Les Arbres

Structures de données
Types

Les structures de données linéaires :

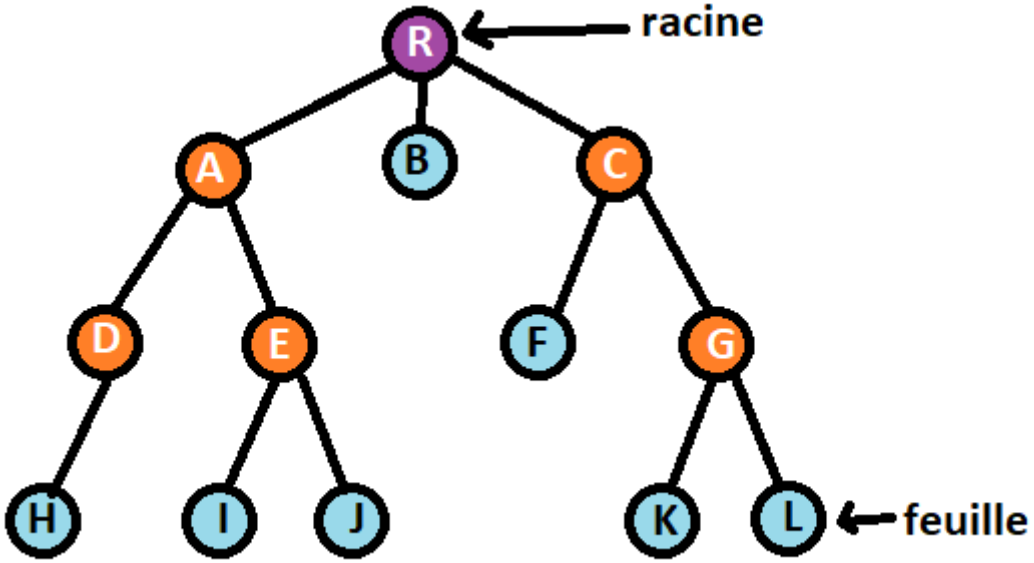
- Les chaînes de caractères
- Les listes
- Les piles
- Les files
- Les dictionnaires
- Les ensembles

Les structures de données non linéaires :

- Les arbres
- Les graphes

Les arbres
Définition

Un arbre est un ensemble organisé de nœuds dans lequel chaque nœud a un père et un seul, sauf un nœud que l'on appelle la racine.

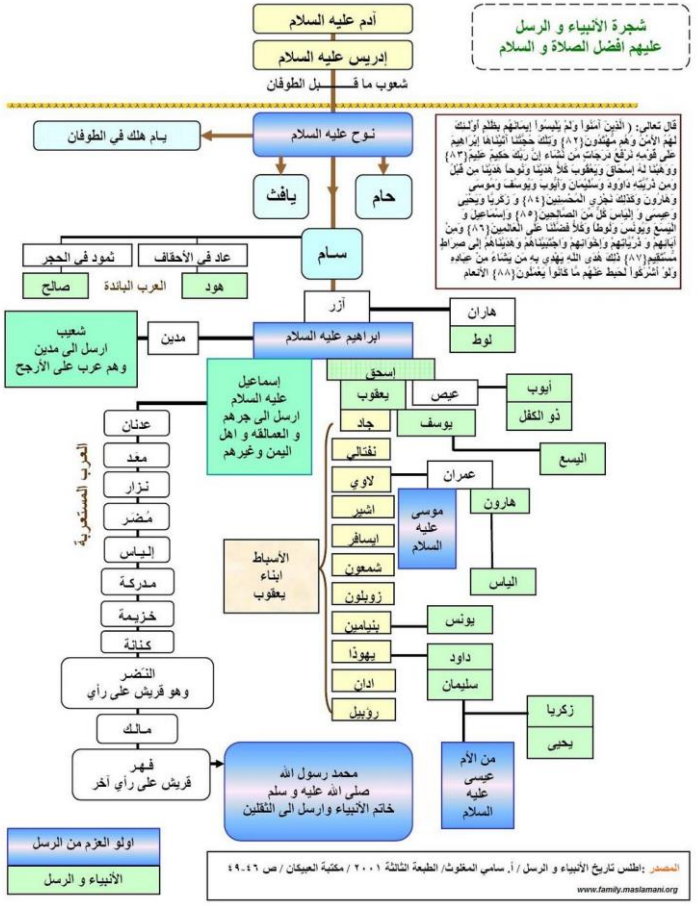


Les arbres
Applications

Modèle pour les structures hiérarchisées:

- Arbres généalogiques :

Exemple :
Arbre de descendance des prophètes



Les arbres
Applications

Modèle pour les structures hiérarchisées:

- Expression arithmétique:

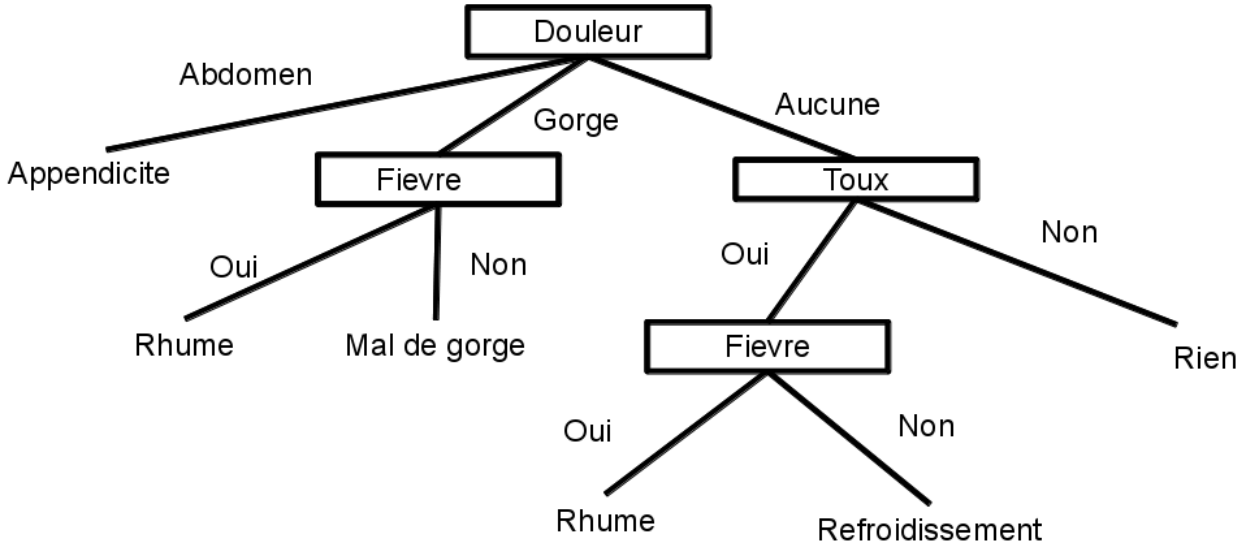
4*3 + 2*7-5

Les arbres
Applications

Modèle pour les structures hiérarchisées:

- Intelligence artificielle :

Exemple :
Système intelligent



Les arbres
Applications

Modèle pour les structures hiérarchisées:

Tri par tas (Tri Maximier)

Base de données : Recherche d’un élément

Système de fichiers : FAT, EXT, HDFS, …

Les arbres
Classification

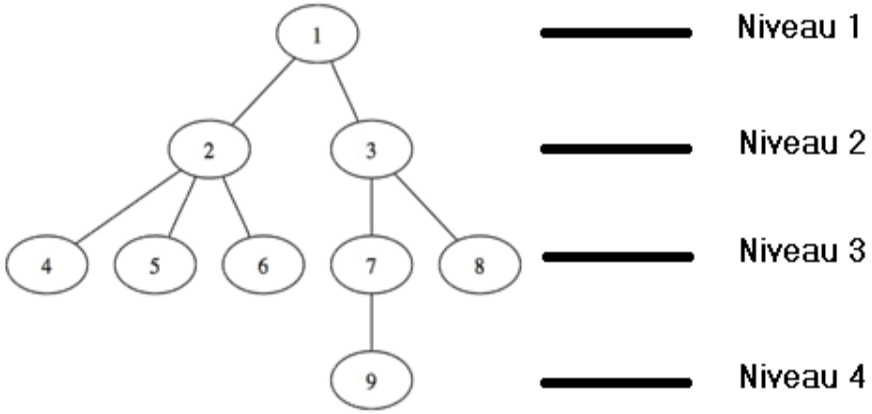
Deux critères de classification :

- Ordre de hiérarchie
- Arité

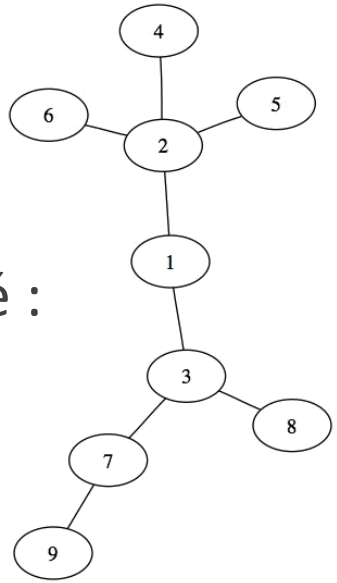
Les arbres
Classification

Classification selon l'ordre de hiérarchie:

- Enraciné :



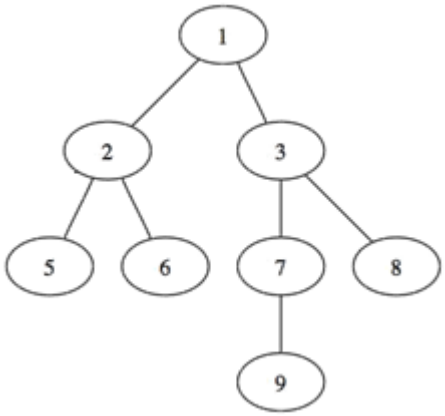
- Non enraciné :



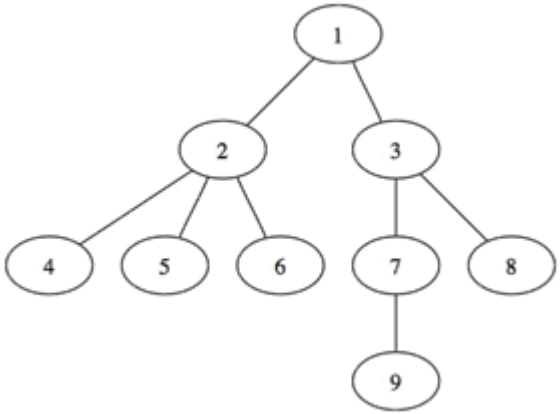
Les arbres
Classification

Classification selon l'arité des nœuds:

- Binaires :



- N-Aires:



Les arbres binaires
Types

Les arbres binaires :

- Les arbres binaires complets
- Les arbres binaires de recherche
- Les arbres binaires équivalents
- Les arbres binaires parfaits
- Les arbres AVL
- Les B-Arbres
- …

Les arbres binaires : Terminologie

Racine : Le seul nœud sans père

r est une racine

Feuille : Un nœud sans fils

b, h et x sont des feuilles

Frère : Un nœud de même parent

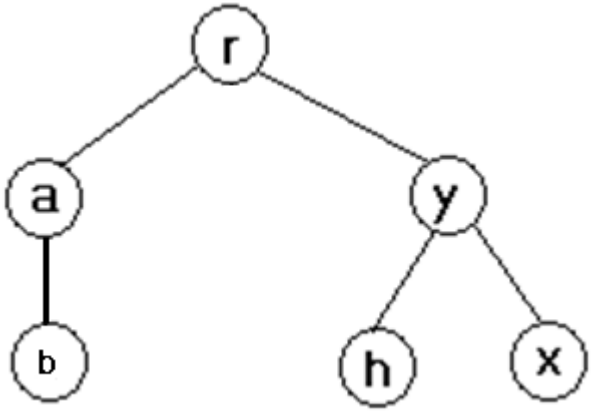
a et y sont des frères

Ancêtre : Un des parents d'un nœud

r est un ancêtre de b, h et x

Descendant : Un des fils d'un nœud

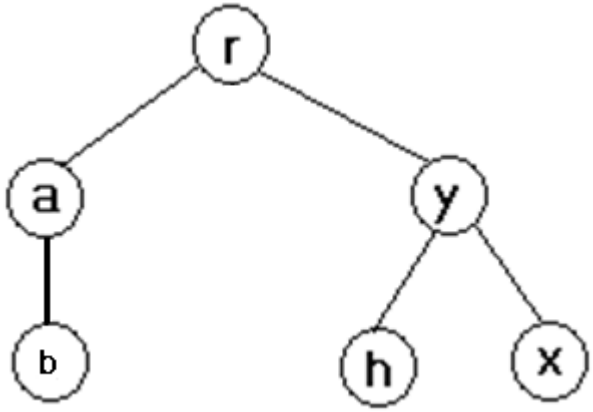
h et x sont des descendants de y



Les arbres binaires : Terminologie

Degré d'un nœud : Nombre de ses enfants

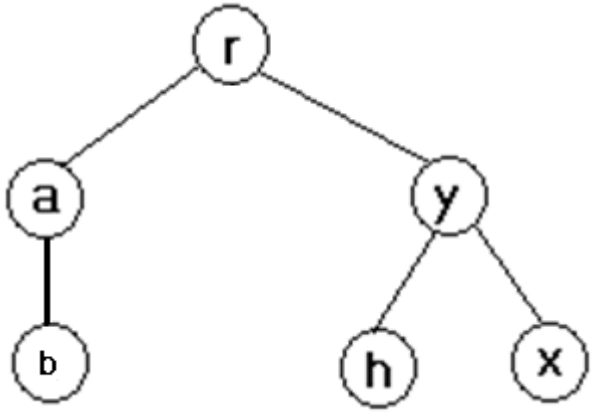
Degré (r) = 2
Degré (y) = 2
Degré (a) = 1
Degré (b) = 0



Les arbres binaires : Terminologie

Profondeur d'un nœud : Longueur du chemin entre la racine et ce nœud

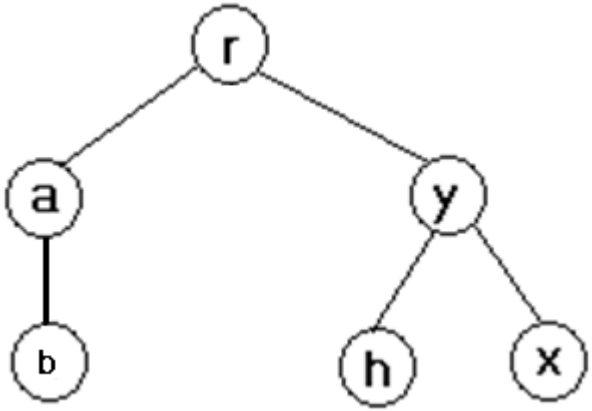
Profondeur (y) = 2
Profondeur (x) = 3
Profondeur (r) = 1



Les arbres binaires : Terminologie

Hauteur d'un arbre : profondeur maximale de ses nœuds

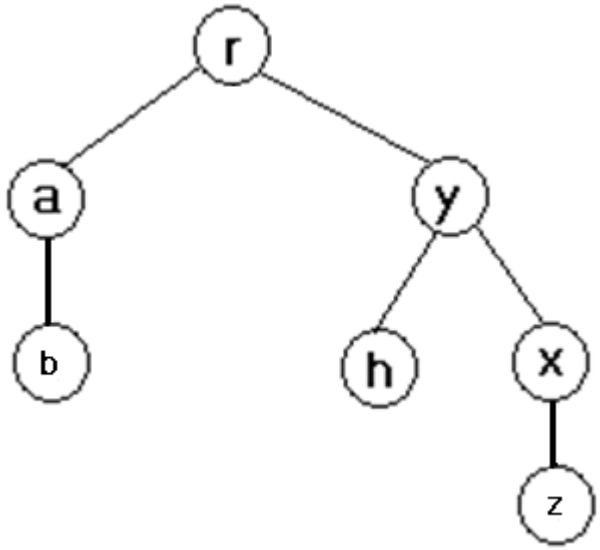
Hauteur = 3



Les arbres binaires : Terminologie

Hauteur d'un nœud : longueur maximale du chemin entre ce nœud et une feuille

Hauteur (x) = 2
Hauteur (y) = 3



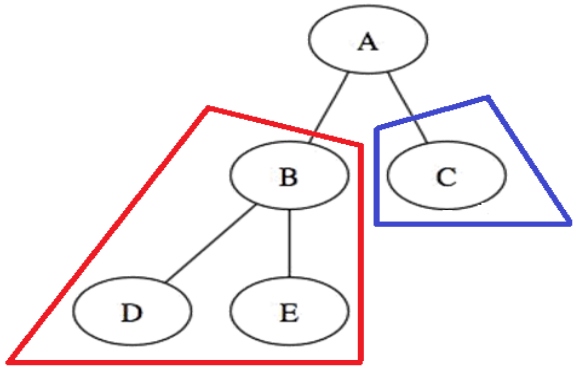
Les arbres
Implémentation

Les arbres binaires forment une structure de données qui peut être définie récursivement de la manière suivante :

Un arbre binaire est :

– soit vide,

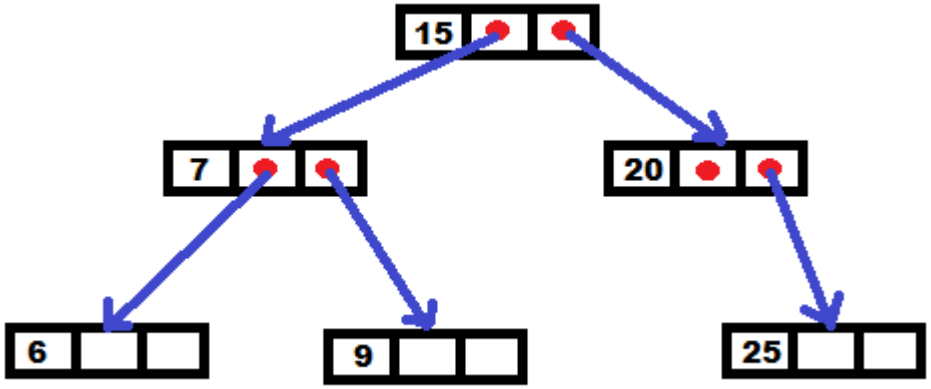
– soit composé d'une racine portant une valeur et d'une paire d'arbres binaires, appelés fils gauche et droit.



Les arbres binaires: Implémentation

- Implémentation par les listes :

On implémente les arbres binaires non vides par des listes de trois éléments :
[valeur , fils gauche , fils droit].



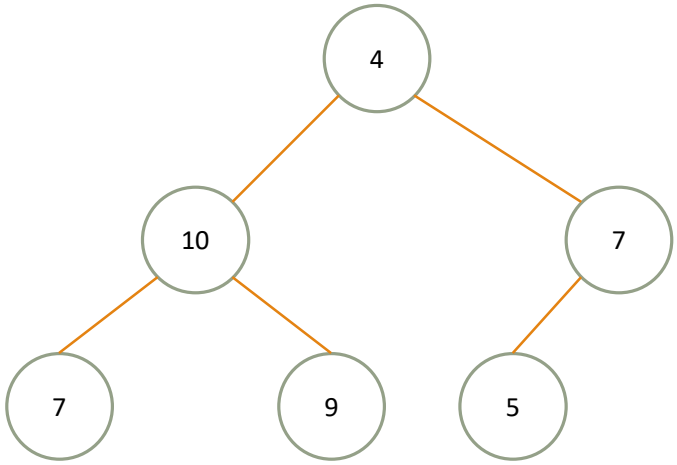
Arb=[15, [7, [6, None, None] , [9, None, None]], [20, None,[25, None, None]]]

Un arbre vide est : Arb = None

Les arbres binaires: Implémentation

Exercice :

Donner l'implémentation par des listes de l'arbre suivante :



A = [4, [10, [7, None, None] , [9, None, None]] , [7, [5,None,None], None]]

Les arbres
Algorithmes de base

Fonction qui teste si un arbre est vide :

```
def est_vide(Arb) :
    return Arb == None
```

Les arbres
Algorithmes de base

Fonction qui retourne la valeur d'un nœud :

```
def valeur(Arb) :
        return Arb[0]
```

Les arbres
Algorithmes de base

Fonction qui retourne le fils à gauche d'un nœud :

```
def fils_gauche(Arb) :
        return Arb[1]
```

Les arbres
Algorithmes de base

Fonction qui retourne le fils à gauche d'un nœud :

```
def fils_droit(Arb) :
        return Arb[2]
```

Les arbres
Algorithmes de base

Fonction qui test si un nœud est une feuille :

```
def est_feuille(Arb) :
        return Arb[1] == None and Arb[2] == None
```

Les arbres
Algorithmes de Parcours

Nous allons découvrir des algorithmes de parcours d'un arbre. Cela permet de visiter tous les nœuds de l'arbre et éventuellement appliquer une fonction sur ces nœuds.

Nous distinguerons deux types de parcours : le parcours en profondeur et le parcours en largeur.

Le parcours en profondeur permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante.

Le parcours en largeur permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau un puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

Les arbres
Algorithmes de Parcours

Variantes du parcours en profondeur :

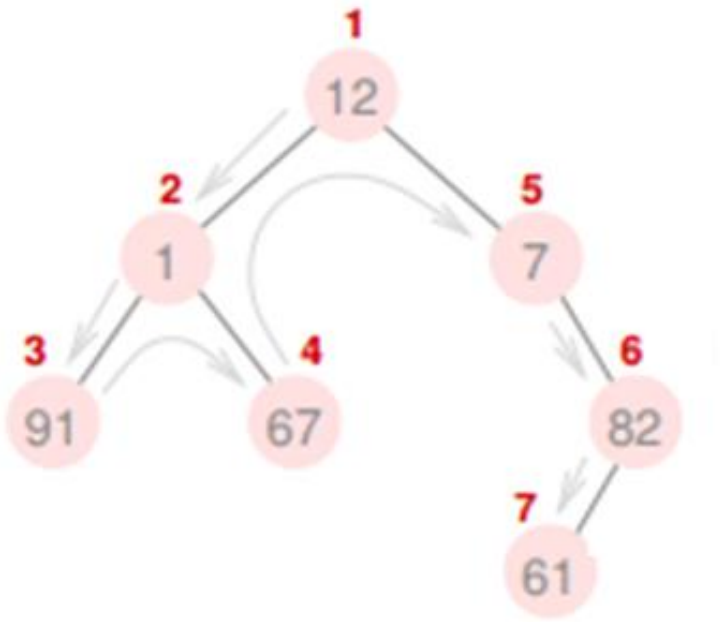
- Parcours préfixe
- Parcours infixe
- Parcours suffixe

Les arbres
Algorithmes de Parcours

Parcours en profondeur préfixe :

Tout Nœud est suivi des nœuds de son sous-arbre Gauche puis des nœuds de son sous-arbre Droit

Le résultat d'affichage est :
12 ; 1 ; 91 ; 67 ; 7 ; 82 ; 61

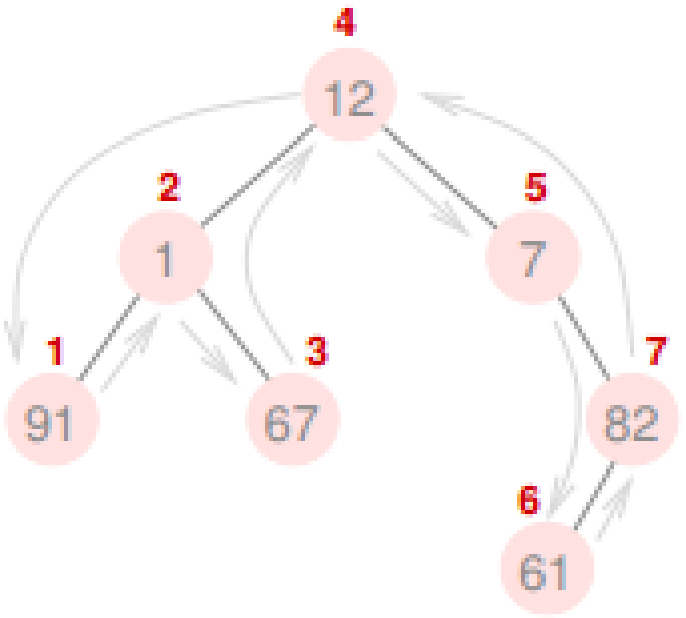


Les arbres
Algorithmes de Parcours

Parcours en profondeur infixe:

Tout Nœud est précédé des nœuds de son sous-arbre Gauche et suivi des noeuds de son sous-arbre Droit

Le résultat d'affichage est :
91 ; 1 ; 67 ; 12 ; 7 ; 61 ; 82

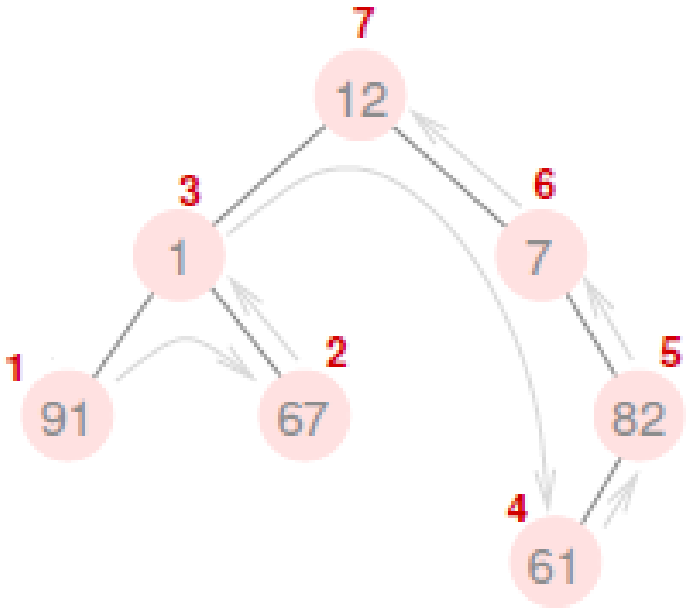


Les arbres
Algorithmes de Parcours

Parcours en profondeur suffixe (postfixe):

Tout Noeud est précédé des noeuds de son sous-arbre Gauche et des noeuds de son sous-arbre Droit

Le résultat d'affichage est :
91 ; 67 ; 1 ; 61 ; 82 ; 6 ; 12

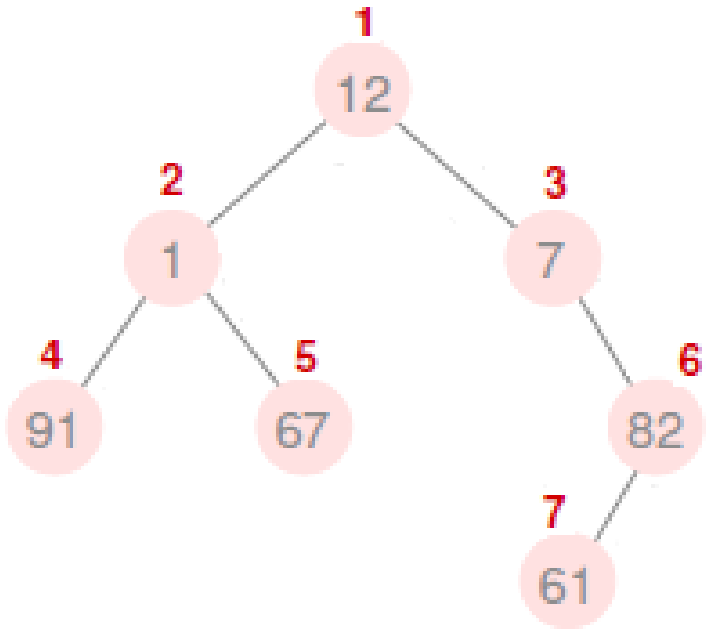


Les arbres
Algorithmes de Parcours

Parcours en largeur :

Dans ce type de parcours on affiche les valeurs par niveau (les nœud de même profondeur).

Le résultat d'affichage est :
12 ; 1 ; 7 ; 91 ; 67 ; 82 ; 61



Les arbres
Algorithmes de Parcours

Exercices :

Ecrire la fonction **parcoursProfondeurPrefix(Arb)** qui affiche les valeurs des nœuds par l'algorithme de parcours préfix.

```
def parcoursProfondeurPrefix(Arb):
        if Arb!=None:
            print(Arb[0])
            parcoursProfondeurPrefix(Arb[1])
            parcoursProfondeurPrefix(Arb[2])
```

Les arbres
Algorithmes de Parcours

Exercices :

Ecrire la fonction **parcoursProfondeurInfix(Arb)** qui affiche les valeurs des nœuds par l'algorithme de parcours infix.

```
def parcoursProfondeurInfix(Arb):
    if Arb!=None:
        parcoursProfondeurInfix(Arb[1])
        print(Arb[0])
        parcoursProfondeurInfix(Arb[2])
```

Les arbres
Algorithmes de Parcours

Exercices :

Ecrire la fonction **parcoursProfondeurPostfix(Arb)** qui affiche les valeurs des nœuds par l'algorithme de parcours postfix.

```
def parcoursProfondeurPostfix(Arb):
    if Arb!=None:
        parcoursProfondeurPostfix(Arb[1])
        parcoursProfondeurPostfix(Arb[2])
        print(Arb[0])
```

Les arbres
Algorithmes de Parcours

Algorithme du parcours en largeur :

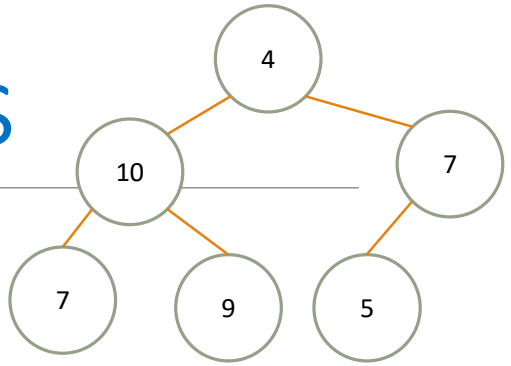
Lorsque nous sommes sur un nœud nous traitons ce nœud (par exemple nous l'affichons) puis nous mettons les fils gauche et droit non vides de ce nœud dans la file d'attente, puis nous traitons le prochain nœud de la file d'attente.

Au début, la file d'attente ne contient rien, nous y plaçons donc la racine de l'arbre que nous voulons traiter.

L'algorithme s'arrête lorsque la file d'attente est vide. En effet, lorsque la file d'attente est vide, cela veut dire qu'aucun des nœuds parcourus précédemment n'avait de sous arbre gauche ni de sous arbre droit.

Par conséquent, on a donc bien parcouru tous les nœuds de l'arbre.

Les arbres
Algorithmes de Parcours



Exemple :

A = [4, [10, [7, None, None] , [9, None, None]] , [7, [5,None,None], None]]

F = [A]

F = [[10, [7, None, None] , [9, None, None]] , [7, [5,None,None], None]]

F = [[7, [5,None,None], None] , [7, None, None] , [9, None, None]]

F = [[7, None, None] , [9, None, None] , [5,None,None]]

F = [[9, None, None] , [5,None,None]]

F = [[5,None,None]]

F = []

Résultat : 4 10 7 7 9 5

Les arbres
Algorithmes de Parcours

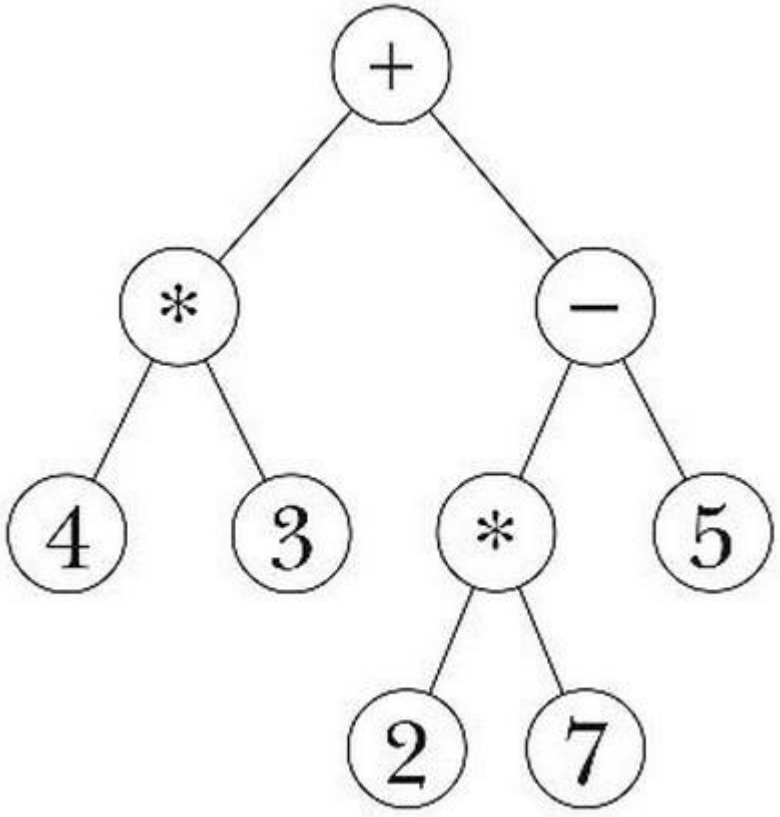
Algorithme du parcours en largeur :

```
def parcoursLargeur(Arb):
        file = [Arb]
        while len(file) > 0:
                N = file.pop(0)
                print(N[0])
                if N[1] != None:
                        file.append(N[1])
                if N[2] != None :
                        file.append(N[2])
```

Les arbres
Exercices

Quel type de parcours doit être utilisé pour évaluer une expression arithmétique ?

- Expression arithmétique: **4*3 + 2*7-5**



Ordre préfixe : +x43-x275
Ordre infixe : 4x3+2x7-5
Ordre postfixe : 43x27x5-+

Exercices :

Ecrire une fonction qui retourne le nombre de nœuds d'un arbre: **nbre_noeuds(arb)**

```
def nbre_noeud(arb):
        if arb==None:
                return 0
        return 1 + nbre_noeud(arb[1]) + nbre_noeud(arb[2])
```

Exercices :

Ecrire une fonction qui retourne la hauteur d'un arbre: **hauteur(arb)**

```
def hauteur(arb):
    if arb==None :
        return 0
    return 1 + max(hauteur(arb[1]), hauteur(arb[2]))
```

Exercices :

Ecrire une fonction qui retourne la valeur maximale d'un arbre binaire : **def max_arb(arb)**

```
def max_arb(A):
    if A[1]==None and A[2]==None :  return A[0]
    if A[1]!=None and A[2]!=None :
            return max(A[0], max_arb(A[1]), max_arb(A[2]))
    elif A[1]!=None :
            return max(A[0], max_arb(A[1]))
    else :   return max(A[0], max_arb(A[2]))
```

Exercices :

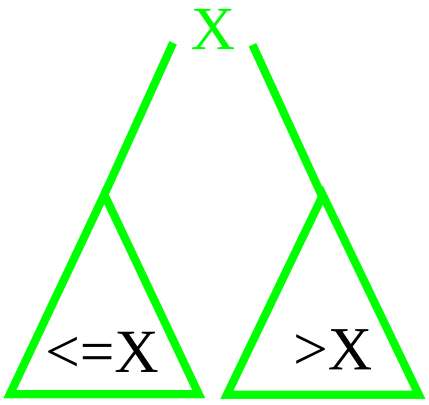
Ecrire une fonction qui retourne la valeur maximale d'un arbre binaire : **def min_arb(arb)**

```
def min_arb(A):
    if A[1]==None and A[2]==None :  return A[0]
    if A[1]!=None and A[2]!=None :
            return min(A[0], min_arb(A[1]), min_arb(A[2]))
    elif   A[1]!=None :
            return min(A[0], min_arb(A[1]))
    else :   return min(A[0], min_arb(A[2]))
```

Arbre binaire de recherche
Définition

Un *arbre binaire de recherche* est un arbre binaire tel que :

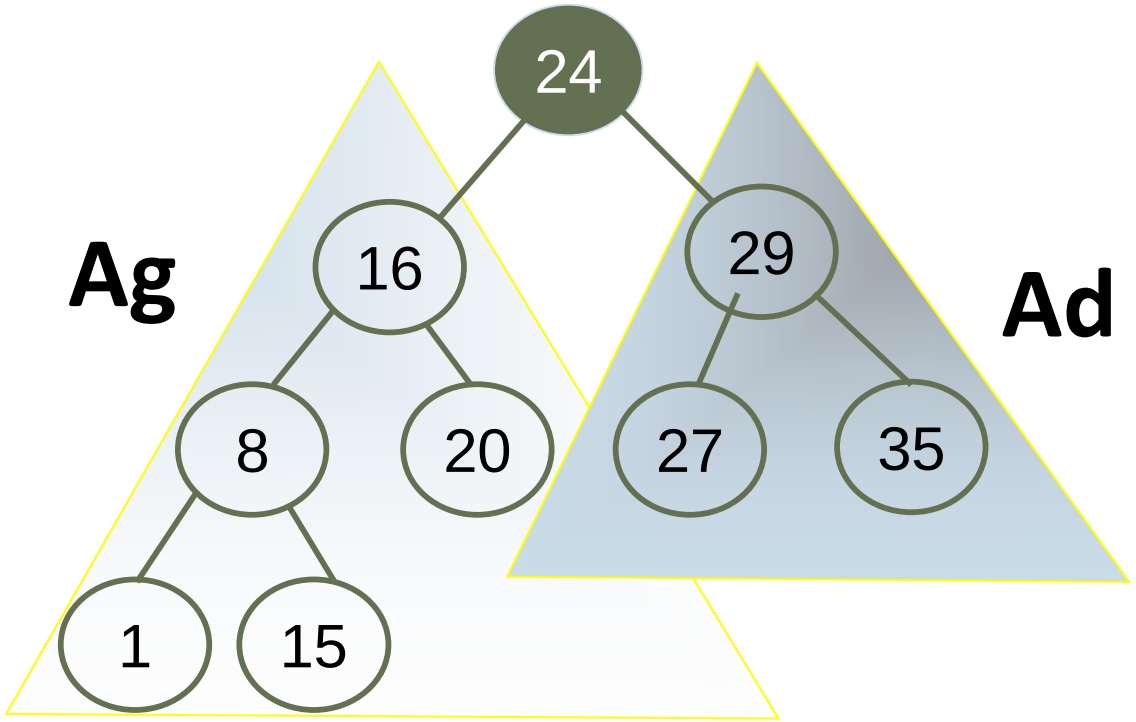
1 - Pour tout nœud x , les nœuds de son sous arbre-gauche s'ils en existent ont des valeurs inférieures ou égales à celle de x, et les nœuds de son sous arbre-droit des valeurs strictement supérieures.



Ce que l'on traduit par $g(A) \leq racine(A) < d(A)$.

Arbre binaire de recherche
Définition

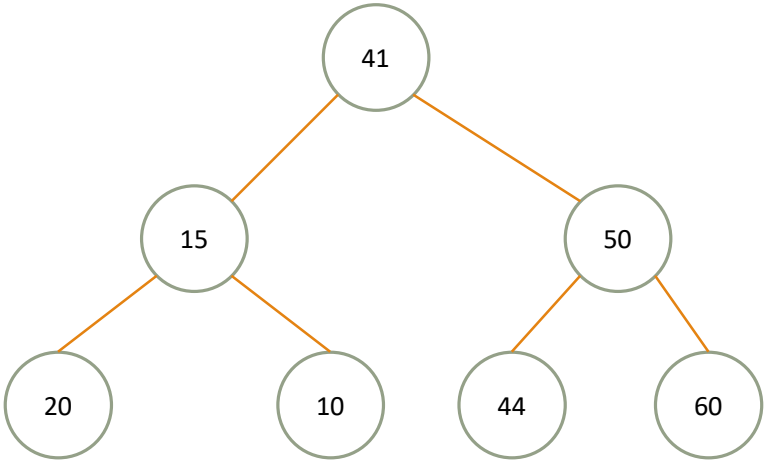
2 - Tout sous-arbre d'un ABR est un ABR :



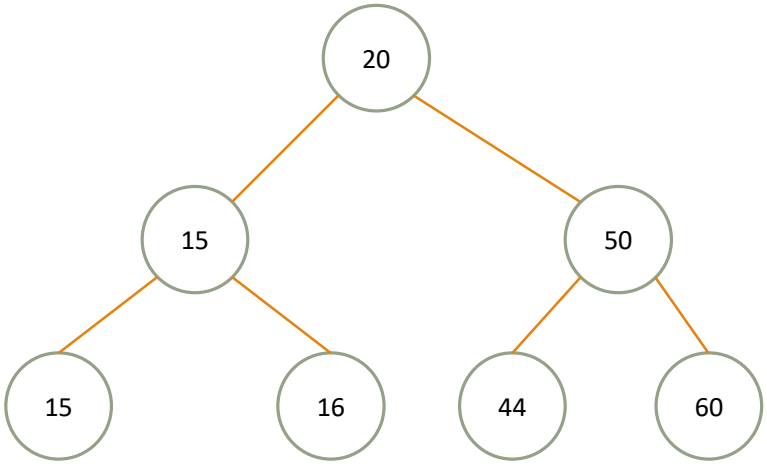
Exemple d'arbre binaire de recherche

Arbre binaire de recherche Définition

Exemples :



Ce n'est pas un arbre binaire de recherche



C'est un arbre binaire de recherche

Exercices :

Ecrire une fonction qui retourne la valeur maximale d'un arbre binaire : **def max_arb(arb)**

```
def max_arb(A):
    if A[1]==None and A[2]==None :  return A[0]
    if A[1]!=None and A[2]!=None :
            return max(A[0], max_arb(A[1]), max_arb(A[2]))
    elif A[1]!=None :
            return max(A[0], max_arb(A[1]))
    else :   return max(A[0], max_arb(A[2]))
```

Exercices :

Ecrire une fonction qui retourne la valeur maximale d'un arbre binaire : **def min_arb(arb)**

```
def min_arb(A):
    if A[1]==None and A[2]==None :  return A[0]
    if A[1]!=None and A[2]!=None :
            return min(A[0], min_arb(A[1]), min_arb(A[2]))
    elif   A[1]!=None :
            return min(A[0], min_arb(A[1]))
    else :   return min(A[0], min_arb(A[2]))
```

Exercices :

Ecrire une fonction qui vérifie si un arbre binaire est un arbre de binaire de recherche: **def est_abr(arb)**

```
def est_abr(A):
        if A==None:
                return True
        return  (A[1]==None or max_arb(A[1]) <=A[0])  and
                (A[2]==None or min_arb(A[2]) > A[0])  and
                est_abr(A[1]) and est_abr(A[2])
```

Exercices :

Ecrire une fonction qui retourne la valeur maximale d'un arbre binaire de recherche: **def max_abr(abr)**

```
def max_abr(abr):
        if abr[2]==None :
                return abr[0]
        return max_abr(abr[2])
```

Exercices :

Ecrire une fonction qui retourne la valeur minimale d'un arbre binaire de recherche: **def min_abr(abr)**

```
def min_abr(arb):
        if abr[1]==None :
                return abr[0]
        return min_abr(abr[1])
```

Exercices : Recherche dichotomique

Ecrire une fonction qui vérifie si un nombre x appartient à un arbre binaire de recherche, par l'algorithme de recherche par dichotomie.

def rechercher(x , arbre) → booléen

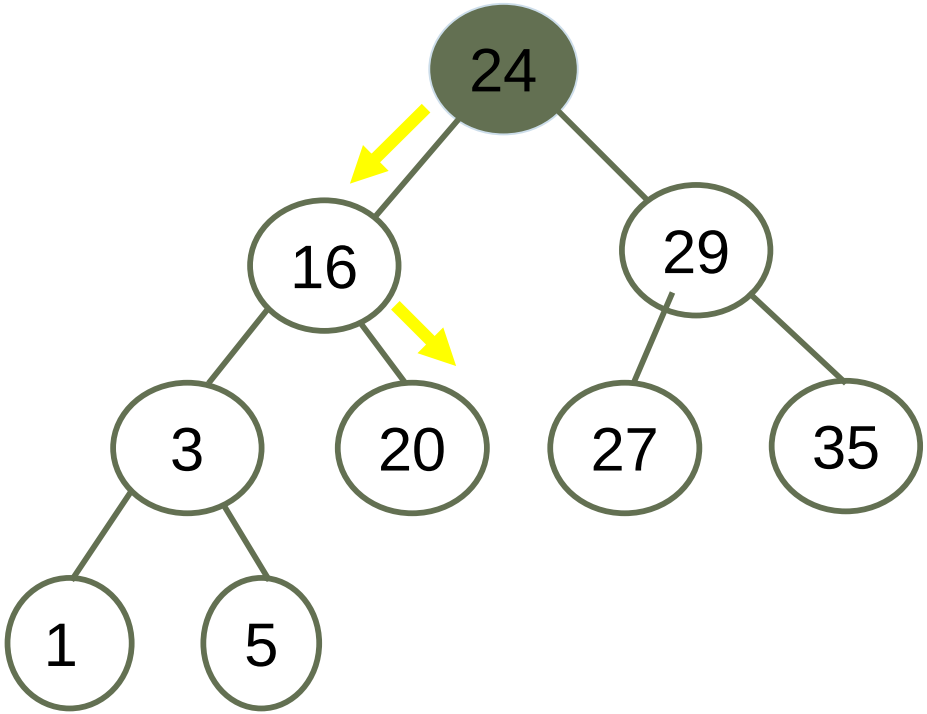
Exercices :
Recherche dichotomique

L'algorithme :

- si le sous-arbre sélectionné est vide, l'élément est absent la fonction retourne False
- On compare l'élément à la valeur de la racine , si égalité la fonction retourne True
- si la valeur est plus petite, on recommence **récursivement** dans le sous-arbre gauche ; et réciproquement si la valeur est plus grande dans le sous-arbre droit.

Exercices :
Recherche dichotomique

Exemple : Soit à rechercher 20 dans l'arbre suivant



20 est plus petit que 24

20 est plus grand que 16

20 est trouvé

Exercices : Recherche dichotomique

Recherche dichotomique : La solution

```
def rechDic(x, Abr):
    if Abr==None:
        return False
    if x == Abr[0]:
        return True
    if x< Abr[0] :
        return rechDic(x, Abr[1])
    if x >  Abr[0] :
        return rechDic(x, Abr[2])
```

Exercices :
Comparer deux ABRs

Ecrire la fonction comparer(arb1 , arb2) qui return True si les deux arbres ont la même la structure et les même valeurs, ou False si non.

```
def comparer(arb1, arb2) :
    if arb1==None  and arb2==None :
        return True
    if arb1!=None  and arb2==None :
        return False
    if arb1==None  and arb2!=None :
        return False
    return    arb1[0]==arb2[0]  and comparer(arb1[1], arb2[1])
                and comparer(arb1[2], arb2[2])
```

Exercices :
Insérer dans un ABR

Ecrire une fonction récursive **inserer(arb , v)** qui permet d'insérer correctement la valeur v dans l'arbre binaire de recherche arb.
La fonction retourne un arbre avec la nouvelle valeur à insérer.

```
def inserer(arb,v):
    if arb==None:
        return [v, None, None]
    if v<=arb[0] :
        arb[1] = inserer(arb[1],v)
    else:
        arb[2] = inserer(arb[2],v)
    return arb
```

Exercices :
Supprimer d'un ABR

Pour supprimer un nœud d'un arbre binaire de recherche, on distingue trois cas à traiter :

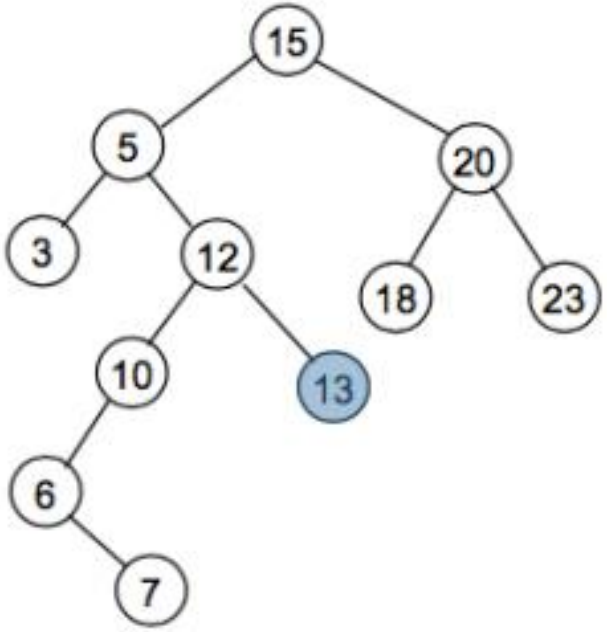
L'élément à supprimer n'a pas de fils : il est terminal et il suffit de le supprimer

L'élément a un fils unique : on supprime le nœud et on relie son fils à son père

L'élément à supprimer a deux fils : on le remplace par son successeur qui est toujours le minimum de ses descendants droits.

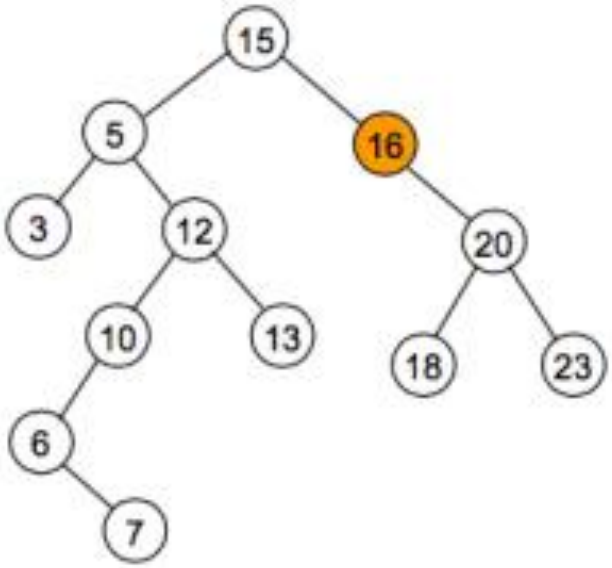
Exercices : Supprimer d'un ABR

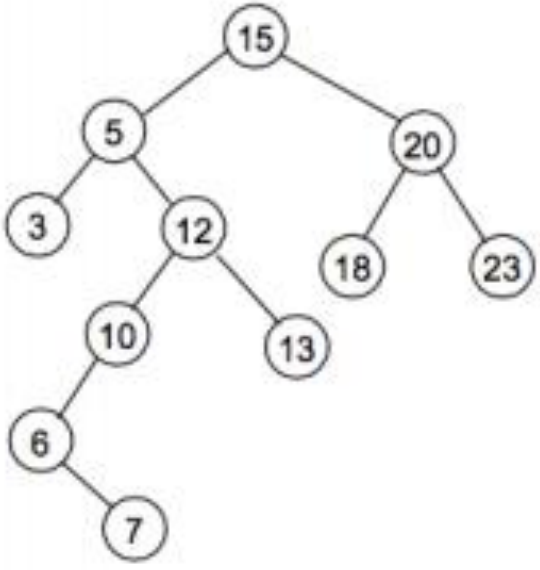
Cas N° 1 : Supprimer un feuille



Exercices :
Supprimer d'un ABR

Cas N° 2 : Supprimer un nœud qui a un seul fils

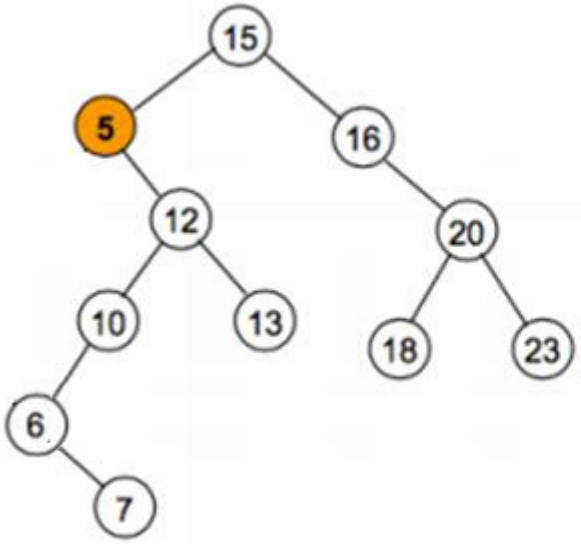


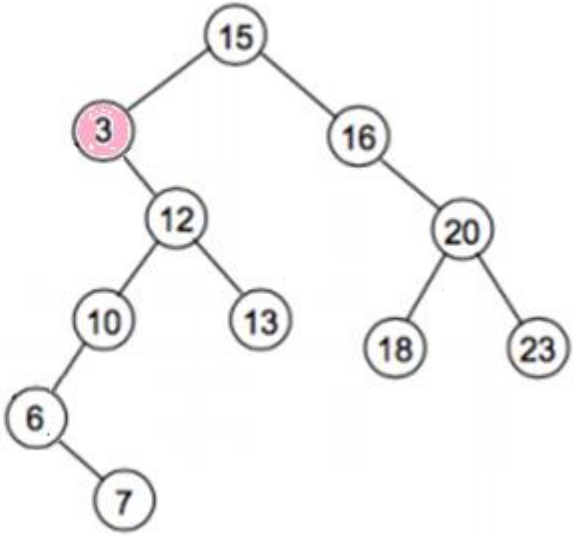


Exercices :
Supprimer d'un ABR

Cas N° 3 : Supprimer un nœud qui a un deux fils

Soit : Dans ce cas, remplacer le noeud à supprimer par la feuille du sous-arbre gauche contenant la valeur maximale

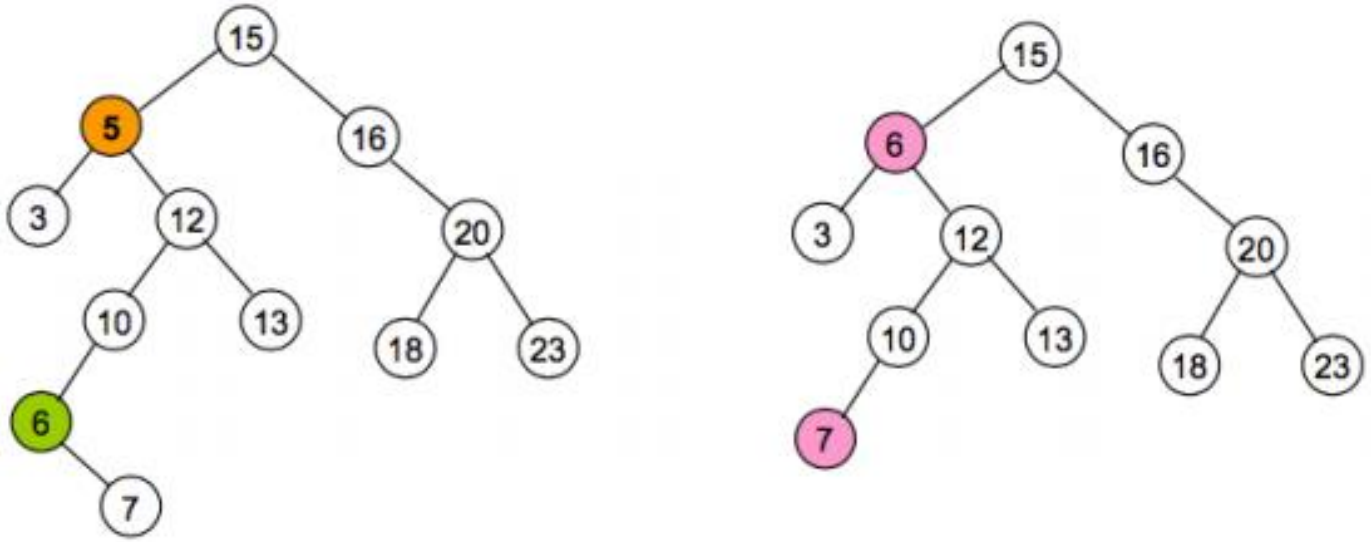




Exercices :
Supprimer d'un ABR

Cas N° 3 : Supprimer un nœud qui a un deux fils

Ou : ou par la feuille du sous-arbre droit contenant la valeur minimale



Exercices :
Supprimer d'un ABR

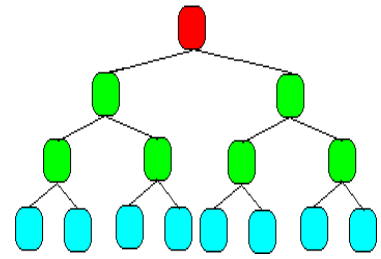
```
def supprime(abr, val):
    if abr==None :
        return abr
    if abr[0]==val :
        if abr[1]==None and abr[2]==None :
            # Si l'élement est une feuille
            return None
        elif abr[1]==None :
            # Si l'élement n'a que le fils de droite
            return abr[2]
        elif abr[2]==None :
            # Si l'élement n'a que le fils de gauche
            return abr[1]
        else :
            # Si l'élement a deux fils
            min = min_arb(abr[2])
            return [min, abr[1], supprime(abr[2], min)]
    elif val > abr[0]:
        return [abr[0], abr[1], supprime(abr[2], val)]
    else :
        return [abr[0], supprime(abr[1], val), abr[2]]
```

Arbre parfait
Définition

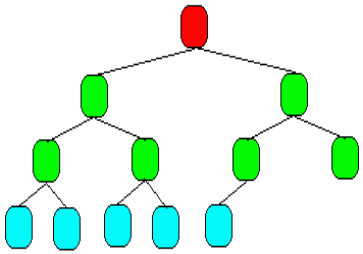
Un arbre binaire est parfait si :

1. Tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux = feuilles), dans ce cas l'arbre parfait est un arbre binaire incomplet

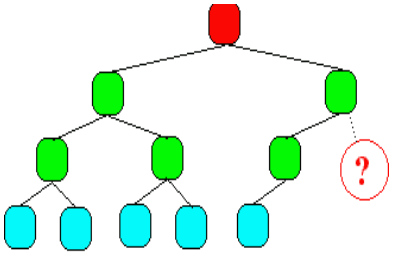
2. Les feuilles du dernier niveau doivent être regroupées à partir de la gauche de l'arbre.



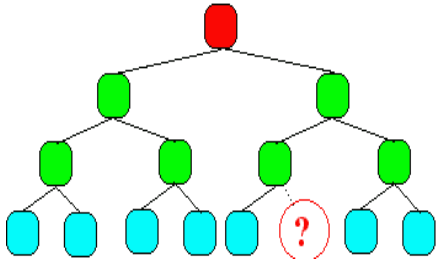
ARBRE BINAIRE PARFAIT



ARBRE BINAIRE PARFAIT



ARBRE BINAIRE NON PARFAIT

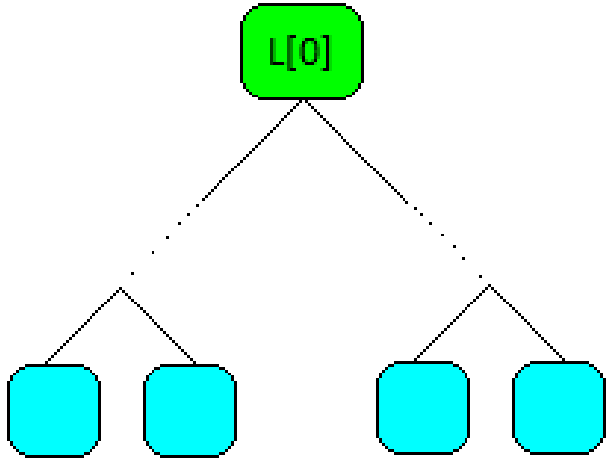


ARBRE BINAIRE NON PARFAIT

Arbre parfait
Implémentation

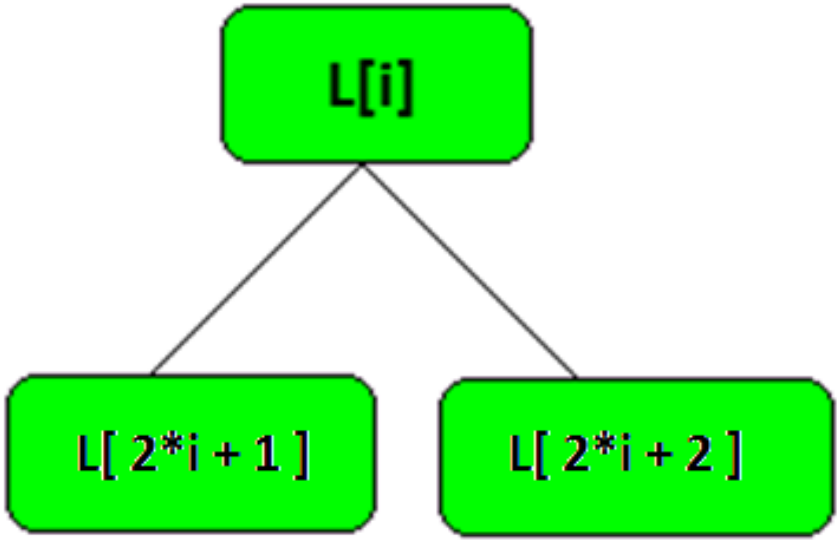
Implémentation : Un arbre binaire parfait peut être implémenter avec une simple liste L , avec les règles suivantes :

Règle 1 : L[0] est la racine de l'arbre



Arbre parfait
Implémentation

Règle 2: L[2*(i+1)-1] et L[2*(i+1)] sont les feuilles de L[i] :



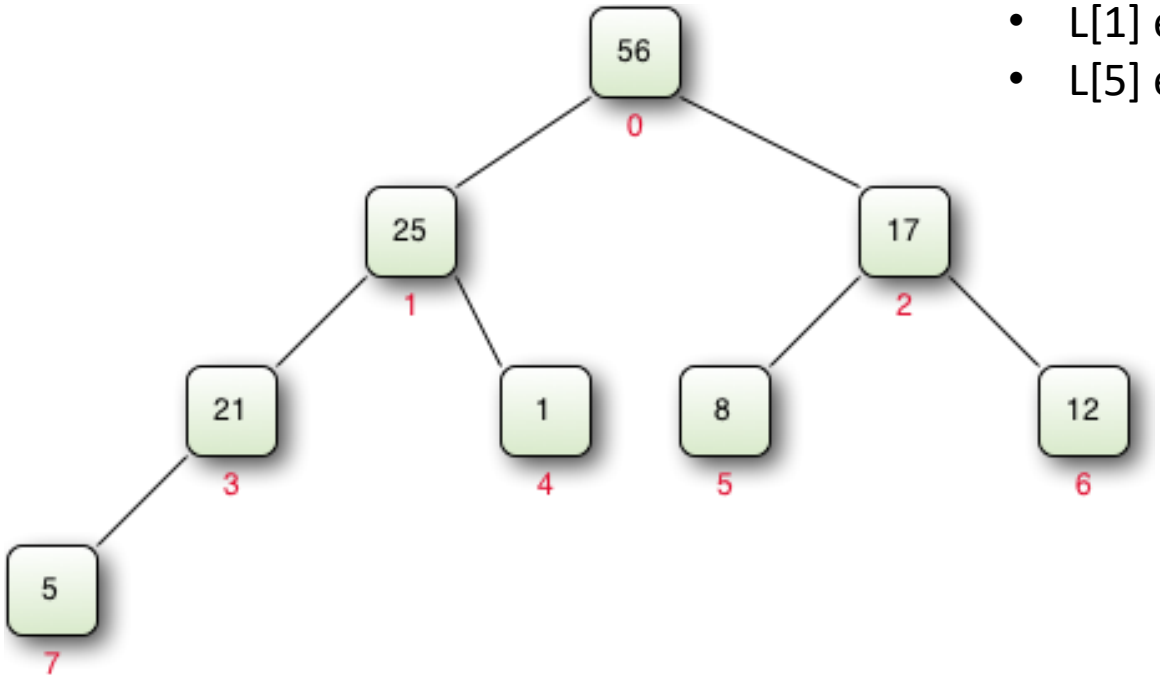
Arbre parfait
Implémentation

Règle 3 : Si i est supérieur ou égale à len(L)//2, L[i] est une feuille.

Arbre parfait
Implémentation

Exemple :

56	25	17	21	1	8	12	5
0	1	2	3	4	5	6	7



- L[1] est le père de L[3] et L[4]
- L[5] est une feuille

Tri Maximier (Tri par tas) Algorithme

L'idée qui sous-tend cet algorithme consiste à voir le tableau comme un arbre binaire.

Le premier élément est la racine, le deuxième et le troisième sont les deux descendants du premier élément, etc.

Tri Maximier (Tri par tas) Algorithme

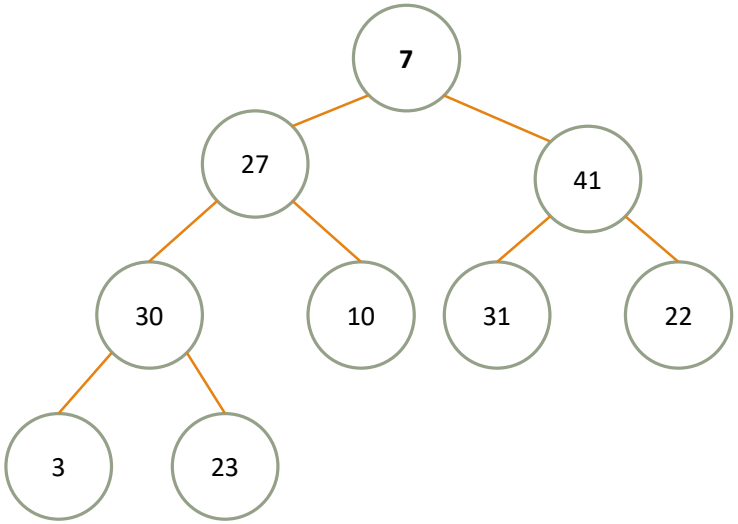
Dans l'algorithme, on cherche à obtenir un tas, c'est-à-dire un arbre binaire parfait vérifiant la propriété suivante :

- chaque nœud est de valeur supérieure (resp. inférieure) à celles de ses deux fils, pour un tri ascendant (resp. descendant).

Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

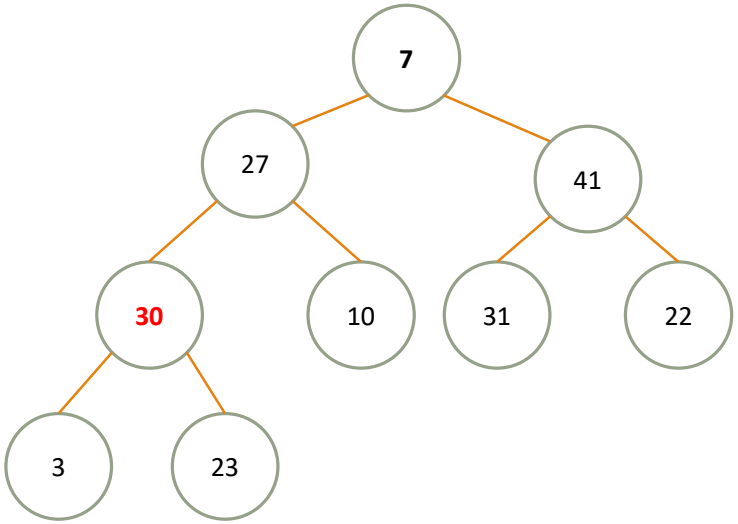
7	27	41	30	10	31	22	3	23



Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

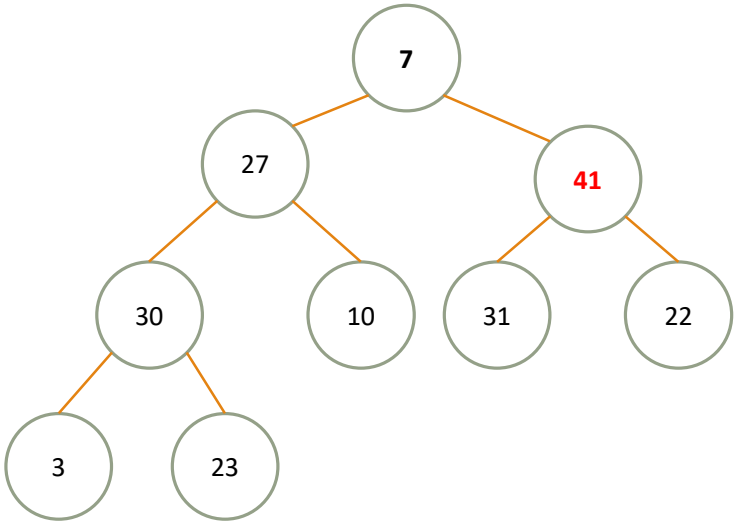
7	27	41	30	10	31	22	3	23



Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

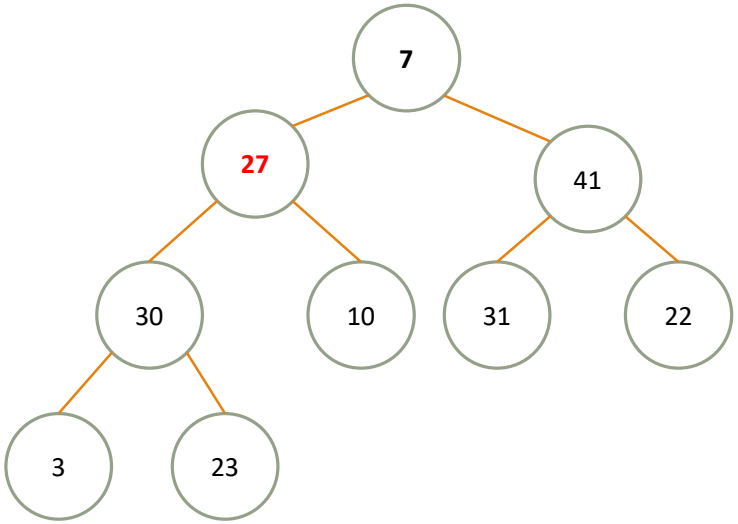
7	27	41	30	10	31	22	3	23



Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

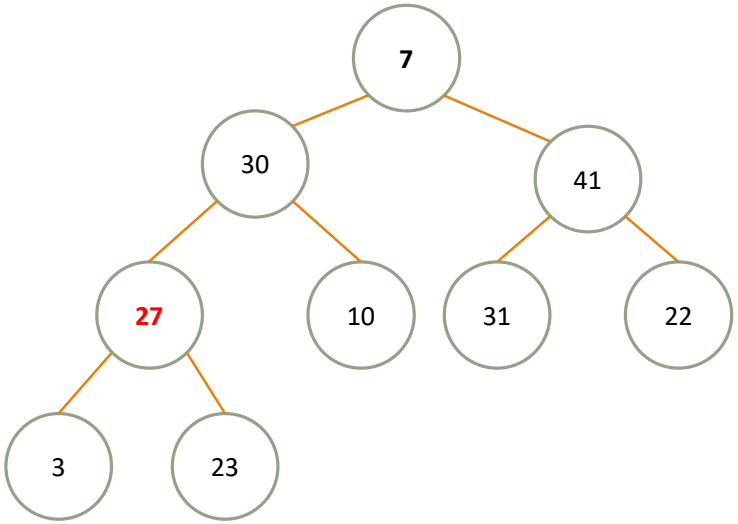
7	27	41	30	10	31	22	3	23



Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

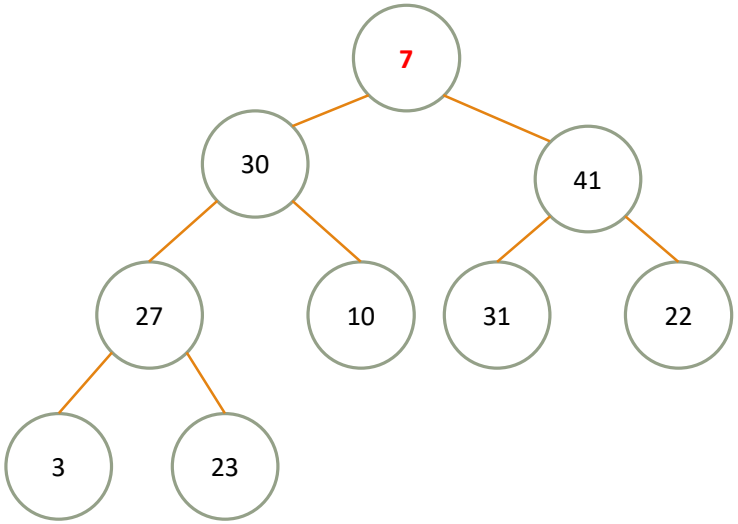
7	27	41	30	10	31	22	3	23



Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

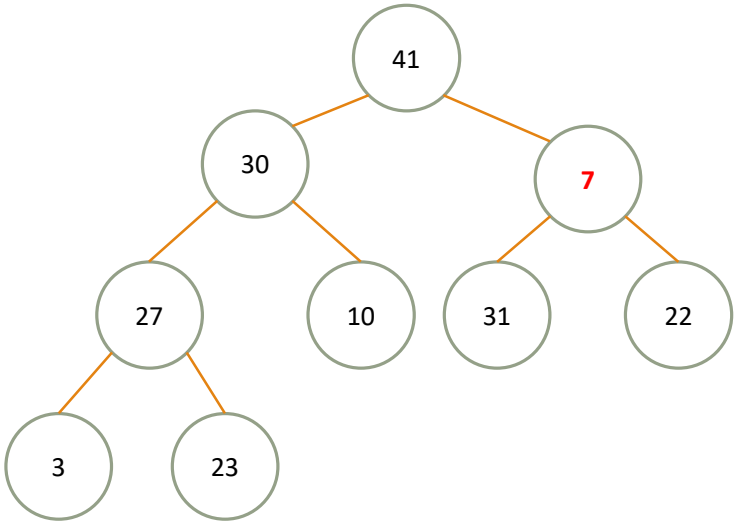
7	27	41	30	10	31	22	3	23



Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

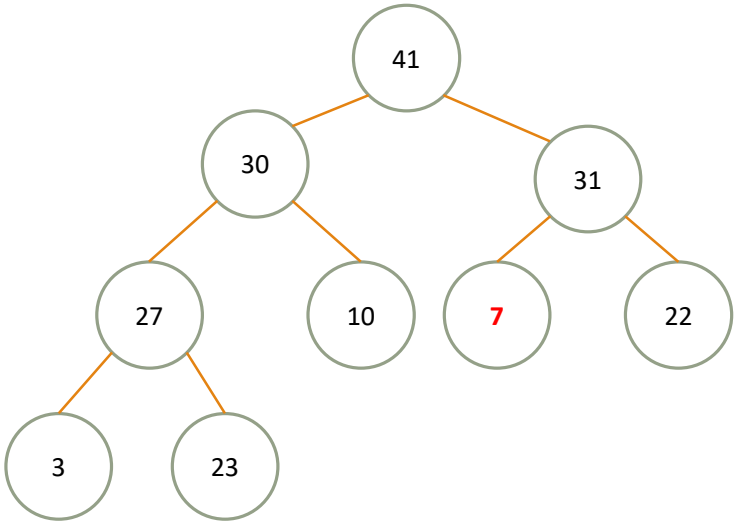
7	27	41	30	10	31	22	3	23



Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

7	27	41	30	10	31	22	3	23



Tri Maximier (Tri par tas) Algorithme

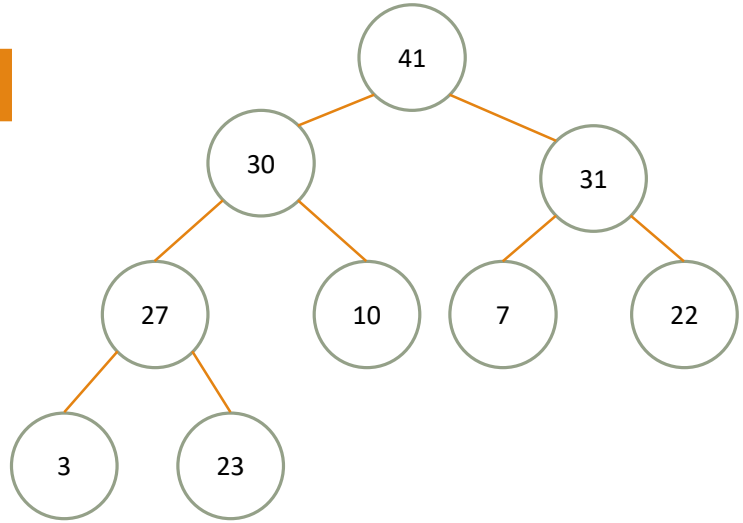
Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

Liste initiale :

7	27	41	30	10	31	22	3	23

Nouvelle liste (Le tas) :

41	30	31	27	10	7	22	3	23



Tri Maximier (Tri par tas) Algorithme

Etape 1 : Création de l'arbre à partir d'une liste (Le tas)

Exercice :

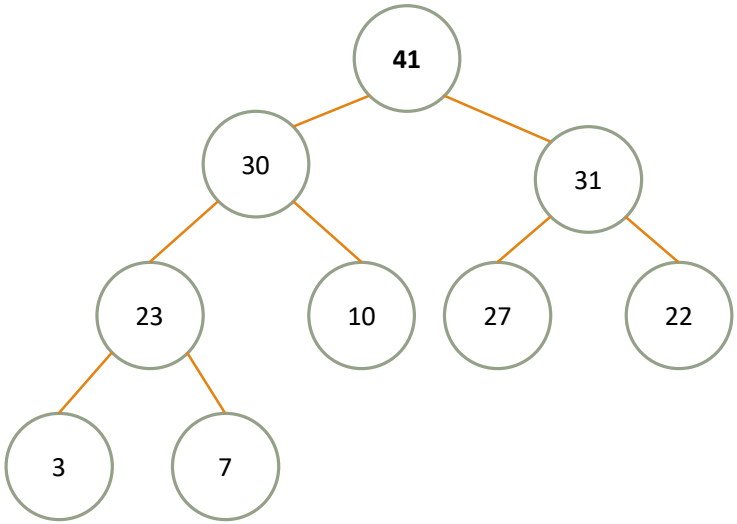
Donner manuellement le tas de la liste suivante

L = [3, 0, 12, 4, 3, 1, 9]

Le tas est : L = [12 , 4, 9 , 0 , 3 , 1 , 3]

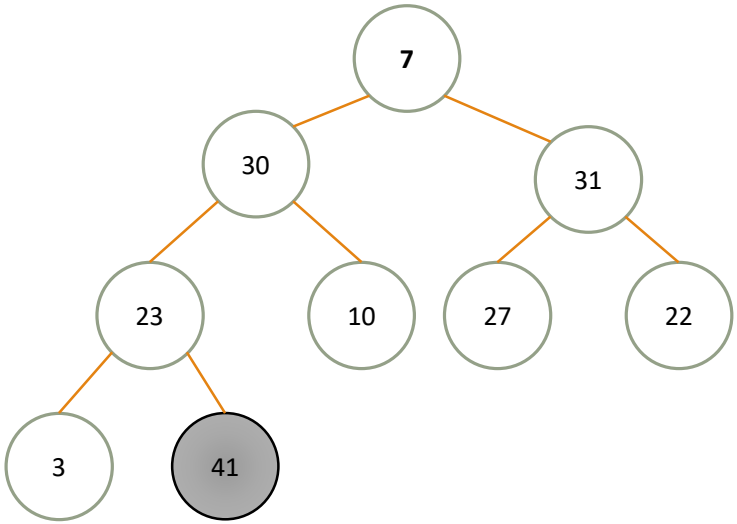
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



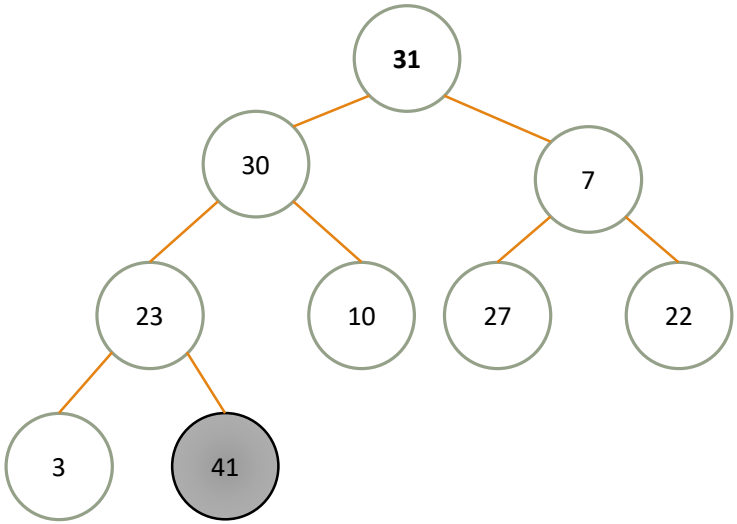
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



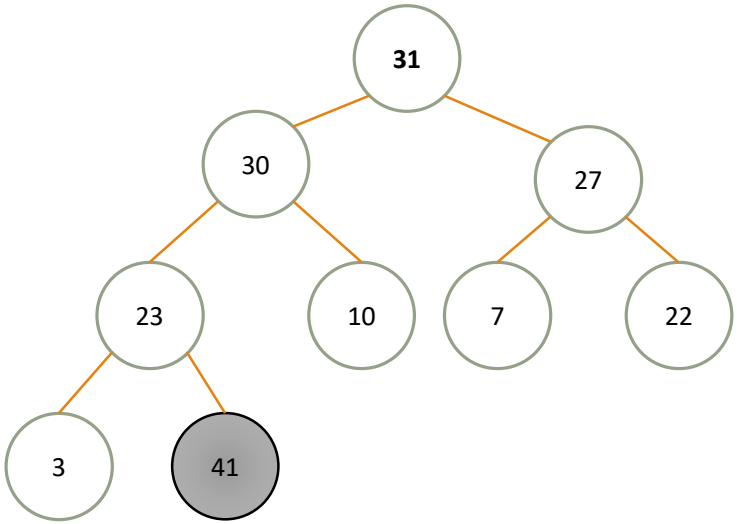
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



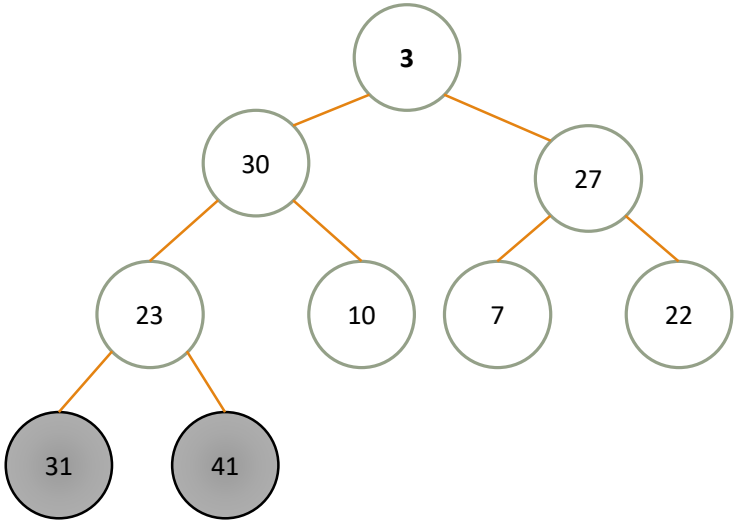
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



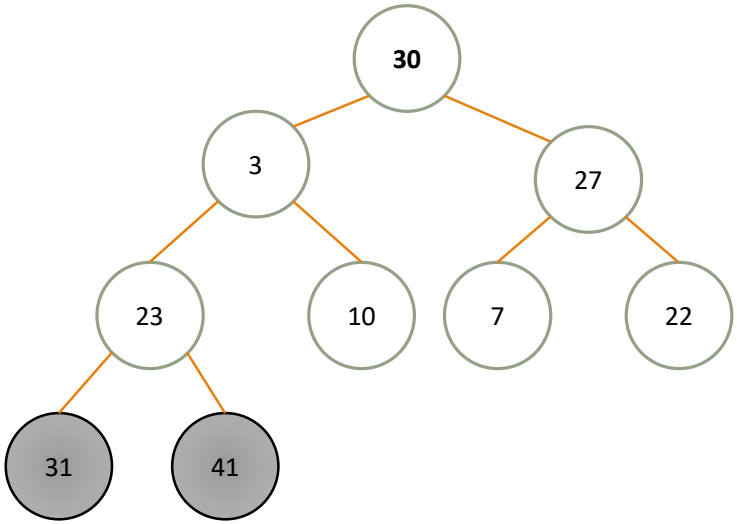
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



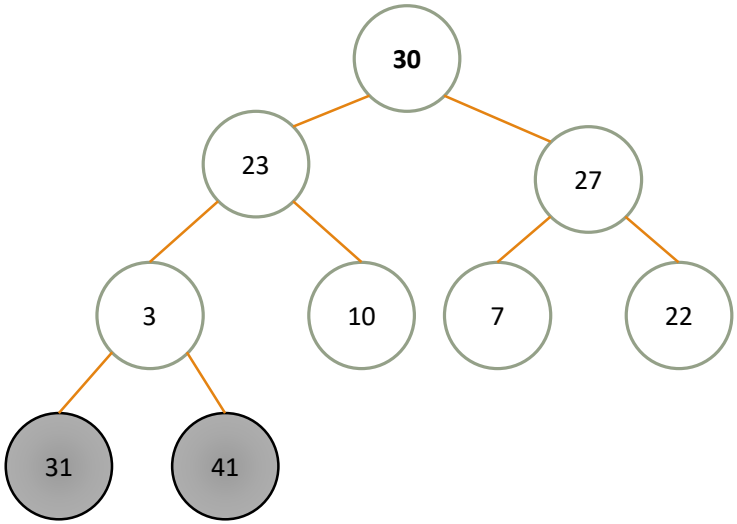
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



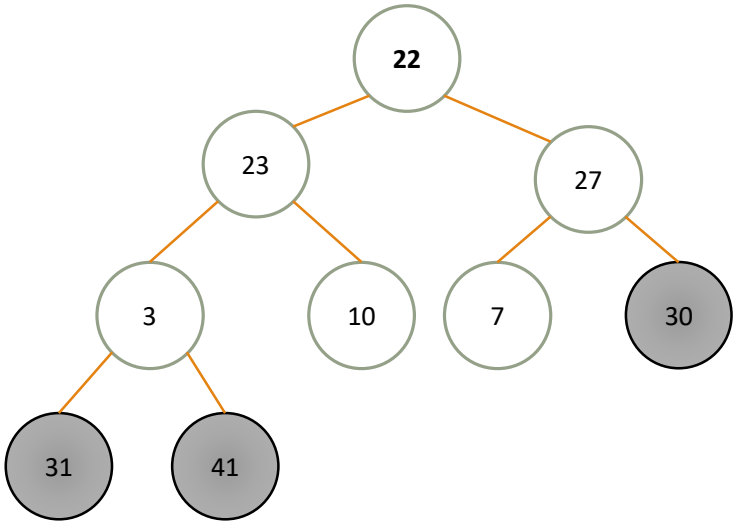
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



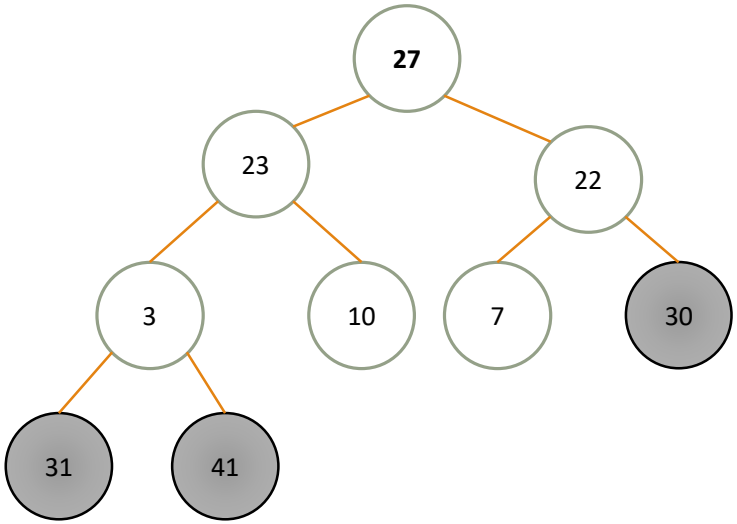
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



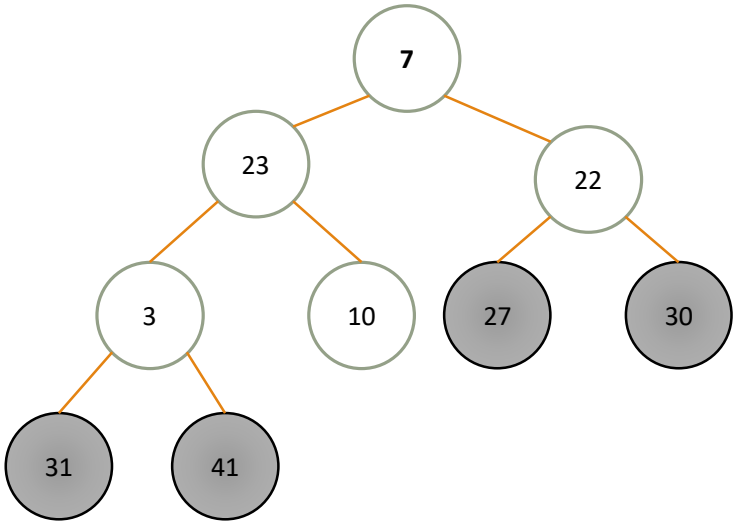
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



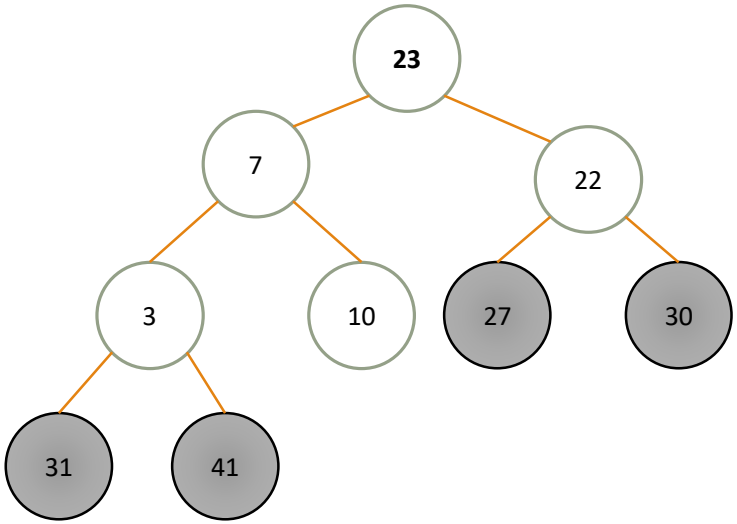
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



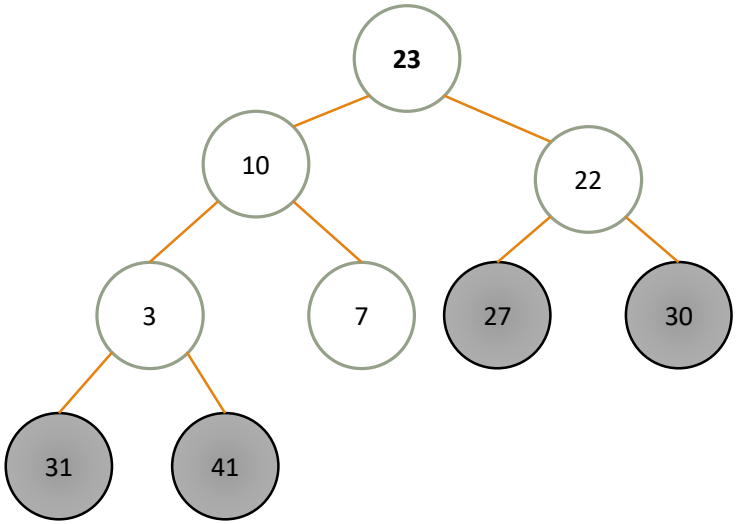
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



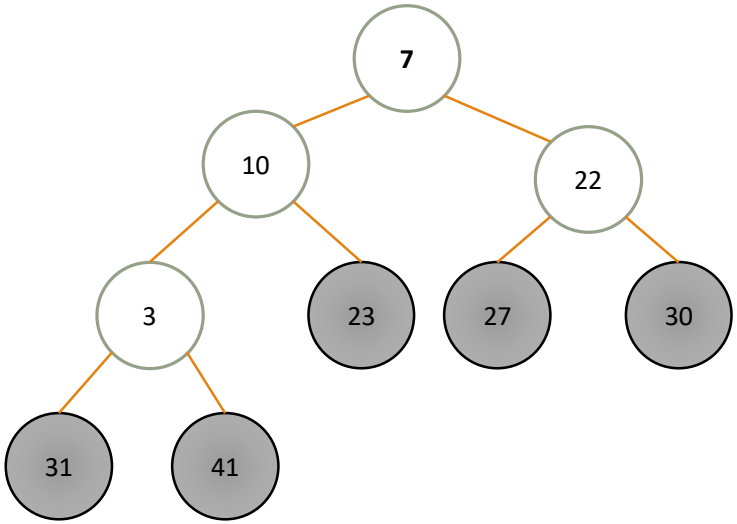
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



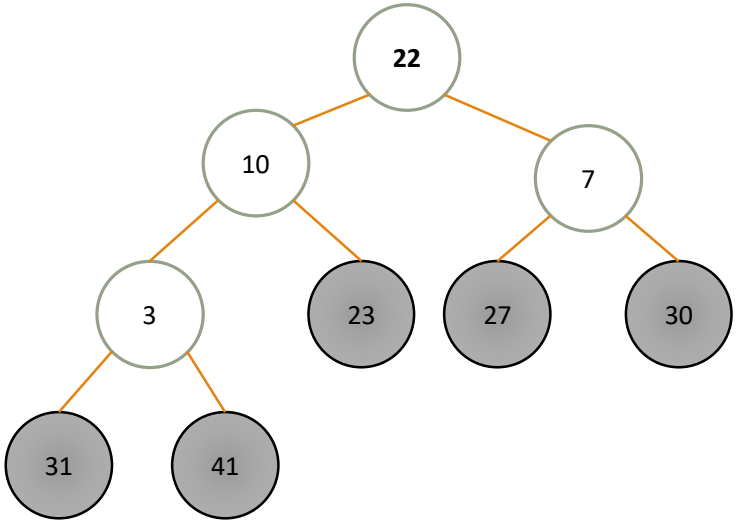
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



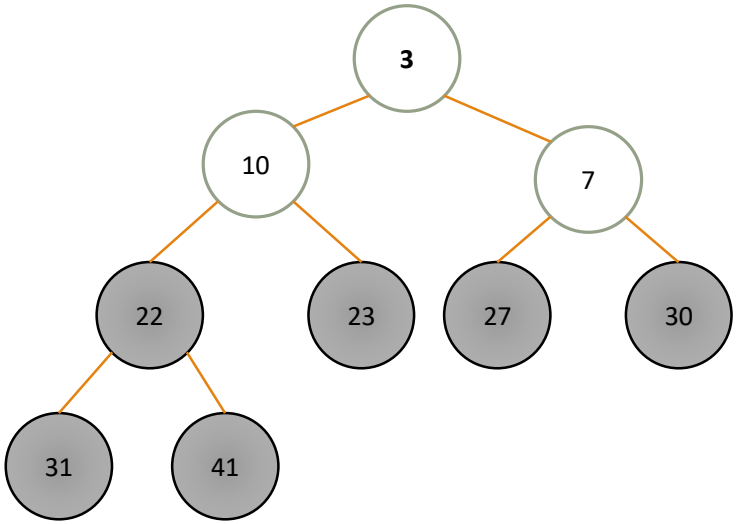
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



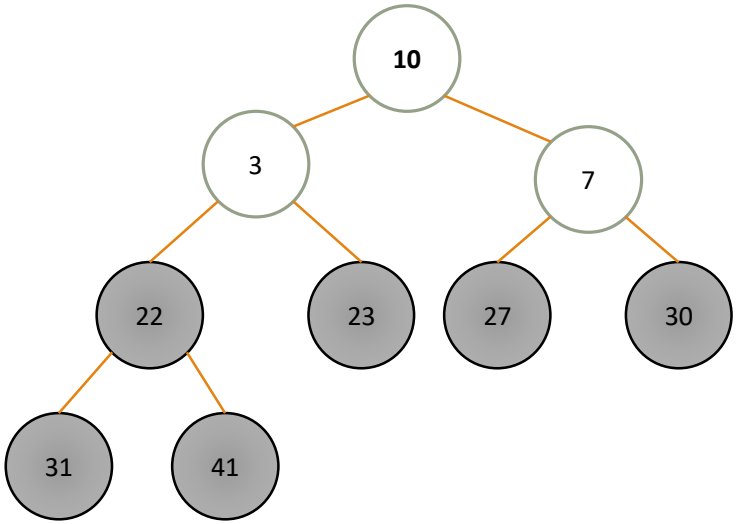
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



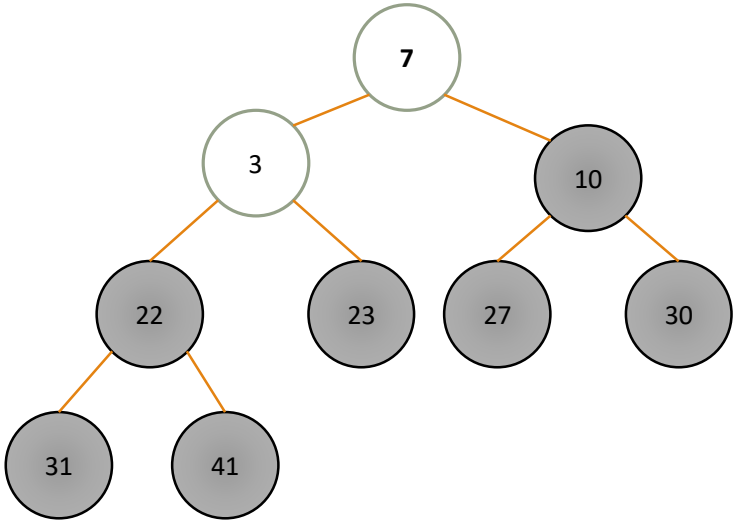
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



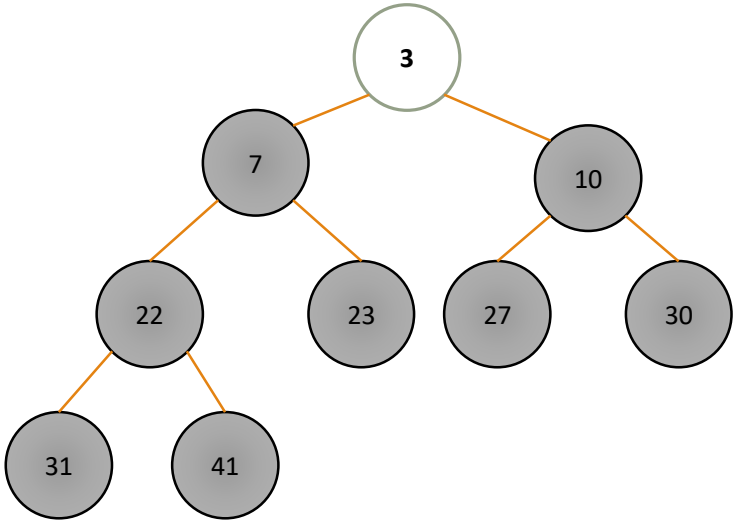
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



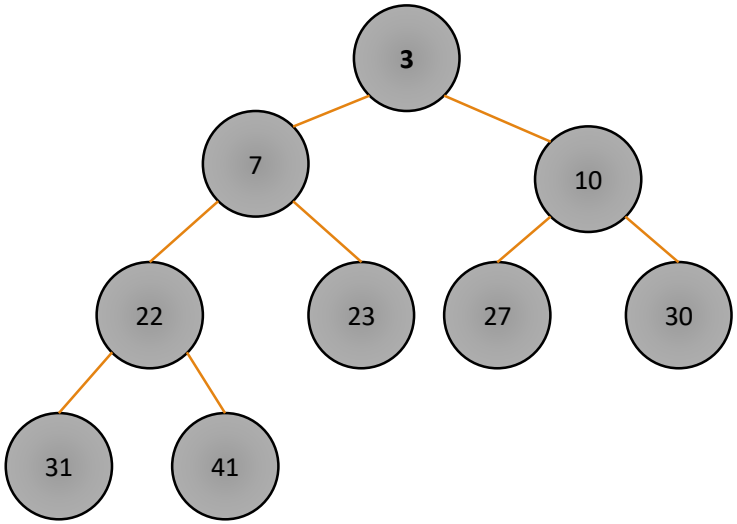
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



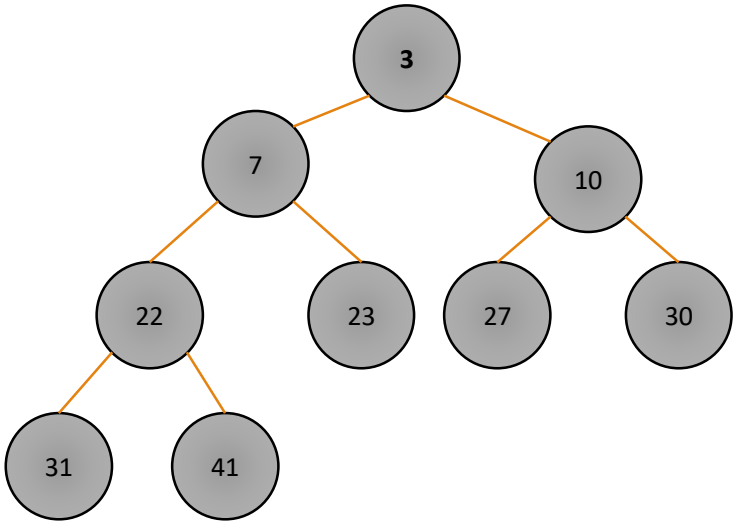
Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



Tri Maximier (Tri par tas) Algorithme

Etape 2 : Tri croissant du tableau et vidage du Tas



3	7	10	22	23	27	30	31	41

Tri Maximier (Tri par tas) Implémentation

Exercices :

1. Ecrire la fonction « **gauche(i)** » qui retourne l'indice du fils à gauche du nœud d'indice i.
2. Ecrire la fonction « **droit(i)** » qui retourne l'indice du fils à droit du nœud d'indice i.
3. Ecrire une fonction «**entasser(L, noeud , limite)**» qui entasse l'élément (L[noeud]) dans l'arbre sans dépasser l'indice **limite**.
4. Ecrire une fonction « **creer_tas(L)** » qui crée un tas de la liste L.
5. Ecrire une fonction « **vider_tas(L)** » qui vide le tas passé en paramètre.
6. Ecrire une fonction « **tri_tas(L)** » qui tri les éléments de L par le tri par tas.

Tri Maximier (Tri par tas) Implémentation

```
def gauche(i):
        return 2*i+1
def droite(i):
        return 2*i+2
```

```
def entasser(L, noeud, limite):
    i=noeud
    while i<limite:
        imax = i
        if gauche(i) < limite and L[gauche(i)]>L[imax]:
            imax = gauche(i)
        if droite(i) < limite and L[droite(i)]>L[imax]:
            imax = droite(i)
        if i!=imax :
            L[i], L[imax] = L[imax], L[i]
            i=imax
        else:
            break
```

Tri Maximier (Tri par tas) Implémentation

```
def creer_tas(L):
    for i in range(len(L)//2-1, -1, -1):
        entasser(L, i, len(L))

def vider_tas(L):
    for i in range(len(L)-1, 0, -1):
        L[0], L[i] = L[i], L[0]
        entasser(L, 0, i)
```

```
def tri_tas(L):
    #Création du tas
    creer_tas(L)
    # Trier
    vider_tas(L)
```