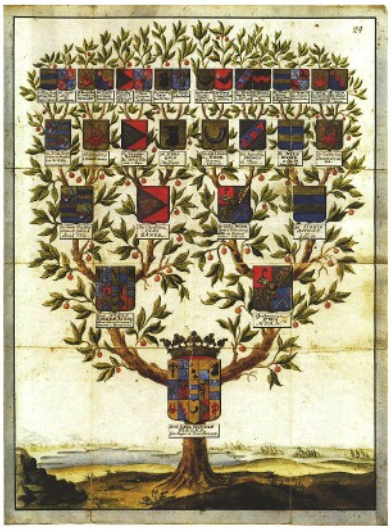
CH6 LES ARBRES

I. GENERALITES SUR LES ARBRES



Un **arbre** est une structure de données constituée de **nœuds**, qui peuvent avoir des **enfants** (qui sont eux-mêmes des nœuds). Cette structure est hiérarchisée et le sommet de l'arbre est appelé **racine**.

Un **nœud** est une position dans un arbre. À chaque nœud correspond un sous-arbre. Chaque nœud est défini par son **étiquette**.

Une **feuille** est un nœud qui n'a pas d'enfant.

Le sommet d'un arbre est appelé la **racine**. C'est le seul nœud qui n'a pas de parents.

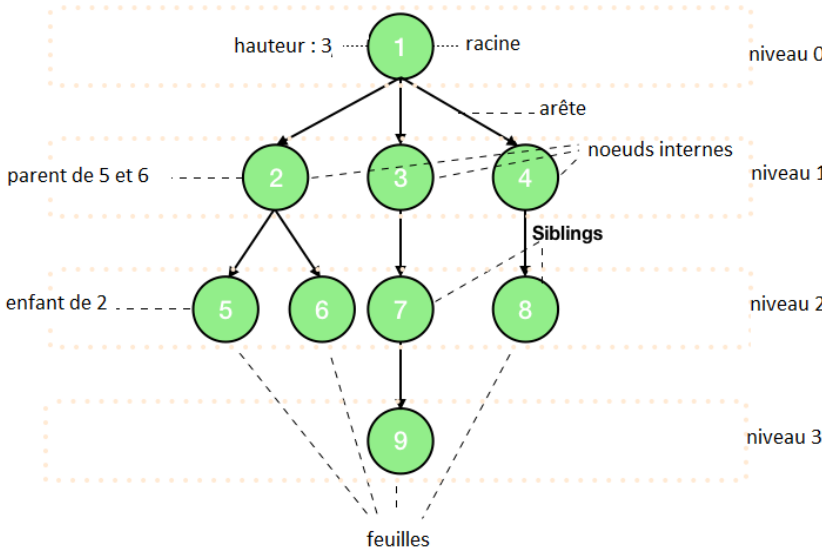
Un **nœud interne** est un nœud autre que la racine et qui n'est pas une feuille.

Une **branche** est une suite finie de nœuds consécutifs de la racine vers une feuille.

La **taille** d'un arbre est le nombre de ses nœuds.

La **profondeur** d'un nœud est la longueur du chemin qui le relie à la racine.

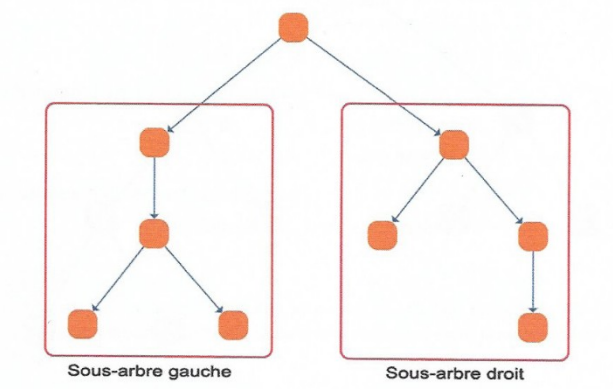
La **hauteur** d'un arbre est la profond de son nœud le plus profond (c'est donc nécessairement une feuille).



II. ARBRES BINAIRES

Un **arbre binaire** correspond à un des cas suivants :

- Un arbre vide (il ne contient aucun nœud).
- Un arbre non vide, dont la structure est la suivante :
  - Un des nœuds est la racine de l'arbre
  - Les nœuds restants sont séparés en deux sous-ensembles, qui forment récursivement deux sous-arbres (binaires) appelés respectivement sous-arbre gauche et sous-arbre droit.
  - La racine est reliée à la racine de ces deux sous-arbres gauche et droit.



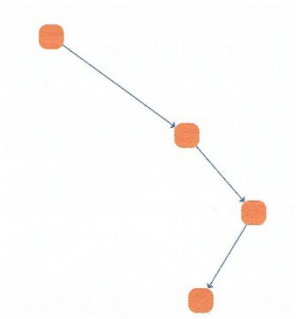
Il existe des cas particuliers d'arbres binaires :

- **Arbre binaire dégénéré (ou filiforme)** : c'est un arbre dont les nœuds possèdent au maximum un enfant.

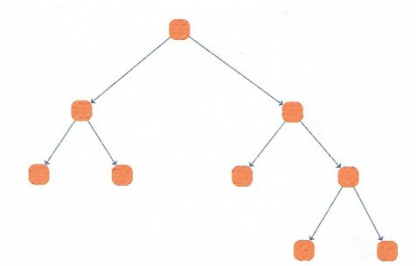
- **Arbre localement complet** : c'est un arbre binaire dont chacun des nœuds possède soit deux, soit aucun enfant.

- **Arbre complet** : c'est arbre localement complet et dont toutes les feuilles sont au niveau hiérarchique le plus bas
-

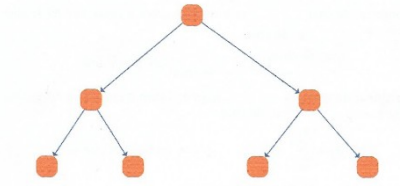
Arbre dégénéré



Arbre localement complet



Arbre complet



Les définitions données sur les arbres (racine, enfant, feuille, taille, hauteur …) sont bien sur valables pour les arbres binaires. Les différents types de parcours sont également les mêmes.

III. PARCOURIR UN ARBRE

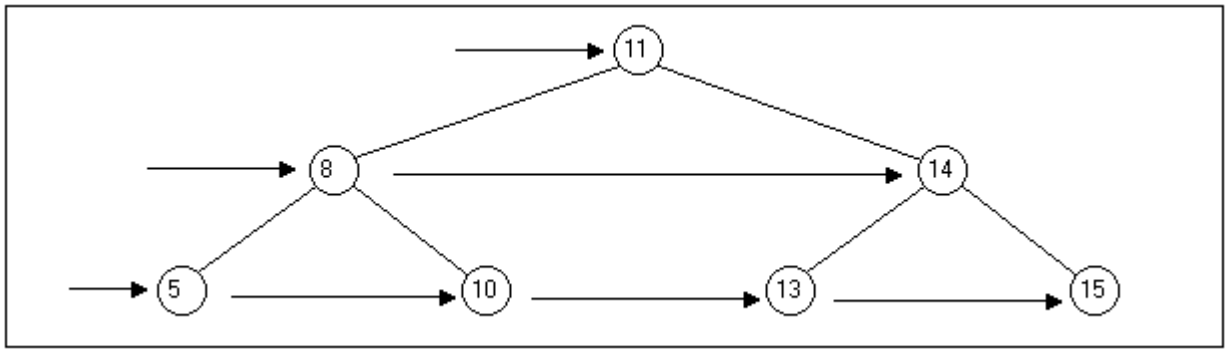
Parcourir un arbre c'est partir d'un nœud et visiter tous les nœuds de l'arbre une seule fois. Ce concept de parcours est très important en algorithmique.
Les parcours permettent notamment de générer une autre représentation de l'arbre (par exemple une liste) ou d'effectuer une recherche dans une structure arborescente.
Il existe différentes façons de parcourir un arbre.

III.1. Parcours en largeur (BFS : Breadth First Search)

Dans un parcours en largeur, les nœuds sont parcourus sur un même niveau, et par profondeur croissante. Avec l'exemple donné ci-dessous, les étiquettes seront traitées dans l'ordre suivant :

11 > 8 > 14 > 5 > 10 > 13 > 15

Plus précisément, on commence par explorer un nœud source, puis ses enfants, puis les enfants non explorés des enfants, etc.



ALGORITHME

- Choisir un nœud de départ et le marquer comme visité.
- Créer une file vide et y ajouter le nœud de départ.
- Tant que la file n'est pas vide :
  1. Extraire le premier élément de la file.
  2. Pour chaque nœud adjacent au nœud extrait, si le nœud n'a pas encore été visité, le marquer comme visité et l'ajouter à la file.

Lorsque l'algorithme est terminé, tous les nœuds ont été visités.

CODE PYTHON

```
 1  def bfs(graph, start):
 2      """Parcours en largeur d'abord d'un graphe
 3      graph : graphe
 4      start : sommet de départ
 5      visited : liste contenant les nœuds visités dans l'ordre du parcours.
 6      """
 7      visited = [start]
 8      file = File()
 9      file.enfiler(start)
10      while not file.est_vide():
11          nd = file.defiler()
12          for voisin in list((graph.voisins(nd))):
```

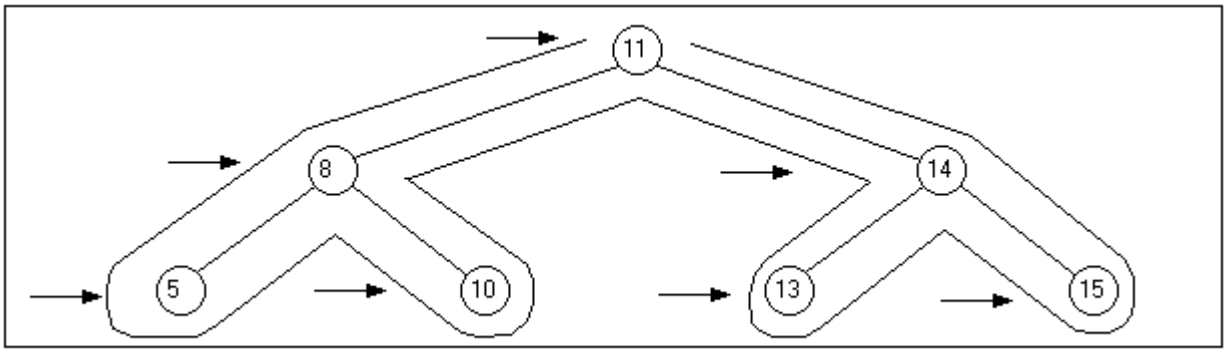
```
            if voisin not in visited:
                file.enfiler(voisin)
                visited.append(voisin)
    return visited
```

III.2. Parcours en profondeur (DFS : Depth First Search)

Ce type de parcours est souvent désigné par le terme anglais *depth-first search* (DFS). Il consiste à parcourir le graphe en partant d'un sommet donné, en suivant un chemin le plus profond possible, puis en revenant en arrière pour suivre un autre chemin. On peut imaginer que l'on se déplace dans le graphe en suivant un chemin qui ressemble à un serpent qui se déplace dans un labyrinthe. Le serpent se déplace le plus profondément possible dans le labyrinthe, puis revient en arrière pour suivre un autre chemin. On peut aussi imaginer que l'on se déplace dans le graphe en suivant un chemin qui ressemble à un arbre. On commence par la racine, puis on descend le plus profondément possible dans l'arbre, puis on remonte en arrière pour suivre un autre chemin.

Avec l'exemple donné ci-dessous, les étiquettes seront traitées dans l'ordre suivant :

11 > 8 > 5 > 10 > 14 > 13 > 15



ALGORITHME

1. Choisir un nœud de départ et le marquer comme visité.
2. Explorer tous les voisins du nœud en cours qui n'ont pas encore été visités, et choisir l'un d'eux.
3. Marquer le nœud choisi comme visité. Il devient le nœud en cours.
4. Répéter l'étape 2 et 3 jusqu'à ce que tous les voisins du nœud en cours aient été visités.
5. Si tous les voisins du nœud en cours ont été visités, revenir en arrière au nœud précédent et explorer ses voisins non visités.
6. Répéter les étapes 2 à 5 jusqu'à ce que tous les nœuds aient été visités.

CODE PYTHON

```
1   1def dfs(graph, start):
2       """Parcours en profondeur d'abord d'un graphe à partir d'un nœud donné.
3       graph: Le graphe à parcourir.
4       start: Le nœud de départ.
5       visited : liste contenant les nœuds visités dans l'ordre du parcours.
6       """
7       visited = []
```

```
 8      pile = Pile()
 9      pile.empliler(start)
10      while not pile.est_vide():
11          nd = pile.depiler()
12          if nd not in visited:
13              visited.append(nd)
14              pile.empiler(graph.voisins(nd)))
15      return visited
```

III.3. Parcours en profondeur : Approche récursive

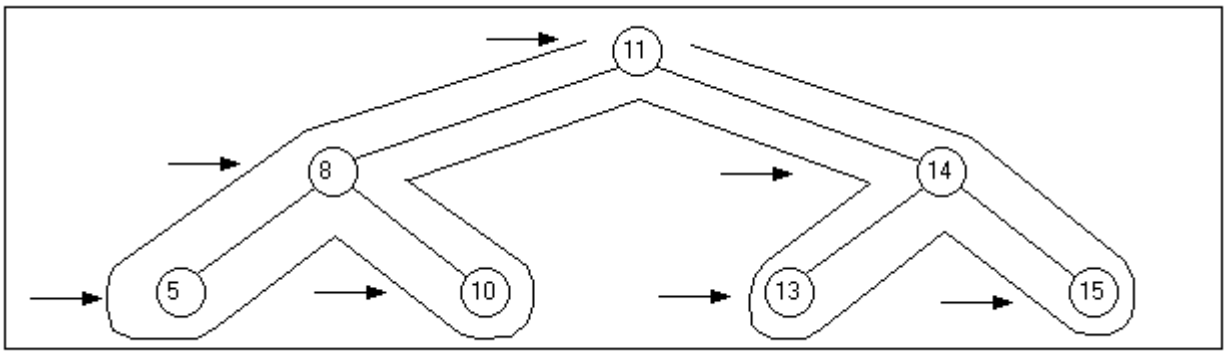
Préfixe

Un parcours en profondeur préfixe est un parcours DFS de type récursif où l'on effectue le traitement de chaque nœud avant d'explorer le sous-arbre correspondant. On retrouve l'approche DFS « classique » précédente.

Avec l'exemple donné ci-avant, les étiquettes seront bien visitées dans l'ordre suivant :

11 > 8 > 5 > 10 > 14 > 13 > 15

```
ParcoursPréfixe(arbre A):
    Afficher la racine de A
    Pour tous les enfants e de la racine :
        ParcoursPréfixe( Sous-Arbre issu de e )
```

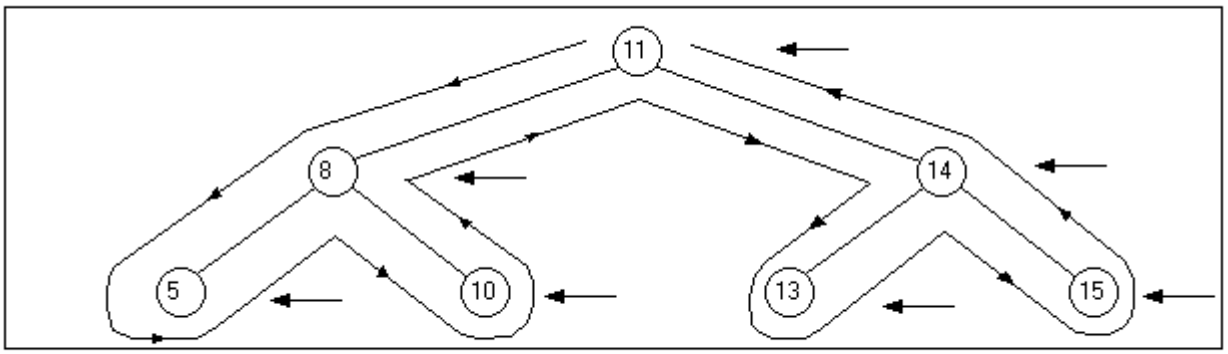


Postfixe

- Un parcours en profondeur postfixe est un parcours où l'on effectue le traitement de chaque nœud après avoir exploré le sous-arbre correspondant. Avec le même exemple donné ci-dessous, les étiquettes seront visitées dans l'ordre suivant :

5 > 10 > 8 > 13 > 15 > 14 > 11

```
ParcoursPostfixe(arbre A):
    Pour tous les enfants e de la racine :
        ParcoursPostfixe( Sous-Arbre issu e )
    Afficher la racine de A
```



Infixe

- Un parcours en profondeur infixe est un parcours où l'on effectue le traitement du sous-arbre gauche, puis la racine puis le traitement du sous-arbre droit.

Avec l'exemple précédent on obtient les étiquettes dans l'ordre suivant :

5 > 8 > 10 > 11 > 13 > 14 > 15

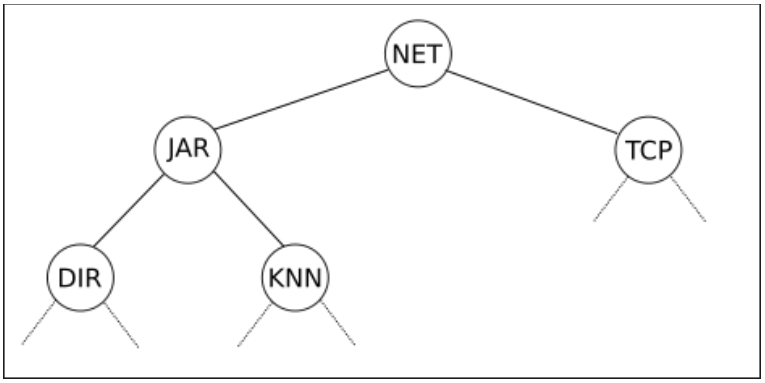
```
ParcoursInfixe(arbre A):
      Pour tous les enfants e de la racine
            ParcoursIInfixe(Sous-arbre gauche issu de e)
            Afficher la racine de A
             ParcoursInfixe(sous-arbre droit issu de e)
```

IV. ARBRE BINAIRE DE RECHERCHE

Imaginons une bibliothèque contenant un extrêmement grand nombre de livres. Cette bibliothèque est organisée de la manière suivante :

- Il y a 17 576 pièces différentes.
- Chaque pièce est repérée par une suite de trois lettres, et dans cette pièce sont rangés tous les livres dont les titres commencent par ces trois lettres.
- Chaque pièce possède deux sorties, une à droite et une à gauche.
- La sortie de gauche mène **toujours** soit à une salle dont les trois lettres sont situées avant dans l'ordre alphabétique, soit nulle part.
- La sortie de droite mène **toujours** soit à une salle dont les trois lettres sont situées après dans l'ordre alphabétique, soit nulle part.

Une représentation de cette bibliothèque peut être donnée sous la forme d'un arbre binaire tel que le suivant :



Cette répartition, pour peu qu'elle soit correctement faite (c'est-à-dire que le choix des lettres soit pertinent), peut être incroyablement efficace. Dans le meilleur des cas, il ne faudra traverser qu'**au maximum 15 salles** pour trouver n'importe quel livre. En effet, si la bibliothèque est correctement organisée, quasiment chaque nœud aura 2 sorties, d'où un arbre (presque) parfaitement équilibré. La hauteur sera donc d'environ soit environ $\log_2(17576)$ soit environ 15.

Cette structure sera particulièrement utile pour effectuer des recherches : on l'appelle ainsi un **arbre binaire de recherche ABR** (ou **BST**, *Binary Search Tree* en anglais).

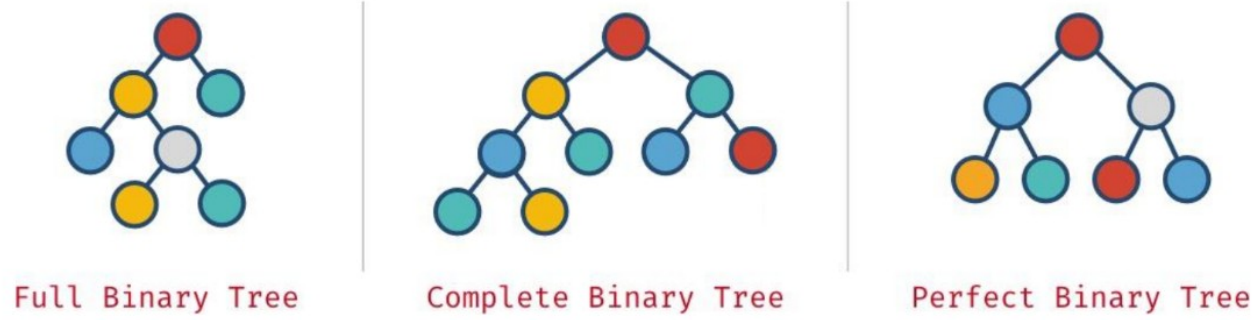
A retenir :

Un **arbre binaire de recherche (ABR)** est un arbre binaire dont les nœuds contiennent des valeurs qui peuvent être comparées entre elles, et tel que, pour tout nœud de l'arbre, toutes les valeurs situées dans le sous-arbre gauche (resp. droit) sont plus petites (resp. plus grandes) que la valeur située dans le nœud.

On emploie également le terme **clé** à la place de valeur.

On distingue :

- les arbres binaires **pleins** : chaque nœud a soit 0 soit 2 enfants
- les arbres binaires **complets** : tous les niveaux sont "remplis" sauf éventuellement le dernier qui est rempli à partir de la gauche
- les arbres binaires **parfaits** : tous les niveaux sans exception sont "remplis"



Remarque :

- *Un **ABR** étant un arbre binaire, le vocabulaire des arbres binaires est valable.*

- *Le parcours en profondeur infixe d'un ABR parcourt les nœuds de l'arbre dans l'ordre croissant de leurs valeurs.*

Efficacite :

Lorsque les éléments sont répartis à peu près équitablement entre les sous-arbres, la recherche élimine environ la moitié des éléments à chaque étape. C'est le même principe que la recherche dichotomique dans un tableau **trié**.

De manière générale, le nombre d'étape ne peut pas dépasser la hauteur de l'arbre. Il est donc intéressant de construire un ABR de hauteur minimale.

Equilibre :

Comme on l'a dit plus haut, le coût de la recherche et de l'ajout dans un ABR dépend de sa structure. Il dépend de la hauteur de l'arbre. Dans le pire des cas, l'arbre est complètement linéaire et coût est alors proportionnel au nombre d'éléments : on est ramené à une recherche dans une liste triée, ce qui a peu d'intérêt !

Il est possible (mais compliqué et hors programme), de s'assurer, lors de la construction de l'arbre, que la hauteur ne sera pas trop grande. On réorganise pour cela les données au fur et à mesure, l'idée étant, pour chaque sous-arbre, de mettre environ la moitié des données dans chaque sous-arbre gauche et droit. On obtient alors une hauteur logarithmique, c'est-à-dire qu'il existe une constante C telle que $h \leq C \log_2(N)$, où h est la hauteur de l'arbre et N le nombre de données. On parle alors d'**arbre équilibré**.

Les opérations de recherche et d'ajout ont dans ce cas également une complexité logarithmique, ce qui les rend très efficaces.