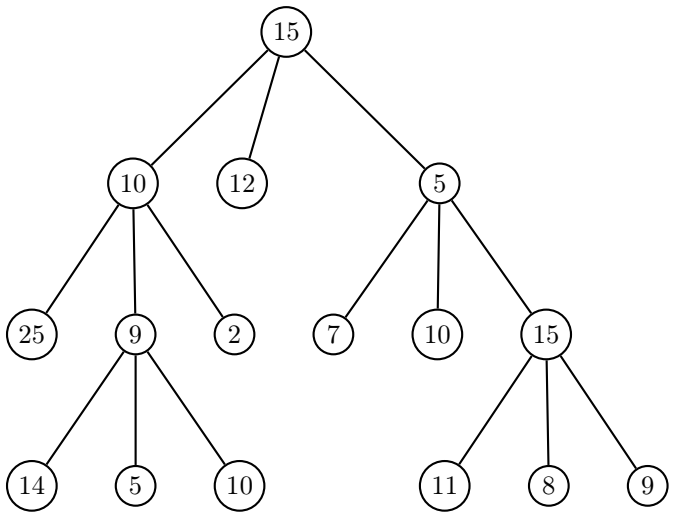
Feuille d'exercices n°8 de l'option d'informatique.

1 arbres

On étudie ici une méthode de parcours d'arbre. On considère des arbres **ternaires** dont tous les sommets (y compris les feuilles) contiennent une information.
On appelle profondeur d'un sommet sa distance (en nombre d'arêtes) à la racine : ainsi la racine est à la profondeur 0, ses fils à la profondeur 1, ... On se propose d'afficher les informations dans l'ordre suivant : les sommets sont rangés par profondeur croissante et de gauche à droite à profondeur égale.

Exemple :



On obtient la suite : 15, 10, 12, 5, 25, 9, 2, 7, 10, 15, 14, 5, 10, 11, 8, 9.
L'algorithme se programme au moyen d'une queue (ou file d'attente) selon le principe «premier entré, premier sorti», dont on peut décrire l'évolution par le schéma :

entrée ⟶						15	⟶ sortie
				5	12	10	
		2	9	25	5	12	
			2	9	25	5	
	15	10	7	2	9	25	

etc ...

Pour passer d'une ligne à l'autre, on enlève le nœud au niveau de la sortie et on ajoute ses fils dans la file d'attente. Cette file pourra être représentée soit par une liste référencée soit par un triplé formé par :

- un tableau de longueur fixe
- un indicateur de la position courante d'entrée
- un indicateur de la position courante de sortie

1. Donner un type de donnée permettant de représenter les arbres ternaires,
Donner un type de donnée permettant de représenter les files d'attente.

2. Écrire une fonction permettant de faire rentrer un élément dans la file et une fonction permettant de retirer l'élément prêt à sortir.

3. Écrire une fonction qui étant donné un arbre ternaire imprime la liste des informations dans l'ordre hiérarchique.

2 automates

On étudie ici une façon de représenter les automates ainsi que des automates simples permettant de reconnaître si un mot est dans un langage ou non.

On choisit de représenter les états par un entier :

```
type état == int;;
type ÉtatsListe == état list;;
```

On prend comme alphabet l'ensemble des caractères :

```
type caractère == char;;
```

Un automate est caractérisé par la liste de ses états finaux (de type `ÉtatsListe`) et la liste de ses transitions qui sera une liste de la forme :

```
type transition == (état*caractère)*état;;
type TransitionListe == transition list;;
```

Par convention, on prendra -1 comme rebut, 0 comme état initial.

La fonction de transition doit donner, à partir d'un état, d'un caractère et de la liste des transitions, l'état suivant.

4. Écrire une fonction `liste_de_mot` de type `string -> char list = <fun>` qui transforme une chaîne de caractères en liste de ses caractères.

5. Écrire la fonction `f_transition` de transition de type `état*caractère -> automate -> état = <fun>` qui, à partir d'un état, d'un caractère et de la liste des transitions, renvoie l'état suivant (éventuellement le rebut).

6. Écrire une fonction `vérif` de type :

```
ÉtatsListe -> TransitionListe -> string -> bool = <fun>
```

qui à partir d'un mot et d'un automate (donné par la liste de ses états finaux et la liste de ses transitions) reconnaissant un langage L retourne `true` si le mot appartient au langage L et `false` sinon.

7. Définir un automate reconnaissant les mots de la forme "abab...ab" qui commencent par `'a'` et se terminent par `'b'` et qui sont constitués d'une alternance de `'a'` et de `'b'`. Le tester sur les mots "*abababab*", "*ababba*", "*abababa*", "*abc*".

8. Définir un automate qui reconnaît les mots de la forme "$abb \ldots ba$" qui commencent et se terminent avec un `'a'` et qui ne contient que des `'b'` entre les deux `'a'` (éventuellement 0). Le tester sur les mots "*abbbbbbb*", "*aa*", "*abbbaba*", "*aca*".

9. Définir un automate qui reconnaît les mots contenant la chaîne de caractères "*abba*".