonnée.
Pour supprimer une feuille, il suffit de la remplacer par l'arbre vide.
Le cas d'un nœud interne est plus délicat. On peut se ramener au cas où on supprime la racine puisque, dans le cas général, il suffira d'appliquer cette procédure au sous-arbre dont le nœud à supprimer est la racine.
Si on souhaite supprimer la racine d'une arbre binaire de recherche, il faut choisir par quel nœud le remplacer. Si la racine n'a pas de fils gauche, le nouvel arbre sera simplement son fils droit. Sinon, on va remplacer la racine par un nœud de son fils gauche et pour préserver la structure d'arbre binaire, il faut que ce soit le nœud d'étiquette maximale du fils gauche.
Bien sûr, on aurait pu procéder en remplaçant la racine par un nœud du fils droit (celui d'étiquette minimale).

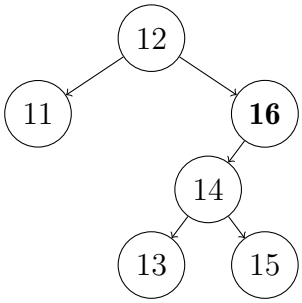
Nous procéderons en 4 étapes que nous détaillons d'abord sur un exemple :

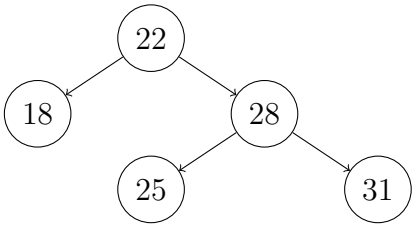



**(i) Suppression de la racine**

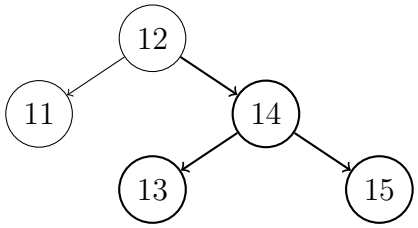


**(ii) Calcul de l'étiquette maximale du fils gauche**
On cherche récursivement le fils droit sans fils droit.

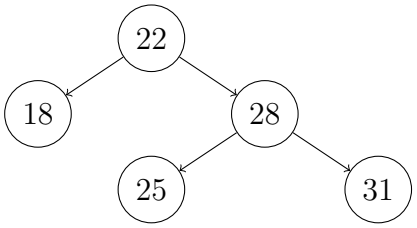


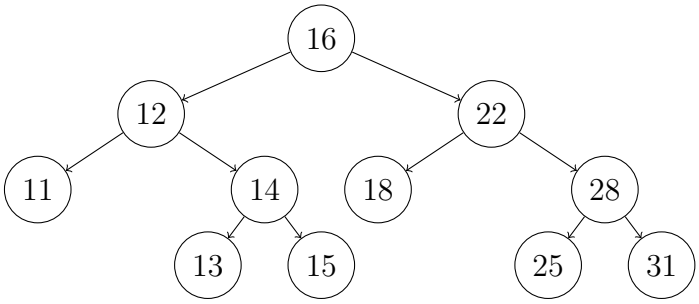

**(iii) Reconstitution du fils gauche**








**(iii) Reconstruction de l'arbre**




Écrivons le code correspondant à ces diverses opérations : tout d'abord une fonction `max_Arbre` donnant l'étiquette maximale d'un arbre binaire de recherche, une fonction `suppr_Max` supprimant le nœud d'étiquette maximale d'un arbre binaire de recherche.

---

```
let rec max_Arbre a = match a with
    |Vide        -> failwith "Arbre vide"
    |N(x,g,Vide) -> x
    |N(_,_,d)    -> max_Arbre g;;

let rec suppr_Max a = match a with
    |Vide        -> failwith "Arbre vide"
    |N(x,g,Vide) -> g
    |N(x,g,d)    -> N(x,g,suppr_Max d);;
```

---

On peut maintenant écrire une fonction supprimant la racine d'un arbre binaire de recherche, puis une fonction supprimant un quelconque de ses nœuds.

```
let rec suppr_Racine a = match a with
    |Vide        -> Vide
    |N(x,Vide,d) -> d
    |N(x,g,d)    -> let m = max_Arbre g in N(m,suppr_Max g,d);;

let rec suppr k a = match a with
    |Vide              -> Vide
    |N(x,_,_) when x=k -> suppr_Racine a
    |N(x,g,d) when k<x -> N(x,suppr k g,d)
    |N(x,g,d)          -> N(x,g,suppr k d);;
```

Montrons une façon plus rapide de coder la méthode précédente, utilisant seulement deux fonctions récursives imbriquées :

```
let rec supprime k a = match a with
    |Vide -> Vide
    |N(x,g,d) when x>k -> N(x, supprime k g,d)
    |N(x,g,d) when x<k -> N(x,g,supprime k d)
    |N(_,Vide,d) -> d
    |N(_,g,d) -> let y,g' = extrait_max g in N(y,g',d)
and extrait_max a = match a with
    |N(x,g,Vide) -> x,g
    |N(x,g,d) -> let y,d' = extrait_max d in y,N(x,g,d');;
```

## 1.6 Arbres équilibrés

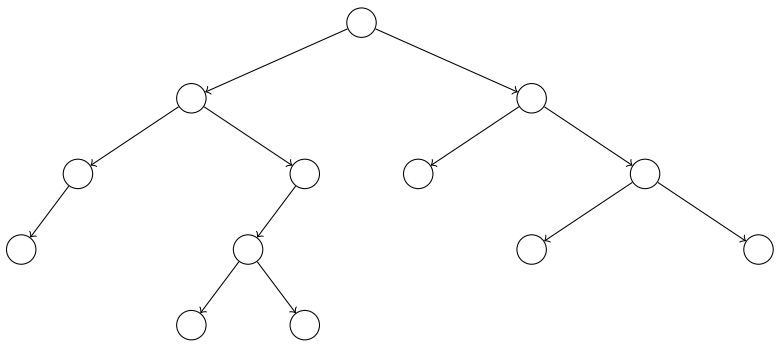
Toutes les fonctions que nous venons de décrire ont une complexité en $O(h)$ où $h$ est la hauteur de l'arbre. Or pour un arbre binaire de hauteur $h$ à $n$ nœuds, on a :

$$\lfloor \log_2 n \rfloor \leqslant h \leqslant n$$

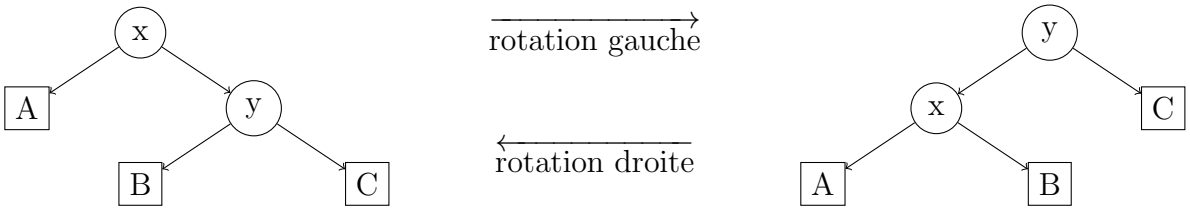
Comme de nombreux algorithmes ont une complexité qui dépend de la hauteur $h$, on a donc tout intérêt à réussir à maintenir nos arbres binaires de recherche à une hauteur minimale lors des étapes d'insertion : de tels arbres sont dits équilibrés.

**Définition 2.** *Un arbre binaire est dit* équilibré *si pour tout nœud interne de cet arbre, la valeur absolue de la différence entre la hauteur de son fils gauche et celle de son fils droit est au plus égale à 1.*

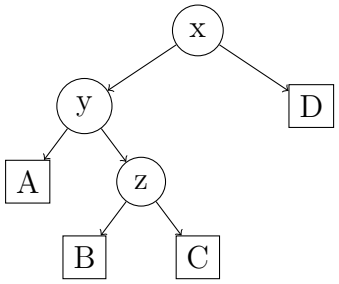
**Exemple :** indiquer si l'arbre ci-dessous est équilibré



Expliquons sur un schéma les opérations de rotation (droite ou gauche) permettant de rééquilibrer un arbre binaire de recherche en conservant sa structure ordonnée.




**Exemple :** Soit $A, B, C, D$ 4 arbres binaires de recherche équilibrés dont les hauteurs diffèrent d'au plus 1. Expliquer comment rééquilibrer l'arbre binaire de recherche ci-dessous à l'aide de deux rotations :




# 2 Tas

## 2.1 Rappels sur les files de priorité

La structure abstraite de file de priorité (en abrégé FP) doit pouvoir gérer des couples de la forme ($p$, élément). Les premières composantes $p$ sont à valeur dans un ensemble totalement ordonné (en pratique les entiers) qui indiquent la priorité de l'élément en deuxième composante. Les éléments présents dans la FP sont tous distincts, par contre ce n'est pas forcément le cas des composantes $p$. Il y a deux choix possibles : décider que les éléments prioritaires sont ceux ayant un $p$ maximal (on parle de FP-max) ou minimal (FP-min). La structure doit garantir les opérations suivantes :

- création d'une FP vide
- test d'égalité d'une FP au vide
- insertion d'un nouvel élément, avec sa priorité
- retrait de l'élément de plus grande priorité
- modification de la priorité d'un couple (augmentation ou diminution)

Cette structure sert notamment pour gérer un agenda, les tâches devant être effectuées par ordre de priorité décroissant. On retrouve notamment cette structure dans les imprimantes de bureau, ou dans la gestion des processus à effectuer par un ordinateur. En algorithmique, on peut utiliser une file de priorité min pour l'implémentation d'un algorithme de recherche de plus courts chemins dans un graphe.

Une première idée pour implémenter une FP-max est d'utiliser une liste de couples.
**Utilisation d'une liste non triée.** Avec une liste non triée, quelques-unes des opérations sont faciles (par exemple l'insertion d'un nouvel élément), par contre l'opération consistant à retirer l'élément prioritaire n'est pas évidente car il faut pour cela parcourir toute la liste pour trouver le $p$ maximum. Cette première idée n'est donc pas judicieuse.
**Utilisation d'une liste triée.** Avec une liste triée (dans l'ordre décroissant), il est facile de retirer l'élément de plus grande priorité. Par contre, l'insertion d'un nouvel élément nécessite de chercher où l'insérer, ce qui prend un temps linéaire en la taille de la liste

dans le pire des cas. L'utilisation d'une liste non triée n'est donc pas non plus une bonne idée.

## 2.2 Structure de tas

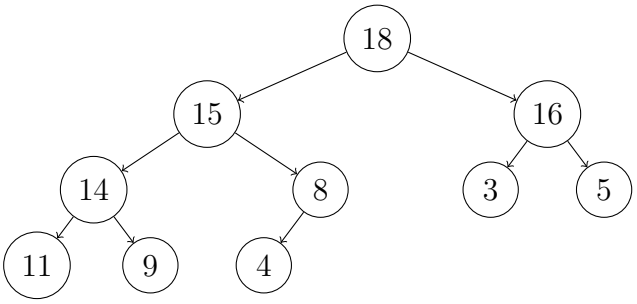
Dans la suite du cours, on se concentrera sur l'implémentation des FP-max, pour laquelle on va définir les tas-max, la démarche étant symétrique pour les FP-min.

**Définition 3.** *On appelle tas de hauteur $h$ tout arbre binaire complet à gauche c'est-à-dire tel que :*

- *toutes les feuilles sont de profondeur $h$ ou $h-1$,*
- *pour tout $p < h$, il y a exactement $2^p$ nœuds de profondeur $p$,*
- *les feuilles de profondeur $h$ sont le plus à gauche de l'arbre,*

*et tel que l'étiquette de tout nœud de l'arbre soit supérieure ou égale à celle de tous ses fils.*

**Exemple :** L'arbre suivant est un tas



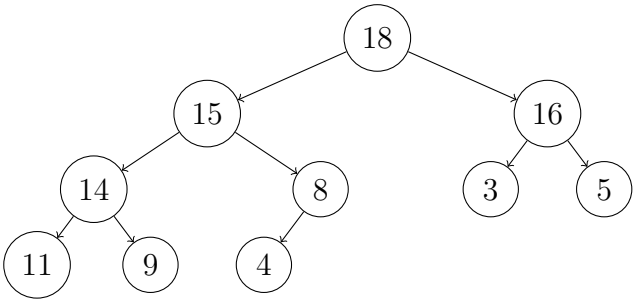

**Proposition :** Un arbre binaire complet à gauche de hauteur $h$ a un nombre $n$ de nœuds compris entre $2^h$ et $2^{h+1}-1$. En particulier on a $h = \lfloor \log_2(n) \rfloor$.

**Remarque :** Comme les fonctions que nous allons définir dans la suite auront une complexité linéaire en $h$ elles auront donc une complexité en $O(\log n)$ ce qui rend cette représentation de la structure de file de priorité bien plus intéressante qu'avec des listes.

## 2.3 Implémentation par tableaux

On peut stocker facilement les étiquettes d'un arbre binaire complet à gauche dans un tableau (ce qui rendra facile les échanges d'étiquettes) : il suffit de stocker les étiquettes de l'arbre dans l'ordre du parcours en largeur.

Par exemple, l'arbre binaire




sera représenté par le tableau

| 18 | 15 | 16 | 14 | 8 | 3 | 5 | 11 | 9 | 4 |
|---|---|---|---|---|---|---|---|---|---|

**Proposition :** Si $i$ est l'indice d'un élément du tableau représentant un AB complet à gauche, alors l'indice du fils gauche de $i$, s'il existe est $2i+1$ et celui de son fils droit, s'il existe est $2i+2$.
Si $i \neq 0$, l'indice du père de $i$ est donc $\lfloor \frac{i-1}{2} \rfloor$.

**Démonstration :** - Si $h$ est la hauteur de notre arbre binaire complet à gauche, pour $p < h$, les nœuds de profondeur $p$, ont pour indices les entiers de $1 + 2 + \cdots + 2^{p-1} + 1 - 1 = 2^p - 1$ à $2^{p+1} - 2$.
Les fils du plus à gauche d'entre eux ont donc pour indices $2^{p+1} - 1$ et $2^{p+1}$ qui valent respectivement $2(2^p - 1) + 1$ et $2(2^p - 1) + 2$. Il en va de même pour les suivants.
- Si l'élément d'indice $i \neq 0$ a pour père l'élément d'indice $j$, on a $i = 2j + 1$ ou $i = 2j + 2$ donc $\frac{i-1}{2} = j$ ou $\frac{i-1}{2} = j + \frac{1}{2}$ donc $j = \lfloor \frac{i-1}{2} \rfloor$ ♯

**Remarque :** Le type tableau est mutable, par contre la taille d'un tableau est définie une fois pour toutes. On conviendra donc de prendre des tableaux de longueur $N$ fixée ($N$ suffisamment grand pour qu'on puisse rajouter des nœuds) et de définir un tas par un tableau de longueur $N$ et un entier $n$ : le tas est alors obtenu, avec les conventions précédentes, comme les $n$ premières cases du tableau.
Par exemple, si on choisit $N = 14$, le tas précédent peut être représenté par l'entier $n = 10$ et le tableau $[|18; 15; 16; 14; 8; 3; 5; 11; 9; 4; 7; 5; 3; 2|]$.
En procédant ainsi, le type tas peut donc être `int*int array`.
Toutefois, dans la pratique, et c'est ce que nous ferons ensuite, on préfère souvent représenter le tas par un tableau de longueur $N + 1$ avec en position 0 l'entier $n$ donnant la longueur effective du tas et les étiquettes du tas entre les indices 1 et $n$. Ainsi, pour $N = 14$ le tas précédent peut être représenté par le tableau $[|10; 18; 15; 16; 14; 8; 3; 5; 11; 9; 4; 7; 5; 3; 2|]$.
Dans ce cas, les fils du nœud d'indice $i$ ont alors pour indices $2i$ et $2i + 1$ et celui de son père est $\lfloor \frac{i}{2} \rfloor$

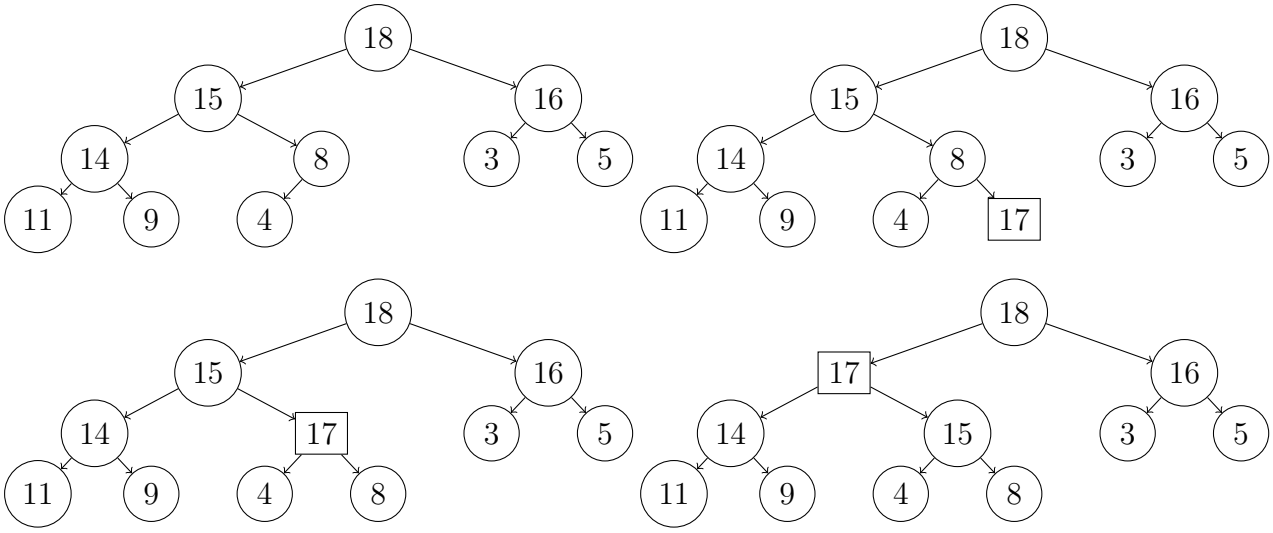
**Question :** Quel est l'inconvénient de cette représentation et comment y remédier ?

## 2.4 Opération d'insertion

Pour ajouter un élément à un tas on procède de la manière suivante :

- on place l'élément à côté de la dernière feuille présente
- on l'échange avec son père tant que son père est plus petit que lui

**Exemple :** Voici les différentes étapes pour insérer 17 dans le tas précédemment considéré




**Codage :**
Nous aurons besoin d'une fonction échangeant deux éléments d'un tableau :

```
let echange t i j = let tampon = t.(i) in t.(i)<-t.(j); t.(j)<-tampon;;
```

Voici la fonction auxiliaire de remontée d'un nœud, `j` étant l'indice du nœud à remonter.

```
let remonte j t = let k = ref j in
    while !k > 1 && t.(!k) > t.(!k/2) do
        echange t !k (!k/2);
        k := !k/2
    done;;
```

On en déduit la fonction d'insertion :

```
let insere x t = let j=t.(0)+1 in
    t.(j) <- x;
    t.(0) <- j;
    remonte j t;;
```

**Proposition :** La fonction d'insertion précédente a une complexité temporelle dans le pire des cas en $\Theta(\log n)$ où $n$ est le nombre d'éléments présents dans le tas.

**Démonstration ;** On peut justifier le résultat précédent de deux façons :
- La complexité de `insere` est celle de `remonte` où l'on constate que la valeur de la référence est à chaque étape divisée par deux jusqu'à atteindre la valeur 1 : il ne peut donc y avoir qu'un nombre logarithmique de passages dans la boucle `while`.
- dans le représentation sous forme d'arbre binaire des tas, la remontée d'un nœud modifie diminue sa profondeur de 1 à chaque étape. Dans le pire des cas il y a donc un nombre d'étapes de l'ordre de la hauteur de l'arbre dont on a vu qu'elle est $\lfloor \log_2(n) \rfloor$
♯

On peut utiliser la fonction d'insertion précédente pour créer un tas à partir d'un tableau :

```
let cree_tas t =
    let n = Array.length t in
    let tas = Array.make (n+1) 0 in
    for i = 0 to n - 1 do
        insere t.(i) tas
    done;
    tas;;
```

**Proposition :** Dans le pire des cas, la création d'un tas à partir d'un tableau de taille $n$ par la méthode précédente a une complexité $\Theta(n \log n)$.

**Démonstration :** On compte la complexité en nombre d'échanges. Dans le pire des cas, chaque remontée aboutit à la racine (c'est le cas par exemple si on part d'un tableau trié initialement en ordre croissant). Dans ce cas, le nombre d'échanges est égal à la somme des profondeurs de tous les nœuds à l'exception de la racine. Si on note $p = \lfloor \log_2(n) \rfloor$ la hauteur du tas, le nombre d'échanges à effectuer est alors égal à :

$$\sum_{h=1}^{p-1} k2^k + p(n - 2^p + 1) = (n+1)p - 2^{p+1} + 2 = \Theta(n \log n)$$

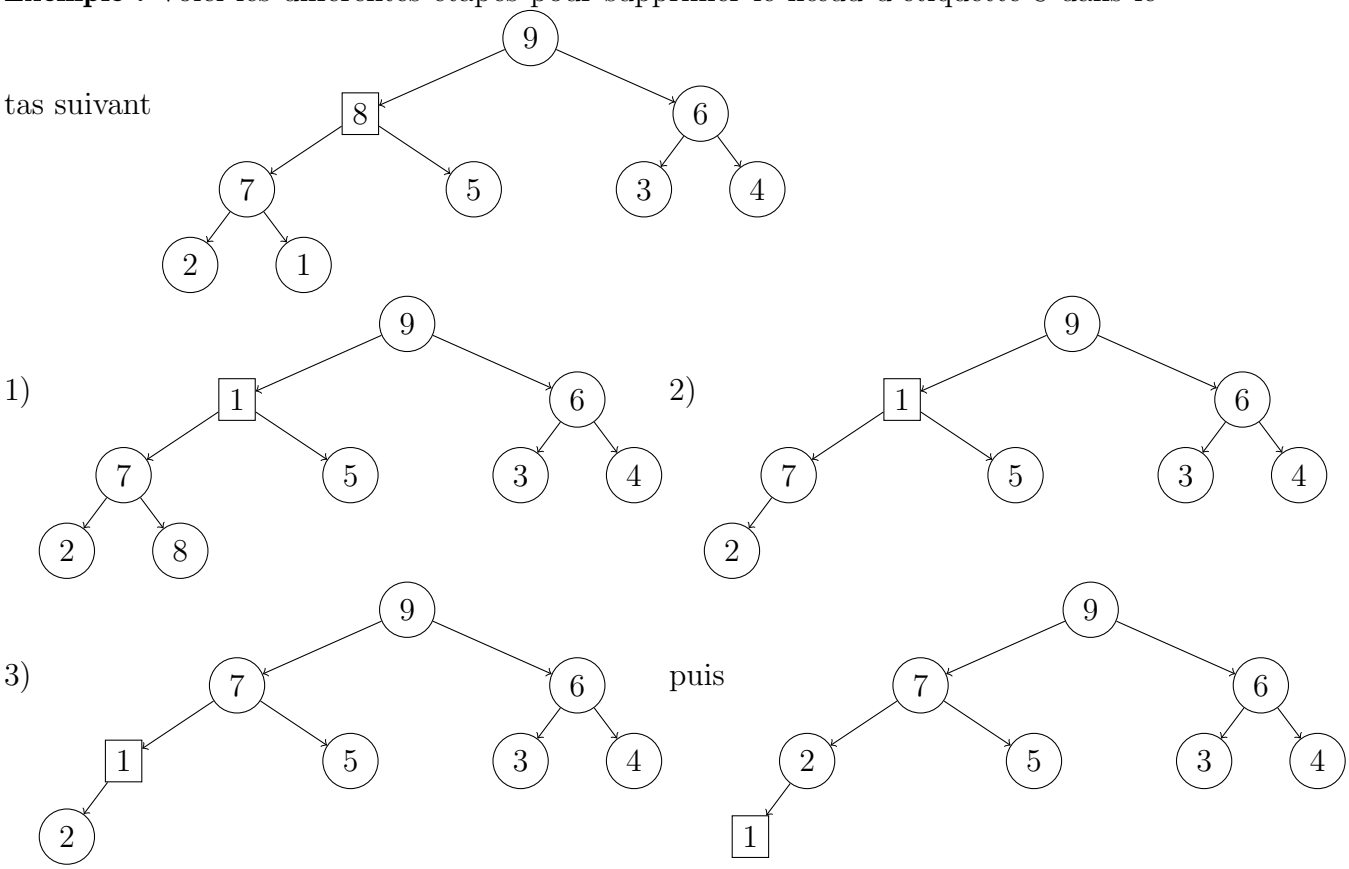
car $2^p \leqslant n < 2^{p+1}$ ♯

## 2.5 Opération de suppression

Pour supprimer un élément d'un tas, on peut procéder de la manière suivante :

- on échange l'élément à supprimer avec la dernière feuille (la plus à droite),
- on supprime cette dernière feuille,
- on descend le nouveau nœud en l'échangeant avec le plus grand de ses fils tant qu'il en existe un plus grand que lui pour rétablir la structure de tas.

**Exemple :** Voici les différentes étapes pour supprimer le nœud d'étiquette 8 dans le tas suivant




**Codage :** Nous avons besoin de deux fonctions auxiliaires, l'une `fils_max` donnant l'indice du plus grand des fils d'un nœud d'indice donné quand un tel fils existe et l'autre `descend` effectuant la descente d'un nœud pour rétablir la structure de tas.

---

```
let fils_max k t = let fg = 2 * k and fd = 2 * k + 1 in
if k > t.(0)/2 then k
else (if fg = t.(0) || t.(fg) > t.(fd) then fg else fd);;

let descend k t =
    let n = t.(0) in
    let p = ref k and f = ref (fils_max k t) in
    while t.(!p) < t.(!f) do
        echange t !p !f;
        p := !f;
        f := fils_max !p t
    done;;
```

---

On en déduit la fonction de suppression :

```
let supprime k t =
    echange t k t.(0);
    t.(0) <- t.(0) - 1;
    descend k t;;
```

**Remarque :** De la même manière qu'on l'a fait pour la fonction d'insertion, on montre que dans le pire des cas, la complexité temporelle de la fonction `supprime` est $\Theta(\log n)$ où $n$ est le nombre d'éléments du tas.

## 2.6 Tri par tas (Heapsort)

On remarque que le plus grand élément d'un tas se trouve à la racine de ce tas, ce qui va nous permettre d'implémenter un tri efficace d'un tableau. On procède de la manière suivante : on crée un tas à partir de notre tableau, puis on effectue des extractions successives de l'élément en racine du tas en plaçant les éléments à partir de la fin du tableau pour obtenir un tri par ordre croissant.

**Codage :** On commence par écrire à partir de `supprime` une fonction d'extraction qui pour un tas donné retourne l'étiquette de son plus grand élément et le supprime dans le tas.

```
let extrait tas =
    supprime 1 tas;
    tas.(tas.(0) + 1);;
```

On en déduit l'algorithme de tri par tas

```
let tri_par_tas t =
    let tas = cree_tas t in
    for i = Array.length t - 1 downto 0 do t.(i) <- extrait tas done;;
```

**Exemple :**

```
# let t = [|35;21;36;4;3;77;25|];;
val t : int array = [|35; 21; 36; 4; 3; 77; 25|]
# tri_par_tas t;;
- : unit = ()
# t;;
- : int array = [|3; 4; 21; 25; 35; 36; 77|]
```

**Proposition :** Dans le pire des cas, la complexité du tri par tas est $\Theta(n \log n)$ où $n$ est la longueur du tableau.
**Démonstration :** On a déjà montré que l'opération de création a une complexité temporelle $\Theta(n \log n)$. Puis, on effectue $n$ extractions, ce qui redonne à nouveau une complexité en $\Theta(n \log n)$. Comme ces deux étapes sont faites successivement, ces complexité s'ajoutent pour donner encore une complexité $\Theta(n \log n)$ ♯

**Exercice 1** Indiquer le nombre de tas ayant pour éléments les entiers de $[\![1, 6]\!]$.

**Exercice 2** On définit le type d'arbres binaires :

```
type 'a arbre =
    |Vide
    |N of 'a * 'a arbre * 'a arbre;;
```

a) Écrire une fonction `arbre_de_tas : int array -> int arbre` qui transforme,un tas codé par un tableau en un arbre codé par le type `int arbre`.
b) Écrire la fonction réciproque `tas_d_arbre` de la fonction précédente.