Initiation aux algorithmes des arbres

I. Introduction

1. Graphes

Un **graphe orienté** G est représenté par un couple (S, A) où S est un ensemble fini et A une relation binaire sur S. L'ensemble S est **l'ensemble des sommets** de G et A est **l'ensemble des arcs** de G.

Il existe deux types de graphes :

⇒ ***graphe orienté** : les relations sont orientées et on parle d'arc. Un arc est représenté par un couple de sommets ordonnés.*

⇒ ***Graphe non orienté** : les relations ne sont pas orientées et on parle alors d'arêtes. Une arête est représentée par une paire de sommets non ordonnés.*

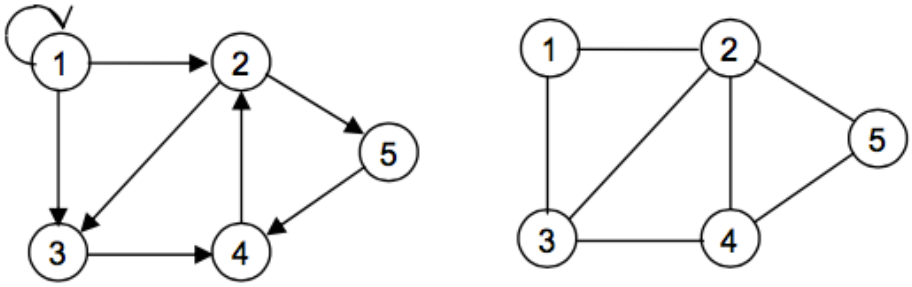


***Figure 1** : Exemple de graphe orienté et de graphe non orienté*

- **Une boucle** est un arc qui relie un sommet à lui-même. Dans un graphe non orienté les boucles sont interdites et chaque arête est donc constituée de deux sommets distincts.
- **Degré d'un sommet :** Dans un *graphe non orienté*, le **degré** d'un sommet est le nombre d'arêtes qui lui sont incidentes.
- **Degré sortant d'un sommet :** Dans un *graphe orienté*, le **degré sortant** d'un sommet est le nombre d'arcs qui en partent,
- **Degré rentrant d'un sommet :** le **degré entrant** est le nombre d'arcs qui y arrivent et le **degré** est la somme du degré entrant et du degré sortant.
- **Chemin :** Dans un *graphe orienté* $G = (S,A)$, un **chemin** de **longueur** k d'un sommet u à un sommet v est une séquence $(u0,u1,..., uk)$ de sommets telle que $u = u0$, $v = uk$ et $(ui\text{-}1, ui)$ appartient à A pour tout i. Un chemin est **élémentaire** si ses sommets sont tous distincts.
- **Un sous-chemin** $p0$ d'un chemin $p = (u0,u1, ... ,uk)$ est une sous-séquence contiguë de ses sommets. Autrement dit, il existe i et j, $0<=i<= j <=k$, tels que $p0 = (ui,ui+1, ... ,uj)$.
- **Circuit :** Dans un *graphe orienté* $G=(S,A)$, un chemin $(u0,u1, ... ,uk)$ forme un circuit si $u0 =uk$ et si le chemin contient au moins un arc. Ce circuit est **élémentaire** si les sommets $u1, ..., uk$ sont distincts. Une boucle est un circuit de longueur 1.
- **Cycle :** Dans le cas non orienté, un chemin v0v1, v1v2, ..., vk−1vk est un cycle si v0 = vk et si une arête n'apparait pas deux fois avec une orientation différente. Par exemple v0 v1, v1 v0 n'est pas un cycle dans le cas non orienté. Ce cycle est élémentaire si les sommets $u1, ..., uk$ sont distincts. Un graphe sans cycle est dit **acyclique**.

➲ **Graphe connexe :** Un *graphe non orienté* est **connexe** si chaque paire de sommets est reliée par une chaîne. Les **composantes connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation « est accessible à partir de ». (***Figure 1***)

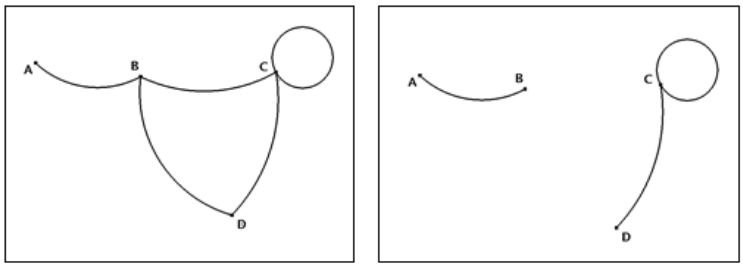


Figure 2* : *graphe connexe et graphe non connexe

➲ **Graphe fortement connexe** : Un graphe orienté est fortement connexe si pour tout couple de sommets x, y il existe un chemin reliant x à y.

Remarque : la définition implique que si x et y sont deux sommets d'un graphe fortement connexe, alors il existe un chemin de x à y et un chemin de y à x.

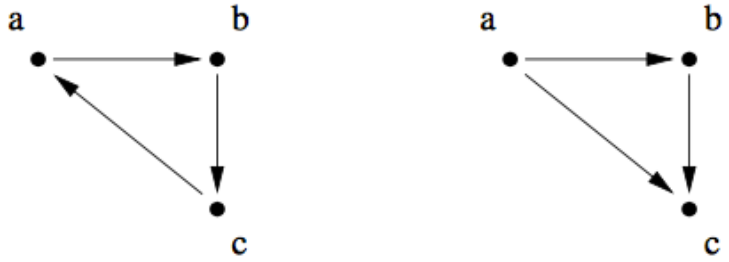


Figure 3* : *graphe fortement connexe

➲ **Sous-graphe :** On dit qu'un graphe $G0 = (S0,A0)$ est un **sous-graphe** de $G = (S,A)$ si $S0$ est inclus dans S et si $A0$ est inclus dans A.

2. *Arbres*

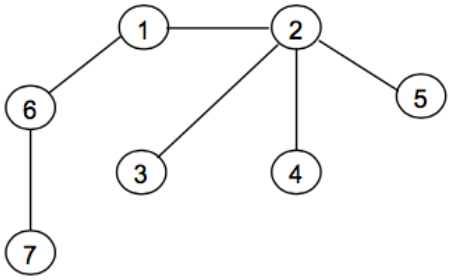


Figure 4* : *Exemple d'arbre

✹ ***Théorème (Propriétés des arbres)***

Soit G un graphe non orienté à n sommets. Les propositions suivantes sont équivalentes :

- Le graphe G est connexe et sans cycle
- Le graphe G est connexe et a n − 1 arêtes
- Le graphe G est connexe et la suppression de n'importe quelle arête le déconnecte
- Le graphe G est sans cycle et a n – 1 arêtes
- Le graphe G est sans cycle et l'ajout de n'importe quelle arête crée un cycle.
- Deux sommets quelconques de G sont reliés par un unique chemin élémentaire.

Arbre enraciné (ou arborescence) : Un arbre enraciné est un arbre hiérarchique dans lequel on peut établir des niveaux. Il ressemblera plus à un arbre généalogique tel qu'on le conçoit couramment.

La ***figure 5*** présente deux arbres qui ne diffèrent que s'ils sont considérés comme des arbres enracinés.

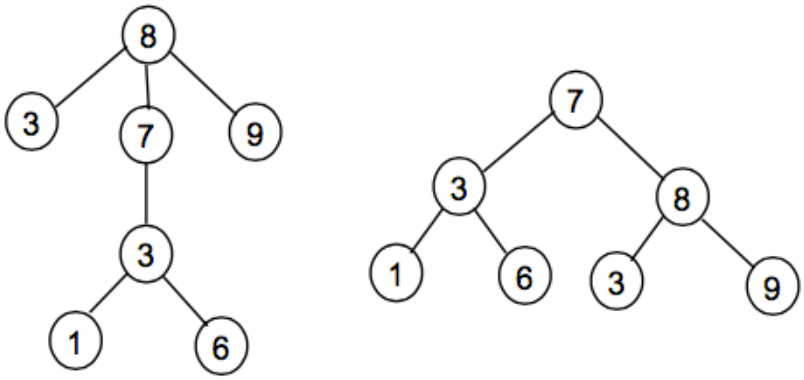


Figure 5 : Exemple de graphe différenciés par la racine

Ancêtre : Soit x un noeud (ou sommet) d'un arbre T de racine r. Un noeud quelconque y sur l'unique chemin allant de r à x est appelé **ancêtre** de x.

Père et fils : Si (y,x) est une arc alors y est le **père** de x et x est le **fils** de y. La racine est le seul noeud qui n'a pas de père.

Feuille ou nœud externe (ou terminal) : Un noeud sans fils est un noeud terminal ou une feuille. Un noeud qui n'est pas une feuille est un **nœud interne**. Si y est un ancêtre de x, alors x est un **descendant** de y.

Sous-arbre : Le sous-arbre de racine x est l'arbre composé des descendants de x, enraciné en x.

Degré d'un nœud : Le nombre de fils du noeud x est appelé le degré de x. Donc, suivant qu'un arbre (enraciné) est vu comme un arbre (enraciné) ou un graphe, le degré de ses sommets n'a pas la même valeur.

Profondeur d'un nœud : La longueur du chemin entre la racine r et le noeud x est la profondeur de x.

Profondeur de l'arbre : c'est la plus grande profondeur que peut avoir un nœud quelconque de l'arbre. Elle est dite aussi la hauteur de l'arbre.

Arbre ordonné : c'est un arbre enraciné dans lequel les fils de chaque noeud sont ordonnés entre eux. Les deux arbres de la ***figure 6*** sont différents si on les regarde comme des arbres ordonnés, mais ils sont identiques si on les regarde comme de simples arbres (enracinés).

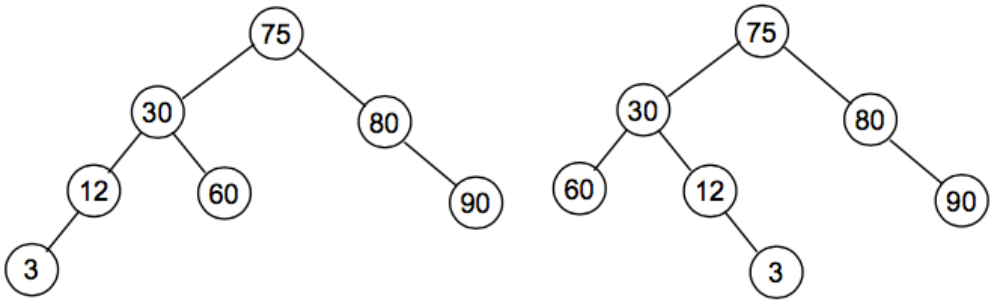


Figure 6 : Exemple d'arbres (enracinés) qui ne diffèrent que s'ils sont ordonnés

3. Arbre binaire

Un arbre binaire est tel que chaque nœud a au plus deux fils. Les arbres binaires se décrivent plus aisément de manière récursive. Un **arbre binaire** T est une structure définie sur un ensemble fini de noeuds et qui :

- Ne contient aucun noeud, ou
- Est formé de trois ensembles disjoints de noeuds : une racine, un arbre binaire appelé son **sous-arbre gauche** et un arbre binaire appelé son **sous-arbre droit**.

Dans un arbre binaire, si un noeud n'a qu'un seul fils, la position de ce fils qu'il soit **fils gauche** ou **fils droit** est importante.

Arbre binaire complet : Dans un arbre binaire complet chaque noeud est soit une feuille, soit de degré deux. Aucun noeud n'est donc de degré un. Un arbre k-**aire** est une généralisation de la notion d'arbre binaire où chaque noeud est de degré au plus k et non plus simplement de degré au plus 2.

4. Parcours d'un graphe

4.1. Parcours des arbres

Nous ne considérons ici que des **arbres ordonnés**. Les parcours permettent d'effectuer tout un ensemble de traitement sur les arbres.

4.1.1. Parcours en profondeur

Dans un *parcours en profondeur*, on descend d'abord le plus profondément possible dans l'arbre puis, une fois qu'une feuille a été atteinte, on remonte pour explorer les autres branches en commençant par la branche « la plus basse » parmi celles non encore parcourues. Les fils d'un noeud sont bien évidemment parcourus suivant l'ordre sur l'arbre.

```
Algorithme ParPro(A)
    si A n'est pas réduit à une feuille alors
        pour tous les fils u de racine(A) faire
            ParPro(u)
End_PP
```

4.1.2. Parcours en largeur

Dans un *parcours en largeur*, tous les noeuds à une profondeur *i* doivent avoir été visités avant que le premier noeud à la profondeur *i*+1 ne soit visité. Un tel parcours nécessite l'utilisation d'une file d'attente pour se souvenir des branches qui restent à visiter.

```
Algorithme Parcours_Largeur(A)
    F : File d'attente
    F.Put (racine(A))
    tantque F != vide faire
        u=F.Get()
        Afficher (u)
        pour « chaque fils v de » u faire
            F.Put (v)
End_PL
```

4.2. Parcours des graphes

Le parcours des graphes est un peu plus compliqué que celui des arbres. En effet, les graphes peuvent contenir des cycles et il faut éviter de parcourir indéfiniment ces cycles. Pour cela, il suffit de colorier les sommets du graphe.

- Initialement les sommets sont tous blancs,
- Lorsqu'un sommet est rencontré pour la première fois il est peint en gris,
- Lorsque tous ses successeurs dans l'ordre de parcours ont été visités, il est repeint en noir.

4.2.1. Parcours en profondeur

```
Algorithme PP(G)
        pour chaque sommet u de G faire
                couleur[u]=Blanc
        Fin pour
        pour chaque sommet u de G faire
                Si couleur[u] = Blanc alors
                        VisiterPP(G, u, couleur)
                Fin Si
        Fin Pour
Fin_PP
Algorithme VisiterPP(G, s, couleur)
        couleur[s]=Gris
        pour chaque voisin v de s faire
                Si couleur[v] = Blanc alors
                        VisiterPP(G, v, couleur)
        couleur[s]=Noir
Fin_VisiterPP
```

4.2.2. Parcours en largeur

Dans un *parcours en largeur d'abord*, tous les noeuds à une profondeur *i* doivent avoir été visités avant que le premier noeud à la profondeur *i*+1 ne soit visité. Un tel parcours nécessite l'utilisation d'une file d'attente pour se souvenir des branches qui restent à visiter.

```
Algorithme PL(G, s)
        F : File d'attente
        pour chaque sommet u de G faire
                couleur[u] = Blanc
        Fin Pour
        couleur[s]=Gris
        F .Put(s)
        Tantque F != Vide faire
                u=F.Get()
                pour chaque voisin v de u faire
                        Si couleur(v) = Blanc faire
                                couleur(v)= Gris
                                F.Put(v)
                        Fin SI
                Fin Pour
                Couleur(u)= Noir
        Fin Tantque
Fin_PL
```

II. Arbres Binaires et Arbres Binaires de Recherche

1. Définition

Un arbre ou arborescence binaire est un graphe qui admet une racine et sans cycle et dont chaque nœud admet deux fils : fils droit et fils gauche.

On peut représenter un arbre binaire sous la forme d'une liste ou une classe.

2. Implémentation d'un arbre binaire

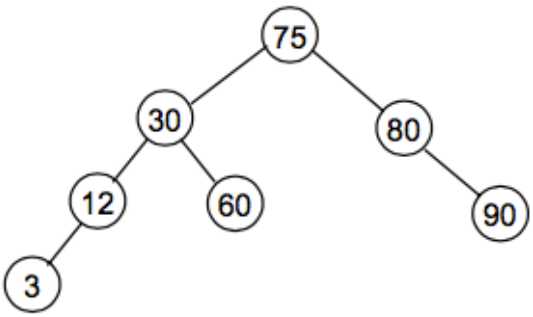
2.1. Définition des structures de données :

Chaque nœud sera composé de 3 champs : un champ pour la donnée, et deux champs pointeurs l'un pour le fils droit et l'autre pour le fils gauche. La structure de donnée arbre contient un seul pointeur 'racine' qui donne le premier élément de l'arbre.

```
class ArbreBinaire:
   self.racine=None
   self.gauche=None
   self.droite=None
   def __init__(self, elms):
     if (isinstance(elms,tuple) or isinstance(elms,list)):
        self.racine=elms[0]
        self.gauche=ArbreBinaire(elms[1])
        self.droite=ArbreBinaire(elms[2])
      else:
        self.racine=elms
```

On va définir un objet arbre à l'aide d'une liste :

```
L=[75,[30,[12,3,None],60],[80,None,90]]
Arbre=ArbreBinaire(L)
```



2.2. Parcours d'un arbre binaire :

Il existe 3 méthodes de parcours d'un arbre binaire :

- Parcours préfixe : père, fils gauche, fils droit
- Parcours infixe : fils gauche, père, fils droit
- Parcours postfixe : fils gauche, fils droit, père

On ne peut réaliser ces différents parcours qu'en utilisant une pile, puisqu'on est obligé de commencer le parcours par la racine et d'empiler les nœuds dont on n'a pas encore rencontré le fils gauche. La pile peut être utilisée d'une manière explicite ou bien au moyen d'une procédure récursive.

2.2.1. Parcours préfixe :

L'algorithme récursif et son implémentation en python de ce parcours sont donnés dans le tableau suivant :

Algorithme	***Programme python***

Algorithme Préfixe(struct node A)
si (A != Nul) alors
AfficheRacine(A)
Préfixe (FilsGauche(A))
Préfixe (FilsDroit(A))
End_If
End_Préfixe
	def Prefixe(self,arbre):
	if arbre==None:
	print(end='')
	else:
	print(arbre.racine,end=',')
	self.Prefixe(arbre.gauche)
	self.Prefixe(arbre.droite)

2.2.2. Parcours infixe :

L'algorithme récursif et son implémentation en python de ce parcours sont donnés dans le tableau suivant :

Algorithme	***Programme python***

Algorithme Infixe(struct node A)
si (A != Nil) alors
Infixe (FilsGauche(A))
AfficheRacine(A)
Infixe (FilsDroit(A))
End_If
End_Préfixe
	def Infixe (self,arbre):
	if arbre==None:
	print(end='')
	else :
	self. Infixe (arbre.gauche)
	print(arbre.racine, end=',')
	self. Infixe (arbre.droite)

2.2.3. Parcours postfixe

L'algorithme récursif et son implémentation en python de ce parcours sont donnés dans le tableau suivant :

Algorithme	***Programme python***

Algorithme Postfixe (struct node A)
si (A != Nil) alors
Postfixe (FilsGauche(A))
Postfixe (FilsDroit(A))
AfficheRacine(A)
End_If
End_Préfixe
	def Postfixe (self,arbre):
	if arbre==None:
	print(end='')
	else :
	self. Postfixe (arbre.gauche)
	self. Postfixe (arbre.droite)
	print(arbre.racine, end=',')

3. *Arbre Binaire de Recherche :*

Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud est supérieur à son fils gauche et inférieur à son fils droit et il n'y a pas de nœuds égaux.

Un arbre binaire de recherche est intéressant puisqu'il est toujours possible de connaître dans quelle branche de l'arbre se trouve un élément et de proche en proche le localiser dans l'arbre. On peut aussi utiliser un arbre binaire de recherche pour ordonner une liste d'éléments.

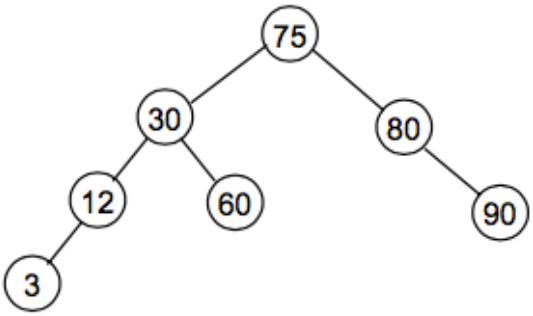


Figure 7* : *arbre binaire de recherche

Pour utiliser un arbre binaire de recherche il va falloir résoudre trois problèmes :

⇒ ***Comment permettre la multiplicité des éléments ?***

Solution : Ajouter un champ dans le nœud pour indiquer la fréquence de la valeur.

⇒ ***Comment construire un arbre binaire de recherche ou comment insérer un nouvel élément dans l'arbre ?***

Solution : Il faut descendre dans l'arbre jusqu'à trouver un pointeur égal à Null.

⇒ ***Comment retrouver l'ordre des éléments ?***

Solution : Effectuer un parcours infixe.

Exemple : *Construisons un arbre binaire de recherche pour la séquence de valeurs suivantes : 75, 30, 60, 12, 80, 3, 90*

Application : *Lire une liste de notes dans le désordre, construire un arbre binaire de recherche et imprimer les notes dans l'ordre 3 croissant.*

3.1. Algorithme de recherche d'un élément :

La recherche d'un élément dans un arbre binaire de recherche est très facile si on profite du caractère récursif d'un arbre binaire.

```
Algorithme Chercher (Parent P , élément X)
        Si ( P = Null) alors
                Return 0
        Sinon  Si ( P.racine = X ) alors
                Return 1
        Sinon  Si (X < P.racine ) alors
                Return Chercher(P.Gauche, X)
        Sinon
                Return Chercher (P.Droite, X)
        Fin SI
End_Chercher
```

3.2. Algorithme d'insertion d'un élément :

L'élément à ajouter est inséré là où on l'aurait trouvé s'il avait été présent dans l'arbre. L'algorithme d'insertion recherche donc l'élément dans l'arbre et, quand il aboutit à la conclusion que l'élément n'appartient pas à l'arbre (il aboutit à la terre), il insère l'élément comme fils du dernier noeud visité.

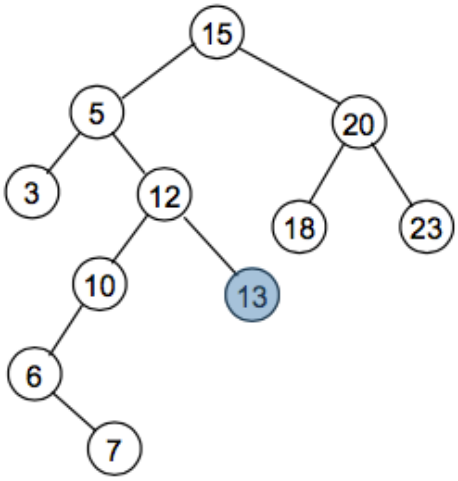
```
Algorithme Insérer (Parent P, élément X)
        Si (P = null) alors
                « ajouter un nœud pour X à cet endroit»
        Sinon Si (P.data = X)
                Incrémenter nombre d'exemplaires
        Sinon  Si(X> P.data)
                Insérer (P.Fils_droit, X)
        Sinon
                Insérer (P.Fils gauche, X)
FinInsérer
```

3.3. Algorithme de suppression d'un élément :

La suppression d'un élément est plus compliquée que l'insertion, car si cet élément est un père à qui confier ses fils ? Il existe trois cas possibles :

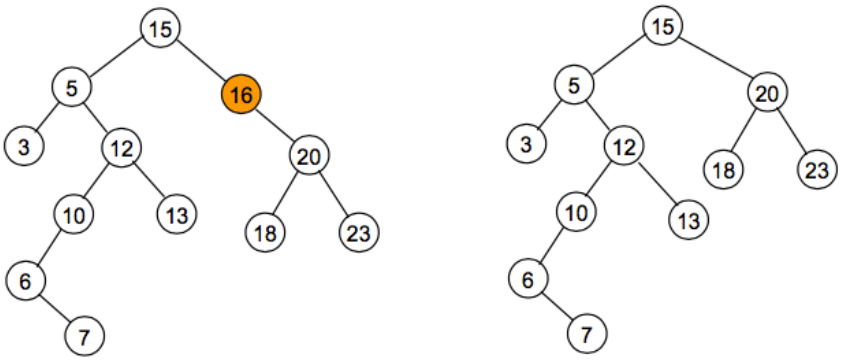
3.3.1. 1èr cas :

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



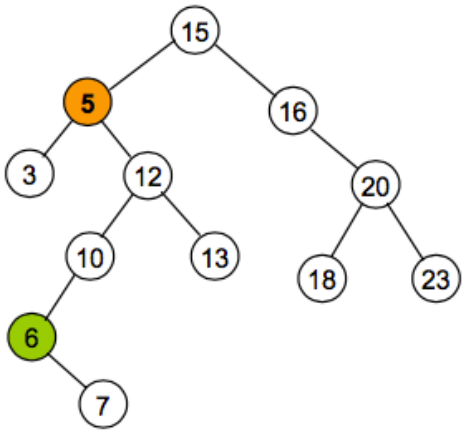
3.3.2. 2ème cas

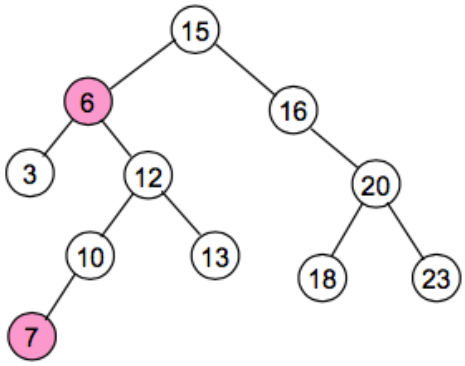
L'élément a un fils unique : on supprime le nœud et on relie son fils à son père



3.3.3. 3ème cas

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





4. Complexité :

Si *h* est la hauteur de l'arbre, on peut aisément montrer que tous les algorithmes précédents ont une complexité en $O(h)$. Malheureusement, un arbre binaire quelconque à *n* noeuds a une hauteur comprise, en ordre de grandeur, entre $\log_2 n$ et *n*. Pour éviter les cas les plus pathologiques, on s'intéresse à des arbres de recherches *équilibrés*.