Les arbres

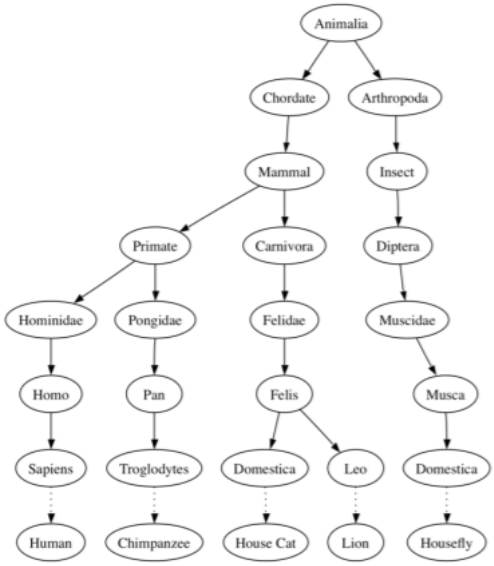
1 Les arbres

1.1 Introduction

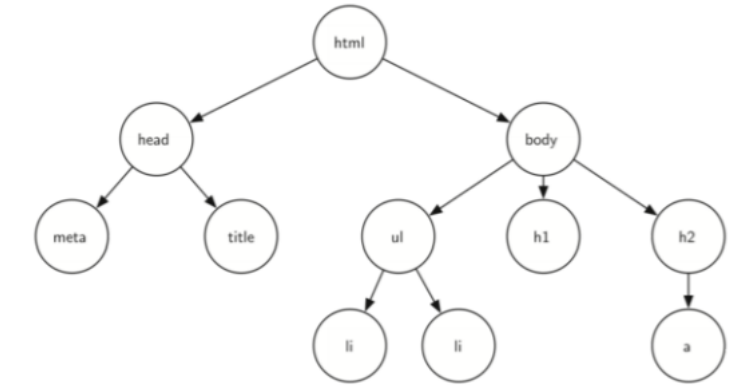
Il n'existe pas qu'une façon linéaire de représenter les données, comme les listes, les tableaux, les dictionnaires, les piles et les files. Nous pouvons également structurer les données de façon hiérarchique.

1.2 Exemples de situations où l'on rencontre des arbres

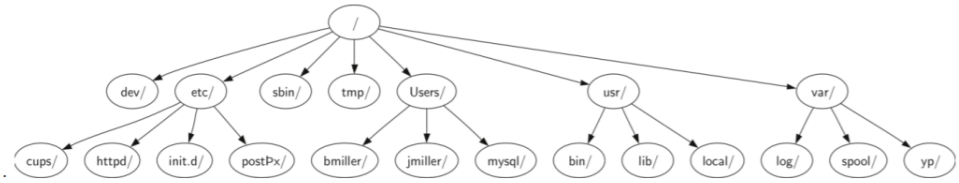
- En biologie



- Les balises d'une page web



- Les dossiers d'un ordinateur



1.3 Définition

Définition 9.1

Un **arbre** est une structure de données constituée de nœuds, qui peuvent avoir des enfants (et qui sont aussi des nœuds)

Définition 9.2

Le sommet de l'arbre est appelé **racine**.

Définition 9.3

Un nœud qui ne possède pas d'enfant est appelé une **feuille**.

Définition 9.4

Les nœuds autres que la racine et les feuilles sont appelés **nœuds internes**.

Définition 9.5

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

Remarque

Un arbre a donc autant de feuilles que de branches !

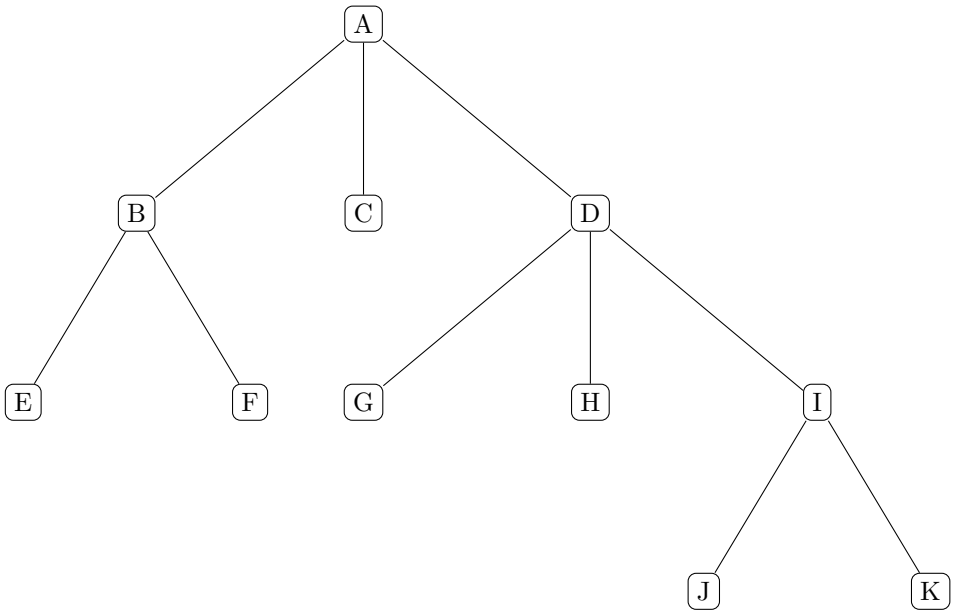
Définition 9.6

l'**arité** d'un arbre est le nombre maximal d'enfants qu'un nœud peut avoir.

Définition 9.7
La **taille** d'un arbre est le nombre de nœuds qui le compose.

Définition 9.8
La **hauteur** d'un arbre est la profondeur à laquelle il faut descendre pour trouver la feuille la plus éloignée de la racine.

Exercice 9.1
On considère l'arbre ci-dessous.



Répondre aux questions suivantes :

- Nombre de nœuds ?
- Nombre de racine ? Laquelle ?
- Nombre de feuilles ? Lesquelles ?
- Nombre de branches ?
- Nombre de nœuds internes ?
- Quel est son arité ?
- Quelle est sa taille ?
- Quelle est sa hauteur ?

Remarque
En informatique, les arbres poussent vers le bas :-)

1.4 Représentation en Python d'un arbre, appelé aussi *arborescence*

1.4.1 Avec une classe

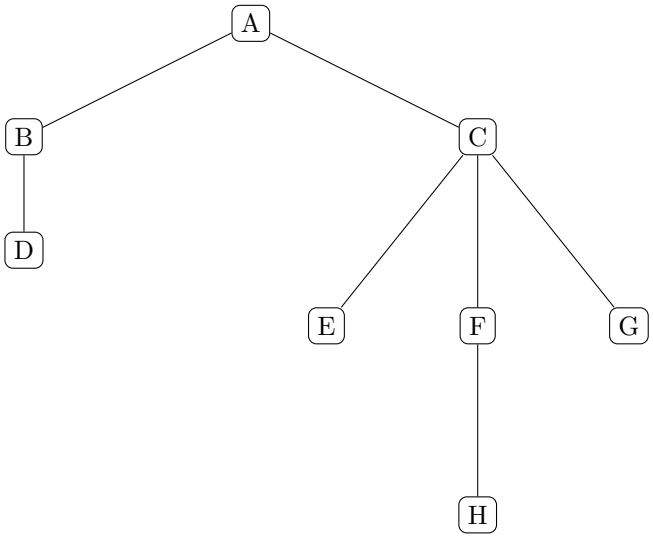
Pour représenter un arbre (une arborescence) en Python, on peut utiliser des objets, comme pour les listes chaînées.

L'objet de la classe contient deux attributs : un attribut valeur (dans lequel on stocke une valeur quelconque, appelée *étiquette* et un attribut fils dans lequel on stocke les fils sous la forme d'un tableau.

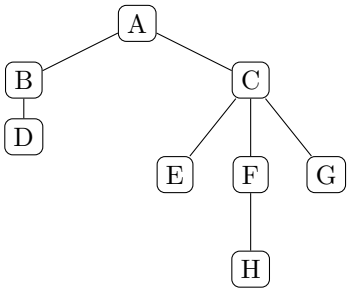
Exercice 9.2
Construire la classe Noeud. Pour les feuilles, on mettra [] pour le fils***

Exercice 9.3
Construire l'arbre donné en exemple plus haut.

Exercice 9.4
Construire l'arbre ci-dessous :



Exercice 9.5
Créer une fonction récursive **represente(arbre,p=0)** qui permet un affichage d'un arbre comme ceci :




```
A
- B
-- D
- C
-- E
-- F
--- H
-- G
```

Représentation possible d'un arbre. Les tirets indiquent la profondeur

***

1.4.2 Avec un dictionnaire

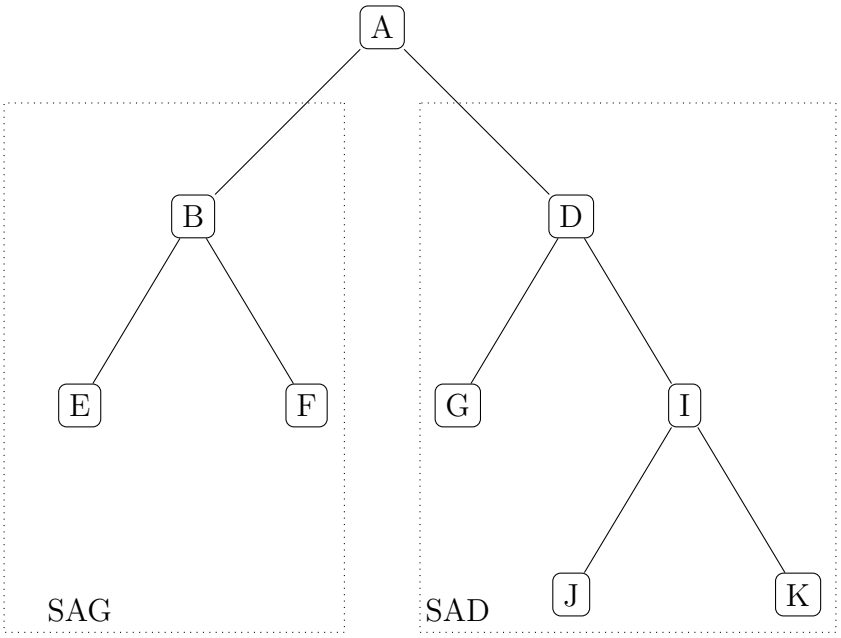
2 Les arbres binaires

2.1 Définition

Définition 9.9

Un arbre dont l'arité est 2 est un **arbre binaire**

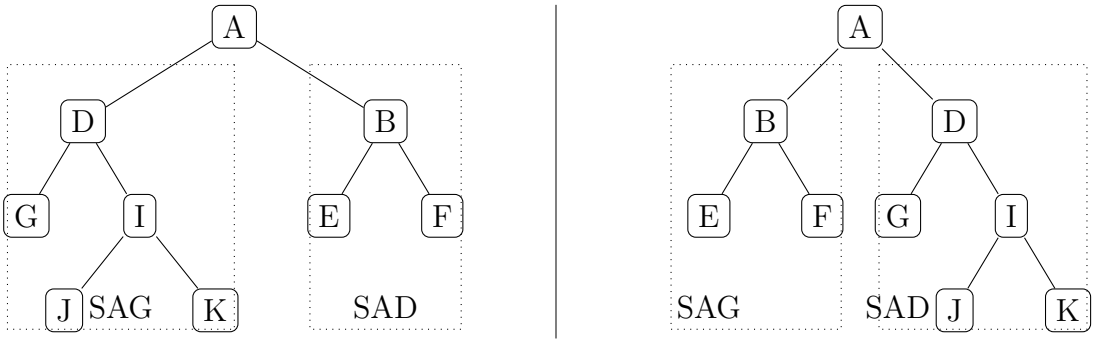
Les arbres binaires sont donc des arbres ou chaque nœud peut donner 0,1 ou 2 enfants.



On distingue généralement à partir du nœud racine 2 sous-arbres disjoints : Le sous-arbre gauche de l'arbre binaire (SAG) et le sous-arbre droit de l'arbre binaire (SAD).

Remarque

⚠ De ce fait, ces deux arbres ne sont pas identiques :



Exercice 9.6

Dessiner tous les arbres binaires possédant 3 nœuds.

Exercice 9.7
Dessiner tous les arbres binaires possédant 4 nœuds.

Exercice 9.8
Sachant qu'il y a 1 arbre binaire vide, 1 arbre binaire contenant 1 nœud, 2 arbres binaires contennant 2 nœuds, 5 arbres binaires contenant 3 nœuds et 14 arbres binaires contenant 4 nœuds, calculer le nombre d'arbres binaires contenant 5 nœuds.
⚠ On cherche seulement ici à les dénombrer.***

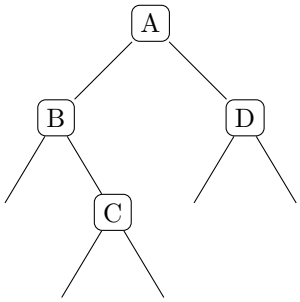
2.2 Représentation en Python d'un arbre binaire

2.2.1 Implémentation avec des classes

Pour représenter un arbre binaire en Python, on peut utiliser des objets.
L'objet de la classe contient trois attributs : un attribut valeur (dans lequel on stocke une valeur quelconque, appelée *étiquette* , un attribut fils droit et un attribut fils gauche.

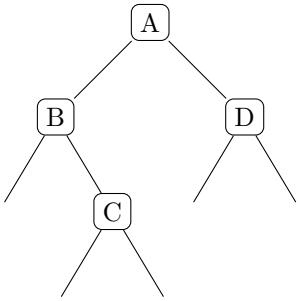
Exercice 9.9
Construire la classe Noeud afin de pouvoir construire en python des arbres binaires. Pour les feuilles, on indiquera None pour le sous arbre droit et le sous arbre gauche.

Exercice 9.10
Construire l'arbre binaire suivant :



2.2.2 Représentation

Exercice 9.11
Réaliser ce type d'affichage d'arbre binaire :




```
>>> represente(a)
  A
- B
--
-- C
---
---
- D
--
```

2.3 Cas particuliers

2.3.1 Arbre dégénéré ou filiforme

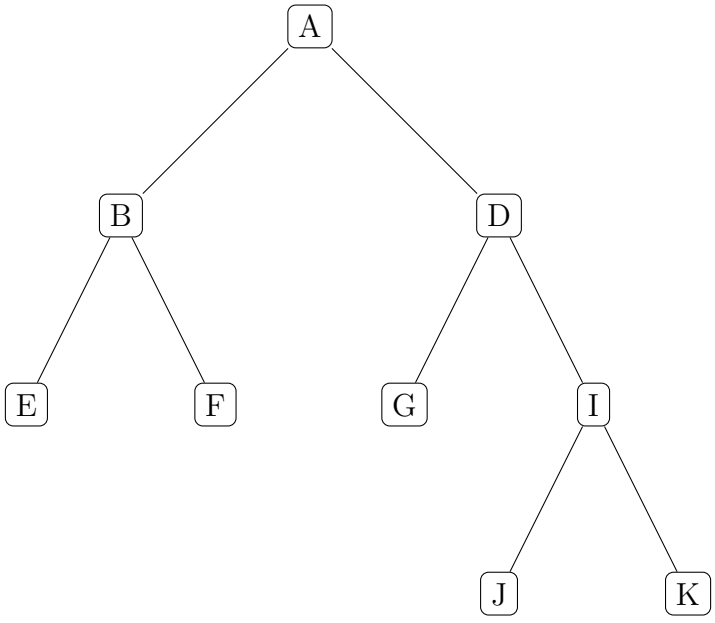
Définition 9.10

Un **arbre dégénéré** est un arbre dont les nœuds ne possèdent au plus un enfant.

2.3.2 Arbre localement complet

Définition 9.11

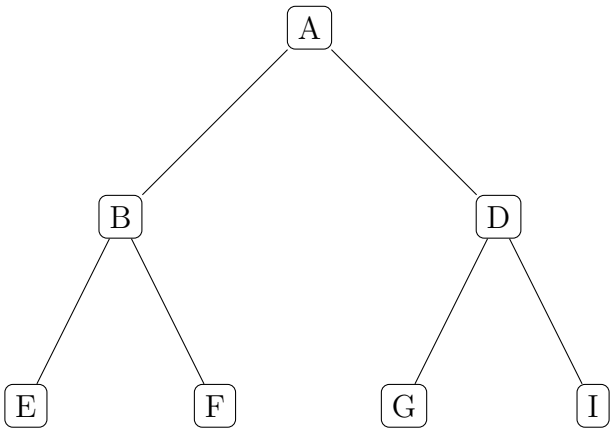
Un **arbre localement complet** est un arbre binaire dont chacun des nœuds possède soit deux enfants, soit aucun.



2.3.3 Arbre complet

Définition 9.12

C'est un arbre qui est localement complet et dont toutes les feuilles sont au niveau hiérarchique le plus bas.

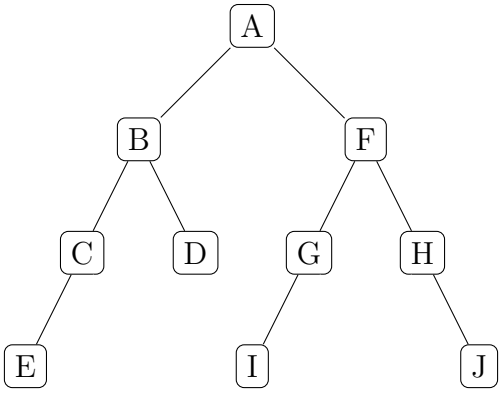


Exercice 9.12

- Combien de nœuds au maximum comporte un arbre localement complet de hauteur h ? Au minimum ?
- Combien de nœuds comporte un arbre complet de hauteur h ?

2.4 Notion de clé

À chaque nœud d'un arbre binaire, on associe une clé ("valeur" associée au nœud)



- Si on prend le nœud ayant pour clé A (le nœud racine de l'arbre) on a :
  - le sous-arbre gauche est composé du nœud ayant pour clé B, du nœud ayant pour clé C, du nœud ayant pour clé D et du nœud ayant pour clé E
  - le sous-arbre droit est composé du nœud ayant pour clé F, du nœud ayant pour clé G, du nœud ayant pour clé H, du nœud ayant pour clé I et du nœud ayant pour clé J
- si on prend le nœud ayant pour clé B on a :
  - le sous-arbre gauche est composé du nœud ayant pour clé C et du nœud ayant pour clé E
  - le sous-arbre droit est uniquement composé du nœud ayant pour clé D
- 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)

Remarque
Un arbre vide est noté NIL

Très important
Un sous-arbre (droite ou gauche) est un arbre (même s'il contient un seul nœud ou pas de nœud de tout (NIL)).

3 Algorithme des arbres binaires

Notations pour les algorithmes : Soit T un arbre :

T.racine est le nœud racine de l'arbre T

Soit un nœud x :

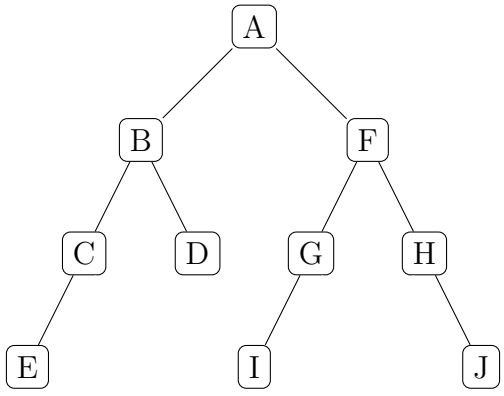
x.gauche correspond au sous-arbre gauche du nœud x

x.droit correspond au sous-arbre droit du nœud x

x.cle correspond à la clé du nœud x

3.1 Calcul de la taille d'un arbre

On considère de nouveau cet arbre :

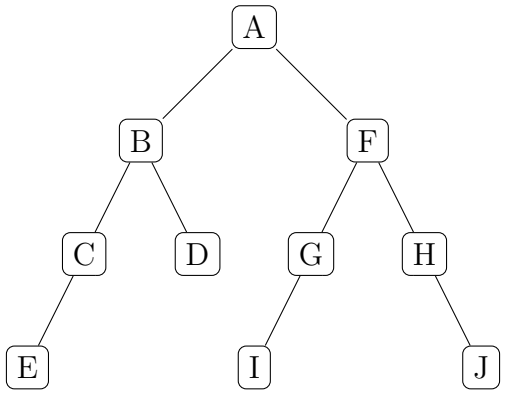


Appliquer cet algorithme à l'arbre ci-dessus.

Exercice 9.13
Créer une fonction **taille(a)** qui renvoie le nombre de noeuds de l'arbre binaire a

```
1  VARIABLE
2  T : arbre
3  x : noeud
4  DEBUT ;
5  Function TAILLE(T)
6  |  if  T ≠ NIL then
7  |  |  x ← T.racine
8  |  |  renvoyer 1 + TAILLE(x.gauche) + TAILLE(x.droit)
9  |  else
10 |  |  renvoyer 0
11 |  end
12 end
13 FIN
```

3.2 Calcul de la hauteur d'un arbre



Appliquer cet algorithme à l'arbre ci-dessus.

```
1  VARIABLE
2  T : arbre
3  x : nœud
4  DEBUT
5  Function HAUTEUR(T)
6  |  if  T ≠ NIL then
7  |  |  x ← T.racine
8  |  |  renvoyer 1 + max(HAUTEUR(x.gauche), HAUTEUR(x.droit))
9  |  else
10 |  |  renvoyer 0
11 |  end
12 end
13 FIN
```

Exercice 9.14

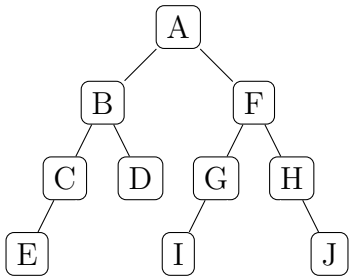
Créer une fonction **hauteur(a)** qui renvoie la hauteur de l'arbre binaire a

3.3 Parcours d'un arbre binaire

On veut ici afficher les différentes valeurs de contenues dans tous les n œuds de l'arbre, par exemple une par ligne. L'ordre dans lequel est effectuée la lecture est donc très important.

3.3.1 Parcours infixe

Un parcours **infixe** fonctionne comme suit : On parcourt le sous arbre de gauche, puis on affiche sa racine, et enfin on parcourt le sous arbre droit.



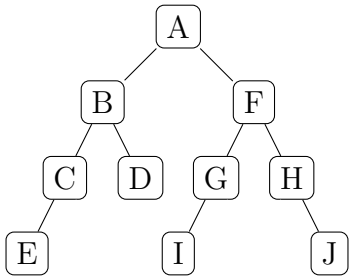

```
1  VARIABLE
2  T : arbre
3  x : noeud
4  DEBUT
5  Function PARCOURS_ INF(T)
6      if T ≠ NIL then
7          x ← T.racine
8          PARCOURS_ INF(x.gauche)
9          affiche x.cle
10         PARCOURS_ INF(x.droit)
11     end
12 end
```

Dans quel ordre a été parcouru cet arbre ?

Exercice 9.15
Créer une fonction **parcours_infixe(a)** qui effectue un parcours infixe de l'arbre binaire a

3.3.2 Parcours de l'arbre ordre préfixe

Un parcours **prefixe** fonctionne comme suit : On affiche la racine et on parcourt les sous arbres.



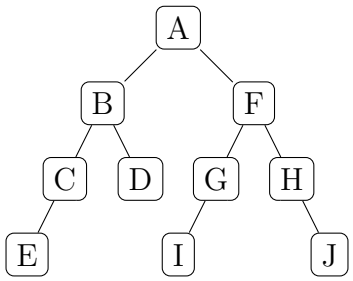
```
1  VARIABLE
2  T : arbre
3  x : nœud
4  DEBUT
5  Function PARCOURS_ PREFIXE(T)
6  |  if T ≠ NIL then
7  |  |  x ← T.racine
8  |  |  affiche x.cle
9  |  |  PARCOURS_ PREFIXE(x.gauche)
10 |  |  PARCOURS_ PREFIXE(x.droit)
11 |  end
12 end
```

Dans quel ordre a été parcouru cet arbre ?

Exercice 9.16
Créer une fonction **parcours_prefixe(a)** qui effectue un parcours prefixe de l'arbre binaire a

3.3.3 Parcours d'un arbre ordre suffixe

Un parcours **postfixe** fonctionne comme suit : on parcourt les sous arbres et on affiche sa racine.

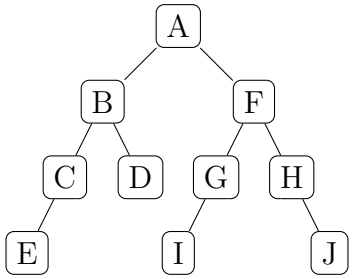



```
1  VARIABLE
2  T : arbre
3  x : nœud
4  DEBUT
5  Function PARCOURS_ SUFFIXE(T)
6  |  if T ≠ NIL then
7  |  |  x ← T.racine
8  |  |  PARCOURS_ SUFFIXE(x.gauche)
9  |  |  PARCOURS_ SUFFIXE(x.droit)
10 |  |  afiche x.cle
11 |  end
12 end
```

Dans quel ordre a été parcouru cet arbre ?

Exercice 9.17
Créer une fonction **parcours_suffixe(a)** qui effectue un parcours prefixe de l'arbre binaire a

3.3.4 Parcourir un arbre en largeur d'abord




```
1  VARIABLE
2  T : arbre
3  Tg : arbre
4  Td : arbre x : nœud
5  f : file
6  Function PARCOURS_ LARGEUR(T)
7      result ← tableau vide
8      if T non vide then
9          f ← file vide
10         f.enfiler(T)
11         while f non vide do
12             T_courant = f.defiler()
13             Ajouter racine de T_courant à result
14             if T_courant.gauche non vide then
15                 f.enfiler(T_courant.gauche)
16             end
17             if T_courant.droite non vide then
18                 f.enfiler(T_courant.droite)
19             end
20         end
21     end
22     Renvoyer result
23 end
```

Dans quel ordre a été parcouru cet arbre ?

Remarque

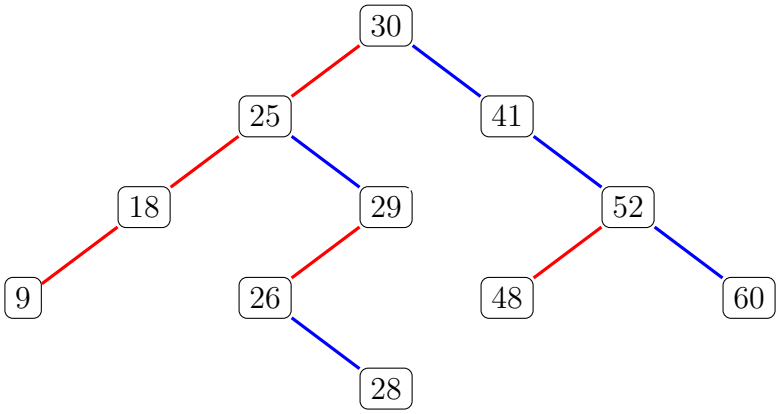
- Cet algorithme utilise une file FIFO
- Cet algorithme n'est pas récursif.

Exercice 9.18
Créer une fonction **parcours_largeur(a)** qui effectue un parcours en largeur d'abord de l'arbre binaire a

4 Les arbres binaires de recherche

Un arbre binaire de recherche est un cas particulier des arbres binaires qui doit satisfaire en plus deux conditions :

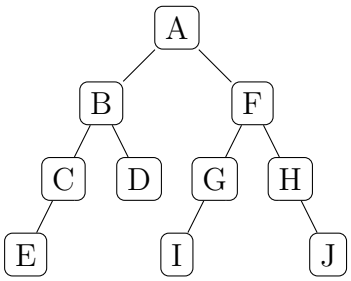
- Les clés de tous les nœuds du sous-arbre gauche d'un nœud X son inférieures ou égales à la clé de X
- Les clés de tous les nœuds du sous-arbre droit d'un nœud X sont strictement supérieures à la clé de X.



Exercice 9.19
Créer l'ABR précédent en python

Exercice 9.20

1. L'arbre suivant est-il un arbre binaire de recherche ?



2. Modifier les étiquettes de cet arbre pour qu'il devienne un arbre binaire de recherche (ABR)
3. Appliquer les différents parcours vus précédemment, et repérer celui, s'il existe, qui permet un affichage des étiquettes dans l'ordre croissant.

Exercice 9.21
Donner tous les arbres binaires de recherche de 3 nœuds et contenant les entiers 1,2 et 3.***

5 Algorithmes sur les arbres binaires de recherche

5.1 Afficher les étiquettes d'un arbre binaire de recherche dans l'ordre croissant

Comme nous l'avons vu dans l'exercice précédent, c'est la parcours infixe qui permet un affichage dans l'ordre croissant (ou l'ordre alphabétique).

5.2 Recherche dans un ABR - Version récursive

5.2.1 L'algorithme

L'intérêt des ABR est d'être plus efficace dans la recherche d'une valeur.
En effet, pour rechercher une valeur, il suffit de la comparer avec la valeur à la racine, puis, si elle est différente, poursuivre la recherche dans un seul sous-arbre.
Pour rechercher une clé dans un arbre binaire de recherche, on peut d'abord la comparer avec la racine. Si la clé est présente à la racine, on renvoie Vrai. Si la clé est inférieure à la racine, on cherche la clé dans le sous-arbre de gauche. Si la clé est supérieure à la racine, on cherche alors dans le sous-arbre de droite. Si la clé n'a pas été trouvée, on retourne Faux.

Savoir-Faire 9.1

Ecrire une fonction **recherche(tree,elt)** qui prend pour argument un arbre binaire de recherche et un elt, et qui retourne False ou True suivant la présence ou non de elt dans l'arbre tree.***

```
def recherche(tree,elt):
```

5.2.2 Efficacité de la recherche

Si l'arbre est équilibré : si les élèments sont a peu près bien répartis entre les deux sous-arbres, la fonction recherche élimine à chaque appel récursif la moitié des éléments !

Approche

Activités qui permettent de déterminer la complexité de l'algorithme précédent.

1. Compléter le tableau suivant :

Nombre n en base 10	$log_2(n)$	n en base 2	Nb de chiffres de l'écriture binaire
15			
28			
100			
90			
64			
2			

☞ Le nombre de chiffre(s) de l'écriture bianaire du nombre n_{10} est $log_2(n) + 1$.

2. Considérons maintenant un nombre n écrit en binaire.

n_{10}	n_2	$\frac{n}{2}$ base 10 à 1 près	$\frac{n}{2}$ base 2 à 1 près
100			
50			
25			
12			
6			
3			
1			

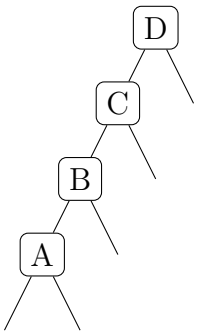
☞ Pour diviser un nombre binaire par deux, il suffit d'enlever le bit de poids faible (à un près)

Savoir-Faire 9.2

Savoir déterminer la complexité de l'algorithme de recherche récursive dans un ABR, quand celui-ci est bien équilibré

Considérons un arbre binaire de recherche, noté T, dont la taille est n (ce qui signifie que le nombre de noeuds est égal à n).
Si, comme supposé, les éléments sont répartis à peu près équitablement entre les sou-arbres, la fonction recherche_rec_abr élimine environ la moitié des éléments à chaque étape.
Considérons m l'écriture de n en base 2. A chaque étape, m étant divisé par 2, m va perdre un chiffre (celui de droite). L'algorithme se termine (dans le pire des cas) lorsque m n'a plus qu'un chiffre.
Ainsi, le nombre d'étapes nécessaires est égal au nombre de chiffres de m, soit $log_2(n) + 1$.
Autrement dit, pour un arbre de taille n, il faut environ $log_2(n)$ étapes.
On peut ainsi en déduire que $O(n) = log_2(n)$

Si l'arbre n'est pas équilibré : Prenons un exemple :



Dans le pire des cas, l'arbre peut être complétement déséquilibré ; c'est le cas de l'exemple où on est en présence d'un **peigne**. Sa hauteur est ici égale ou nombre de nœuds.
Dans ce cas, notre algorithme n'est pas très efficace, car il est suceptible de parcourir l'arbre en entier. Dans ce cas, l'algorithme n'est pas meilleur que celui recherchant une valeur sans une liste chaînée.

D'une façon générale , le coût de la recherche dans un ABR dépend de sa hauteur ; plus précisémment, ce coût est majoré par sa hauteur. En effet, à chaque étape, on descend dans le sous-arbre droite ou le sous-arbre gauche, et on ne peut donc pas répéter ceci un nombre de fois supérieur à sa hauteur.

5.3 Recherche dans un ABR - Version itérative

Créer une fonction **recherche_iteratif_abr(tree,elt) avec tree un ABR et elt un int ou str. La fonction renvoie True ou Fase en fonction du résultat de la rechereche**

5.4 Recherche du minimum dans un ABR

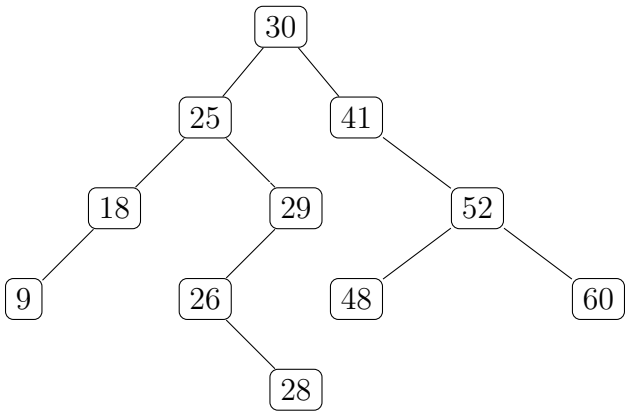
Exercice 9.22
Construire une fonction **min(tree)** qui accepte en argument un ABR et qui renvoie la clé minimale

5.5 Recherche du maximum dans un ABR

Exercice 9.23
Construire une fonction **max(tree)** qui accepte en argument un ABR et qui renvoie la clé maximale.

5.6 Insertion d'une clé dans

☞ Insertions aux feuilles : cela signifie que l'on suppose que element va trouver sa place au niveau des feuilles.



On souhaite inserer la clé "23" dans l'ARB précédent.
☞ Combien y a-t-il de places possibles ?
☞ Quelle instruction permet d'effectuer cet ajout ?

Savoir-Faire 9.3

On souhaite maintenant créer une fonction **ajoute(x,tree)** qui prend en argument un ABR et une clé, et modifie en place l'ABR, en ajoutant la clé au bon endroit.

5.7 Insertion d'une clé dans un arbre binaire de recherche

6 Annexes

Algorithme de recherche d'une clé dans un ABR

```
1  VARIABLE
2  T : arbre
3  cle : clé
4  x : nœud
5  DEBUT
6  Vider f # la file est vide
7  Function RECHERCHER_ CLE(T,cle)
8  |  if T = NIL then
9  |  |  retouner Faux
10 |  else
11 |  |  x ← T.racine
12 |  |  if x = cle then
13 |  |  |  retourner Vrai
14 |  |  end
15 |  |  if x <cle then
16 |  |  |  retourner RECHERCHER_ CLE (cle,x.droit)
17 |  |  else
18 |  |  |  retourner RECHERCHER_ CLE (cle,x.gauche)
19 |  |  end
20 |  end
21 end
```