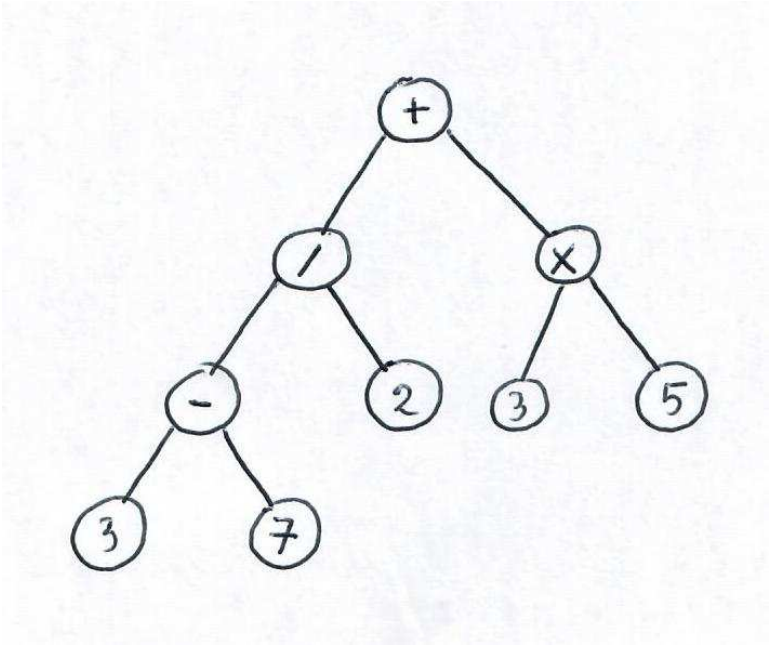
Structure de données : Les Arbres binaires

L'idée de départ : Les structures de données vues jusqu'à présent sont mal adaptées à la représentation de certaines informations.

Par exemple :

- Les données phylogénétiques
- Les procédures d'aide à la décision
- Arborescence des répertoires dans un système d'exploitation
- Représentation d'un tournoi de tennis
- Représentation d'une expression algébrique



Toutes ces données se représentent naturellement sous la forme d'un arbre.

Nous verrons également que la manipulation de données structurées sous forme d'arbre permet parfois de diminuer très significativement la complexité de certains problèmes : tri, recherche dans un index...

I] Définition de la structure d'Arbre Binaire

La structure d'Arbre Binaire : Que veut-on ?

On souhaite :

- Une structure récursive :
  - → définie par un noeud (appelé "racine") relié à deux sous-arbres
  - → définie par des "cas de base" qui sont soit un arbre vide, soit une feuille
- Qui permet de stocker
  - → au niveau des noeuds des données de même type
  - → au niveau des feuilles des données évenuellement d'un autre type
- Qui répartit les données dans les espaces disponibles de la mémoire de l'ordinateur.
  Ce qui impose un accès aux données en descendant l'arbre depuis la racine.

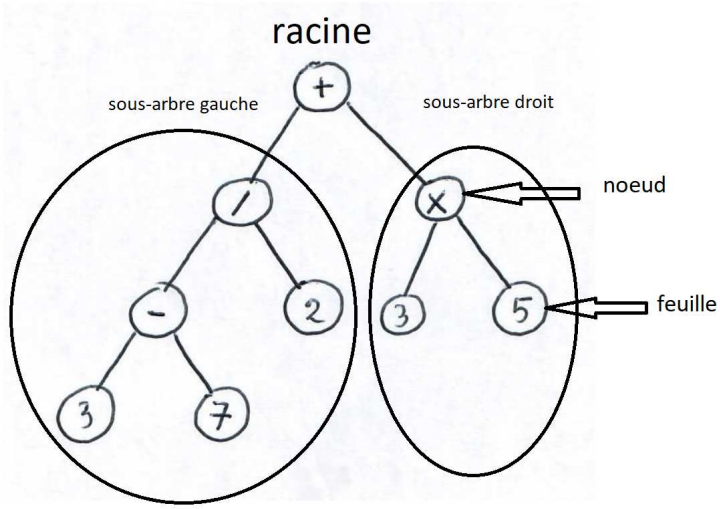
Nature du type

Comment construire un type "arbre" ?

- C'est un type "somme" pour pouvoir réunir sous le nom "arbre" :
  - → les noeuds (ou sous-arbres non triviaux)
  - → les feuilles (ou sous-arbres réduit à un élément)
  - → l'arbre vide

  L'arbre vide, les feuilles et les noeuds sont alors des arbres particuliers.

- les noeuds sont des tuples : `arbre * racine * arbre`



Type 1 :

Arbre binaire simple ne contenant que des données de type `'a`

OCaml
```
type 'a arbre1 =                                    (* données de type 'a *)
    |Nil1                                           (* arbre vide *)
    |N1 of ('a arbre1) * 'a * ('a arbre1);;         (* noeud *)
```

Type 2 :

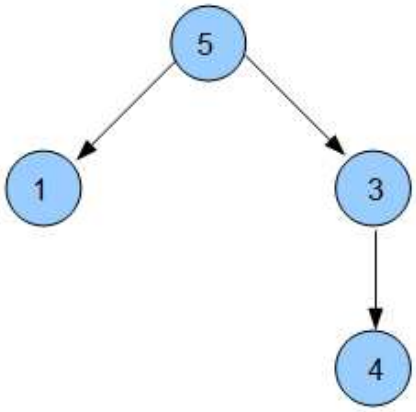
Arbre binaire dont les feuilles et les noeuds sont de type distincts

OCaml
```
type ('n,'f) arbre2 =                                    (* 2 types de données *)
    |Nil2                                                (* arbre vide *)
    |F2 of 'f                                            (* feuille *)
    |N2 of (('n,'f) arbre2) * 'n * (('n,'f) arbre2);;(* noeud *)
```

Les noms `arbre, Nil, F, N` choisis ici sont usuels mais peuvent être remplacés par tout autre nom.
On rappelle que `N1, N2, F2` sont alors des fonctions permettant de convertir une donnée en une donnée de type arbre.

Exemple 1 : Classements

`int arbre1`

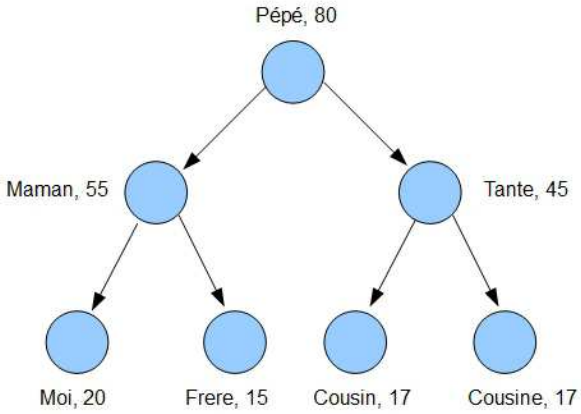


Par convention, les arbres vides ne sont pas représenté sur les dessins.

OCaml
```
let rang = N1(N1(Nil1,1,Nil1),5,N1(N1(Nil1,4,Nil1),3,Nil1));;     (* arbre1 *)

let N1(g1,r1,d1) = rang;;      (* Pour récupérer les composantes de la racines *)
let N1(g2,r2,d2) = g1;;        (* Pour récupérer les composantes du sous-arbre gauche *)
r2;;                           (* renvoie la valeur 1 *)
```

Exemple 2 : Arbre généalogique

string*int arbre1

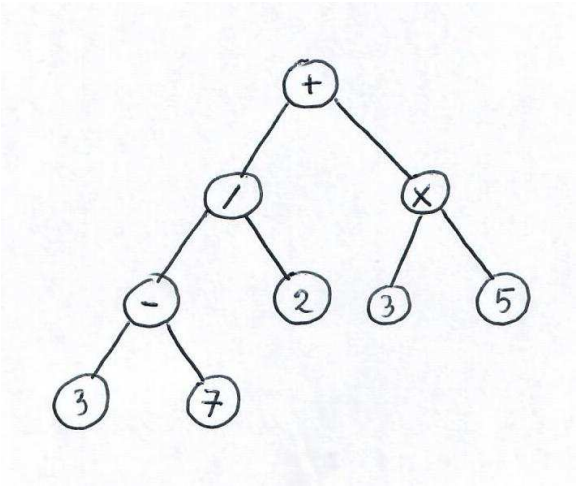


OCaml
```
let famille = N1(N1(N1(Nil1,("moi",20),Nil1) ,        (* sous-arbre gauche *)
                    ("maman",55) ,                    (*       fils gauche *)
                    N1(Nil1,("frere",15),Nil1)),      (*            gauche *)

                ("pépé",80),                          (* La racine de l'arbre *)

                N1(N1(Nil1,("cousin",17),Nil1),       (* sous-arbre droit *)
                   ("tante",45) ,                     (*       fils droit *)
                   N1(Nil1,("cousine",17),Nil1 )));; (*            droit *)

let N1(g1,r1,d1) = famille;;      (* Pour récupérer les deux arbres fils et la racine *)
```

On évitera de définir un arbre sur une même ligne lorsque celui-ci n'est pas trivial.

Exemple 3 : Expression algébrique $(3-7)/2+3*5$

(char,int) arbre2

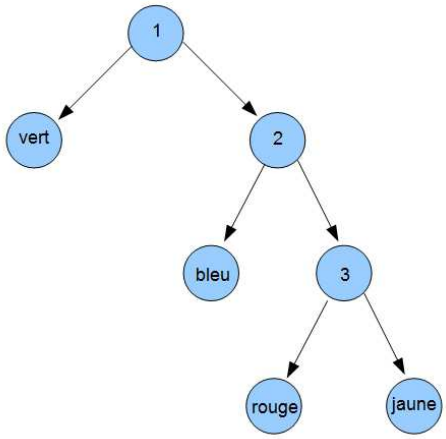


OCaml

```
let expression = N2(
                    N2(N2(F2(3),'-',F2(7)),'/',F2(2)),  (* s-arbre gauche *)
                   '+',                                 (* racine *)
                    N2(F2(3),'*',F2(5))                 (* s-arbre droit *)
                    );;

let N2(g,r,d) = expression;;      (* Pour récupérer les deux arbres fils et la racine *)
```

En Python ?

On utilise des listes emboîtées
Inutile de redéfinir un nouveau type !



On écrit directement :

```
arbre = [["vert"],1,[["bleu"],2,[["rouge"],3,["jaune"]]]]
```

Contrairement à OCaml, cela est possible car les composantes des listes Python peuvent être de types distincts

II] Vocabulaire

Vocabulaire de base :

1. Chaque élément de l'arbre est appelé un *sommet*.

2. Chaque sommet renvoyant sur d'autres données est appelé un *noeud*.
   On parle aussi parfois de *noeuds internes*.
   Chaque noeud contient une information d'un type donné et deux liens (adresses mémoires) vers deux sous-arbres.
   On peut donc considérer un noeud comme un sous-arbre de l'arbre initial.

3. Le sommet de l'arbre est un noeud particulier appelé la *racine* de l'arbre

4. Les sommets terminaux (extrémités) de l'arbre sont appelés les *feuilles* de l'arbre.
   On parle aussi parfois de *noeuds externes*.
   Chaque feuille contient une information d'un type qui peut différer du type des données stockées dans les noeuds.

5. Un noeud donné possède deux *noeuds fils*, qui éventuellement peuvent être des feuilles :
   - un fils gauche
   - un fils droit

   Les feuilles quant à elles, renvoient soit sur des noeuds vides (avec la définition 1) soit sur rien (avec la définition 2).

6. A l'exception de la racine, tout noeud possède un unique *père* : le noeud dont il est le fils.

7. Les noeuds et feuilles situés sous un noeud donné sont appelés les *descendants* du noeud.
   La relation "*est un descendant de*" définit un ordre partiel sur les noeuds et feuilles de l'arbre.
   La racine de l'arbre est le plus petit élément pour cette relation d'ordre.

8. Les noeuds situés au dessus d'un noeud ou d'une feuille donné sont appelés les *ancêtres* du noeud.

9. Les liens entre les noeuds et leurs fils sont appelés les *branches* de l'arbre.

Vocabulaire complémentaire :

1. La *taille* d'un arbre est le nombre de ses noeuds (internes et externes).

2. La *hauteur ou profondeur d'un noeud ou d'une feuille* est le nombre de branches qui le ou la relie à la racine.

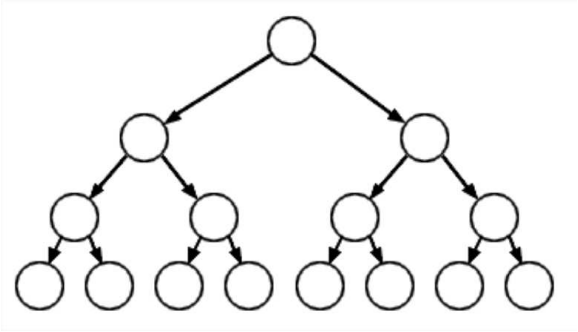
3. La *hauteur ou profondeur* d'un arbre est le maximum des hauteurs des feuilles.

4. La somme des hauteurs des différents noeuds et feuilles est appelée sa *longueur de cheminement*.

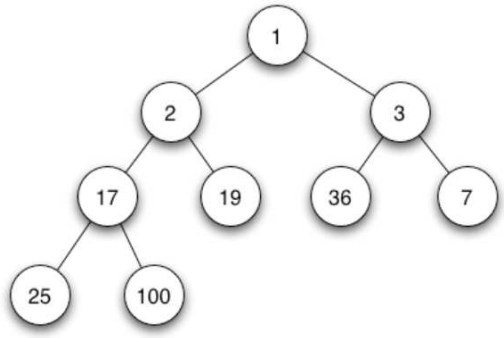
5. Si l'on enlève un noeud à un arbre binaire donné ainsi que toutes les branches qui en partent ou qui y mènent, on obtient :
   - soit un nouvel arbre
   - soit 2 ou 3 arbres que l'on appelle alors une *forêt*.

6. On appelle l'*arité* d'un noeud, le nombre de branche qui partent de ce noeud.
   Dans le cas des arbres binaires, cette arité est égale à 2.

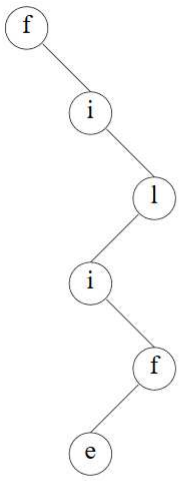
4 arbres binaires particuliers :



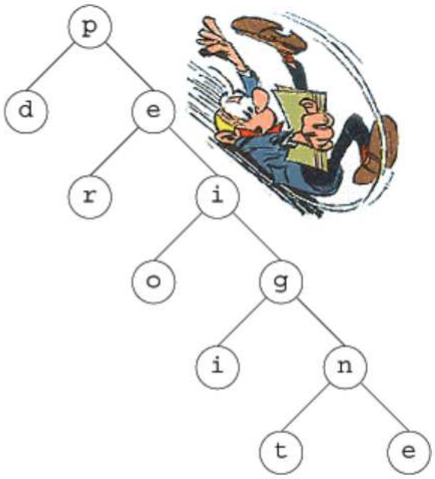
Arbre binaire parfait



Arbre binaire complet



Arbre filiforme



Peigne droit

L'arbre filiforme est également appelé "arbre dégénéré".

III] Opérations élémentaires

Définition du type utilisé ici :

Afin de couvrir tous les cas possibles, nous prenons le type suivant :

OCaml
```
type ('n,'f) arbre =
             | Nil
             | F of 'f
             | N of ('n,'f) arbre * ('n) * ('n,'f) arbre;;
```

Trois constructeurs :

- Une instruction qui crée un arbre vide.

OCaml
```
let arbre1 = Nil;;
```

- `F : element -> arbre` qui construit un arbre limité à une feuille contenant `element`.

OCaml
```
let arbre2 = F(element);;;;    (* création d'un arbre limité à une feuille *)
```

- `N : arbre * element * arbre -> arbre` qui construit un arbre à partir $\left\{ \begin{array}{l} \text{d'un élément} \\ \text{de deux sous-arbres} \end{array} \right.$.

OCaml
```
let arbre3 = N(arbre1,'+',arbre2);;    (* création d'un arbre à partir de 2 sous-arbres *)
```

3 prédicats :

1. Instructions qui teste si un arbre est vide, est une feuille ou est un noeud

OCaml
```
(arbre = Nil) ;;       (* teste si un arbre est vide *)
(arbre = F(f));;       (* teste si un arbre est une feuille *)
(arbre = N(g,r,d));;   (* teste si un arbre est noeud *)
```

Quatre fonctions de sélection :

1. On récupère les fils et la racine d'un noeud grâce à l'instruction :

OCaml
```
let N(g,r,d) = arbre;;          (* g contient le sous-arbre gauche *)
                                (* r contient la racine *)
                                (* d contient le sous-arbre droit *)
```

2. On récupère la donnée stockée dans une feuille grace à l'instruction suivante :

OCaml
```
let F(f) = feuille;;          (* f contient la valeur de la feuille *)
```

IV] Quelques algorithmes usuels

La structure `arbre` étant définie de façon récursive elle est adaptée à la programmation récursive.

Types utilisés dans les exercices suivants :

Selon la nature des exercices, nous utiliserons l'un des deux types d'arbre binaire suivants :

OCaml
```
type 'a arbre1 =                              (* type simple *)
             | Nil
             | F of 'a                        (* on pouvait s'en dispenser *)
             | N of 'a arbre1 * 'a * 'a arbre1;;
```

ou

OCaml
```
type ('n,'f) arbre2 =                              (* type complexe *)
             | Nil
             | F of 'f
             | N of ('n,'f) arbre2 * ('n) * ('n,'f) arbre2;;
```

Exercice 1 : Nombre de feuilles et nombre de noeuds

1. Donner une fonction renvoyant le nombre de feuilles d'un arbre :

OCaml
```
let rec nbrf = function
               | Nil -> 0
               | F(_) -> 1
               | N(g,_,d) -> nbrf g + nbrf d;;
```

2. Donner une fonction renvoyant la taille d'un arbre (nombre total de noeuds et de feuilles) :

OCaml
```
let rec taille = function
               | Nil -> 0
               | F(_) -> 1
               | N(g,_,d) -> 1 + taille g + taille d;;
```

3. Relation entre le nombre de noeuds et le nombre de feuilles :
Pour un arbre t, on note :
   - $n(t)$ le nombre de noeuds (internes) de l'arbre t
   - $f(t)$ le nombre de feuilles de l'arbre t

   Montrer par induction fonctionnelle (sorte de récurrence sur les noeuds de l'arbre en prenant une feuille pour initialisation) que dans le cas d'un **arbre binaire complet**, on a la relation :

$$f(t) = n(t) + 1$$

   Réponse :

   - Initialisation : Cette relation est vraie pour un arbre feuille.
   - Hérédité : Soit `t = N(g,r,d)` un arbre non trivial.
     On suppose que les arbres fils vérifient les relations $\begin{cases} f(g) = n(g) + 1 \\ f(d) = n(d) + 1 \end{cases}$.
     Montrons que `t` vérifie $f(t) = n(t) + 1$.

     → Nous savons que $f(t) = f(g) + f(d)$.
     Ainsi, $f(t) = (n(g) + 1) + (n(d) + 1) = n(g) + n(d) + 2$ et donc $n(g) + n(d) = f(t) - 2$.
     → Or, nous avons également $n(t) = n(g) + n(d) + 1$
     → Les deux relations précédentes donnent alors : $n(t) = f(t) - 2 + 1 = f(t) + 1$

Exercice 2 : Ajouter une feuille à un arbre

Déterminer un algorithme récursif permettant d'ajouter une feuille à un arbre de type `'a arbre`.

Pour éviter un déséquilibre systématique, on pourra utiliser une variable aléatoire permettant de choisir l'arbre fils auquel cette feuille est ajoutée. La commande `Random.int 2` *renvoie un entier choisi entre* 0 *et* 1.

OCaml
```
let rec ajout_f f = function
                | Nil -> F(f)
                | N(Nil,r,d) -> N(F(f),r,d)
                | N(g,r,Nil) -> N(g,r,F(f))
                | N(g,r,d) -> match Random.int 2 with
                                  | 0 -> N(ajout_f f g,r,d)
                                  | 1 -> N(g,r,ajout_f f d);;
```

Cette fonction est loin d'être optimale car elle peut renvoyer un message d'erreur même si l'arbre n'est pas complet.

Exercice 3 : Hauteur d'un arbre

1. Il s'agit de la profondeur maximale d'une feuille de l'arbre où la profondeur d'une feuille est définie comme le nombre de branches qui séparent la feuille de sa racine.

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

2. En reprenant les notations précédentes, et en notant $h(t)$ la hauteur d'un arbre t, prouver par induction fonctionnelle que :
   $$h(t) + 1 \leq f(t) \leq 2^{h(t)}$$

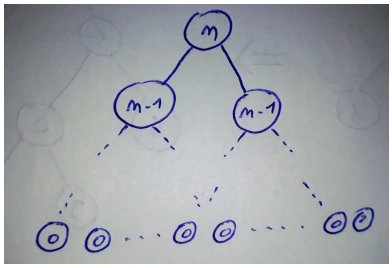
   Réponse : A faire sur le Forum.

3. Montrer, en considérant les arbres parfaits et les arbres peignes, que les bornes peuvent être atteintes.

   Réponse : A faire sur le Forum.

Exercice 4 : Construction d'un arbre parfait

Construire une fonction qui permet de construire un arbre parfait de hauteur $n \in \mathbb{N}^*$ dont les noeuds contiennent la hauteur de l'arbre dont ils sont la racine.
Vérifier que $c(n) = O(n)$.

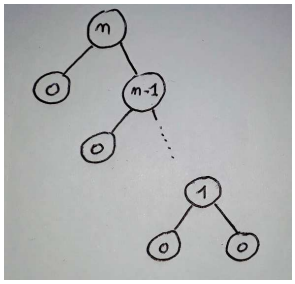


OCaml
```
let rec arbre_parfait = function
           |0 -> F(0)
           |n -> let t = arbre_parfait (n-1) in N(t,n,t);;

arbre_parfait 4;;
```

Complexité : On a $c(n) = c(n-1) + 1$ ce qui nous donne $c(n) = O(n)$.

Exercice 5 : Construction d'un peigne droit

Construire une fonction qui permet de construire un arbre peigne droit de hauteur $n \in \mathbb{N}^*$ dont les noeuds contiennent la hauteur de l'arbre dont ils sont la racine.
Vérifier que $c(n) = O(n)$.

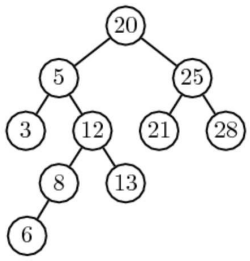


OCaml
```
let rec peigne_droit = function
           |0 -> F(0)
           |n -> let t = peigne_droit (n-1) in N(F(0),n,t);;

peigne_droit 4;;
```

Exercice 6 : EPITA 2019

Dans cet exercice, on laisse à l'élève le choix de la structure d'arbre utilisée.

1. Proposer un algorithme permettant de calculer le maximum des valeurs des branches d'un arbre.
   - *Une branche désigne ici le chemin allant de la racine à une feuille.*
   - *La valeur d'une branche est la somme des valeurs des noeuds et feuille de la branche.*



   Réponse : A faire sur le Forum.

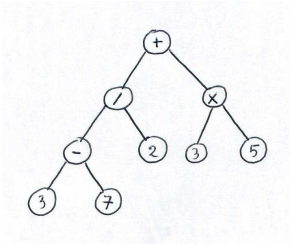
2. Proposer un algorithme vérifiant si un arbre est symétrique.

   On pourra utiliser deux fonctions auxilliaires, l'une qui transforme un arbre en son symétrique et l'autre qui teste l'égalité de deux arbres.

   Réponse : A faire sur le Forum.

Exercice 7 : Evaluation d'une expression algébrique

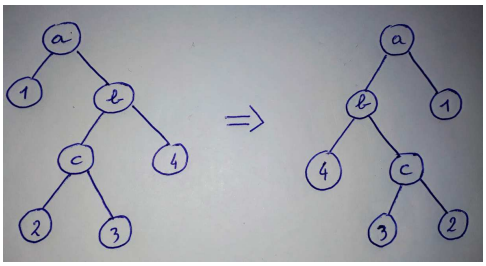
Proposer une procédure OCaml permettant d'évaluer une expression algébrique donnée sous la forme d'un arbre.



OCaml
```
let rec eval = function
                | Nil -> failwith "l'expression est vide"
                | F(f) -> f
                | N(g,r,d) -> let a = eval g and
                                  b = eval d in
                                  match r with
                                     | '+' -> a +. b
                                     | '-' -> a -. b
                                     | '*' -> a *. b
                                     | '/' -> a /. b;;

let expression = N(N(F(2.),'*',F(5.)), '+' , N(F(3.),'-',F(5.)));;
eval expression;;
```

Exercice 8 : Symétrie

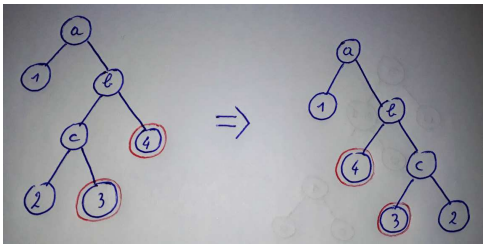
Proposer une fonction OCaml qui renvoie le symétrique d'un arbre.



OCaml
```
let rec sym = function
     |Nil -> Nil
     |F(a) -> F(a)
     |N(g,r,d) -> N(d,r,g)

let expression = N(N(F(2.),'*',F(5.)), '+' , N(F(3.),'-',F(5.)));;
sym expression;;
```

Exercice 9 : Déplacement des feuilles

Proposer une fonction OCaml qui place à gauche les feuilles qui se trouvent à droite d'un arbre.



OCaml
```
let rec feuilles_g = function
     |Nil -> Nil
     |F(a) -> F(a)
     |N(g,r,F(a)) -> N(F(a),r,feuilles_g g)
     |N(g,r,d) -> N(feuilles_g g,r, feuilles_g d);;

let expression = N(N(F(2.),'*',F(5.)), '+' , N(F(3.),'-',F(5.)));;
feuilles_g expression;;
```

V] Modes de parcours d'un arbre

Quel intérêt ? :

Il est important de savoir parcourir tous les noeuds et feuilles d'un arbre afin par exemple, de :

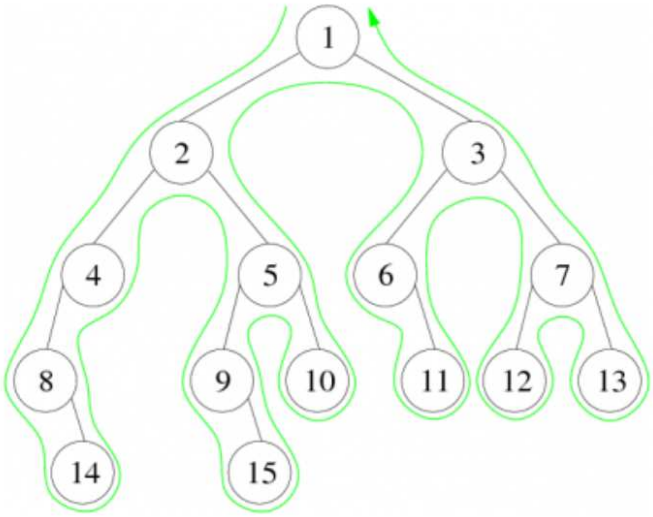
- Rechercher si un arbre contient une valeur particulière
- Compter les noeuds qui contiennent une valeur donnée
- Etablir un ordre total entre les données stockées... etc...

Il existe deux façons usuelles de parcourir un arbre : $\left\{ \begin{array}{l} \text{le parcours en profondeur} \\ \text{le parcours en largeur} \end{array} \right.$.

Parcours en profondeur :

On visite entièrement le sous-arbre de gauche avant de visiter celui de droite.
Cela donne l'algorithme suivant :

OCaml
```
let rec parcours_prof = function
                        | Nil      -> on ne fait rien
                        | F(f)     -> action à effectuer sur "f"
                        | N(g,a,d) -> action à effectuer sur "a"
                                      parcours_prof g;
                                      parcours_prof d;;
```



Si l'on considère les noeuds comme des îles et les branches comme des bandes de terres, on constate que le parcours effectué par cet algorithme correspond au parcours effectué par un navigateur qui circulerait autour de l'arbre dans le sens trigonométrique en partant de la racine.

Exercice 1 : Maximum

Contruire une fonction qui renvoie la plus grande valeur contenu dans un arbre de type `int arbre`.

1. Avec un programme récursif naturel qui ne se soucie pas du mode de parcours
2. En utilisant une référence de stockage du maximum temporaire et une fonction auxilliaire récursive de

parcours en profondeur.

On utilisera la structure :

OCaml
```
type 'a arbre =
              | Nil
              | F of 'a
              | N of 'a arbre * ('a) * 'a arbre;;
```

Réponse 1 : Sans se soucier du mode de parcours

OCaml
```
let rec max_noeud1 = function
                 | Nil -> 0
                 | F(f) -> f
                 | N(g,r,d) -> let a = max_noeud1 g and
                                   b = max_noeud1 d in
                               max r (max a b);;

let test1 = N(N(N(F(4),8,Nil),6,N(Nil,12,F(2))),1,N(N(F(1),4,F(3)),3,F(7)));;
max_noeud1 test1;;

# max_noeud1 test1;;
- : int = 12
```

Réponse 2 : Avec un parcours en profondeur

OCaml
```
let max_noeud2 a = let N(g,r,d) = a in
                   let m = ref r in               (* pour stocker le maximum *)
                   let rec aux = function         (* pour parcourir l'arbre *)
                           | Nil -> m := !m
                           | F(f) -> m := max (!m) f
                           | N(g1,r1,d1) -> m := max (!m) r1;
                                            aux g1;
                                            aux d1
                   in aux a;
                      !m;;

let test1 = N(N(N(F(4),8,Nil),6,N(Nil,12,F(2))),1,N(N(F(1),4,F(3)),3,F(7)));;
max_noeud2 test1;;

# max_noeud2 test1;;
- : int = 12
```

Exercice 2 : Liste des noeuds

Déterminer une fonction qui renvoie la liste des valeurs des noeuds obtenue en parcourant un arbre en profondeur.

1. A l'aide d'une procédure récursive naturelle utilisant la concaténation.
2. A l'aide d'une référence permettant de stocker les éléments et une fonction auxilliaire de parcours en profondeur.

On utilisera la structure :

OCaml
```
type 'a arbre =
             | Nil
             | F of 'a
             | N of 'a arbre * ('a) * 'a arbre;;
```

Réponse 1 : Avec une concaténation

OCaml
```
let rec liste_noeud1 = function
      |Nil -> []
      |F(a) -> [a]
      |N(g,r,d) -> (r::liste_noeud1 g)@(liste_noeud1 d);;

let a = N(N(N(F(3),5,F(7)),-1,F(2)),0,N(F(3),-4,F(5)));;
liste_noeud1 a;;

# liste_noeud1 a;;
- : int list = [0; -1; 5; 3; 7; 2; -4; 3; 5]
```

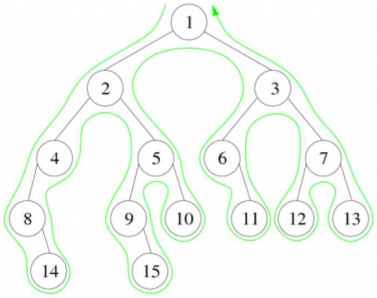
Réponse 2 : Avec un parcours en profondeur

OCaml
```
let list_noeud2 a =  let l = ref [] in                  (* liste de stockage des noeuds *)
                     let rec aux = function             (* fonction de parcours en profondeur *)
                                   | Nil -> l := !l
                                   | F(f) -> l := f::(!l)
                                   | N(g,r,d) -> l := r::(!l);
                                                 aux g;
                                                 aux d
                     in aux a;
                     List.rev (!l);;

let test1 = N(N(N(F(4),8,Nil),6,N(Nil,12,F(2))),1,N(N(F(1),4,F(3)),3,F(7)));;
list_noeud test1;;

# list_noeud2 test1;;
- : int list = [1; 6; 8; 4; 12; 2; 3; 4; 1; 3; 7]
```

Exercice 3 : Parcours infixe

Déterminer l'ordre de lecture des éléments de l'arbre de la figure suivante lorsque l'algorithme est le suivant :

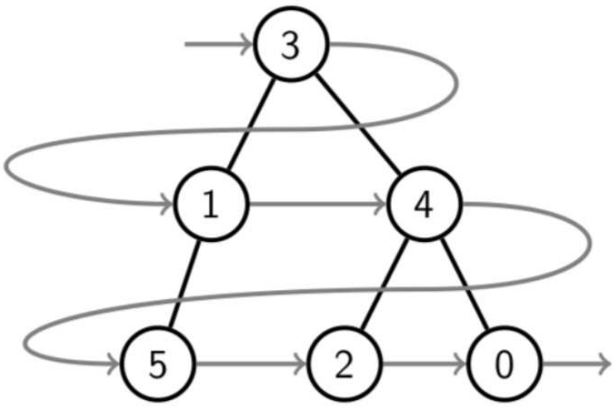
OCaml
```
let rec parcours_infixe = function
                        | Nil      -> on ne fait rien
                        | F(f)     -> lecture de "f"
                        | N(g,r,d) -> parcours_prof g;
                                      lecture de "r" ;
                                      parcours_prof d;;
```



Cela donne : 8 - 14 - 4 - 2 - 9 - 15 - 5 - 10 - 1 - 6 - 11 - 3 - 12 - 7 - 13

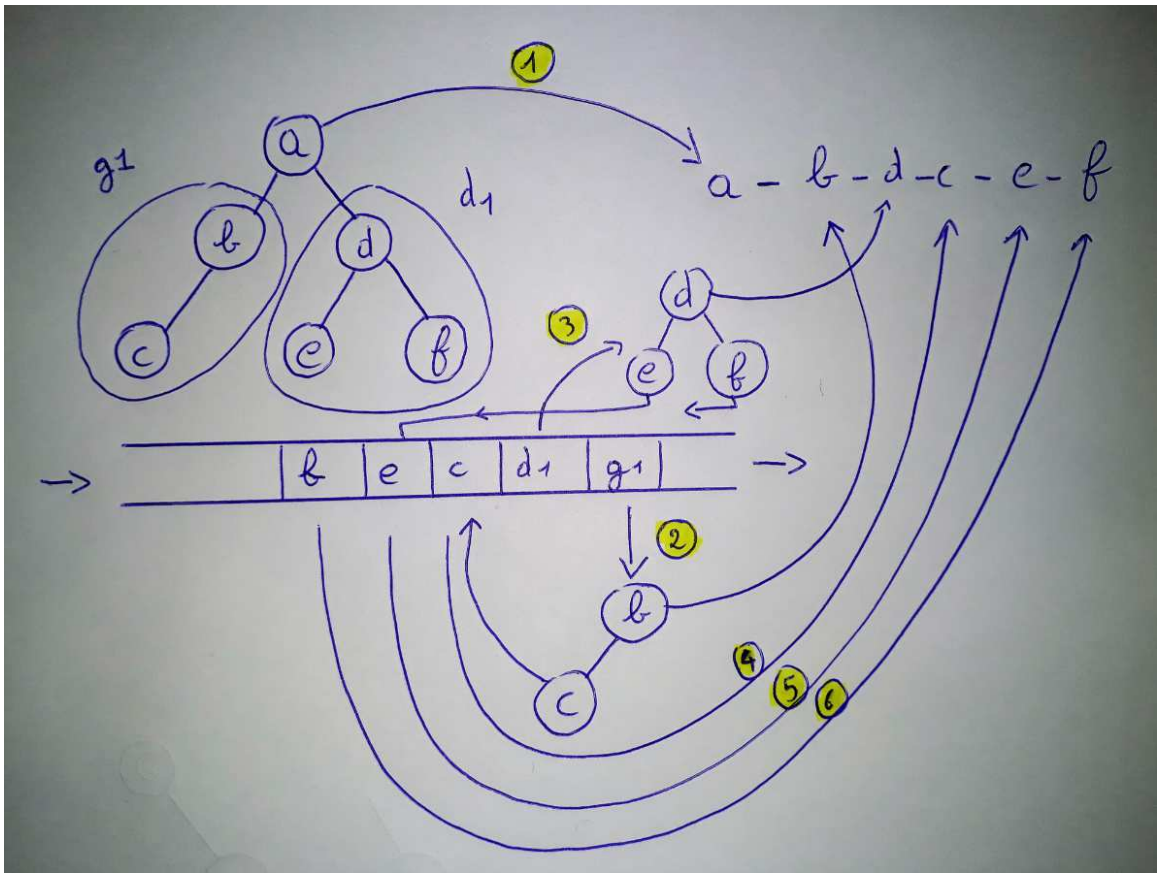
Parcours en largeur :

On commence par la racine, puis on visite chacun de ses fils (qui sont en profondeur 1) en commençant par exemple la gauche. On visite les noeuds et les feuilles situées en profondeur 2... etc...



La programmation de cette méthode utilise une file de stockage.
L'idée est alors la suivante :

OCaml
```
- On commence par lire la valeur du sommet S puis on stocke dans
  la file ses deux arbres fils g1 puis d1, puis

- On retire de la file g1, on récupère la valeur de la racine, puis
  on stocke dans la file les deux fils de g1 : g2 puis d2, puis

- On retire de la file d1, on récupère la valeur de la racine, puis
  on stocke dans la file les deux fils de d1 : g3 puis d3 puis... etc...

- on poursuit ainsi l'algorithme jusqu'à ce que la file soit vide ...
```



Exercice : Parcours en largeur

Déterminer en pseudo-langage une fonction qui renvoie la liste des valeurs obtenues en parcourant un arbre en largeur.

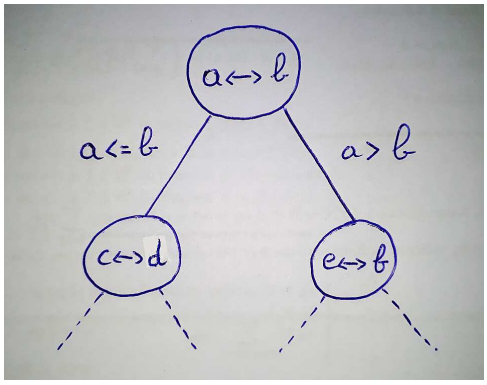
Réponse :

```
Pour un arbre a = N(g,r,d)
- création d'une file vide : f
  création d'une liste vide : l
- stockage de r dans l
- stockage de g puis de d dans f
- tant que la file l est non vide faire :
     - retirer un element de la file f
     - si c'est une feuille : stocker le contenu dans l
       sinon : - stocker la racine dans l
               - stocker le fils gauche dans la file f
               - stocker le fils droite dans la file f
- renvoyer l
```

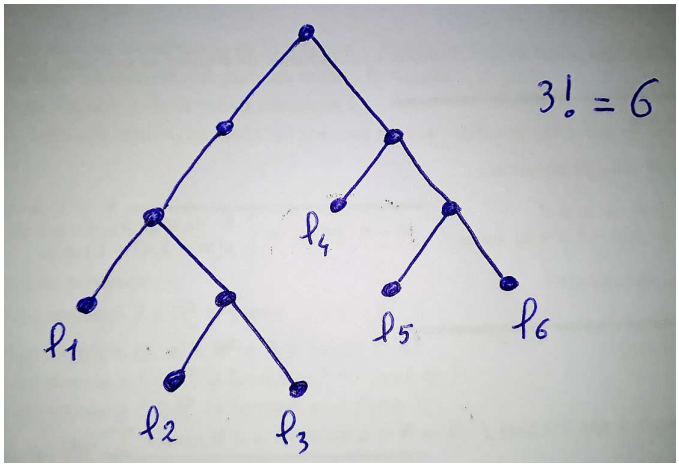
VI] Trois applications usuelles des arbres binaires

Complexité minimale d'un algorithme de tri par comparaison :

- On fixe un algorithme de tri par comparaisons 2 à 2 des éléments donnés.
- Les différentes étapes de la mise en oeuvre de cet algorithme peuvent se représenter par un arbre binaire où chaque branche partant d'un noeud correspond aux deux éventualités lors de la comparaison de 2 éléments.



- L'arbre droit correspond à la suite de l'algorithme dans l'un des cas et l'arbre gauche la suite dans l'autre cas.
- Chaque branche de l'arbre (de la racine à une feuille) représente les étapes nécessaires au tri d'une liste donnée.



- Sachant qu'il y a $n!$ listes possibles, alors notre arbre binaire contiendra $n!$ branches (ou feuilles).
- Or, le plus petit arbre contenant $n!$ feuilles est au mieux un arbre parfait de hauteur h_0 tel que $2^{h_0} = n!$.
  La hauteur $h \geq h_0$ de l'arbre correspondant à notre tri vérifie donc $2^h \geq n!$ c'est à dire

$$h \geq \frac{\ln n!}{\ln 2}$$

- Or, la hauteur h de notre arbre correspond à la liste qui nécessite le plus de comparaisons pour être traitée.
- La complexité de notre algorithme (en nombre de comparaisons) dans le pire des cas vérifie :

$$c(n) \geq \frac{\ln n!}{\ln 2}$$

- Or, la formule de Stirling nous permet de vérifier que $\ln n! = O(n \ln n)$.

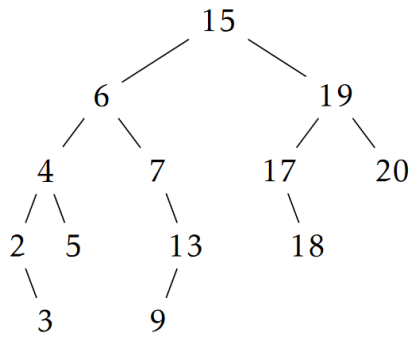
Preuve : Comme $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \to +\infty$ *alors* $\ln n! \sim \ln\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)$.

Or $\ln\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = \ln\sqrt{2\pi} + \frac{1}{2}\ln n + n\ln n - n\ln e = n\ln n + o(n\ln n) \sim n\ln(n)$

Nous avons ainsi démontré que la complexité dans le pire des cas (en comptant le nombre de comparaisons) d'un algorithme de tri est au minimum un $O(n \ln n)$.

Arbre binaire de recherche :

Il s'agit d'un arbre stockant des données de la forme `(cle, contenu)` et vérifiant les proprétés suivantes.



Seules les clés ont été représentées sur la figure

Pour chaque noeud de l'arbre :

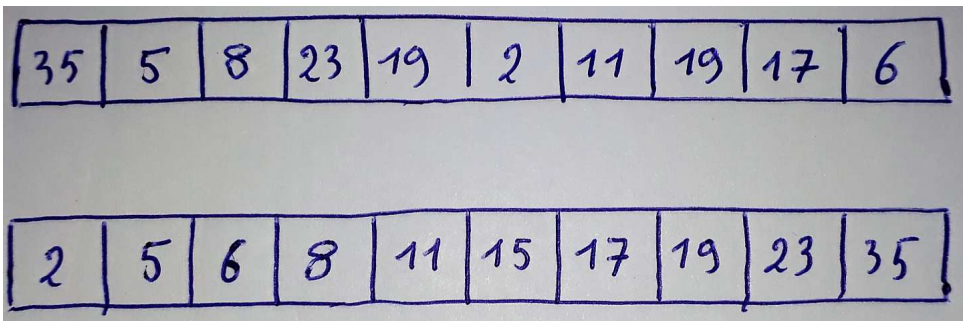
- La clé de d'élément stocké dans le noeud-fils gauche est strictement plus petite
- La clé de d'élément stocké dans le noeud-fils droit est strictement plus grande

Cette structure est donc une alternative à la structure de dictionnaire vue dans le premier chapitre. Lorsque l'arbre est équilibré, cette façon de stocker les données présente l'avantage de diminuer la complexité de l'algorithme de recherche.

Quel intérêt ? : *Lire l'article de Hervé Lehning...*

Supposons que l'on recherche une valeur dans une collection de n données :

Exemple où $n = 10$



- Dans le tableau non trié, la recherche de "6" demande 10 comparaisons
- Dans le tableau trié, la recherche d'un élément demande au maximum 4 comparaisons

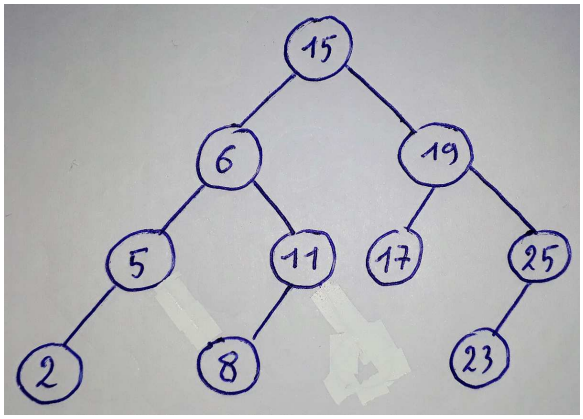
Plus généralement :

- Dans un tableau non trié, l'algorithme de recherche est de complexité $O(n)$
- Dans un tableau trié, l'algorithme de recherche est de complexité $O(\ln(n))$

La structure de tableau trié semble donc adaptée, mais l'ajout d'un élément a une complexité en $O(n)$ car pour placer un nouvel élément, il nous faut décaler tout ceux qui sont plus grands.. Ce type de structure est donc adaptée aux bases de données qui évoluent peu...

Dans le cas contraire, on peut utiliser un arbre équilibré de recherche appelé *arbre AVL*.

Exemple où $n = 10$ (suite)



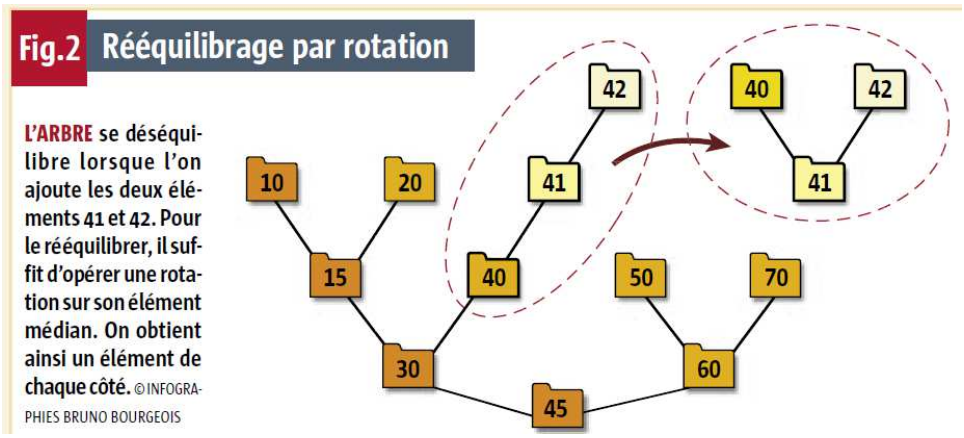
Dans cet arbre la recherche d'un élément se faire en au plus de 4 étapes.

Exercice : 1

Prouver que pour un arbre équilibré de recherche contenant n valeurs, la complexité dans le pire des cas de la recherche d'une clé est un $O(\ln(n))$.

A retenir :

- La recherche d'un élément dans un arbre AVL a une complexité logarithmique
- L'ajout d'un élément dans un AVL a une complexité logarithmique
- L'ajout de nouveaux éléments engendre un déséquilibre progressif de l'arbre qui augmente peu à peu la complexité moyenne. Pour conserver la complexité logarithmique, on procède alors régulièrement à des ré-équilibrage par "rotations".

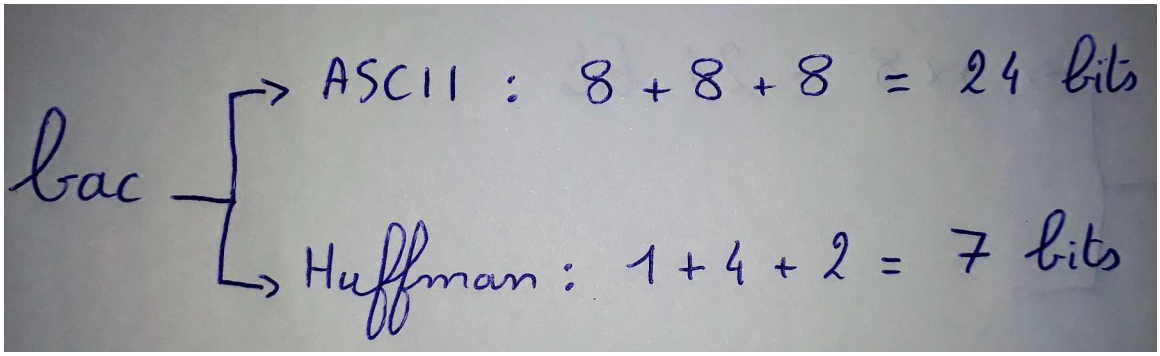


Algorithme de Huffmann pour la compression de données :

Lire l'article de Hervé Lehning...

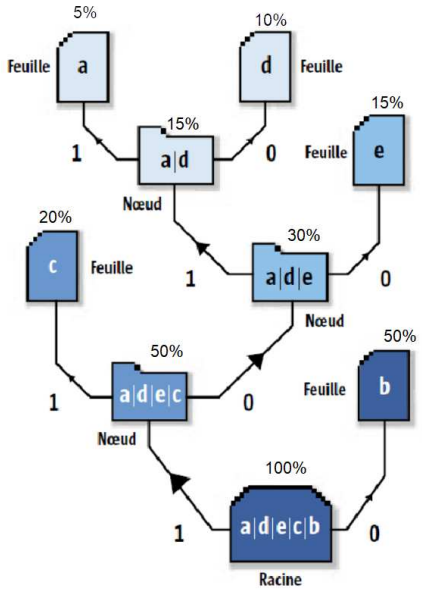
On souhaite coder un texte donné.

- Usuellement, les caractères sont codés en ASCII sur 8 bits.
- L'idée de Huffmann est de coder les caractères les plus fréquents du texte sur moins de bits, quitte à coder les moins fréquents sur un peu plus de bits. En moyenne, on gagne de la place !



Méthode :

- On commence par classer les caractères du texte à compresser par fréquence croissante.
- On répartit alors ces lettres dans un arbre de Huffman.
  Voir l'article sur le mode de construction de cet arbre.



Exemple : Texte ne contenant que les lettres :

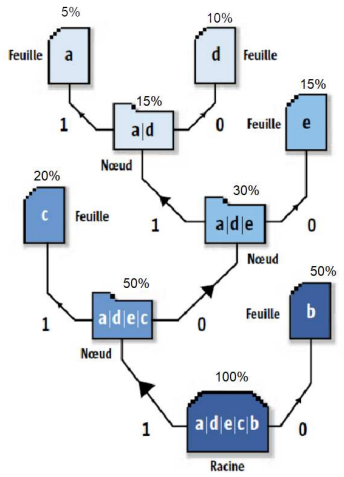
a (5%) - b (50 %) - c (20 %) - d (10 %) - e (15 %)

- On peut alors associer à chacune des lettres les codes : 0 - 11 - 100 - 1011 - 10110 - etc...
- Ces codes présentent l'avantage ne n'être le préfixe d'aucun autre (car « le chemin qui mène à une feuille ne peut passer par une autre feuille ») et peuvent ainsi être représentés successivement dans la mémoire de l'ordinateur.

On parvient ainsi à une compression moyenne de 40%, valeur encore inégalée lorsqu'il s'agit de coder un texte via le codage de ses caractères.

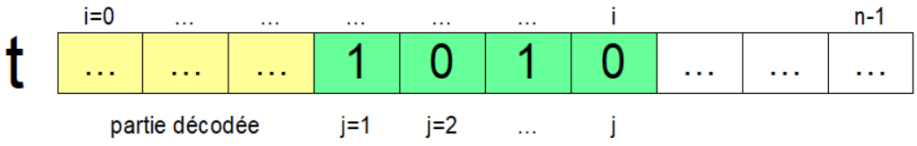
Exercice : 2

Concevoir un programme de décodage d'un texte codé selon l'arbre de Huffmann ci-dessous.



Réponse :

```
nom : decodage
Argument : t : tableau contenant le code
           a : arbre de Huffmann
Sortie : la liste contenant les caractères du mot décodé
Algorithme :
```




```
l = []        Pour le stockage des caractères décodés
n = taille du tableau t
i=0                                        (* indice du bit traité *)
j=1                                        (* rang du bit traité dans le code d'une lettre *)
fils = a                                   (* arbre servant au décodage *)

tant que i < n faire :   - tant que t.(i) = (j mod 2) faire :
                                        - j = j+ 1
                                        - i = i + 1
                                        - si t.(i)=1 alors fils = fils droit
                                                     sinon fils = fils gauche
                         (**** fin du décodage d'une lettre ****)

                         - stocker la feuille de a contenant le caractère dans l
                         - j = 1          (* réinitialisation de j *)
                         - i = i + 1      (* incrémentation de j *)
                         - fils = a       (* réinitialisation de fils *)
                         (**** fin de lecture du tableau t ****)

renvoyer la liste l inversée
```

Définition du type :

OCaml
```
type 'a arbre =                                          (* Définition du type d'arbre *)
             | Nil
             | F of 'a
             | N of 'a arbre * ('a) * 'a arbre;;
```

Programmation de la fonction de décodage :

OCaml
```
let decode t a = let n = Array.length t and              (* Fonction de décodage *)
                     fils = ref a and
                     i = ref 0 and
                     j = ref 1 and
                     l = ref [] in
   while !i < n do while t.(!i) = (!j mod 2) do let N(g,r,d) = !fils in
                                                 if t.(!i) = 1 then fils := d
                                                               else fils := g;
                                                 j := !j + 1;
                                                 i := !i + 1;
                                           done;
                 if t.(!i) = 0 then let N(F(f),r,d) = !fils  in l := f::(!l);
                                                                j := 1;
                                                                i := !i + 1;
                                                                fils := a
                                    else let N(g,r,F(f)) = !fils in l := f::(!l);
                                                                    j := 1;
                                                                    i := !i + 1;
                                                                    fils := a
              done;
   List.rev (!l);;
```

Vérification :

OCaml
```
let a = N(F('b') ,                    (* Définition de l'arbre de Huffmann *)
          '_' ,
          N(N(F('e') ,
              '_' ,
                N(F('d') ,
                  '_' ,
                    F('a'))) ,
            '_' ,
             F('c')));;
let t = [|0;1;0;1;1;1;1|];;           (* Défiition du tableau à décoder *)

decode t a;;
# decode t a;;
- : char list = ['b'; 'a'; 'c']      (* résultat *)
```