Les arbres

Chapitre 1

Cette année nous reprenons la notion d'arbre qui a été développée l'année passée. On va étudier la manière dont les arbres permettent de réaliser certaines structures de données.

1 Les arbres

1.1 Rappels

Commençons par quelques rappels terminologiques sur les structures de données qui ont été vue en première année.

Définition 1.1.1 (Structure de données abstraite)

1. *On appelle structure de données abstraite un type de données muni d'opérations.*
2. *Si on peut modifier les données sans devoir toutes les réécrire, on dit que la structure est :*
3. *Si une modification de la structure nécessite de construire un nouvel objet on parle de structure de données :*

Exemples :

1. En Caml, les listes sont des structures de données :

2. Les tableaux sont des structures de données :

Pour définir une structure de données, on précise les opérations que l'on veut pouvoir réaliser. Par exemple, les files vues en première année qui sont une structure de données de type FIFO (*first in first out*), on désire les opérations suivantes :

-
-
-
-

ATTENTION

La notion de structure de données abstraite ne doit pas être confondue avec l'implémentation d'une structure de données.
On peut imaginer que Alice code une structure de données et elle précise juste dans la documentation, les fonctions que l'on peut utiliser (ainsi que leur complexité la plupart du temps). Si Bob veut utiliser cette structure de données dans son programme, il ne va pas chercher à comprendre comment Alice a codé ses fonctions. Il va se contenter d'utiliser les fonctions définies. Il l'utilise comme une « boite noire ».

1.2 Définition

Définition 1.2.2

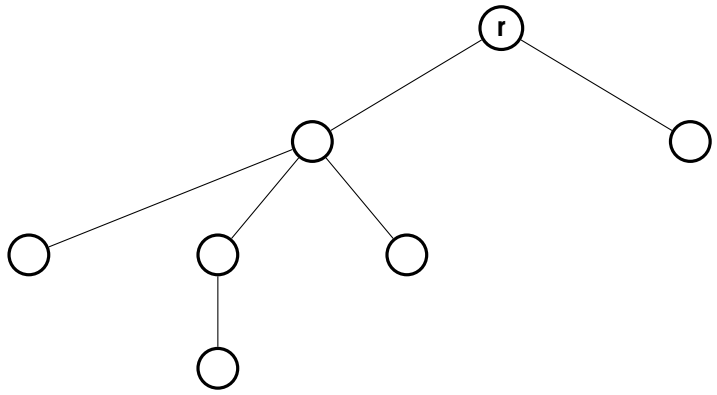
Un arbre (enraciné) est un ensemble fini non vide A avec un élément r que l'on appelle la racine.
Il est muni d'une application $p : A \setminus \{r\} \to A$ telle que pour tout $x \in A \setminus \{r\}$, il existe $k \in \mathbf{N}$ tel que $r = p^k(x)$.
La plupart du temps, on se donne de plus un ensemble B (ensemble des étiquettes) et une application de A dans B.

Terminologie :

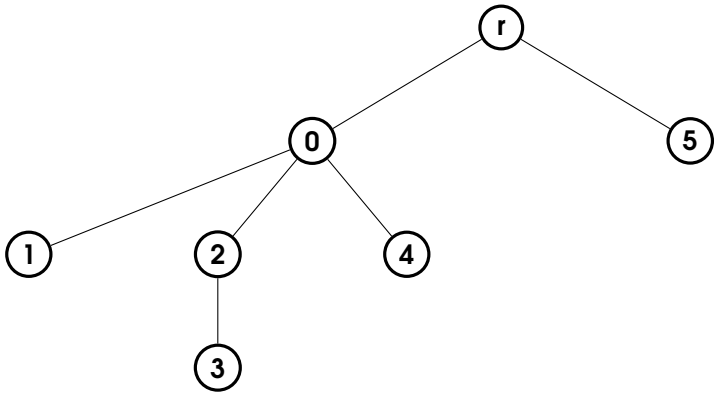
1. L'élément r s'appelle la racine de l'arbre.
2. Pour tout élément $x \neq r$, l'élément $p(x)$ s'appelle le père de x. Il est unique. Il arrive que, par convention, on note aussi $p(r) = r$.
3. Soit x un élément de A, les antécédents de x par p, $p^{-1}(\{x\}) = \{y \in A \mid p(y) = x\}$ s'appellent les fils de x.
4. On appelle nœuds les éléments de A.
5. On appelle feuille les éléments de A qui n'ont pas de fils et on appelle nœuds internes les nœuds qui ne sont pas des feuilles.
6. Soit x un élément de A. L'arité de x (ou le degré de x) est le nombre de ses fils. Les feuilles d'un arbre sont d'arité 0.

Remarque : Il existe de nombreuses définitions équivalentes pour les arbres. Nous en verrons une autre dans le cours sur les graphes. Dans ce chapitre nous parlerons d'arbre qui ne sont pas enracinés (les arbres libres).

Les arbres se représentent de la manière suivante.



Le même arbre mais étiqueté



Définition 1.2.3

Soit A un arbre.

1. *La taille de l'arbre est le nombre de ses nœuds. On la note $|A|$.*
2. *Soit x un élément de A. La profondeur de x, notée $\mathsf{prof}(x)$, vaut 0 si $x = r$ et vaut 1 de plus que celle de son père si $x \neq r$. On a donc*
$$\mathsf{prof}(x) = \begin{cases} 0 & \text{si } x = r \\ 1 + \mathsf{prof}(p(x)) & \text{sinon.} \end{cases}$$
3. *La hauteur d'un arbre est la profondeur maximale d'un de ses nœuds*

Remarques :

1. La profondeur d'un nœud $x \neq r$ est le plus petit entier k tel que $p^k(x) = r$. C'est donc le nombre d'arêtes traversées pour aller de la racine r au nœud x.
2. En pratique on parle indifféremment de hauteur et de profondeur.
3. Par convention l'arbre vide est de hauteur -1.

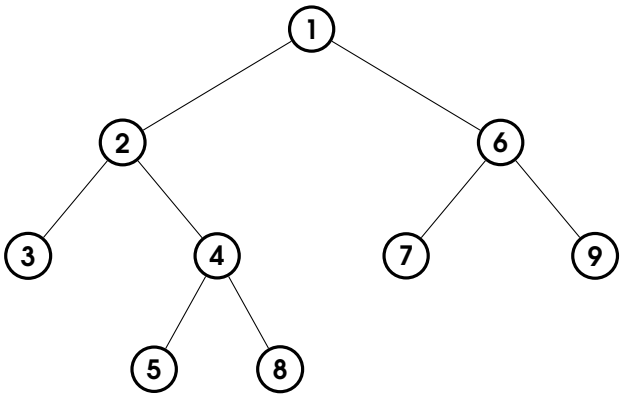
ATTENTION

Il arrive de trouver une convention différente en prenant la racine de hauteur 1. Pensez à bien lire l'énoncé !

Définition 1.2.4

*Si tous les nœuds **internes** sont d'arité au plus deux (resp. exactement deux) on dit que l'arbre est un arbre binaire (resp. un arbre binaire entier).*

Remarque : Avec les notations précédentes, les arbres binaires sont des sous-cas des arbres. De ce fait on n'ordonne pas les deux fils. En pratique (voir les implémentations) on ordonne les deux fils en précisant le fils de gauche et le fils de droite. La structure obtenue n'est plus un sous-cas des arbres généraux mais un sous-cas des arbres planaires où, pour chaque noeud, la liste de ses fils est ordonnée.
Exemple : Les arbres se représentent de la manière suivante.



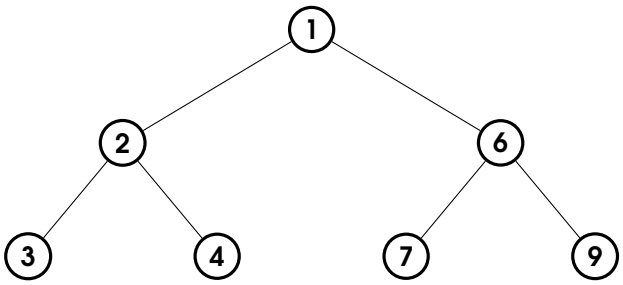
Dans cet exemple,

- La racine est
- est le père 5
- et sont les fils de 4.
- est une feuille
- 4 est de profondeur
- L'arbre est de hauteur et de taille
- C'est un arbre binaire entier

Définition 1.2.5

On appelle arbre binaire complet (ou parfait) un arbre binaire entier tel que toutes les feuilles aient la même profondeur.

Exemple :



Exercices :

1. Quel est la taille d'un arbre binaire complet de hauteur h ? Combien a-t-il de feuilles ?

2. Justifier que si A est un arbre binaire de taille n et de hauteur h alors :

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

En déduire un encadrement de la hauteur en fonction de la taille. *On veut juste l'encadrement, nous verrons plus loin comment rédiger une preuve précise.*

3. Reprendre l'exercice précédent avec un arbre dont les nœuds sont d'arité maximale a.

2 Implémentation

Il y a de nombreuses manières d'implémenter des arbres binaires en CAML. Pour la définition mathématique des arbres on utilise essentiellement la notion de père mais pour l'implémentation on utilise essentiellement la notion de fils.

2.1 Défintion du type

Nous utiliserons des types récursifs.

- Un arbre binaire étiqueté peut être implémenté par

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

  Ici une feuille est de la forme

- Si l'arbre n'est plus binaire

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

  Les feuilles sont les noeuds de la forme

ATTENTION

Il existe plein d'autres implémentations récursives du même genre (voir plus loin et en exercice). Il peut aussi être intéressant d'implémenter un arbre binaire par un tableau, nous le ferrons plus loin.

2.2 Induction structurelle

On voit que les arbres sont donc définis de manière récursive par :

- un cas de base qui est l'arbre vide.
- un constructeur qui correspond à un noeud dont les fils sont des arbres.

La plupart des preuves vont donc se faire par induction structurelle[1] qui est l'analogue de la récurrence. Pour montrer qu'un prédicat $\mathscr{P}$ est vrai pour tous les arbres il faut :

- Montrer que le prédicat est vrai pour le cas de base.
- Montrer que le prédicat est vrai pour un arbre obtenu par application du constructeur en supposant qu'il l'était pour les données auxquelles on a appliquées le constructeur.

Exemple : Remontrons que si a est un arbre binaire de taille n et de hauteur h alors :

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

- Cas de base :

Comme le cas de l'arbre vide est un peu pathologique on peut tester un arbre qui n'a qu'un noeud : $n=1$ et $h=0$.

[1] Nous reviendrons plus tard sur une définition précise d'une induction structurelle.

- Induction. Soit a un arbre non vide de la forme `N(g,x,d)`. On note n_g, n_d, h_g et h_d les tailles et hauteurs de d et g. Par hypothèses :

$$h_g + 1 \leqslant n_g \leqslant 2^{h_g+1} - 1 \text{ et } h_d + 1 \leqslant n_d \leqslant 2^{h_d+1} - 1.$$

Remarque : C'est ce même principe d'induction structurelle qui permet de d'assurer que les fonctions comme la taille ou la hauteur sont bien définies.

2.3 Quelques fonctions

Nous allons donner quelques exemples de fonctions. Nous travaillerons avec le type d'arbre binaire `'a arbreBin` défini ci-dessus.

- Commençons par une fonction donnant la hauteur d'un arbre :

```
let rec hauteur a = match a with
  |Vide -> -1
  |N(g,_,d) -> 1 + max (hauteur d) (hauteur g);;
```

- Écrivons maintenant une fonction pour la taille d'un arbre.

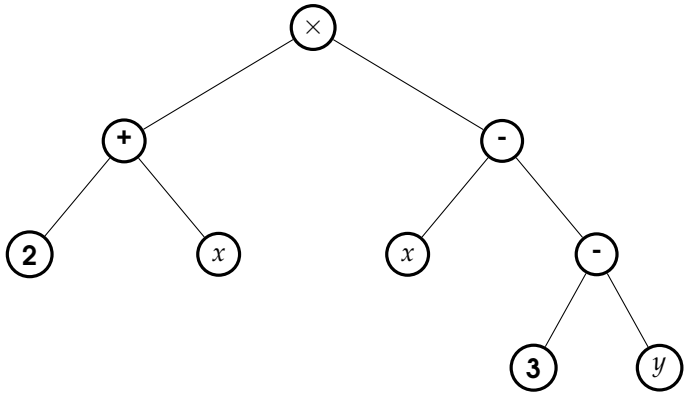
Exercice : Écrire une fonction qui calcule le nombre de nœuds internes et une qui calcule le nombre de feuilles.

2.4 Arbre de calcul

Les calculs algébriques peuvent se représenter dans des arbres. Par exemple

$$(2 + x) * (x - (3 - y))$$

se représente par



La différence par rapport aux arbres étudiées pour le moment c'est que les étiquettes des feuilles et celles des nœuds internes n'ont pas le même type. On utilise donc un type plus compliqué qui traite les feuilles différemment des noeuds internes.

```
type ('a,'b) arb =
        |F of 'a
        |N of ('a,'b) arb * 'b * ('a,'b) arb;;
```

Remarque : Ici, il n'est plus nécessaire de considérer l'arbre vide car :

-
-

3 Arbres binaires de recherche

3.1 Dictionnaires

Définition 3.1.6 (Dictionnaire)

La structure de données abstraite de dictionnaire consiste en un ensemble de couples (clés, données) tels que les clés soient dans un ensemble totalement ordonné (la plupart du temps des entiers). On dispose des opérations suivantes :

-
-
-
-

Remarque : Dans la majorité des cas, on demande à ce qu'un même clé n'apparaissent pas deux fois dans le dictionnaire. L'exemple le plus classique est le dictionnaire usuel.
Regardons deux implémentations naïves :

- Une liste de couples. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille du dictionnaire.

  - La recherche :
  - L'insertion :
  - La suppression :

- Un tableau de couples ordonnés par les clés. Là encore, on peut regarder les complexité :
  - La recherche :
  - L'insertion :
  - La suppression :

Exercices :

1. On considère un type `dico1` défini par `type 'a dico1 = (int*'a) list`. Écrire les fonctions d'insertion, de recherche d'un élément et de suppression d'une clé.

2. On considère un type `dico2` défini par `type 'a dico2 = (int*'a) array`. Écrire les fonctions d'insertion, de recherche d'un élément et de suppression d'une clé. On fera attention au fait que les entrées du tableaux devront toujours être classées par ordre croissant des clés.

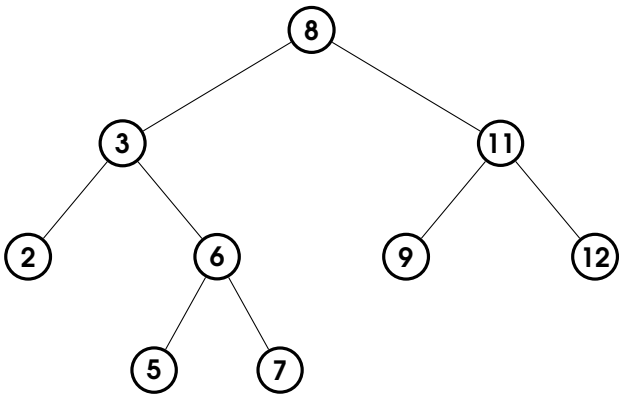
   On pourra commencer par coder une fonction `trouveIndice : int -> (int*'a) array -> (bool*int)` telle que `trouveIndice x t` renvoie le couple `(true,i)` s'il la première composante de `t.(i)` est `x` et renvoie un couple `false,i` sil la clé n'apparait dans le tableau. Dans ce cas, `i` sera l'indice du tableau où on pourra insérer une clé valant `x`.

3.2 Définition

Définition 3.2.7

*On appelle **arbre binaire de recherche** un arbre binaire étiqueté par des éléments dans un ensemble ordonné tel que l'étiquette de chaque nœud est supérieure à toutes les étiquettes des nœuds du sous arbre de gauche et inférieure à toutes les étiquettes des nœuds du sous arbre de droite.*

Exemple : L'arbre suivant est un arbre binaire de recherche.



Remarques :

1. Nous utiliserons souvent des entiers comme étiquettes mais on peut aussi utiliser des lettres ou des mots triés par ordre alphabétique.
2. On peut formaliser cela. Si on note, pour un arbre a, $E(a)$ l'ensemble de ses étiquettes alors un arbre binaire est un arbre binaire de recherche s'il est vide ou de la forme $N(x, g, d)$ où,
   -
   -
   -

   Comme on le voit, on a donc une définition récursive, cela implique que l'on pourra faire des preuves par des raisonnements par induction.
3. Il existe trois parcours d'un arbre binaire. En effet, en parcourant l'arbre de gauche à droite et en considérant les noeuds vides, on va passer par tous les sommets trois fois. On peut donc définir trois parcours :
   - Le parcours préfixe où l'on liste le nœud la première fois où on le rencontre. Dans notre exemple :
   - le parcours postfixe où l'on liste le nœud la dernière fois où on le rencontre. Dans notre exemple :
   - le parcours infixe où l'on liste le nœud lors de sa deuxième rencontre. Dans notre exemple :

   On peut définir un arbre binaire de recherche par le fait que lors du parcours infixe, la liste obtenue est croissante.

4. En fonction de ce que l'on veut faire, on demande à ce que toutes les étiquettes soient différentes ou pas. Si on autorise à avoir des doublons, on doit faire un choix, par exemple, les étiquettes du sous-arbre de gauche sont **inférieures ou égales** à l'étiquette du noeud et les étiquettes du sous-arbre de droite sont **strictement supérieures** à l'étiquette du noeud.

3.3 Recherche

L'utilité d'un arbre binaire de recherche est de pouvoir rechercher facilement (et rapidement) si un élément est dans l'arbre (nous verrons plus bas comment en déduire un structure de dictionnaire). Le principe est de mettre en place une recherche semblable à la recherche dichotomique dans un tableau trié.

Le principe de l'algorithme est le suivant. Pour chercher un élément x dans un arbre binaire de recherche :

- Si l'arbre est vide (cas de base) l'élément n'est pas dans l'arbre.
- Si la racine est x : c'est gagné !
- Si x est strictement supérieur à la racine, on cherche x dans le sous-arbre de droite.
- Si x est strictement inférieur à la racine, on cherche x dans le sous-arbre de gauche.

Cela donne en CAML avec l'implémentation d'arbre binaire complet vu précédemment

```
type arbreBin =
  |Vide
  |N of int * arbreBin* arbreBin;;
```

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

Exercice : Adapter cette fonction au cas d'un `'a arbre Bin` défini par

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

où on suppose avoir une fonction `comparaison : 'a -> 'a -> int` tel que `comparaison x y` vaut 0 si $x = y$, est strictement positif si x est plus grand que y et strictement négatif sinon.

Proposition 3.3.8 (Complexité de la recherche dans un arbre binaire de recherche)

L'algorithme précédent permet de trouver (ou pas) un élément dans un arbre binaire en un temps $O(h)$ où h est la hauteur de l'arbre.

3.4 Insertion

On dispose d'un arbre binaire de recherche et d'un élément x que l'on veut insérer dans l'arbre. On veut bien évidemment garder un arbre binaire de recherche. Le principe est de faire comme si on cherchait x.

- Si on le trouve alors on ne fait rien (même si on pourrait insérer une deuxième copie)
- Si on ne le trouve pas, c'est donc que l'on est à une feuille ou à un arbre vide, on insère l'élément.

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

Exercices :

1. Quels arbres obtient-on en appliquant cet algorithme aux listes `[1;2;4;6;7]` et `[6;2;4;1;7]` ?

2. Ecrire une fonction de type `int list -> arbre` qui insère successivement les termes d'une liste dans un arbre binaire de recherche.

Proposition 3.4.9 (Complexité de l'insertion dans un arbre binaire de recherche)

L'algorithme précédent permet d'insérer un élément dans un arbre binaire en un temps $O(h)$ où h est la hauteur de l'arbre.

ATTENTION

On a vu que la recherche et l'insertion ont une complexité en $O(h)$. Or, pour une taille d'arbre donné (disons n), la hauteur de l'arbre peut varier de n (dans le cas d'un arbre « peigne ») à $\log(n)$ dans le cas d'un arbre complet.
C'est pour cela que l'on va essayer de garder un arbre le plus « étoffé »possible où on peut espérer avoir $h \simeq \log(n)$.

ATTENTION

On peut ainsi construire un arbre binaire de recherche à partir d'une liste en insérant les uns après les autres les éléments dans un arbre initialement vide. Il faut noter que si la liste est initialement triée, on obtient un arbre « peigne ».
Il existe des algorithmes qui permettent de construire un ABR de sorte que sa profondeur soit $O(\log(n))$. On peut donc construire un tel arbre en

$$\sum_{k=1}^{n} \ln k \sim$$

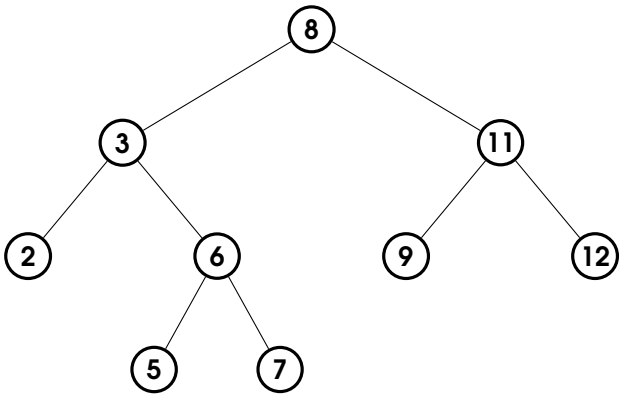
3.5 Suppression

L'algorithme de suppression est un peu plus compliqué. Bien évidemment, supprimer une feuille est simple, par contre, si c'est un noeud interne il faut « boucher le trou ».

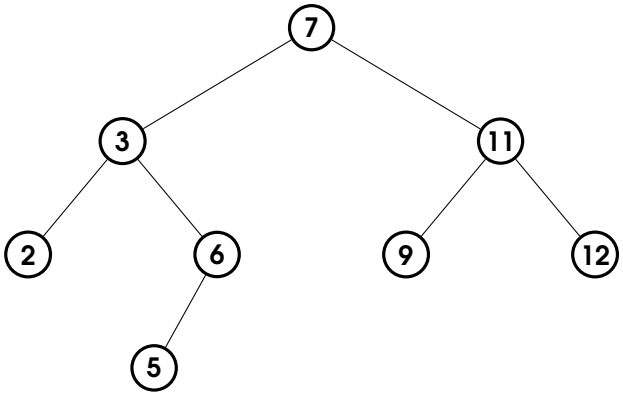
On voit que l'on peut, par récursivité, se ramener au cas où l'on supprime la racine.

- Si le sous-arbre de gauche est vide, on remplace l'arbre par son sous-arbre de droite
- Si le sous-arbre de gauche n'est pas vide :
  - on cherche le plus grand élément du sous-arbre de gauche (en allant toujours à droite)
  - on enlève cet élément du sous-arbre de gauche en remarquant que son fils droit est nécessairement l'arbre vide
  - on place cet élément au sommet du sous-arbre de gauche

Remarque : Cette méthode permet d'assurer que la hauteur de l'arbre ne peut que descendre.
Exemple : Si on considère l'arbre binaire de recherche suivant et que l'on veut enlever la racine



On obtient



Exercice : On considère l'arbre binaire de recherche obtenu en insérant successivement les éléments de la liste `[3;1;8;11;4;7;12;2;5]`.

Dessiner l'arbre obtenu puis dessiner l'arbre obtenu en supprimant 8.

On en déduit la fonction :

Et donc le programme

Proposition 3.5.10 (Complexité de la suppression dans un arbre binaire de recherche)

L'algorithme précédent permet de supprimer un élément dans un arbre binaire en un temps $O(h)$ où h est la hauteur de l'arbre.

3.6 Application - Dictionnaire

Un dictionnaire est une structure de données contenant des élément repérés par une clé. L'exemple typique est le dictionnaire usuel où les éléments sont les définitions qui sont repérées par les mots qu'elles définissent. Dans un dictionnaire on veut :

- Créer un dictionnaire vide
- Insérer un élément
- Rechercher des éléments par la clé pour éventuellement le modifier
- Supprimer des éléments

On voit donc que l'on peut réaliser un dictionnaire où la recherche sera efficace en utilisant un arbre binaire de recherche. En effet, toutes les opérations (insertion, suppression, recherche) se font de manière au plus linéaire en la **hauteur** de l'arbre binaire de recherche. Si on arrive à maintenir l'arbre le plus étoffé possible (voir exercices) on peut donc garantir une complexité en $O(\ln n)$ où n est la taille du dictionnaire.

4 Tas et liste de priorité

4.1 Liste de priorité

Nous voulons étudier la structure de liste de priorité. Le principe est le suivant, on imagine des personnes qui arrivent à un guichet mais, contrairement à la file classique, les personnes ont des importances différentes (on peut par exemple penser à des processus qui arrivent à un processeur). L'opérateur du guichet doit donc pouvoir traiter la personne qui a la plus grande importance. De ce fait, on veut faire trois opérations :

- Trouver le maximum
- Insérer un élément
- Retirer le maximum.

Remarque : Comme dans le cas des dictionnaires, on peut penser aux implémentations naïves :

- Une liste. On peut regarder la complexité des opérations que l'on veut réaliser par rapport à la taille de la file.
  - Trouver le maximum :
  - Insérer un élément :
  - Suppression du maximum :
- Un tableau ordonné. Là encore, on peut regarder les complexité :
  - Trouver le maximum :
  - Insérer un élément :
  - Suppression du maximum :

Nous allons voir une implémentation qui permet d'insérer et de retirer le maximum en $O(\log n)$ et de trouver le maximum en temps constant où n désigne la taille de la file.

4.2 Tas

Définition 4.2.11

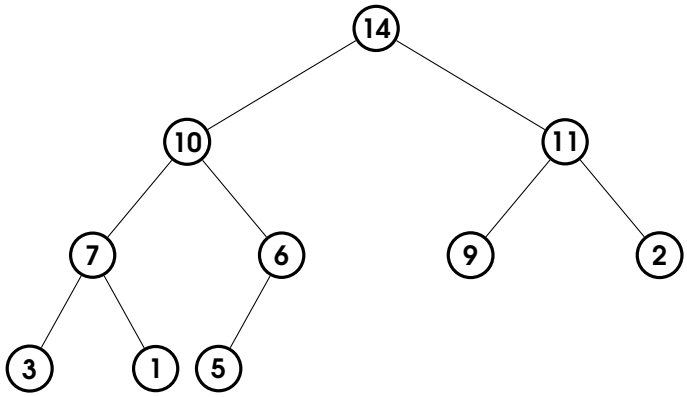
Un tas (maximal) de hauteur h est un arbre binaire tel que :

- *Toutes les feuilles sont de profondeur h ou $h-1$.*
- *Pour tout $p < h$, il y a exactement 2^p nœuds de profondeur p.*
- *Tous les nœuds de profondeur h sont « à gauche ».*
- *Chaque nœud a une étiquette supérieure ou égale à celle de ses fils. En particulier, la racine est le maximum de toutes les étiquettes.*

Remarques :

1. Les trois premières conditions concernent le squelette, la dernière concerne les étiquettes.
2. On peut aussi considérer un tas minimal où, au contraire, la racine est le plus petit élément et ainsi de suite.

Exemple :



Remarques :

1. On peut encadrer la taille n de l'arbre en fonction de sa hauteur h (et réciproquement) en décomposant les cas selon le nombre de nœuds du dernier niveau. On a :

$$2^h = 1 + 2 + 2^2 + \cdots + 2^{h-1} + 1 \leqslant n \leqslant 1 + 2 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$$

   Et donc

$$\log_2(n+1) - 1 \leqslant h \leqslant \log_2(n).$$

2. L'intéret est que l'on accède en temps constant au plus grand élément du tas (c'est la racine). Cela permet donc d'utiliser cela pour implémenter une file de priorité.

4.3 Insertion et suppression

On veut pouvoir insérer un élément dans un tas (de manière à garder un tas) de plus, cette méthode doit être en temps logarithmique par rapport à la taille du tas (ce qui revient à linéaire par rapport à la hauteur[2]).

La méthode est la suivante :

- On insére l'élément « en bas » du tas, c'est-à-dire que si le tas est un arbre binaire complet on augmente la profondeur de 1 et, dans le cas contraire, on rajoute l'élément « dans la dernière couche », juste à droite du dernier élément. De fait, c'est la seule place disponible au niveau du squelette du tas.

[2] on rappelle que, par définition, un tas est presque complet.

- On remonte l'élément ajouté en l'échangeant avec son père tant que ce dernier est inférieur à l'élément qui vient d'être ajouté.

Proposition 4.3.12 (Complexité de l'insertion d'un élément dans un tas)

L'algorithme d'insertion ci-dessus a une complexité linéaire en la profondeur du tas c'est-à-dire une complexité logarithmique en la taille du tas.

On veut aussi pouvoir supprimer la racine (qui est le maximum). La méthode est la suivante

- On échange la racine avec « le dernier du tas »,
- On supprime le dernier élément du tas. C'est la seule possibilité au niveau du squelette du tas.
- On « descend » l'élément qui a été échangé.

Proposition 4.3.13 (Complexité de la suppression du maximum dans un tas)

L'algorithme de suppression ci-dessus a une complexité linéaire en la profondeur du tas c'est-à-dire une complexité logarithmique en la taille du tas.

Exemple : Si on reprend le tas ci dessus et que l'on insère 13, on obtient successivement

Maintenant si on veut supprimer la racine, on obtient

4.4 Implémentation à l'aide d'un tableau

On peut implémenter une file de priorité à l'aide d'un tableau. On se donne un tableau `t` de type `int array` de longueur N (supposée suffisamment grande). La case `t.(0)` sert à repérer la taille du tas. En pratique on ne considère dans le tas que les éléments `t.(i)` pour i compris en 1 et `t.(0)`. Ensuite on remplit le tableau de la manière suivante :

- La racine du tas est `t.(1)`.
- Ses deux fils sont `t.(2)` et `t.(3)`.
- De manière plus générale, les éventuels fils de `t.(k)` sont `t.(2k)` et `t.(2k+1)`. De ce fait, le père de `t.(k)` est `t.(k/2)`[3].

Remarque : Si on veut utiliser une file de priorité dont les éléments ne sont pas des entiers, on peut utiliser un type enregistrement :

```
type filePriorite = {mutable taille : int ; donnee : 'a array}
```

On peut alors garder les mêmes conventions que ci-dessus (on suppose alors que `t.(0)` ne sert pas) ou commencer à `t.(0)` et il y a lieu de décaler les formules ci-dessus.
Exercice : Si on stocke les éléments à partir de `t.(0)` quel sont les indices des fils éventuels de `t.(k)` ? Quel est l'indice du père de `t.(k)` ?

On peut alors écrire nos fonctions

- Création d'un tas vide

```
let creTasVide taille_max = Array.make taille_max 0;;
```

- Recherche du maximum :

[3] ici k/2 désigne la partie entière de $\frac{k}{2}$

```
let maxiFile t = t.(1)
```

Remarque : On peut bien évidemment tester si la file est vide `t.(0) = 0` et éventuellement lever une exception.

- Insertion :

  On commence par écrire une fonction `echange : 'a array -> int -> int -> unit` telle que, par effet de bords, `t i j` échange les éléments d'indice `i` et `j` dans `t`.

  On peut alors écrire la fonction `insert : 'a array -> 'a -> unit` telle que `insert t x` insert l'élément `x` dans le tas donné par le tableau `t`.

  Remarque : La fonction procède par effets de bords (on a une implémentation impérative de la structure de file de priorité).

- Suppression : On veut écrire une fonction `supprime : 'a array -> unit` telle que `supprime t` supprime le maximum (c'est-à-dire la racine) d'une file de priorité donnée par le tableau `t`.

  On commence par une fonction récursive `descente : 'a array -> int -> unit` telle que `descente t k` va « descendre » l'élément d'indice `k` afin d'obtenir un tas.

On peut alors écrire la fonction `supprime`

4.5 Tri par tas

On peut utiliser la structure de tas pour construire un tri. En effet en partant d'une liste on peut insérer successivement ses éléments dans un tas. Par la suite, en retirant tous les éléments les uns après les autres du tas (en les prenant au sommet) on récupère une liste triée.

Les ajouts et suppression dans un tas étant de complexité logarithmique (par rapport à la taille du tas) on en déduit un algorithme de tri quasi-linéaire. Notons que cela peut se faire en place.

Exemple : Trions le tableau | 7 | 1 | 8 | 4 | 2 | 10 |.

On considère que le début du tableau est un tas. Il a donc un élément qui est l'élément 7. On note

1	7	1	8	4	2	10