Attendus

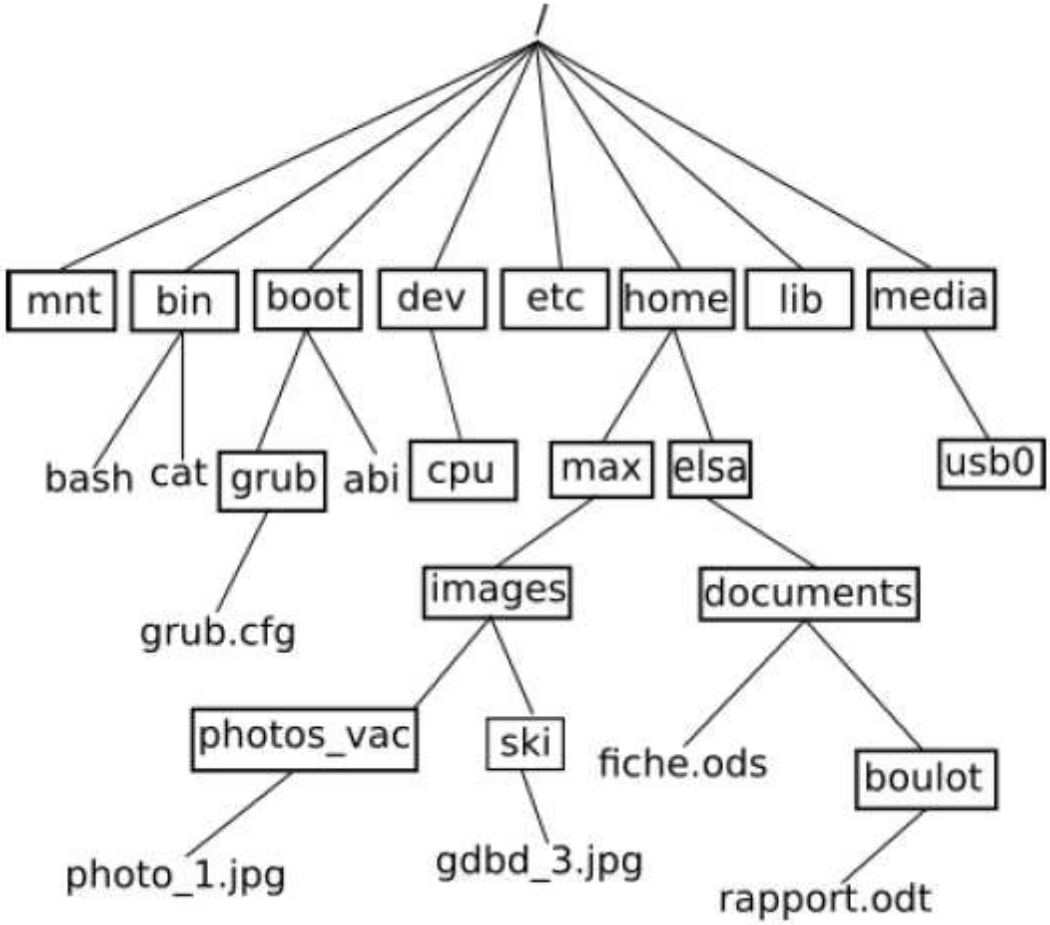
- Identifier des situations nécessitant une structure de données arborescente.
- Évaluer quelques mesures des arbres binaires (taille, encadrement de la hauteur, etc.).
- Calculer la taille et la hauteur d'un arbre.
- Parcourir un arbre de différentes façons (ordres infixe, préfixe ou suffixe ; ordre en largeur d'abord).
- Rechercher une clé dans un arbre de recherche, insérer une clé.

I Qu'est-ce qu'un arbre ?

1) Notion d'arbre

Les **structures de données** que nous avons étudiées jusqu'à présent (tableaux, listes, piles, files ...) sont **linéaires**, dans la mesure où elles stockent les éléments les uns à la suite des autres, « à la queue leu leu ».

Un **arbre** est une structure constituée de **noeuds**, qui peuvent avoir des enfants qui sont eux-mêmes des noeuds. Les systèmes de fichiers dans les systèmes de type UNIX (Linux et Mac OS) ont par exemple une structure en arbre définissant une arborescence :



Les arbres sont des structures mathématiques abstraites très utilisées en informatique. C'est le cas notamment lorsque l'on a besoin d'une structure **hiérarchisée** des données : dans l'exemple ci-dessus le fichier `grub.cfg` ne se trouve pas au même niveau de l'arborescence que le fichier `rapport.odt` (le fichier `grub.cfg` se trouve « plus proche » de la racine `/` que le fichier `rapport.odt`). On ne pourrait pas avec une simple liste qui contiendrait les noms des fichiers et des répertoires, rendre compte de cette hiérarchie (plus ou moins « proche » de la racine).

2) Notion d'arbre binaire

Les arbres binaires sont des cas **particuliers** d'arbres : dans un arbre binaire, on a au maximum **2 branches** qui partent d'un élément. Dans la suite nous travaillerons *uniquement* sur les arbres binaires.

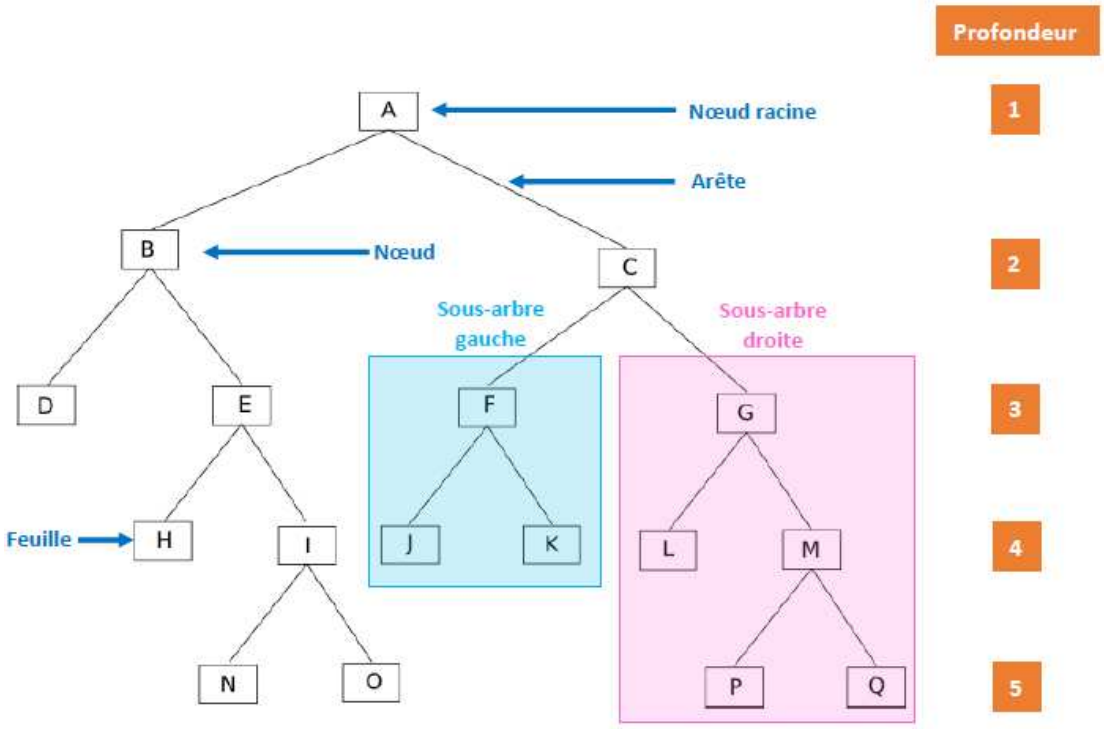
Un peu de vocabulaire

En informatique, un *arbre binaire* est une *structure de données* qui peut se représenter sous la forme d'une *hiérarchie* dont chaque élément est appelé *noeud*, le *noeud initial* étant appelé *racine*. Dans un arbre binaire, chaque élément possède au plus *deux éléments fils* au *niveau inférieur*, habituellement appelés *gauche* et *droit*. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé *père*.

Au niveau le plus élevé il y a donc un noeud racine. Au niveau directement inférieur, il y a au plus deux noeuds fils. En continuant à descendre aux niveaux inférieurs, on peut en avoir quatre, puis huit, seize, etc. c'est-à-dire la suite des puissances de deux. Un noeud n'ayant aucun fils est appelé *feuille*. Le nombre de niveaux total, autrement dit la distance entre la feuille la plus éloignée et la racine, est appelé *hauteur* de l'arbre.

Le niveau d'un noeud est appelé *profondeur*.

D'après WIKIPEDIA



Quelques précisions concernant l'arbre binaire précédent :

- Chaque élément de l'arbre est appelé noeud (par exemple : A, B, C, D, ..., P et Q sont des noeuds)
- Le noeud initial (A) est appelé **noeud racine** ou plus simplement **racine**.
- On appelle **arête** le segment qui relie 2 noeuds.
- Le noeud E et le noeud D sont les **fils** du noeud B. Réciproquement, on dira que le noeud B est le **père** des noeuds E et D.
- Dans un arbre binaire, un noeud possède au plus **2** fils.
- Un noeud n'ayant aucun fils est appelé **feuille** (exemples : D, H, N, O, J, K, L, P et Q sont des feuilles).
- À partir d'un noeud (qui n'est pas une feuille), on peut définir un sous-arbre gauche et un sous-arbre droit (exemple : à partir de C on va trouver un sous-arbre gauche composé des noeuds F, J et K et un sous-arbre droit composé des noeuds G, L, M, P et Q).

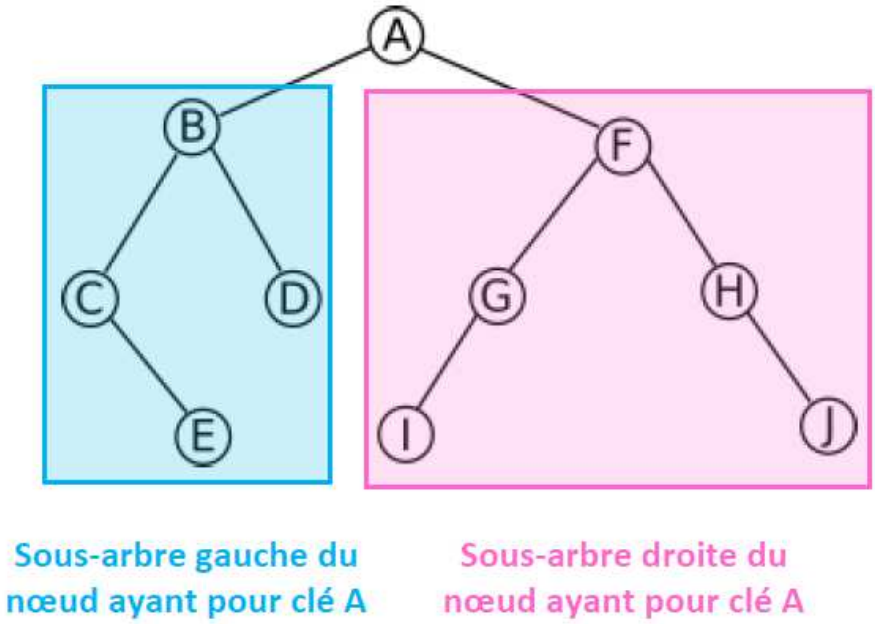
- La **profondeur** d'un noeud ou d'une feuille dans un arbre binaire est le nombre de noeuds du chemin qui va de la racine à ce noeud. La racine d'un arbre est à une profondeur 1, et la profondeur d'un noeud est égale à la profondeur de son prédécesseur plus 1. Si un noeud est à une profondeur p, tous ses successeurs sont à une profondeur $p+1$.
  Exemples : profondeur de B = 2 ; profondeur de I = 4 ; profondeur de P = 5.
- La **hauteur** d'un arbre est la profondeur **maximale** des noeuds de l'arbre.
  Exemple : la profondeur de P = 5, c'est un des noeuds les plus profond, donc la hauteur de l'arbre est de 5.

Il est important de remarquer que l'on peut aussi voir les arbres comme des structures **récursives** : les fils d'un noeud sont des arbres (sous-arbre gauche et un sous-arbre droite dans le cas d'un arbre binaire), ces arbres sont eux-mêmes constitués d'arbres...

Remarque : Python ne propose pas de façon native l'implémentation des arbres binaires. Il faudra pour les modéliser faire appel à la programmation orientée objet (P.O.O.) comme nous allons le voir plus tard.

3) Arbre binaire et clés

À chaque noeud d'un arbre binaire, on peut associer une **clé** correspondant au nom du noeud et servant d'étiquette.



En partant du noeud racine dont la clé est A on peut distinguer :

- le sous-arbre gauche composé du noeud ayant pour clé B, du noeud ayant pour clé C, du noeud ayant pour clé D et du noeud ayant pour clé E ;
- le sous-arbre droit est composé du noeud ayant pour clé F, du noeud ayant pour clé G, du noeud ayant pour clé H, du noeud ayant pour clé I et du noeud ayant pour clé J.

Un **arbre** (ou un **sous-arbre**) **vide** est noté `NIL` (abréviation du latin `nihil` qui veut dire rien).
Si on prend le noeud ayant pour clé G on a :

- le sous-arbre gauche est uniquement composé du noeud ayant pour clé I
- le sous-arbre droit est vide (`NIL`)

Il faut bien avoir à l'esprit qu'un sous-arbre (gauche ou droite) est un arbre (même s'il contient un seul noeud ou pas de noeud du tout (`NIL`)).

4) Choix d'interface pour le TAD arbre binaire

Le TAD *arbre binaire* dont le type sera noté `arbre_binaire` peut être défini, en vue du paragraphe précédent, sous forme récursive avec trois attributs :

- `racine` : de type `E` (qui est une clé)
- `gauche` : de type `arbre_binaire`
- `droit` : de type `arbre_binaire`

Nous pouvons définir alors les méthodes suivantes :

- **Constructeur :**
  `créerArbre` : `E` → `arbre_binaire` (les sous-arbres gauche et droit sont vides)
- **Sélecteurs :**
  `get_cle` : `arbre_binaire` → `E` (pour obtenir la racine de l'arbre)
  `get_gauche` : `arbre_binaire` → `arbre_binaire` (pour obtenir le sous-arbre gauche)
  `get_droite` : `arbre_binaire` → `arbre_binaire` (pour obtenir le sous-arbre droit)
- **Mutateurs :**
  `insert_gauche` : `arbre_binaire` × `E` → `arbre_binaire` (pour insérer une clé fils gauche)
  `insert_droite` : `arbre_binaire` × `E` → `arbre_binaire` (pour insérer une clé fils droite)
- **Prédicat :**
  `estVide` : `arbre_binaire` → `booléen`

II Hauteur et taille d'un arbre binaire

1) Notations utilisées dans les algorithmes

Soit un arbre `T` :

- `T.racine` correspond au noeud racine de l'arbre `T`
- `T.gauche` correspond au sous-arbre gauche de l'arbre `T`
- `T.droit` correspond au sous-arbre droit de l'arbre `T`

Il faut noter que si `T` est une feuille, `T.gauche` et `T.droit` sont des arbres vides (`NIL`).

2) Calculer la hauteur d'un arbre

- **Principe :**
  - un arbre vide est de hauteur 0 ;
  - un arbre non vide a pour hauteur 1 + la hauteur maximale entre ses fils.
- **Algorithme en pseudo-code :**

```
DEBUT
  HAUTEUR(T) :
    Si T ≠ NIL :
      renvoyer 1 + max(HAUTEUR(T.gauche), HAUTEUR(T.droit))
    Sinon :
      Renvoyer 0
    Fin Si
FIN
```

Remarque :
La fonction `max` renvoie la plus grande valeur des 2 valeurs passées en paramètre, par exemple : `max` (5,6) renvoie 6.

Question 1 : Quelle est la particularité de la fonction `HAUTEUR()` ?

Exercice 1 : Applique l'algorithme ci-dessus sur l'arbre binaire du paragraphe I.3. afin de calculer sa hauteur en détaillant ton raisonnement.

3) Calculer la taille d'un arbre (nombre de noeuds)

- **Principe :**
  - si l'arbre est vide : renvoyer 0
  - sinon renvoyer 1 plus la somme du nombre de noeuds des sous-arbres.
- **Algorithme en pseudo-code :**

```
DEBUT
  TAILLE(T) :
    Si T ≠ NIL :
      Renvoyer 1 + TAILLE(T.gauche) + TAILLE(T.droit)
    Sinon :
      Renvoyer 0
    Fin si
FIN
```

Question 2 : Quelle est la particularité de la fonction `TAILLE()` ?

Exercice 2 : En t'inspirant du paragraphe précédent, applique l'algorithme ci-dessus sur l'arbre binaire du paragraphe I.3. afin de calculer sa taille en détaillant votre raisonnement.

III Parcours en profondeur d'un arbre binaire

1) Parcourir un arbre dans l'ordre infixe

- **Principe :**
  Dans un *parcours infixe*, chaque noeud est visité après son enfant gauche mais avant son enfant droit. Le parcours infixe affiche la racine après avoir traité le sous-arbre gauche, après traitement de la racine, on traite le sous-arbre droit (c'est donc un parcours de type G R D : Gauche Racine Droite).
- **Algorithme en pseudo-code :**

```
DEBUT
  PARCOURS_INFIXE(T) :
    Si T ≠ NIL :
      PARCOURS_INFIXE(T.gauche)
      Afficher T.racine
      PARCOURS_INFIXE(T.droit)
    Fin si
FIN
```

Question 3 : Quelle est la particularité de la fonction `PARCOURS_INFIXE()` ?

Exercice 3 : Dans quel ordre est parcouru l'arbre binaire du paragraphe I.3. par la fonction `PARCOURS_INFIXE()` ?

2) Parcourir un arbre dans l'ordre préfixe

- **Principe :**
  Dans un *parcours préfixe* (preorder traversal), chaque noeud est visité avant que ses enfants soient visités. Cela signifie que l'on affiche la racine de l'arbre, on parcourt tout le sous-arbre de gauche, une fois qu'il n'y a plus de sous-arbre gauche on parcourt les éléments du sous-arbre droit. Ce type de parcours peut être résumé en trois lettres : R G D (pour Racine Gauche Droit).
- **Algorithme en pseudo-code :**

```
DEBUT
  PARCOURS_PREFIXE(T) :
    Si T ≠ NIL :
      Afficher T.racine
      PARCOURS_PREFIXE(T.gauche)
      PARCOURS_PREFIXE(T.droit)
    Fin si
FIN
```

Question 4 : Quelle est la particularité de la fonction `PARCOURS_PREFIXE()` ?

Exercice 4 : Dans quel ordre est parcouru l'arbre binaire du paragraphe I.3. par la fonction `PARCOURS_PRFIXE()` ?

3) Parcourir un arbre dans l'ordre suffixe (ou postfixe)

- **Principe :**
  Dans un *parcours suffixe* (ou postfixe), chaque noeud est visité après que ses enfants aient été visités. Le parcours postfixe effectue schématiquement le parcours suivant : sous arbre-gauche, sous-arbre droit puis la racine, c'est donc un parcours G D R (Gauche Droite Racine).
- **Algorithme en pseudo-code :**

```
DEBUT
  PARCOURS_SUFFIXE(T) :
    Si T ≠ NIL :
      PARCOURS_SUFFIXE(T.gauche)
      PARCOURS_SUFFIXE(T.droit)
      Afficher T.racine
    Fin si
FIN
```

Question 5 : Quelle est la particularité de la fonction `PARCOURS_SUFFIXE()` ?

Exercice 5 : Dans quel ordre est parcouru l'arbre binaire du paragraphe I.3. par la fonction `PARCOURS_SUFFIXE()` ?

Le choix du parcours infixe, préfixe ou suffixe dépend du problème à traiter, on pourra retenir pour les parcours préfixe et suffixe (le cas du parcours infixe sera traité un peu plus loin) que :

- dans le cas du parcours **préfixe**, un noeud est affiché **avant** d'aller visiter ses enfants
- dans le cas du parcours **suffixe**, on affiche chaque noeud **après** avoir affiché chacun de ses fils

IV Parcours en largeur d'un arbre binaire

- **Principe :**
  L'*algorithme de parcours en largeur* (ou BFS, pour Breadth First Search en anglais) permet le parcours d'un arbre de la manière suivante : on commence par explorer un noeud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. À partir d'un noeud source S, il liste d'abord les voisins de S pour ensuite les explorer un par un. Ce mode de fonctionnement utilise donc une **file** dans laquelle il prend le premier sommet et place en dernier ses voisins non encore explorés. Le principe de l'algorithme est le suivant :

1. Mettre le noeud source dans la file.
2. Retirer le noeud du début de la file pour le traiter.
3. Mettre tous les voisins non explorés dans la file (à la fin).
4. Si la file n'est pas vide reprendre à l'étape 2.

- **Algorithme en pseudo-code :**

```
DEBUT
  PARCOURS_LARGEUR(T) :
    Si T == Nil :
      Arrêt
    Fin si
    f ← vide
    enfiler(T,f)
    Tant que f non vide :
      Afficher (f.tete).racine
      T ← defiler(f)
      Si T.gauche ≠ NIL :
        enfiler(T.gauche, f)
      Fin si
      Si T.droit ≠ NIL :
        enfiler(T.droit, f)
      Fin si
    Fin tant que
FIN
```

Question 6 :
Dans quel ordre est parcouru l'arbre binaire du paragraphe I.3. par la fonction `PARCOURS_LARGEUR( )` ?

Exercice 6 : Selon toi, pourquoi parle-t-on de parcours en largeur ?

Remarque : on utilise une file (FIFO) pour cet algorithme de parcours en largeur. En outre, cet algorithme n'utilise pas de fonction récursive.

V Arbre binaire de recherche

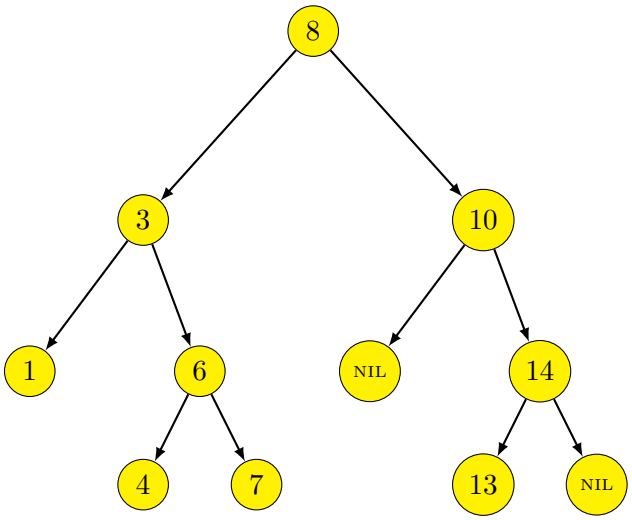
1) Notion d'arbre binaire de recherche

Les éléments stockés dans un *arbre binaire de recherche ABR* (en anglais, **B**inary **S**earch Tree ou **BST**) doivent posséder une *relation d'ordre totale*, c'est-à-dire qu'il doit exister une manière de dire si un élément est plus grand ou plus petit qu'un autre. Par exemple, on peut trier des nombres par valeur ou des personnes par ordre alphabétique de leur nom de famille.

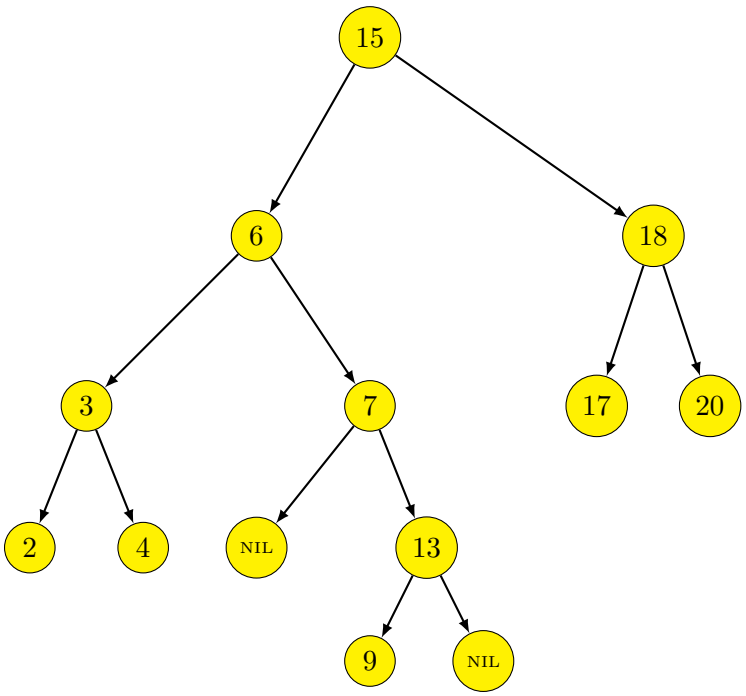
Définition
Un *arbre binaire de recherche* est un arbre binaire dont l'ensemble des noeuds vérifient les propriétés suivantes :
- Les valeurs du sous-arbre gauche sont inférieures ou égales à la valeur du noeud.
- Les valeurs du sous-arbre droit sont strictement supérieures à la valeur du noeud.

Le premier élément inséré dans l'arbre devient la racine. Ensuite, il suffit de mettre à gauche les éléments plus petits et à droite les éléments plus grands. C'est cette particularité qui rend les BST intéressants : la plupart des opérations réalisées sur les arbres binaires de recherche ont une complexité temporelle logarithmique $O(\log n)$ dans le cas moyen. Cela est dû au fait que, grâce à la relation d'ordre liant les valeurs, on peut naviguer dans l'arbre avec une logique rappelant une recherche dichotomique, ce qui est **plus performant** que les listes, par exemple, que l'on doit parcourir élément par élément.

Ainsi la séquence {8 3 10 1 6 14 4 7 13} peut-être traduite sous la forme de l'arbre binaire de recherche suivant :



Exercice 7 : Quelle est la séquence associée à l'arbre binaire de recherche ci-dessous ?



2) Recherche d'une clé dans un arbre binaire de recherche

Nous allons maintenant étudier un algorithme permettant de rechercher une clé de valeur `k` dans un arbre binaire de recherche. Si `k` est bien présent dans l'arbre binaire de recherche, l'algorithme renvoie vrai, dans le cas contraire, il renvoie faux.

- **Principe :**
  La recherche dans un arbre binaire d'un noeud ayant une clé particulière est un procédé **récursif**. On commence par examiner la racine. Si sa clé est la clé recherchée, l'algorithme se termine et renvoie la racine. Si elle est strictement inférieure, alors elle est dans le sous-arbre gauche, sur lequel on effectue alors récursivement la recherche. De même si la clé recherchée est strictement supérieure à la clé de la racine, la recherche continue dans le sous-arbre droit. Si on atteint une feuille dont la clé n'est pas celle recherchée, on sait alors que la clé recherchée n'appartient à aucun noeud, elle ne figure donc pas dans l'arbre de recherche. On peut comparer l'exploration d'un arbre binaire de recherche avec la recherche par dichotomie qui procède à peu près de la même manière sauf qu'elle accède directement à chaque élément d'un tableau au lieu de suivre des liens. La différence

entre les deux algorithmes est que, dans la recherche dichotomique, on suppose avoir un critère de découpage de l'espace en deux parties que l'on n'a pas dans la recherche dans un arbre.

- **Algorithme en pseudo-code :**

```
DEBUT
  ARBRE_RECHERCHE(T,k) :
    Si T == NIL :
      Renvoyer Faux
    Fin si
    Si k == T.racine :
      Renvoyer Vrai
    Fin si
    Si k < T.racine :
      ARBRE_RECHERCHE(T.gauche,k)
    Sinon :
      ARBRE_RECHERCHE(T.droit,k)
    Fin si
FIN
```

Exercice 8 : Quelle est la particularité de la fonction `ARBRE_RECHERCHE()` ?

Exercice 9 : Applique l'algorithme de recherche d'une clé dans un arbre binaire de recherche sur l'arbre binaire de recherche de l'exercice 7. On prendra `k = 13`.

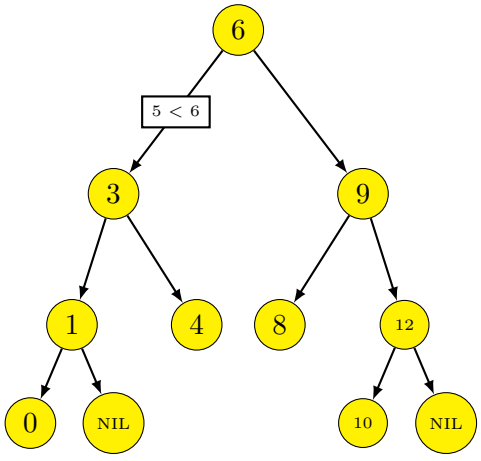
Exercice 10 : Applique l'algorithme de recherche d'une clé dans un arbre binaire de recherche sur l'arbre binaire de recherche de l'exercice 7. On prendra `k = 16`.

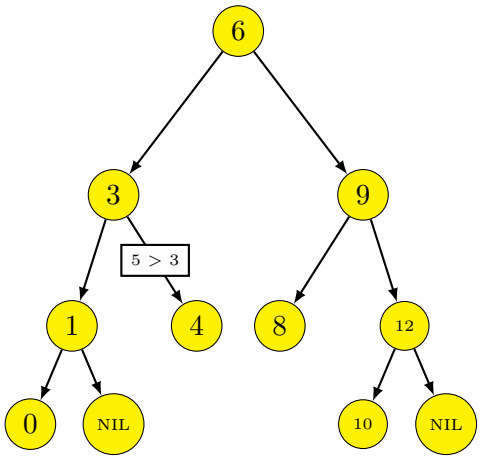
3) Insertion d'une clé dans un arbre binaire de recherche

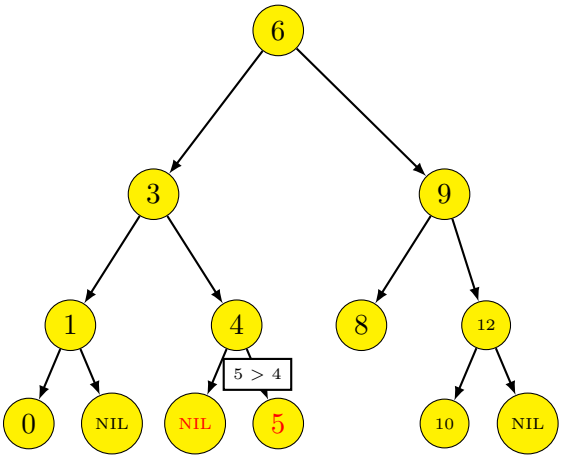
Il est tout à fait possible d'insérer un noeud « y » dans un arbre binaire de recherche (non vide).

- **Principe :**
  L'insertion d'un noeud commence par une recherche : on cherche la clé du noeud à insérer ; lorsqu'on arrive à une feuille, on ajoute le noeud comme fils de la feuille en comparant sa clé à celle de la feuille : si elle est inférieure, le nouveau noeud sera à gauche ; sinon il sera à droite.

  Exemple : insertion de de la valeur 5 dans l'arbre de recherche binaire suivant







- **Algorithme en pseudo-code :**

```
DEBUT
  ARBRE_INSERTION(T,y) :
    A ← T
    Tant que A ≠ NIL :
      x ← A
      Si y < A.racine :
        A ← A.gauche
      Sinon :
        A ← A.droit
      Fin si
    Fin tant que
    Si y < x.racine :
      x.insertgauche(y)
    Sinon :
      x.insertdroite(y)
    Fin si
    Renvoyer T
FIN
```

Exercice 11 : Applique l'algorithme d'insertion d'un noeud `y` dans un arbre binaire de recherche sur l'arbre ci-dessous, en prenant `y = 16`.