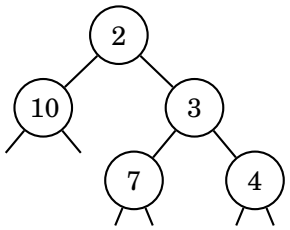
TP n°15 – Arbres et tries

1 Mise en jambes : arbres binaires

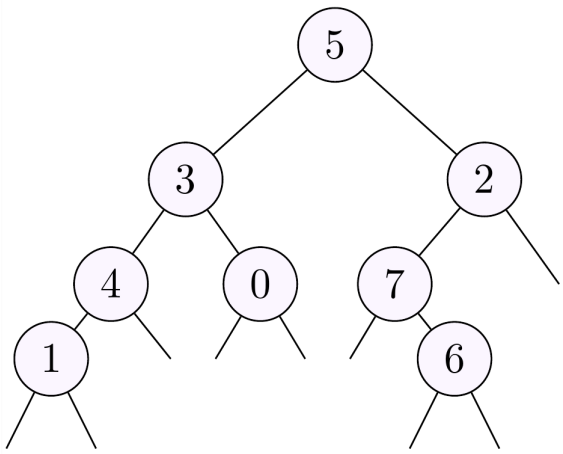
On rappelle qu'en Ocaml les arbres binaires peuvent être représentés par le type suivant :

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



- **Q1.** Représenter l'arbre ci-dessus en Ocaml.
- **Q2.** Écrire une fonction récursive `hauteur : 'a arbrebin -> int` qui calcule la hauteur d'un arbre.
- **Q3.** Écrire une fonction récursive `somme : int arbrebin -> int` qui calcule la somme des étiquettes d'une arbre.

On considère l'arbre suivant :



- **Q4.** Donner l'ordre dans lequel les noeuds sont traités par :
  - Un parcours en profondeur infixe
  - Un parcours en profondeur préfixe
  - Un parcours en profondeur suffixe
  - Un parcours en largeur
- **Q5.** Écrire une fonction en Ocaml `est_dans a e: 'a arbrebin-> 'a -> bool` qui vérifie si une étiquette est dans un arbre en utilisant un parcours en largeur.
  On pourra utilsier le module `Queue` de Ocaml. Les primitives ont les noms suivants, le type `'a Queue.t` désignant une file d'éléments de type `'a` :
  - `Queue.create : unit -> 'a Queue.t` qui crée une file vide
  - `Queue.is_empty : 'a Queue.t -> bool` qui vérifie si une file est vide
  - `Queue.push : 'a -> 'a Queue.t -> unit` qui ajoute un élément (equivalent de enfile)
  - `Queue.pop : 'a Queue.t -> 'a` qui retire et renvoie l'élément le plus récent (équivalent de defile)
  - `Queue.peek : 'a Queue.t -> 'a` qui renvoie sans retirer l'élément le plus récent.
- **Q6.** Dessiner l'arbre parfait à 15 sommets. Que remarquez-vous sur le nombre de fils d'un sommet d'un arbre parfait ? Que remarquez vous sur la hauteur des deux sous-arbres d'un noeud dans un arbre parfait ? On admettre que ces deux propriétés caractérisent les arbres parfaits.
- **Q7.** En utilisant la remarque précédente, écrire en Ocaml une fonction qui vérifie si un arbre est parfait en utilisant un parcours en profondeur.

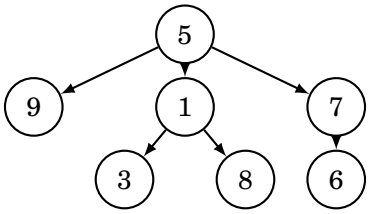
2 Arbres n-aires

On définit les arbres n-aires ainsi en Ocaml (**Ici on utilise une liste pour ranger les sous-arbres**) :

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

On définit aussi un alias pour les listes d'arbres, qui sont des forêts.

```
type 'a foret = 'a arbre list;;
```



Exemple d'arbre

- **Q8.** Définir l'arbre précédent en Ocaml.

On peut calculer la hauteur en Caml à l'aide de deux fonctions mutuellement récursives :

```
let rec hauteur arbre =
  match arbre with
    | Noeud(_,[]) -> 0
    | Noeud (_, fils) -> 1 + max_hauteur_foret fils
and max_hauteur_foret foret = match foret with
    | [] -> 0
    | arbre :: autres -> max (hauteur arbre) (max_hauteur_foret autres)
;;
```

- **Q9.** Compléter le code suivant pour écrire la fonction `taille : 'a arbre -> int`.

```
 let rec taille arbre = (*calcule le nombre de sommets d'un arbre*)
  match arbre with
    | Noeud(_,[]) ->
    | Noeud (_, fils) ->
and taille_foret foret = (*calcule le nombre de noeuds dans une foret d'arbres*)
    match foret with
    | [] ->
    | arbre :: autres ->
;;
```

On peut étendre la notion de parcours vue sur les arbres binaires aux arbres quelconques.

- Le parcours préfixe consiste à traiter l'étiquette avant de traiter les sous-arbres.
- Le parcours infixe n'a plus de sens puisqu'on peut avoir un nombre quelconque de sous-arbres.
- Le parcours suffixe consiste à traiter les sous-arbres avant de traiter l'étiquette.
- Le parcours en largeur est défini de la même façon.

- **Q10.** Écrire une fonction `parcours_prefixe : int arbre -> unit` qui affiche les noeuds dans l'ordre du parcours préfixe. Comme pour la taille, on peut faire une fonction pour traiter un noeud et une fonction pour traiter une forêt.
- **Q11.** Écrire les fonctions `ordre_suffixe : int arbre -> int list` qui renvoient les listes des nœuds dans l'ordre du parcours suffixe.
- **Q12.** Écrire une fonction `degre : 'a arbre -> int` qui donne le degré d'un arbre, c'est-à-dire la plus grande arité de ses nœuds. L'arité d'un noeud est son nombre de sous-arbres.

3 Tries

Dans cette partie on cherche à implémenter un algorithme d'auto-complétion, comme lorsqu'on appuie sur la touche tab dans le terminal.

On dispose d'un vocabulaire v, une liste de mots qu'on veut pouvoir compléter une fois qu'on en a écrit un préfixe p.

Par exemple si notre vocabulaire est `let mots = ["diane"; "dire"; "diva"; "divan"; "divin"; "do"; "dodo"; dodu"; "don"; "donc"; "dont"];;` et que le préfixe déjà tapé est `let p = "dir"`, alors on peut auto-compléter en "dire". Par contre si on a `let p = "dod"`, on peut compléter en "dodo" ou "dodu".

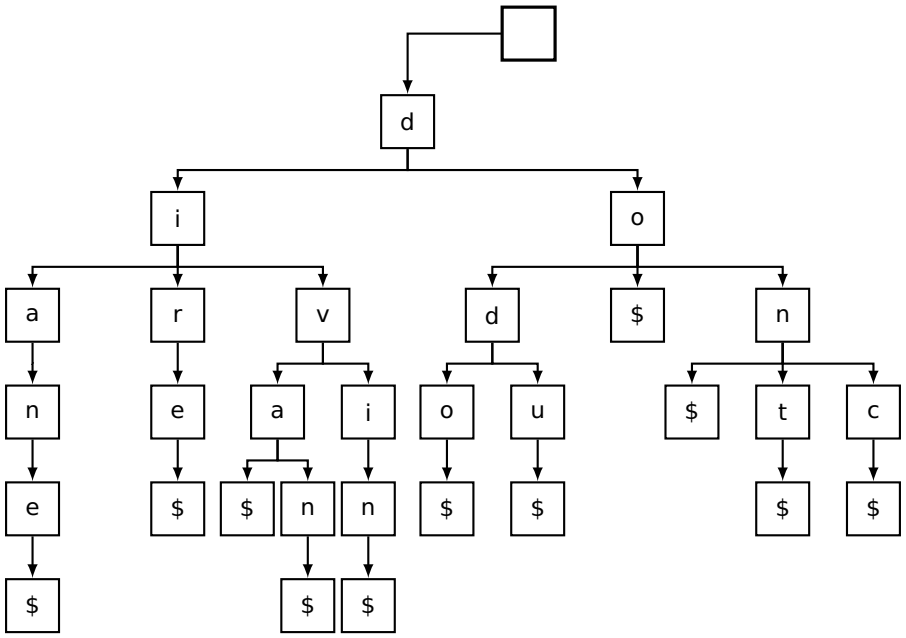
Une approche naive serait de parcourir tous les mots de v, lettre à lettre jusqu'à trouver un mot dont le préfixe est p, ce qui est en $O(|v| * |p|)$.

Une manière plus efficace quand on veut faire la completion plusieurs fois de suite est de créer un trie.

Un trie (ou arbre préfixe) est un arbre dont la racine est étiquetée d'un caractère vide, les feuilles sont étiquetées d'un symbole spécial \$ qui indique une fin de mot et chaque autre sommet est étiqueté d'une lettre. Les mots qui ont les mêmes préfixes sont rangés dans les mêmes branches de l'arbre.

S'il existe un chemin entre la racine et une feuille \$, alors les lettres qu'on croisent sur le chemin forment un mot du vocabulaire.

Pour le vocabulaire exemple précédent, on obtient le trie suivant :



3.1 Utilitaires pour les mots

En Ocaml, les **string** sont comme des tableaux de caractères, mais avec une syntaxe différente.
On accède à la lettre en position i d'une chaine `chaine` (la numérotation commence à 0) avec `chaine.[i]`
On calcule la longueur d'une chaine de caractères avec `String.length`
On concatène deux chaines de caractères avec `^`
Pour transformer un caractère **c** en une chaine de caractères de longueur 1, on utilise `String.make 1 c`.

On définit l'alias de type suivant : `type mot = char list`

- **Q13.** Écrire une fonction `mot_of_string : string -> mot` prenant en entrée une chaîne de caractères et renvoyant la liste de ses caractères, avec un caractère \$ rajouté à la fin.

```
# mot_of_string "bonjour";;
- : char list = ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'; '$']
```

- **Q14.** Écrire une fonction `afficher: mot -> unit` qui prend en entrée une liste de caractères et affiche le mot correspondant, suivi d'un retour à la ligne. Les éventuels caractères \$ seront ignorés.

```
# afficher_liste ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'; '$'];;
bonjour
- : unit = ()
```

3.2 Opérations élémentaires sur les tries

- **Q15.** Écrire une fonction `cardinal: trie -> int` renvoyant le nombre de mots contenus dans le vocabulaire d'un trie.
- **Q16.** Écrire une fonction `appartient : trie -> mot -> bool` qui détermine si un mot est dans le vocabulaire d'un trie.
- **Q17.** Écrire une fonction `ajouter : trie -> mot -> trie` qui ajoute un mot à un trie.

- **Q18.** Écrire une fonction `trie_of_list : string list -> trie` prenant en entrée une liste de chaînes de caractères et renvoyant un trie contenant exactement les mots de cette liste (auxquels on a ajouté un $).
- **Q19.** Écrire une fonction `afficher_mots : trie -> unit` qui affiche tous les mots appartenant au vocabulaire d'un trie (sans les $ finals), à raison d'un mot par ligne.

3.3 Autocomplétion

- **Q20.** Écrire une fonction `autocompletion : trie -> string -> unit` qui étant donné un trie et la partie du mot déjà tapée, affiche tous les mots que l'utilisateur pourrait être en train de taper.
  On pourra refaire la question mais en renvoyant cette fois la liste des mots qu'il est possible de former.

3.4 Annagrammes

- **Q21.** Écrire une fonction `calculer_occurrences : string -> int array` qui prend en entrée une chaîne de caractères `s` et renvoie un `int array` de longueur 256 tel que `t.(i)` soit égal au nombre d'occurrences de `int_of_char i` dans `s`.
- **Q22.** Écrire une fonction `afficher_mots_contenus : trie -> string -> unit` qui prend en entrée un mot sous forme de chaîne de caractères (sans $ final) et un trie, et affiche tous les mots du trie que l'on peut former en utilisant tout ou partie des lettres du mot fourni. Si une lettre est répétée dans l'entrée elle peut être utilisée autant de fois qu'elle est répétée, sinon elle ne peut être utilisée qu'une seule fois.
- **Q23.** Écrire une fonction `afficher_anagrammes : trie -> string -> unit` qui prend en entrée un mot sous forme de chaîne de caractères et affiche toutes ses anagrammes présentes dans le trie. Une anagramme d'un mot est un mot constitué exactement des mêmes lettres (avec le même nombre d'occurrences) mais dans un ordre différent (on considérera qu'un mot est anagramme de lui-même).