# Structures de données

# Structure de données

Une structure de données est une organisation logique des données permettant de simplifier ou d'accélérer leur traitement.

# Pile

# Pile ( stack)

- « dernier arrivé, premier sorti » (ou LIFO pour Last In, First Out)
- Le fonctionnement est donc celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

# Pile ( stack)

Primitives

- push/empiler  : ajoute un élément sur la pile.
- « pop/dépiler » : enlève un élément de la pile et le renvoie. En anglais : « Pop »
- « isEmpty » : renvoie vrai si la pile est vide, faux sinon

# Pile ( stack)

- Applications
- Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « Afficher la page précédente ».
- L'évaluation des expressions mathématiques en notation post-fixée (ou polonaise inverse) utilise une pile.
- La fonction « Annuler la frappe » (en anglais « Undo ») d'un traitement de texte

# Pile ( stack)

- Applications
- mémorise les modifications apportées au texte dans une pile.
- Un algorithme de recherche en profondeur utilise une pile pour mémoriser les nœuds visités.
- Les algorithmes récursifs admis par certains langages (LISP, Algol, Pascal, C, etc.)
- utilisent implicitement une pile d'appel.  On peut donc toujours simuler la récursion en créant les primitives de gestion d'une pile.

# Python:stack

```
# Using Python list directly as stack
stack = []
stack.append(1)  # push
stack.append(2)
stack.append(3)
print(stack.pop())  # 3
print(stack[-1])    # 2 (peek)
```

# Stack

```
from collections import deque
stack = deque()
stack.append('item1')      # push
stack.append('item2')
stack.append('item3')
print(stack.pop())         # item3 (pop)
print(stack.pop())         # item2
print(stack.pop())         # item1
```

# File ( Queue)

Une file (queue  ) est une structure de données basée sur le principe « Premier entré, premier sorti »,  FIFO (First In, First Out), ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés.

# File ( Queue)

- **Primitives**
- « enqueue/ajouter » : ajoute un élément dans la file. Terme anglais correspondant : «  »
- « dequeue/enlever » : renvoie le prochain élément de la file, et le retire de la file.  correspondant : « dequeue »
- « isEmpty » : renvoie VRAI si la file est vide, FAUX sinon

# File ( Queue)

**Applications**

- En général, on utilise des files pour mémoriser temporairement des transactions qui doivent attendre pour être traitées.
- Les serveurs d'impression, qui doivent traiter les requêtes dans l'ordre dans lequel elles arrivent.
- dans un système d'exploitation, qui doivent accorder du temps-machine à chaque tâche.
- Un algorithme de parcours en largeur utilise une file pour mémoriser les nœuds visités.

# File ( Queue)

```
# Using Python list directly as queue
queue = []
queue.append(1)  # enqueue
queue.append(2)
queue.append(3)
print(queue.pop(0))  # 1 (dequeue)
print(queue[0])      # 2 (front/peek)
```

# File ( Queue)

```
# Using Python list directly as queue
queue = []
queue.append(1)  # enqueue
queue.append(2)
queue.append(3)
print(queue.pop(0))  # 1 (dequeue)
print(queue[0])      # 2 (front/peek)
```

# File ( Queue)

```
from collections import deque
queue = deque()
queue.append('task1')      # enqueue
queue.append('task2')
queue.append('task3')
print(queue.popleft())     # task1 (dequeue)
print(queue.popleft())     # task2
print(queue.popleft())     # task3
```

# File (Priorty  Queue)

**Files à priorités**

Dans une file à priorités, on peut effectuer trois opérations :

- insérer un élément
- supprimer le plus grand élément
- tester si la file à priorités est vide ou pas
- Les principales implémentations de ces files à priorités sont le tas , le tas binomial et le tas de Fibonacci.

# Python: PriorityQueue

```
from queue import PriorityQueue

# Create priority queue

pq = PriorityQueue()

# Put items with priority

pq.put((2, "task B"))

pq.put((1, "task A"))

pq.put((3, "task C"))

# Get items (lowest priority number comes first)

print(pq.get())  # (1, 'task A')

print(pq.get())  # (2, 'task B')

print(pq.get())  # (3, 'task C')
```

# Python: heapq

```
Import heapq

# Create priority queue

pq = []

# Push items with priority

heapq.heappush(pq, (2, "medium task"))

heapq.heappush(pq, (1, "high priority task"))

heapq.heappush(pq, (3, "low priority task"))

# Pop items (lowest priority number comes first)

print(heapq.heappop(pq))  # (1, 'high priority task')

print(heapq.heappop(pq))  # (2, 'medium task')

print(heapq.heappop(pq))  # (3, 'low priority task')
```

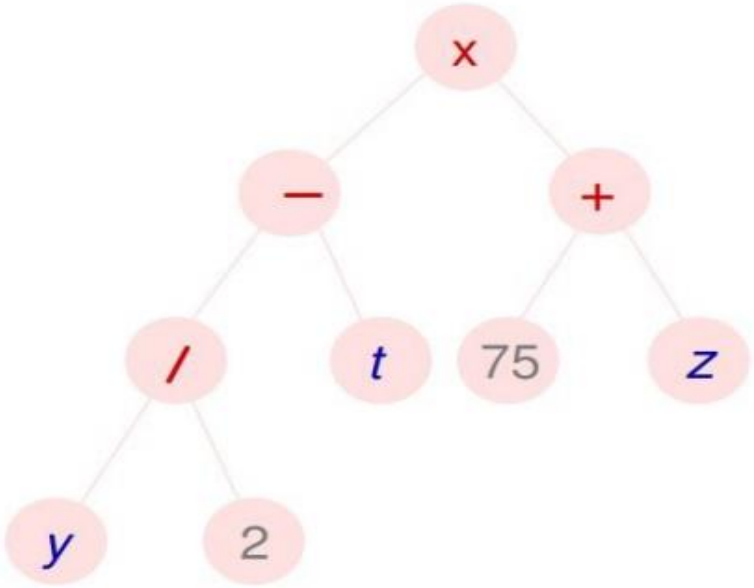
# Arbres

# Arbre(Tree)

- Un arbre est un graphe sans cycle, où des nœuds sont reliés par des arêtes. On distingue trois sortes de nœuds :
- les nœuds internes, qui ont des fils ;
- les feuilles, qui n'ont pas de fils ;
- la racine de l'arbre, qui est l'unique nœud ne possédant pas de père.
- Traditionnellement, on dessine toujours la racine en haut et les feuilles en bas

# Arbre(Tree)

- L'arbre binaire ci-dessous représente l'expression arithmétique (y/2 - t) x (75 + z).



# Arbre(Tree)

- La profondeur d'un nœud est la distance, i.e. le nombre d'arêtes, de la racine au nœud. La hauteur d'un arbre est la plus grande profondeur d'une feuille de l'arbre.
- La taille d'un arbre est son nombre de  nœuds (en comptant les feuilles ou non).
- Les arbres sont en fait rarement utilisés en tant que tels, mais de nombreux types d'arbres avec
- une structure plus restrictive existent et permettent alors des recherches rapides et efficaces.

# Arbre(Tree)

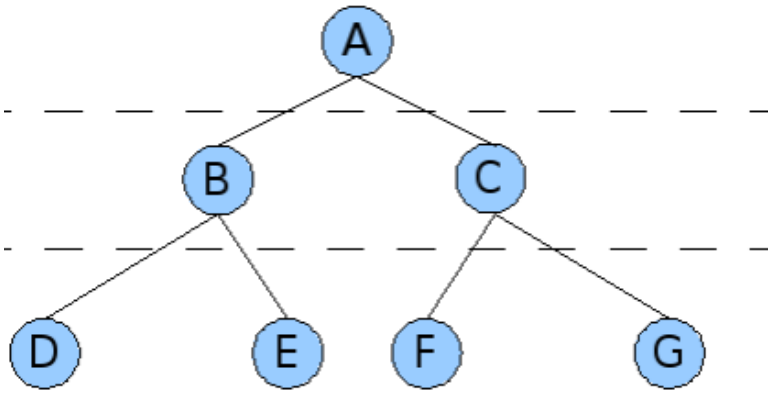
## Parcours en largeur/Breadth-First Search (BFS)

- Le parcours en largeur correspond à un parcours par niveau de nœuds de l'arbre. Un niveau est un ensemble de nœuds ou de feuilles situés à la même profondeur.

# Arbre(Tree)

## BFS

Ainsi, si l'arbre ci-contre est utilisé, le parcours sera A, B, C, D, E, F, G.



# Arbre(Tree)

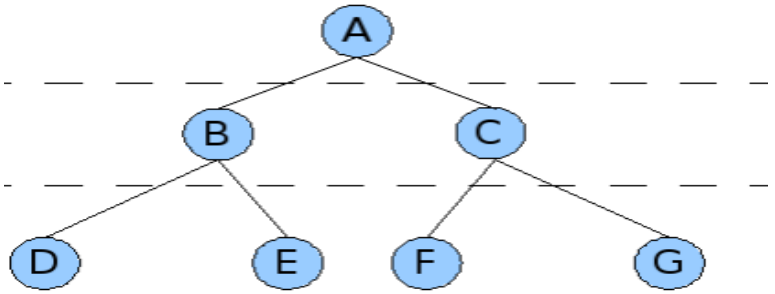
## Parcours en profondeur/ Depth-First Search (DFS)

Le parcours en profondeur est un parcours récursif sur un arbre. Il existe trois ordres pour cette méthode de parcours. Le parcours en profondeur préfixé est le plus courant.

# Arbre(Tree)

## Parcours en profondeur  préfixé(preorder)

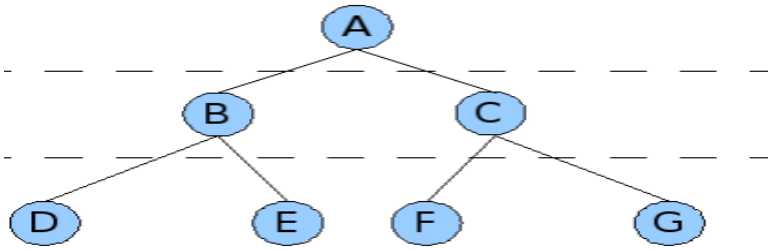
Dans ce mode de parcours, le nœud courant est traité avant le traitement des nœuds gauche et droit. Ainsi, si l'arbre précédent est utilisé, le parcours sera A, B, D, E, C, F, G.



# Arbre(Tree)

- **Parcours en profondeur suffixé(postorder)**

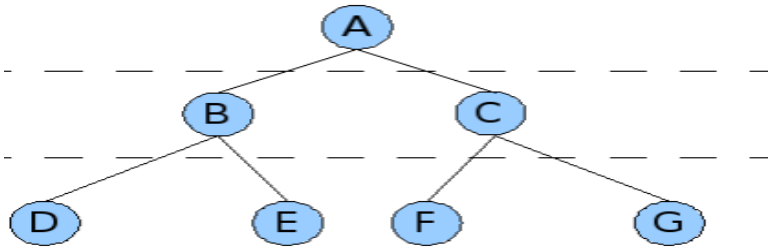
- Dans ce mode de parcours, le nœud courant est traité après le traitement des nœuds gauche et droit. Ainsi, si l'arbre précédent est utilisé, le parcours sera D, E, B, F, G, C, A.



# Arbre(Tree)

**Parcours en profondeur infixé(inordre)**

Dans ce mode de parcours (qui n'est applicable qu'à des arbres binaires), le nœud courant est traité entre le traitement des fils gauche et droit. Ainsi, si l'arbre précédent est utilisé, le parcours sera : D, B, E, A, F, C puis G.



# Arbre(Tree)

## Arbres binaires(binary  tree)

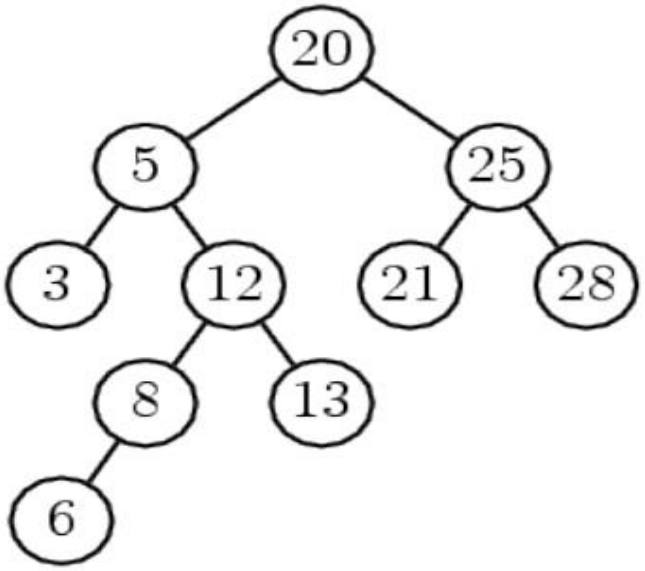
- Dans un arbre binaire, chaque nœud possède au plus deux fils, habituellement appelés« gauche » et « droit ».
- Du point de vue des fils, l'élément dont ils sont issus au niveau supérieur est logiquement appelé père.

# Arbre(Tree)

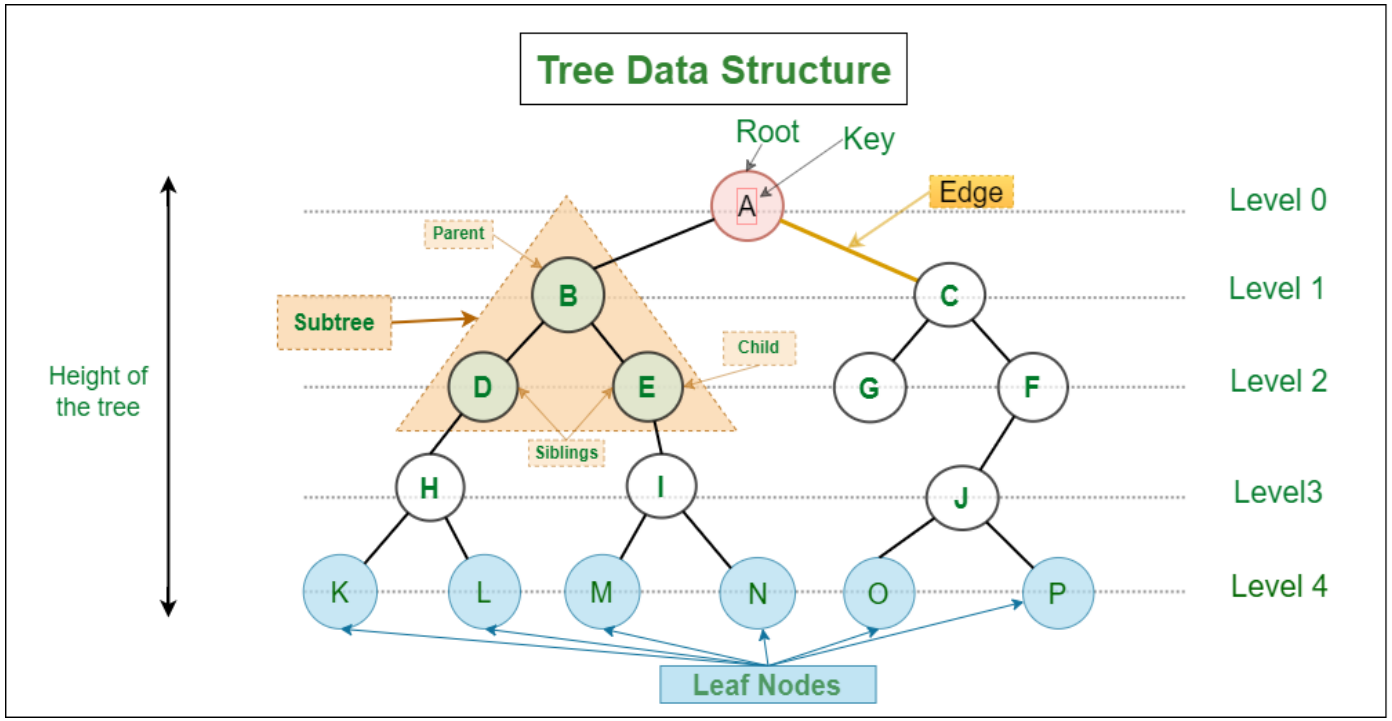
**Types d'arbres binaires**

- Un arbre binaire entier est un arbre dont tous les nœuds possèdent zéro ou deux fils.
- Un arbre binaire parfait est un arbre binaire entier dans lequel toutes les feuilles sont à la même hauteur.
- L'arbre binaire parfait est parfois nommé arbre binaire complet. Cependant certains définissent un arbre binaire complet comme étant un arbre binaire entier dans lequel les feuilles ont pour profondeur n ou n-1 pour un n donné.

# Arbre(Tree)



# Arbre(Tree)


Tree Data Structure
Root
Key
Edge
A
B
C
D
E
G
F
H
I
J
K
L
M
N
O
P
Parent
Subtree
Child
Siblings
Height of the tree
Level 0
Level 1
Level 2
Level3
Level 4
Leaf Nodes

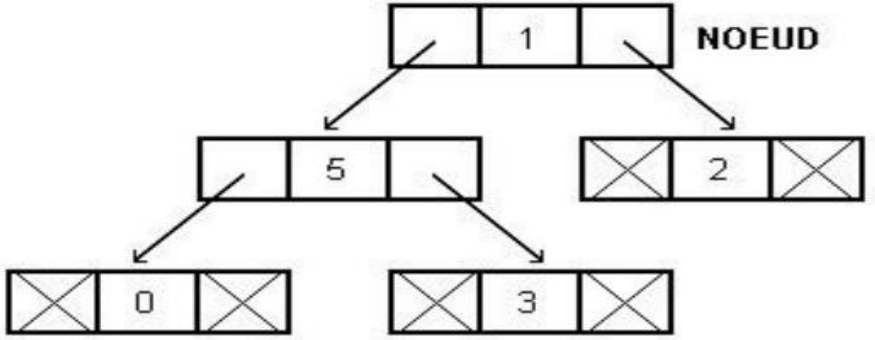
# Arbre(Tree)

Méthodes pour stocker des arbres binaires

- Structure à 3 nœuds
- Tableau

# Arbre(Tree)

**Structure à 3 nœuds**



# Arbre(Tree)

## Structure à 3 nœuds

**Avantages**

- Conçu pour contenir un nombre variable de nœuds.
- Pas de gaspillage de mémoire.

**Inconvénients**

- Implémentation délicate à réaliser
- Pas d'accès direct à un nœud de l'arbre

# Arbre(Tree)

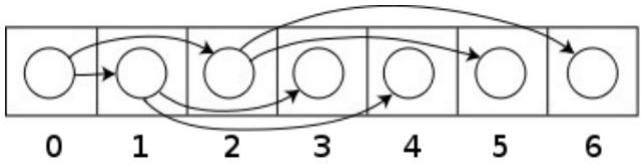
## Tableau

- Les arbres binaires peuvent aussi être rangés dans des tableaux, et si l'arbre est un arbre binaire  complet, cette méthode ne gaspille pas de place, et la donnée structurée résultante est appelée un tas.
- Dans cet arrangement compact, un nœud a un indice i, et ses fils se trouvent aux indices 2i+1 et 2i+2, tandis que son père se trouve à l'indice floor((i-1)/2), s'il existe.

# Arbre(Tree)

## Tableau

- La racine (nœud 0) a pour fils les nœuds 1 et 2. Le nœud 1 a pour fils 3 et 4, etc




- **Avantages**
- Implémentation facile à réaliser.
- Possibilité d'accès direct à un nœud de l'arbre (un seul accès en mémoire).
- **Inconvénients**
- Conçu pour contenir un nombre fixe de nœuds.
- Si l'arbre est profond mais contient peu de nœuds, il se produit un gaspillage important de mémoire.

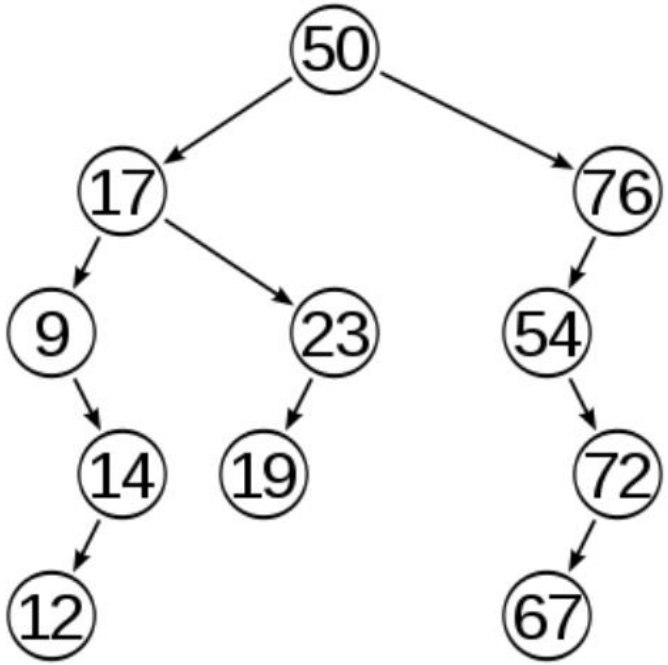
# Arbre(Tree)

**Arbres binaires de recherche(binary search tree)**

- Un arbre binaire de recherche (BST) est un arbre binaire dans lequel chaque nœud possède une étiquette, telle que chaque nœud du sous-arbre gauche ait une étiquette inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une étiquette supérieure ou égale à celle-ci
- Selon la mise en œuvre de l'ABR, on pourra interdire ou non des étiquettes de valeur égale.

# Arbre(Tree)

- Essayez d’explorer cet arbre selon un parcours en profondeur infixé ?



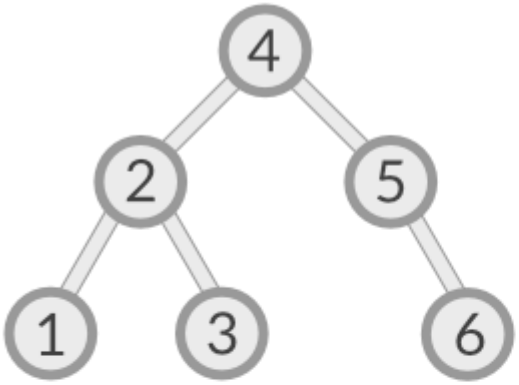
# Arbre(Tree)

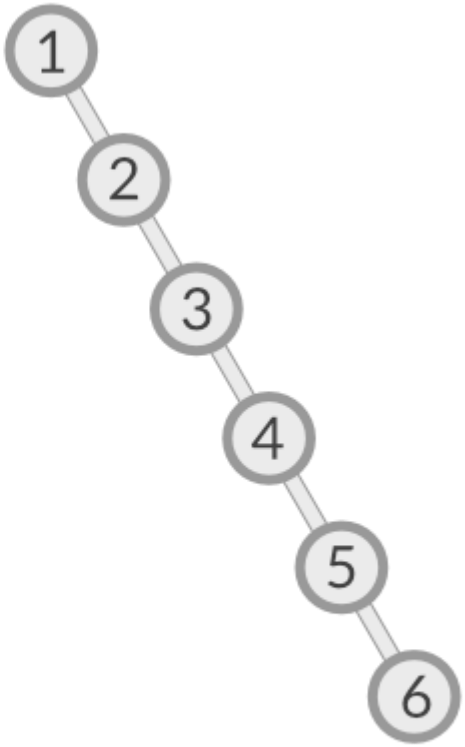
- **Recherche**

- La recherche dans un arbre binaire d'un nœud ayant une étiquette particulière est un procédé récursif. On commence par examiner la racine. Si l'étiquette de la racine est l'étiquette recherchée , l'algorithme se termine et renvoie la racine. Si l'étiquette cherchée est inférieure, alors elle est dans le sous-arbre gauche, sur lequel on effectue alors récursivement la recherche. De même, si l'étiquette recherchée est strictement supérieure à l'étiquette de la racine, la recherche continue sur le sous-arbre droit. Si on atteint une feuille dont l'étiquette n'est pas celle recherchée, on sait alors que cette étiquette n'est pas dans l'arbre.

- Cette opération requiert un temps en O(log(n)) dans le cas moyen, mais O(n) dans le pire des cas où l'arbre est complètement déséquilibré où chaque père a un seul fils.

# Arbre(Tree)



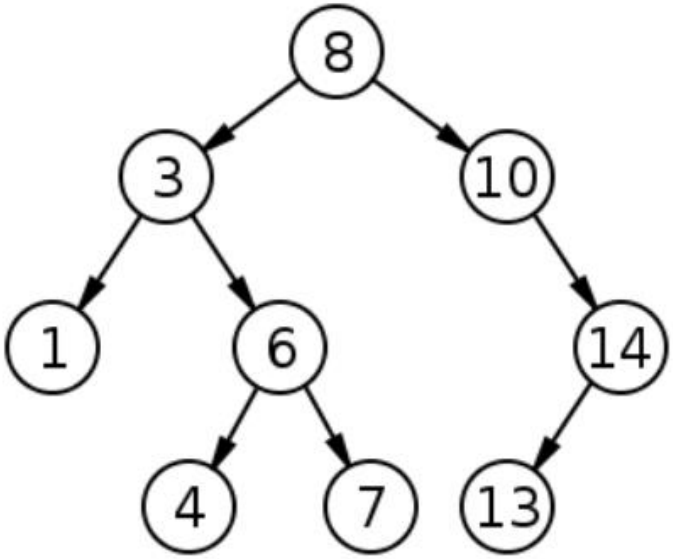
# Arbre(Tree)

**Insertion**

- L'insertion d'un nœud commence par une recherche : on cherche l'étiquette du nœud à insérer. Si on la trouve, on ne fait rien. Sinon, lorsqu'on ne peut plus descendre dans l'arbre, cela signifie qu'on a trouvé le père. On insère le nœud en comparant son étiquette à celle de son père : si elle est inférieure, le nouveau nœud sera son fils gauche ; sinon il sera son fils droit. Ainsi, chaque nœud ajouté sera une feuille.
- La complexité de l'insertion est O(log(n)) dans le cas moyen et O(n) dans le pire des cas

# Arbre(Tree)

- Insérez dans l'arbre binaire de recherche ci-après les valeurs 11 et 5.



# Arbre(Tree:Suppression)

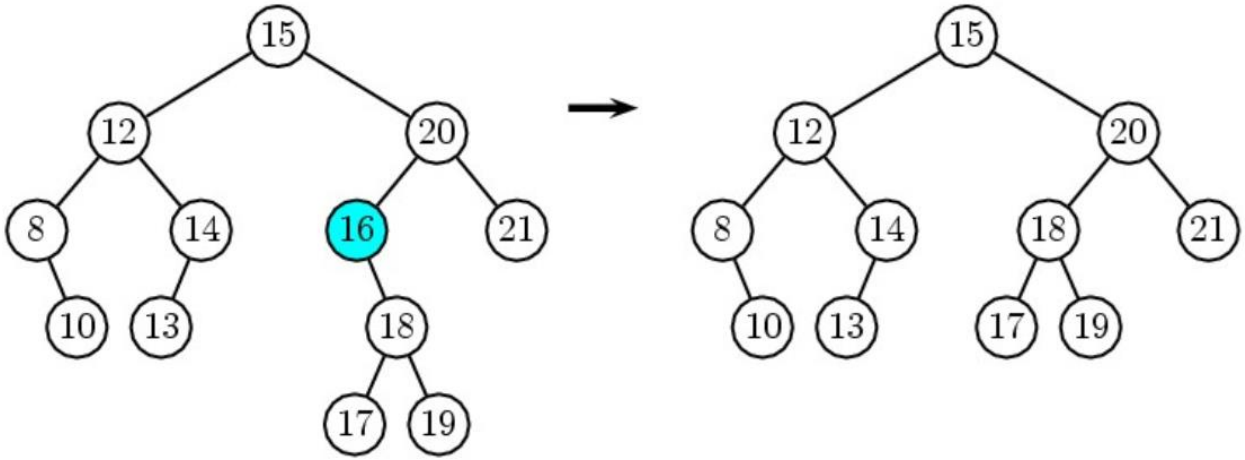
**Suppression**

- Plusieurs cas sont à considérer, une fois que le nœud à supprimer a été trouvé :
  - Suppression d'une feuille
  - Suppression d'un nœud avec un seul fils
  - Suppression d'un nœud avec deux fils

# Arbre(Tree:Suppression)

## Suppression d'un nœud avec un seul fils

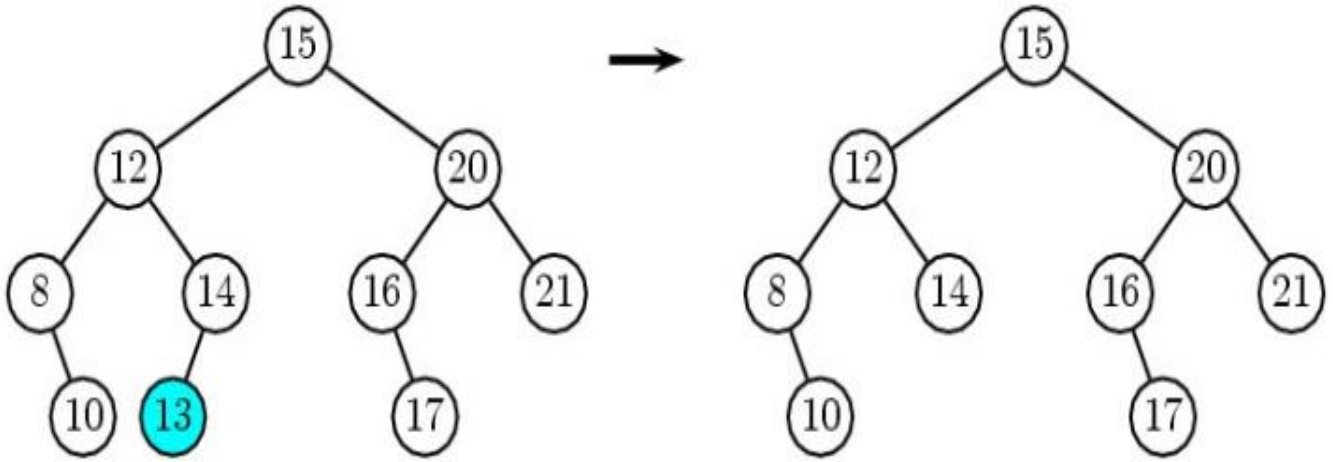
- On l'enlève de l'arbre et on le remplace par son fils



# Arbre(Tree:Suppression)

**Suppression d'une feuille**

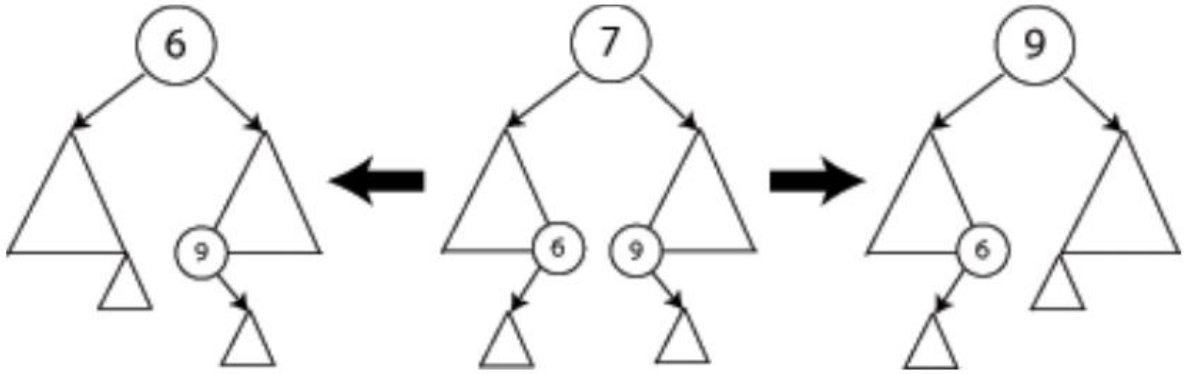
- Il suffit de l'enlever de l'arbre étant donné qu'elle n'a pas de fils.



# Arbre(Tree:Suppression)

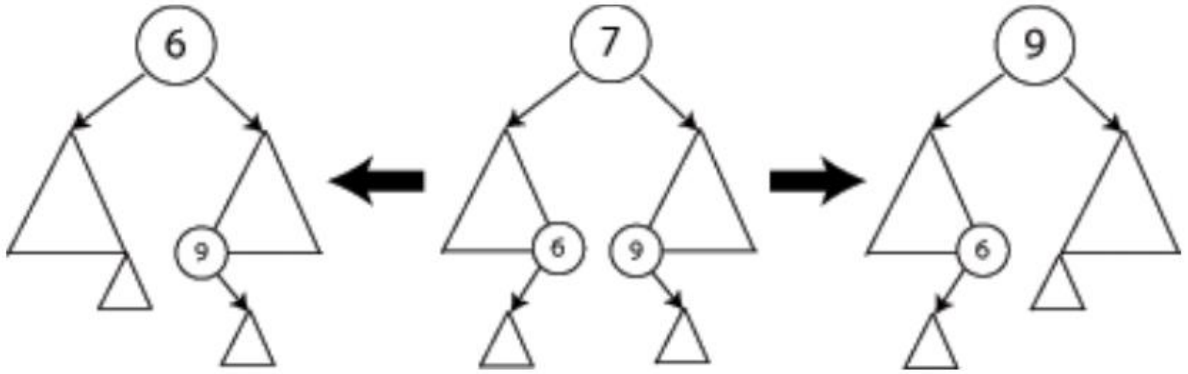
**Suppression d'un nœud avec deux fils**

- Supposons que le nœud à supprimer soit appelé N (le nœud de valeur 7). On le remplace alors par son successeur le plus proche, donc le nœud le plus à gauche du sous-arbre droit ( le nœud de valeur 9) ou son plus proche prédécesseur, donc le nœud le plus à droite du sous-arbre gauche (le nœud de valeur 6).



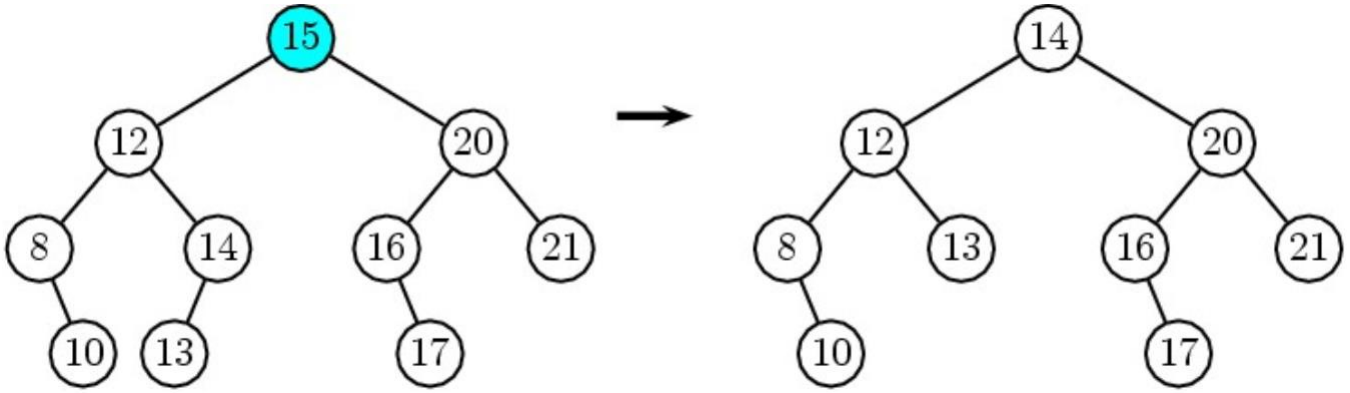
# Arbre(Tree:Suppression)

- Cela permet de garder une structure d'arbre binaire de recherche. Puis on applique à nouveau la procédure de suppression à N, qui est maintenant une feuille ou un nœud avec un seul fils. Cela permet de garder une structure d'arbre binaire de recherche. Puis on applique à nouveau la procédure de suppression à N, qui est maintenant une feuille ou un nœud avec
- un seul fils



# Arbre(Tree:Suppression)

- nouveau la procédure de suppression à N, qui est maintenant une feuille ou un nœud avec un seul fils.



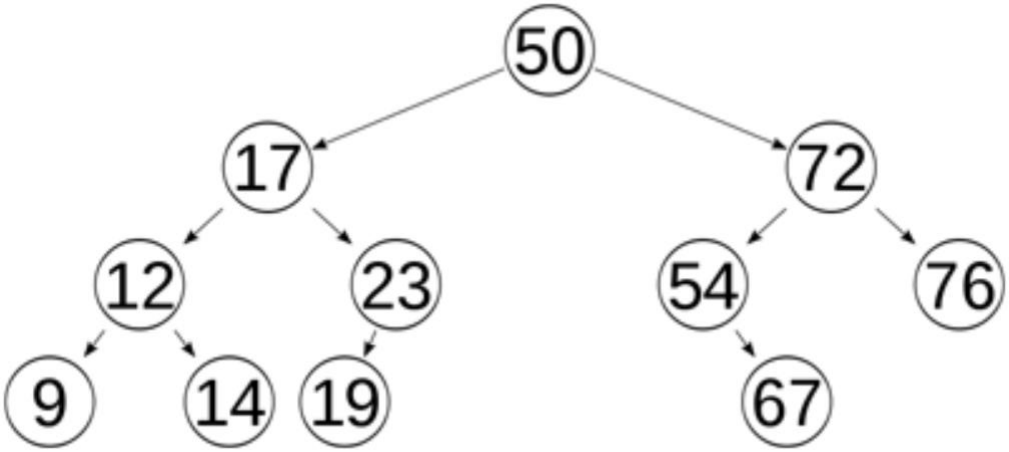
# AVL

**Arbres AVL**

La dénomination « arbre AVL » provient des noms de ses deux inventeurs russes, Georgy Maximovich Adelson-Velsky et Evgenii Mikhailovich Landis.

Les arbres AVL ont été historiquement les premiers arbres binaires de recherche automatiquement équilibrés. Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un. La recherche, l'insertion et la suppression sont toutes en O(log(n)) dans le pire des cas.

# AVL



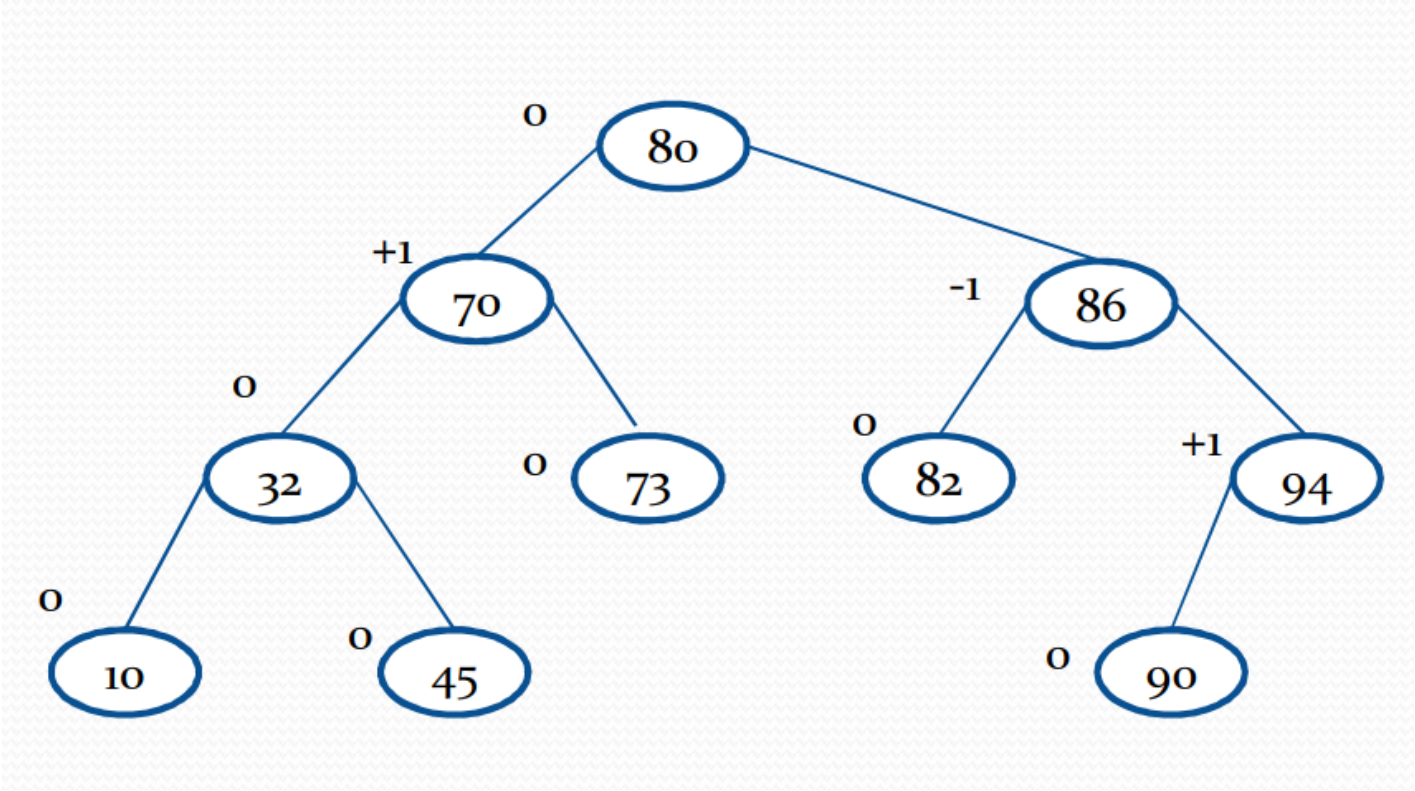
# AVL

- Le facteur d'équilibrage (balance Factor)d'un nœud est la différence entre la hauteur de son sous-arbre droit et celle de son sous-arbre gauche. Un nœud dont le facteur d'équilibrage est +1, 0, ou -1 est considéré comme équilibré.

# AVL

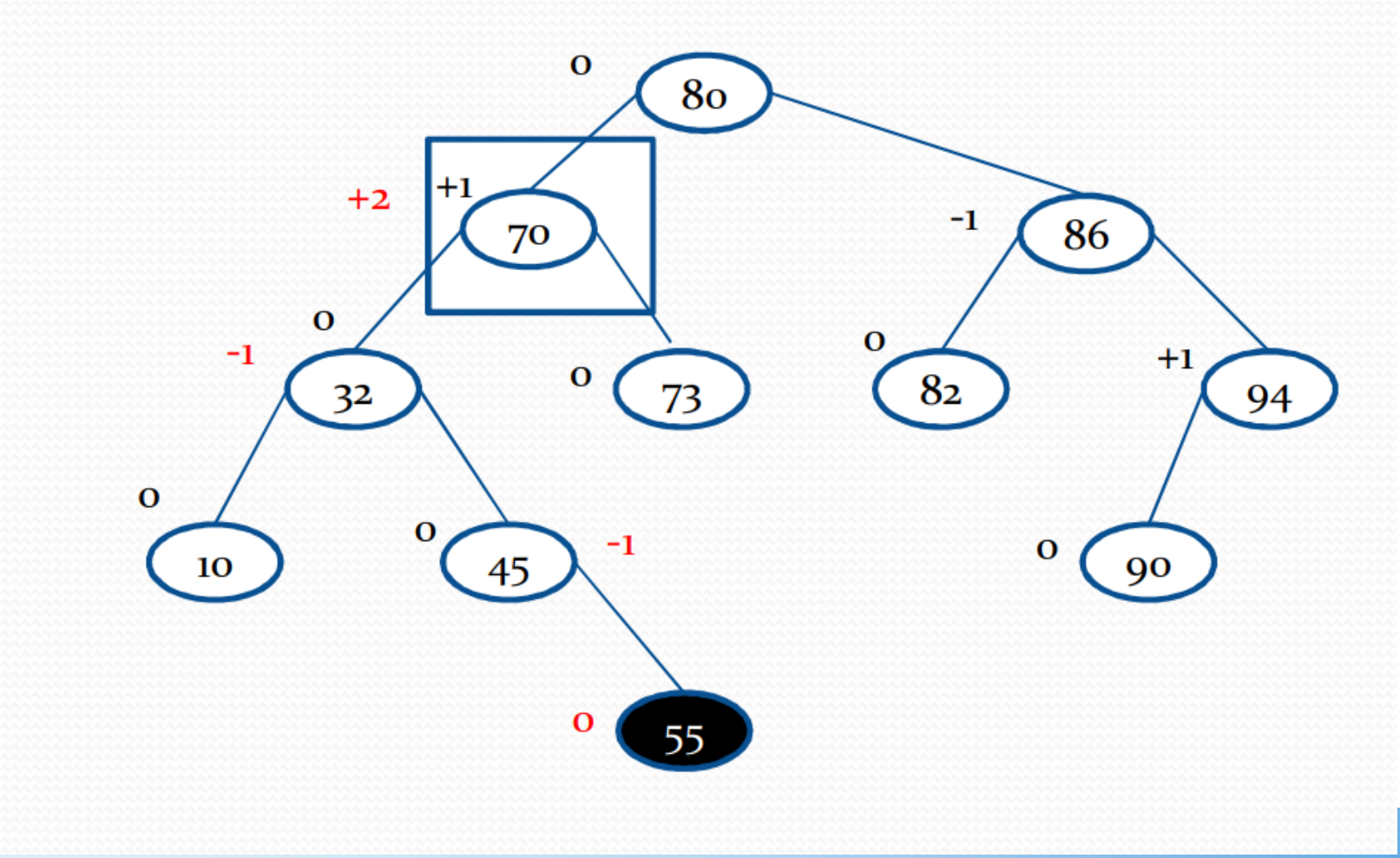
- Ajouter un champ balance (facteur d'équilibrage ) au niveau de chaque

Nœud | h(fg(n) )– h(fd(n)) | <= 1



# AVL

- Arbres AVL (Cas de déséquilibre)



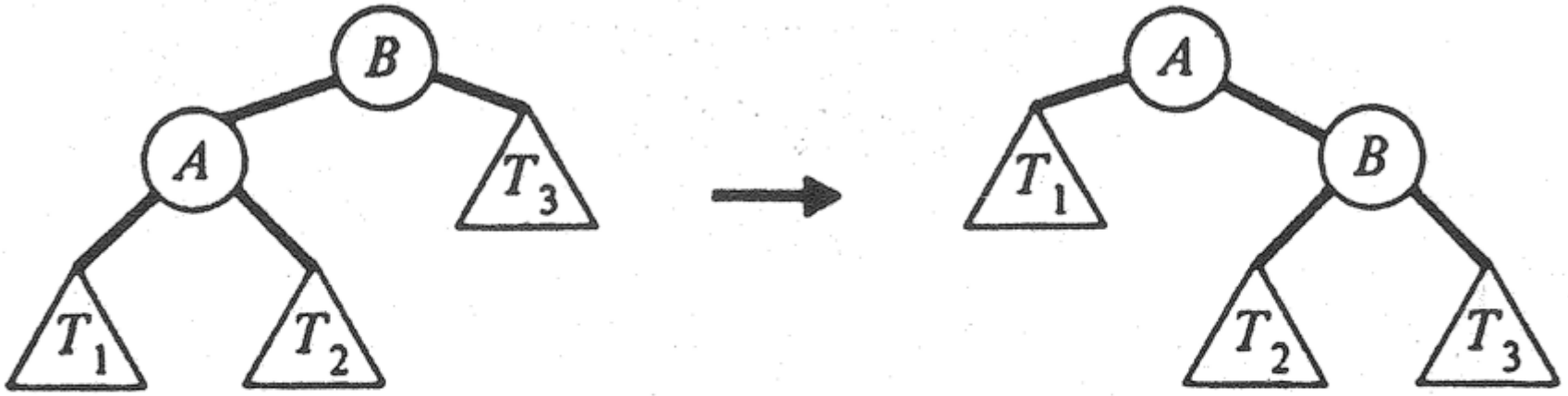
# AVL

## Rotation

- L’objectif principal d’une rotation d’arbre est de garder un ABR équilibrés après chaque opération de suppression ou d’insertion d’un élément dans l’arbre.
- Le sens et le nombres de rotations nécessaires pour rééquilibrer un ABR dépend des valeurs des facteurs d’équilibrage d’un nœud et ainsi que ces fils droit et gauche.

# Arbre(Tree)

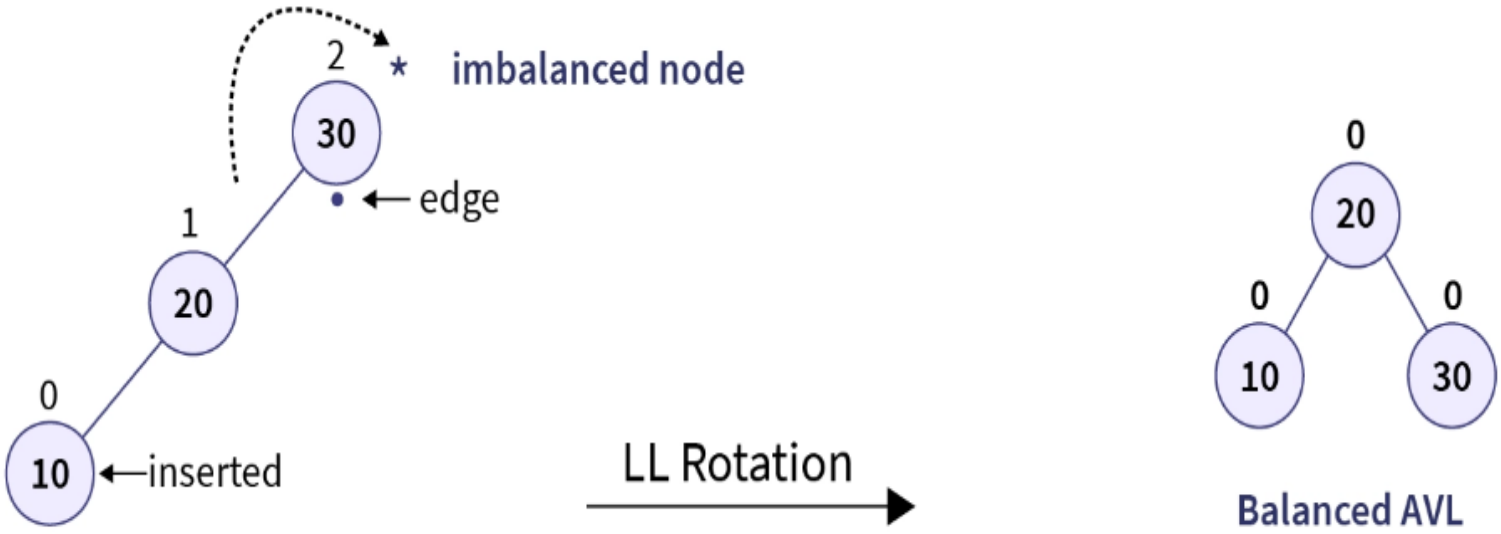
- Rotation à droite(Right rotation)




$T_1 < A < T_2 < B(\text{root}) < T_3 \quad \rightarrow \quad T_1 < A(\text{root}) < T_2 < B < T_3$

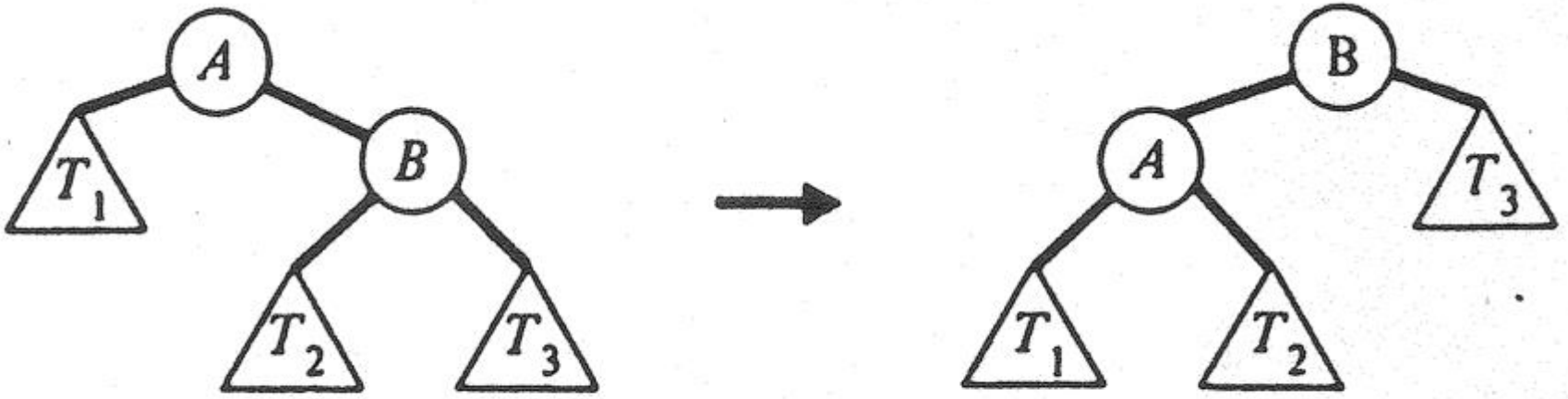
# Arbre(Tree)

- RIGHT rotation(single rotation)
- Left-Left (LL) insertion)



# AVL

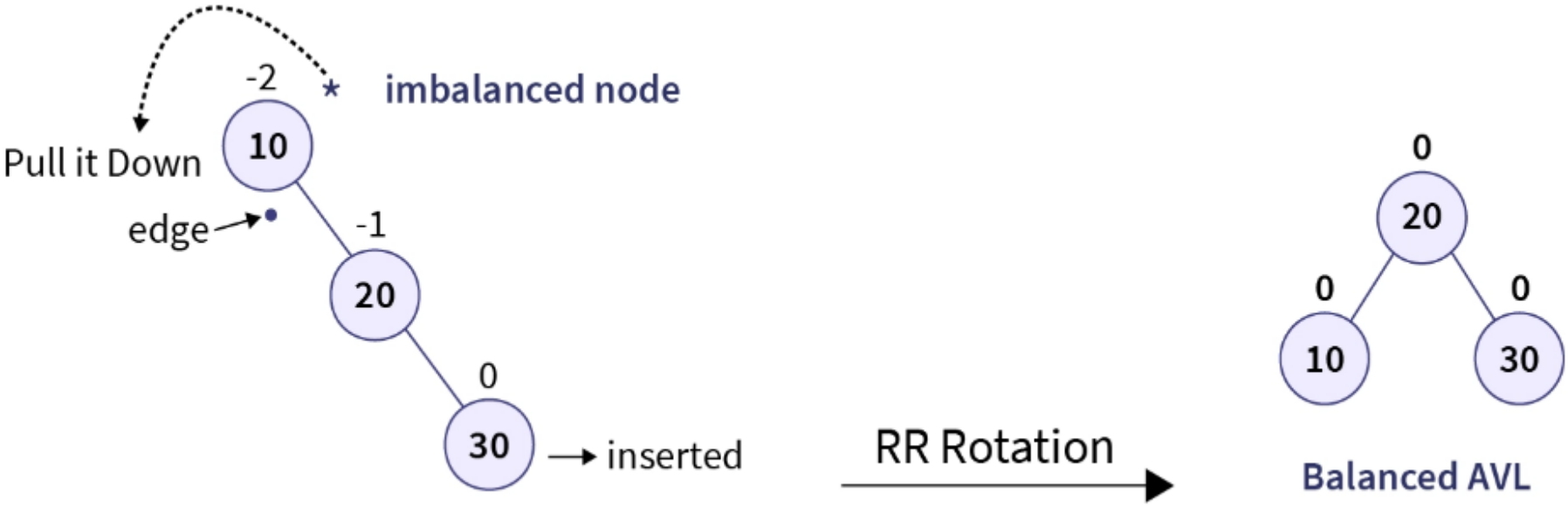
- Rotation à gouche(Left rotation)




$T_1 < A(\text{root}) < T_2 < B < T_3 \quad \rightarrow \quad T_1 < A < T_2 < B(\text{root}) < T_3$

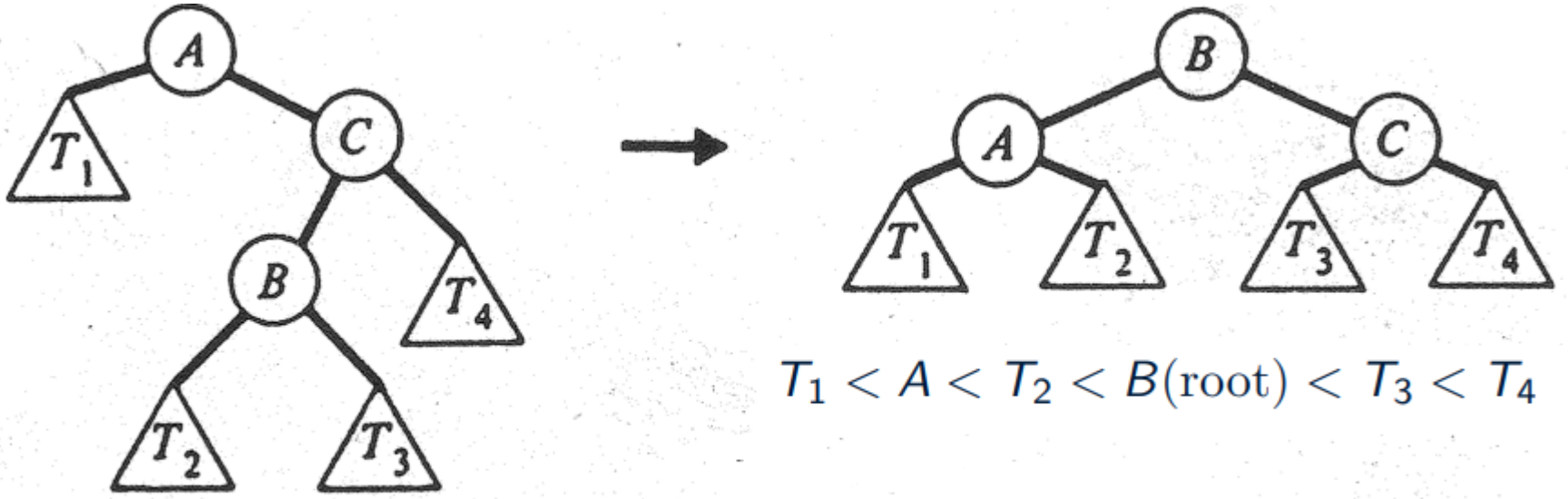
# AVL

- Left rotation (single rotation)
- Right-Right (RR) insertion



# Arbre(Tree)

- Rotation double droite-gauche
- (Right Left rotation RL)

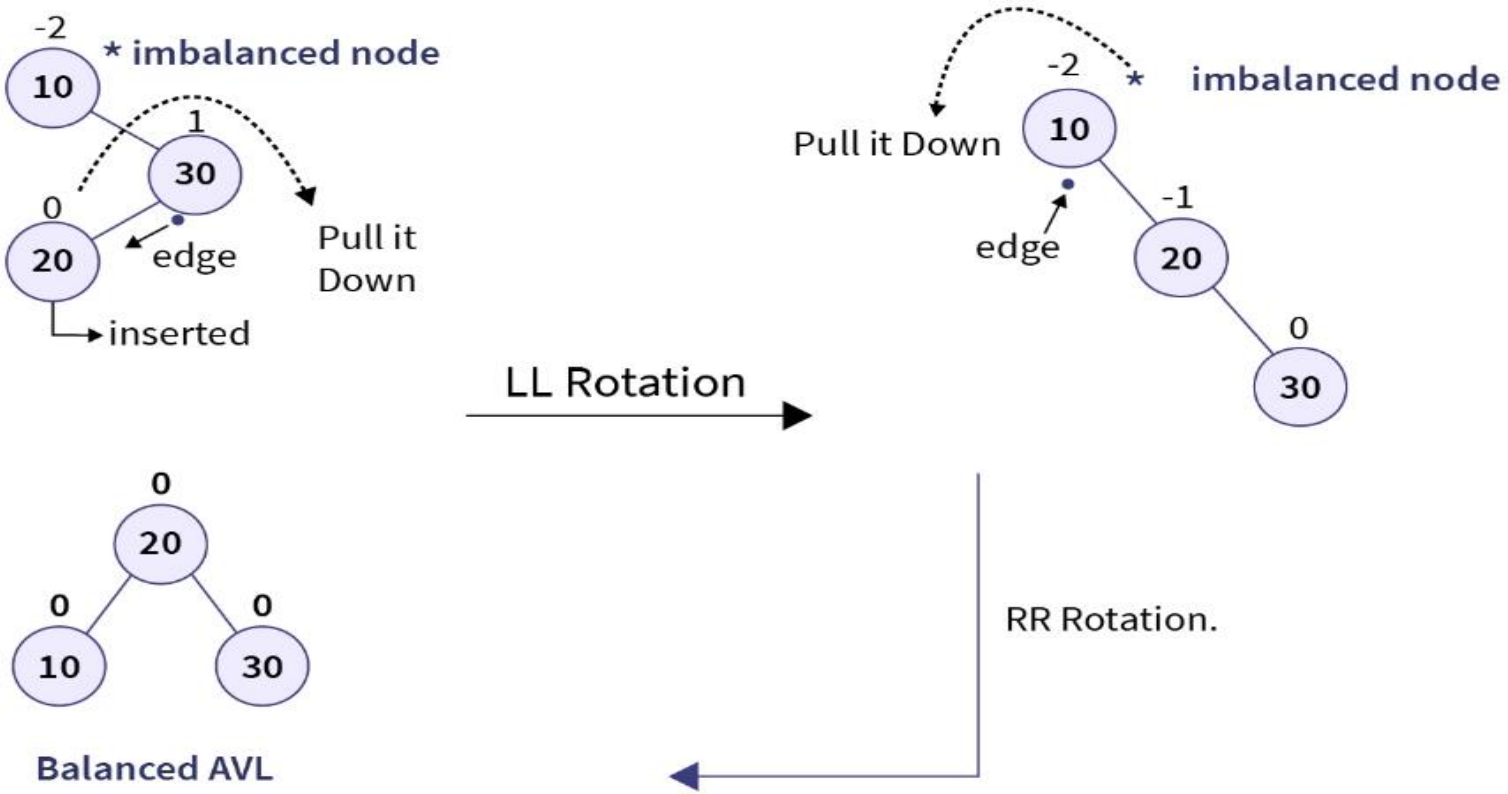



$T_1 < A < T_2 < B(\text{root}) < T_3 < T_4$

$T_1 < A(\text{root}) < T_2 < B < T_3 < C < T_4 \rightarrow$

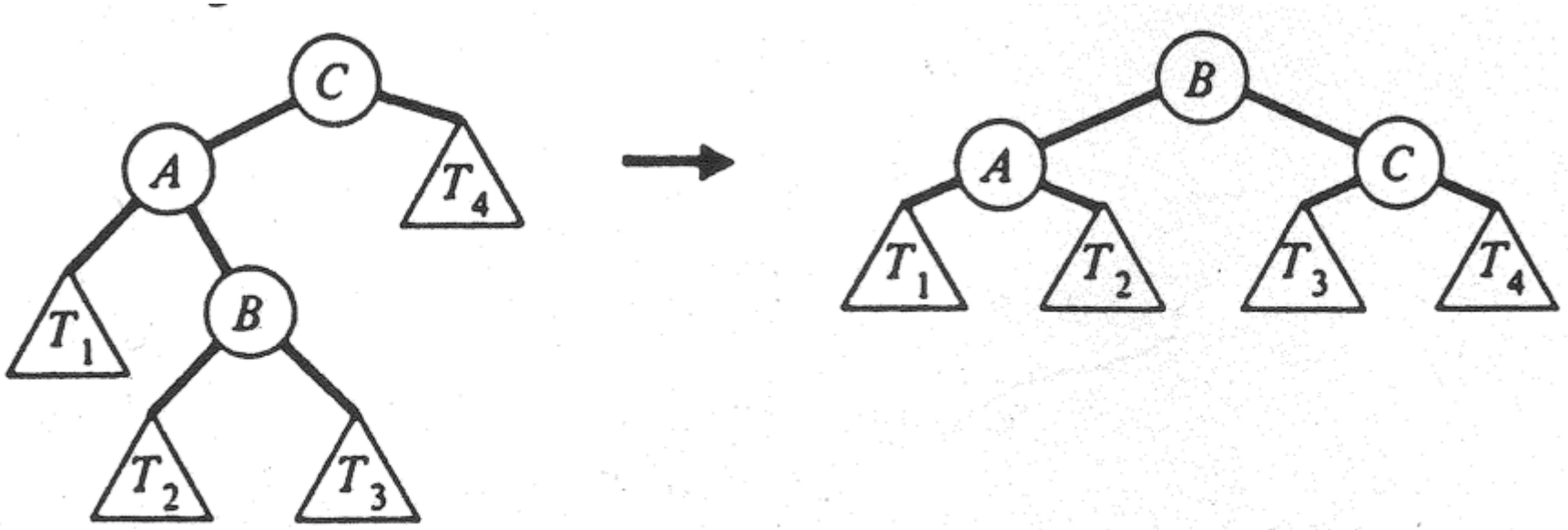
# AVL

- Double rotation(Right Left rotation)
- Right-Left (RL) insertion



# AVL

- Rotation double gauche-droite
- (Left  Right rotaion LR)

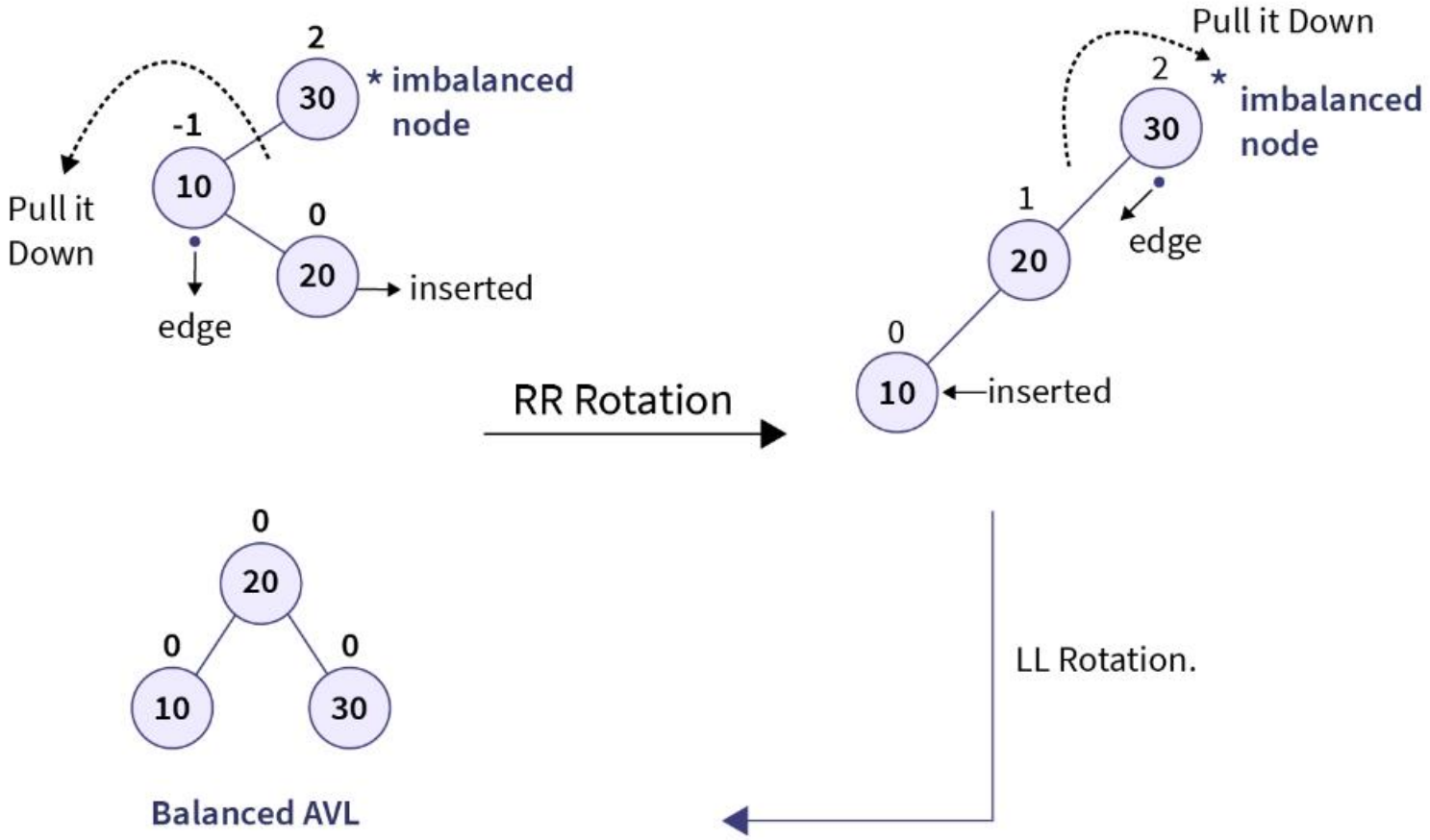



$T_1 < A < T_2 < B < T_3 < C(\text{root}) < T_4 \rightarrow$

$T_1 < A < T_2 < B(\text{root}) < T_3 < T_4$

# AVL

- Double rotation(Left Right rotation)
- Left-Right (LR) insertion,



# AVL Insertion

- **Étape 1**: Insérez le nœud dans AVL en utilisant le même algorithme d'insertion de BST

- **Étape 2:** Une fois le nœud ajouté, le facteur d'équilibre (Balance Factor **BF**)de chaque nœud est mis à jour

- **Étape 3**: Vérifiez maintenant si un nœud viole la plage du facteur d'équilibre si le facteur d'équilibre est violé, puis effectuez des rotations en utilisant le cas ci-dessous.

- Si BF(node) = +2 et BF(node -> left-child) = +1,(une insertion **LL**)

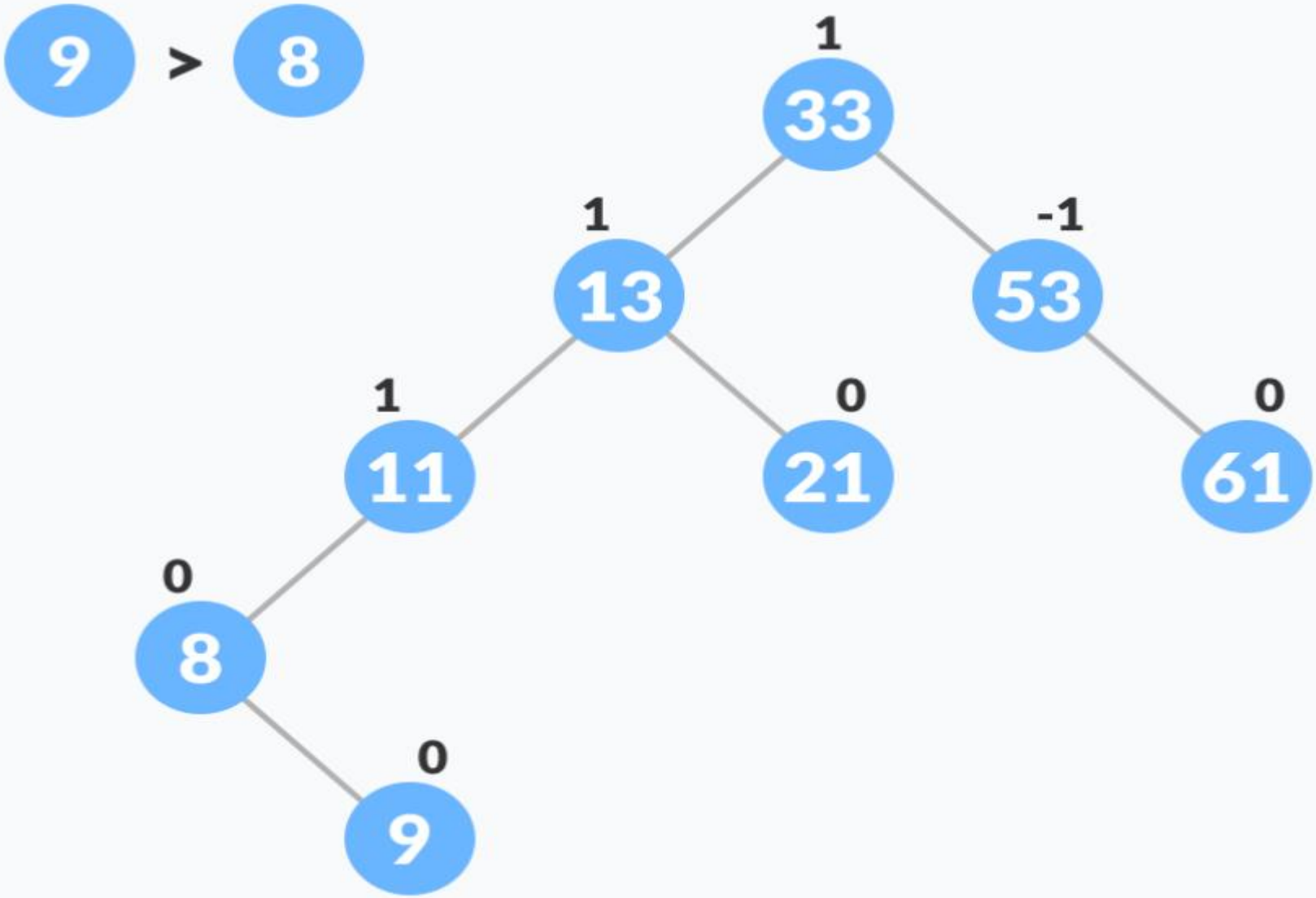
  effectuez rotation à droite ( **right rotation)** .

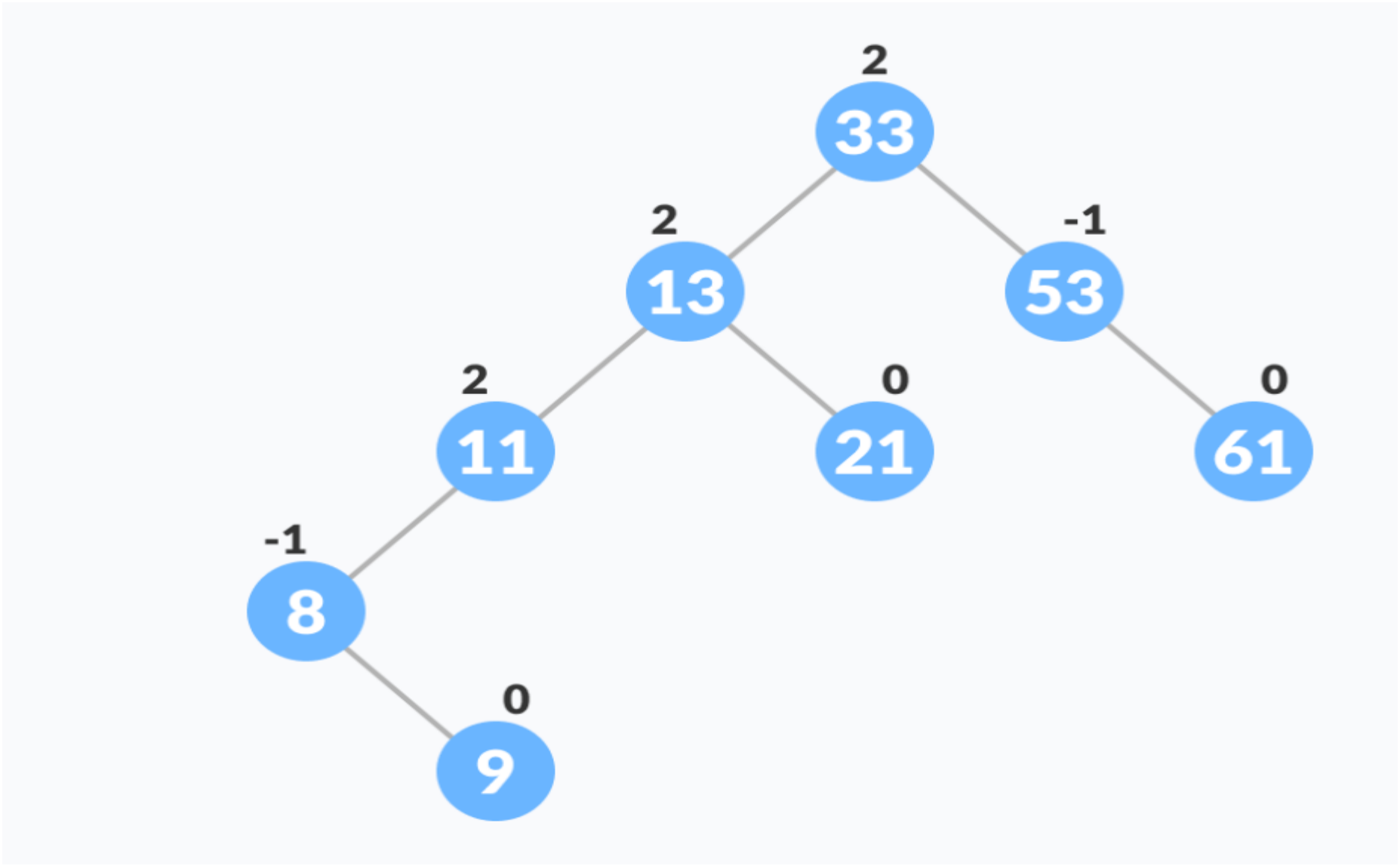
- Si BF(node) = -2 et BF(node -> right-child) = -1, (une insertion **RR**)

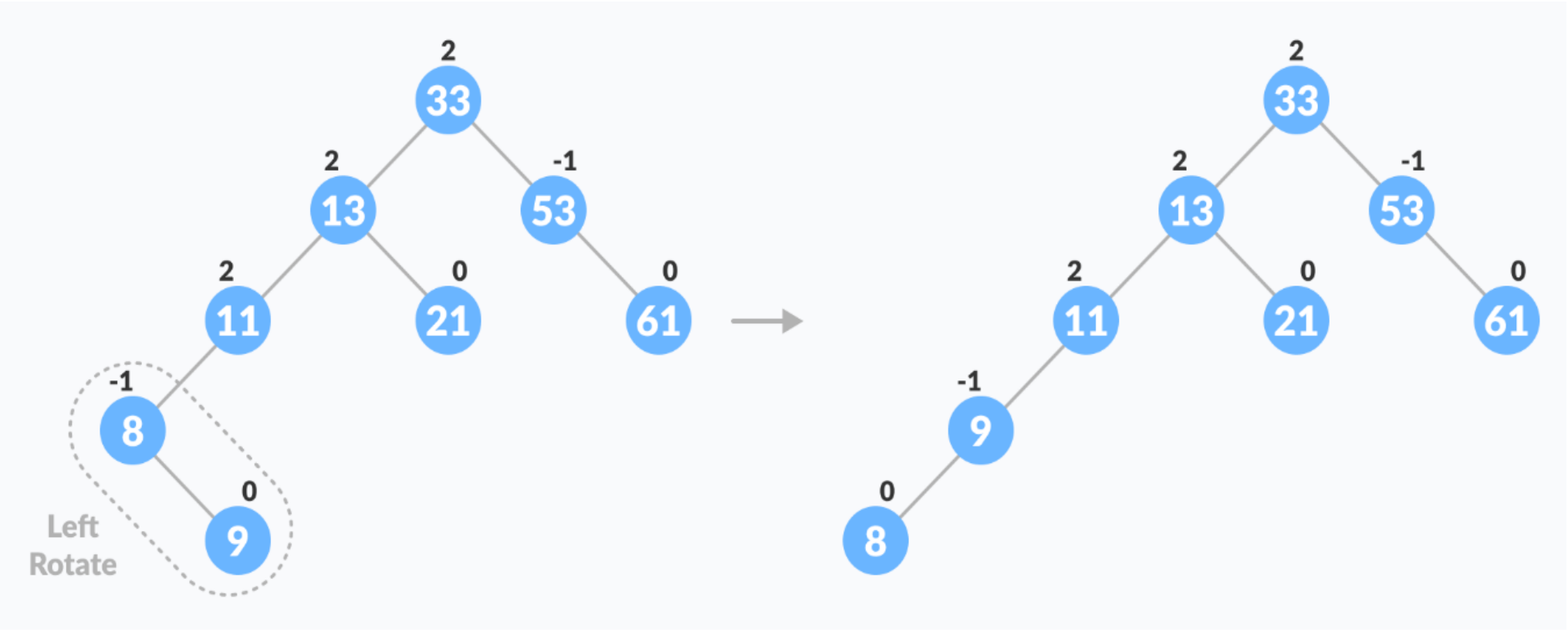
- effectuez rotation à gauche ( **Left rotation)** .

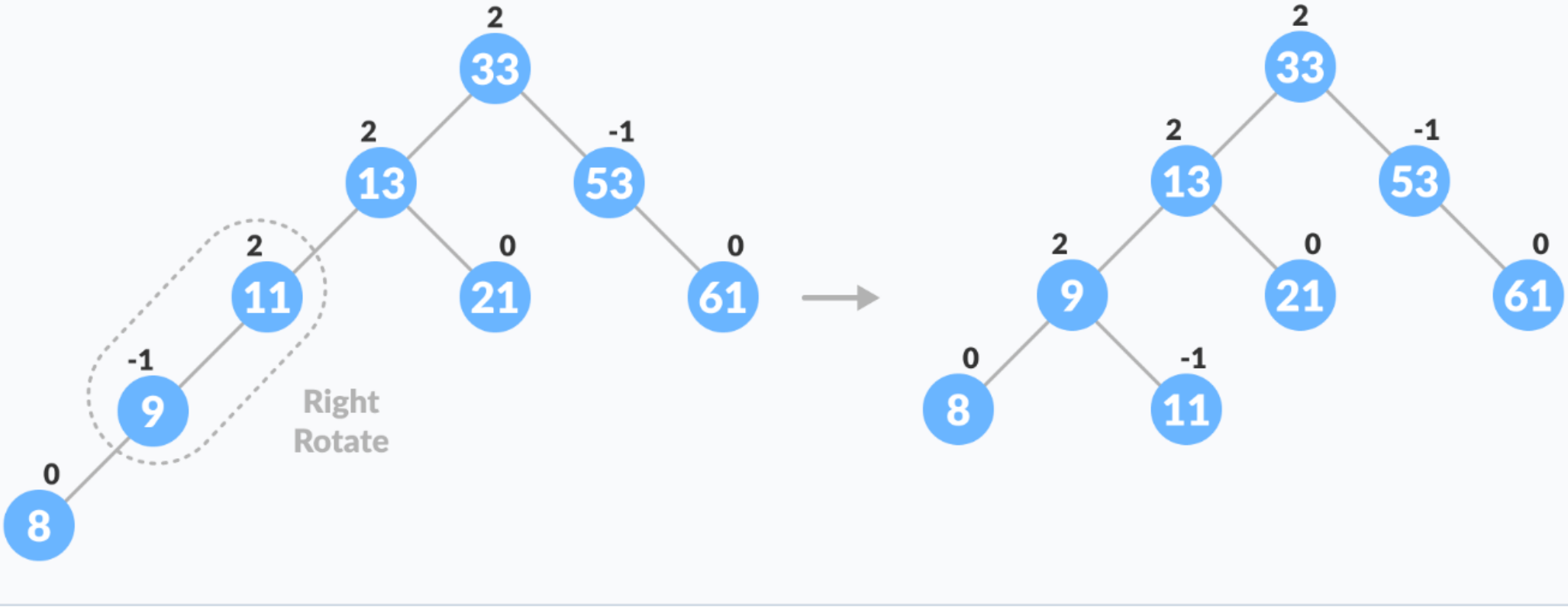
- Si BF(node) = -2 et BF(node -> right-child) = +1, effectuez une rotation **RL**.

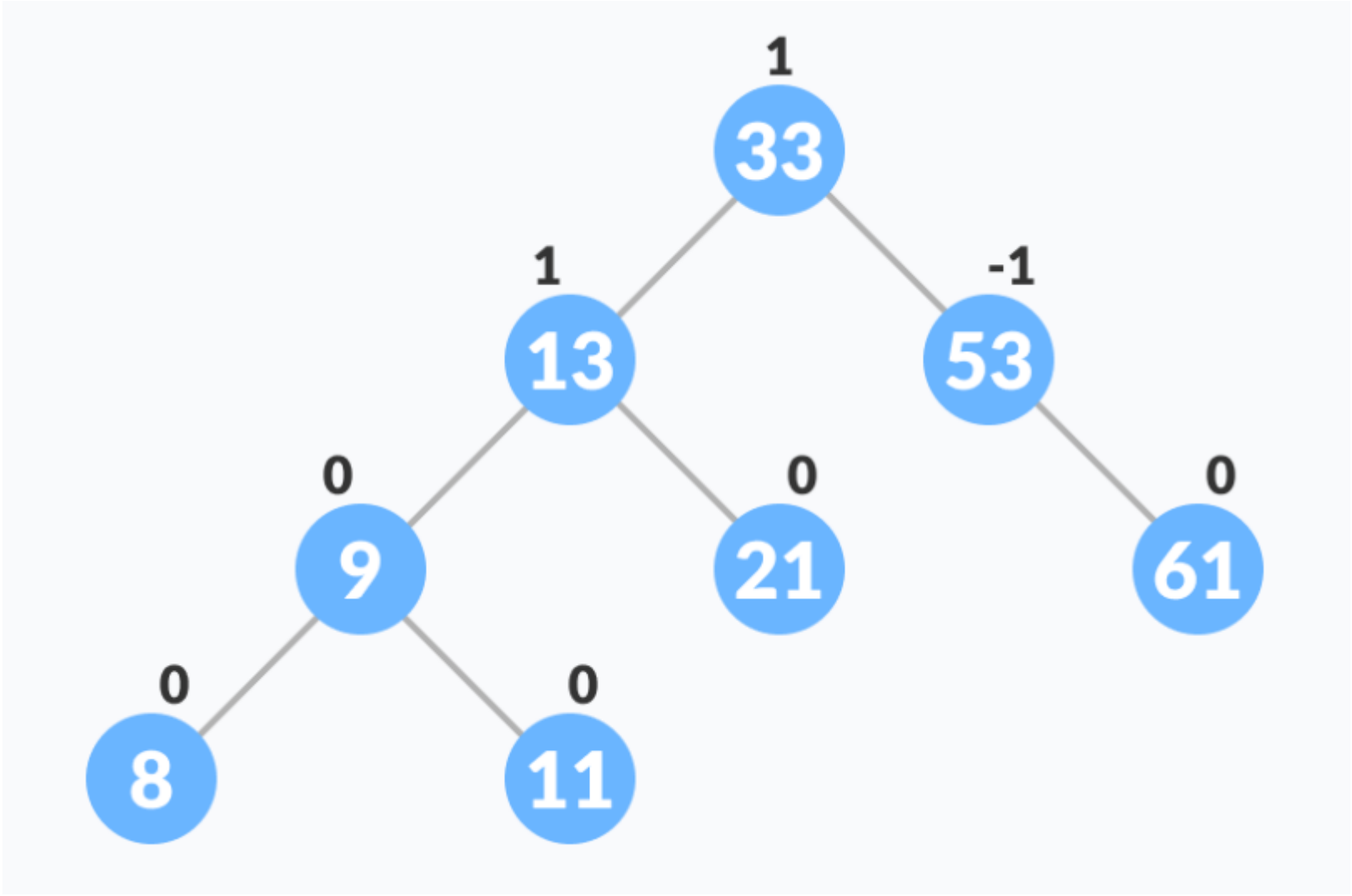
- Si BF(node) = +2 et BF(node -> left-child) = -1, effectuez une rotation **LR**.

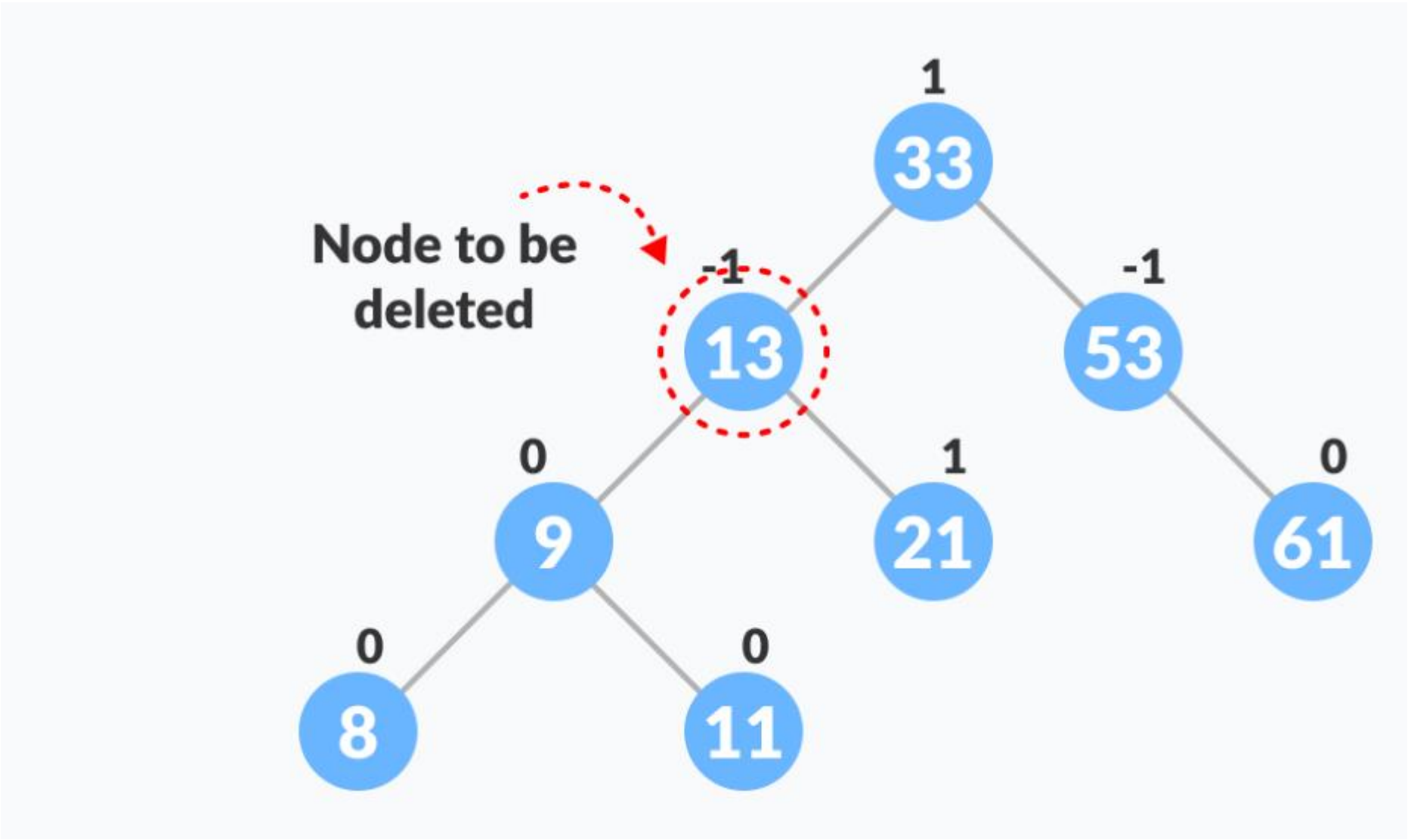

9 > 8
1
33
1
13
-1
53
1
11
0
21
0
61
0
8
0
9

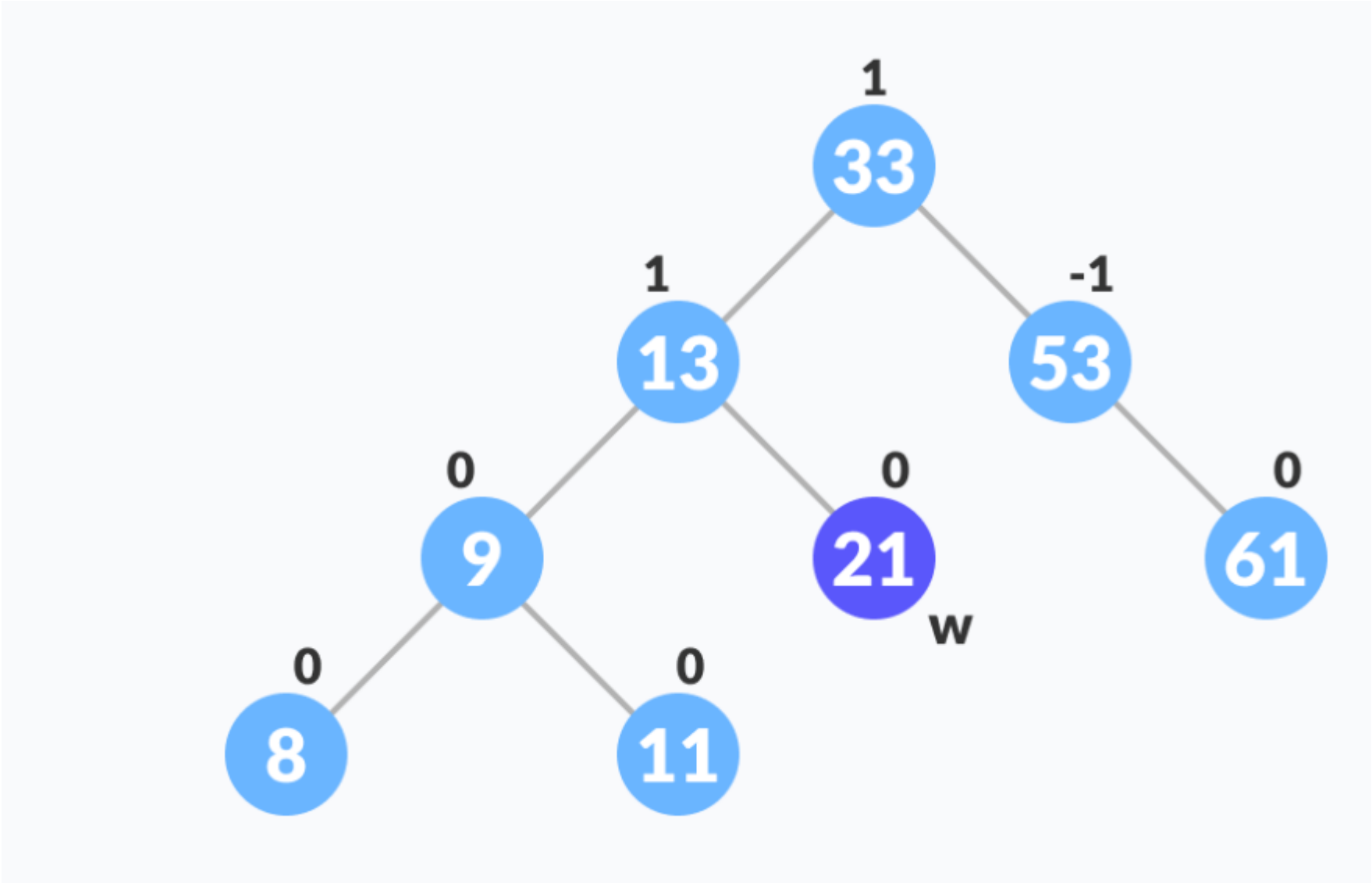

2
33
2
13
-1
53
2
11
0
21
0
61
-1
8
0
9

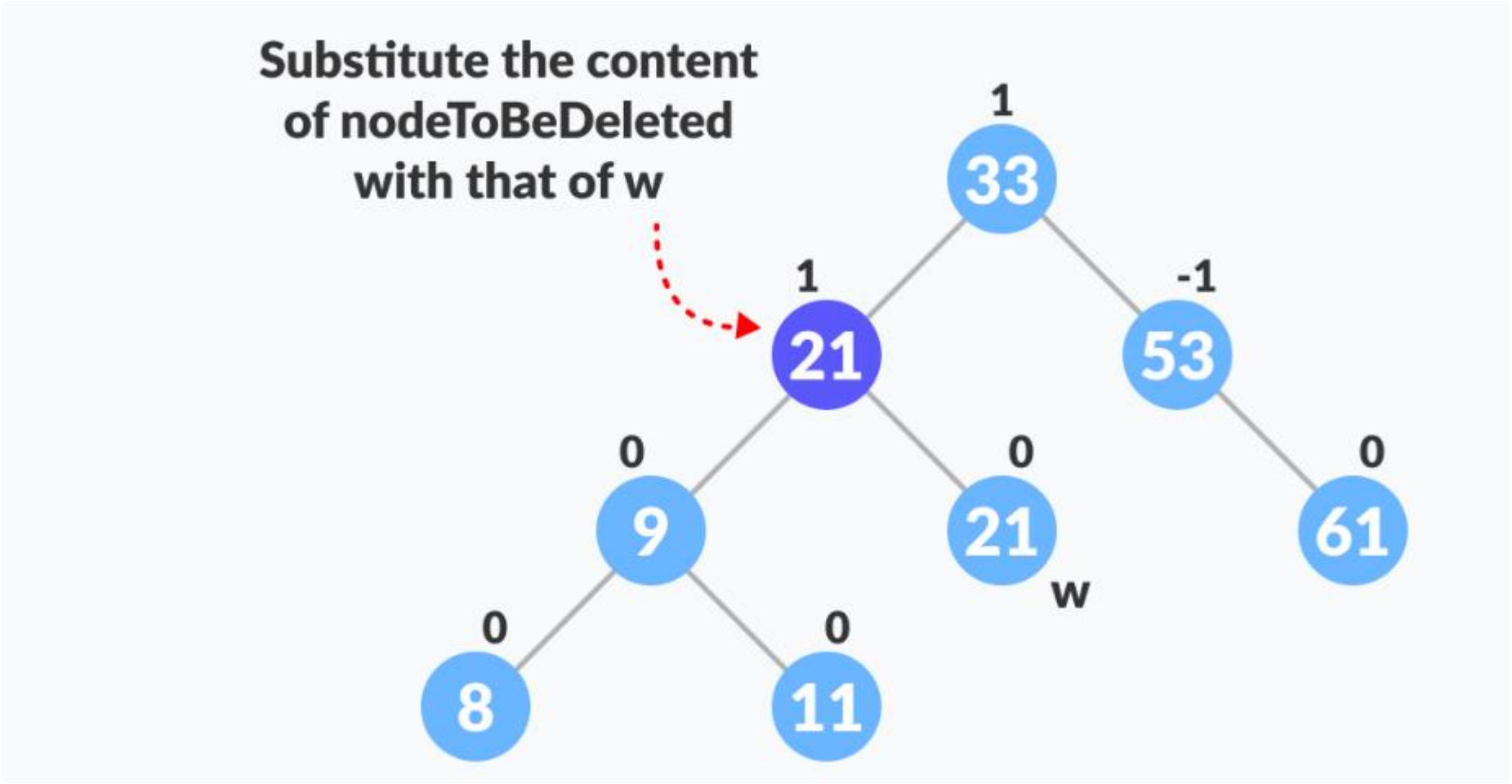

2
33
2
13
-1
53
2
11
0
21
0
61
-1
8
0
9
Left
Rotate
2
33
2
13
-1
53
2
11
0
21
0
61
-1
9
0
8

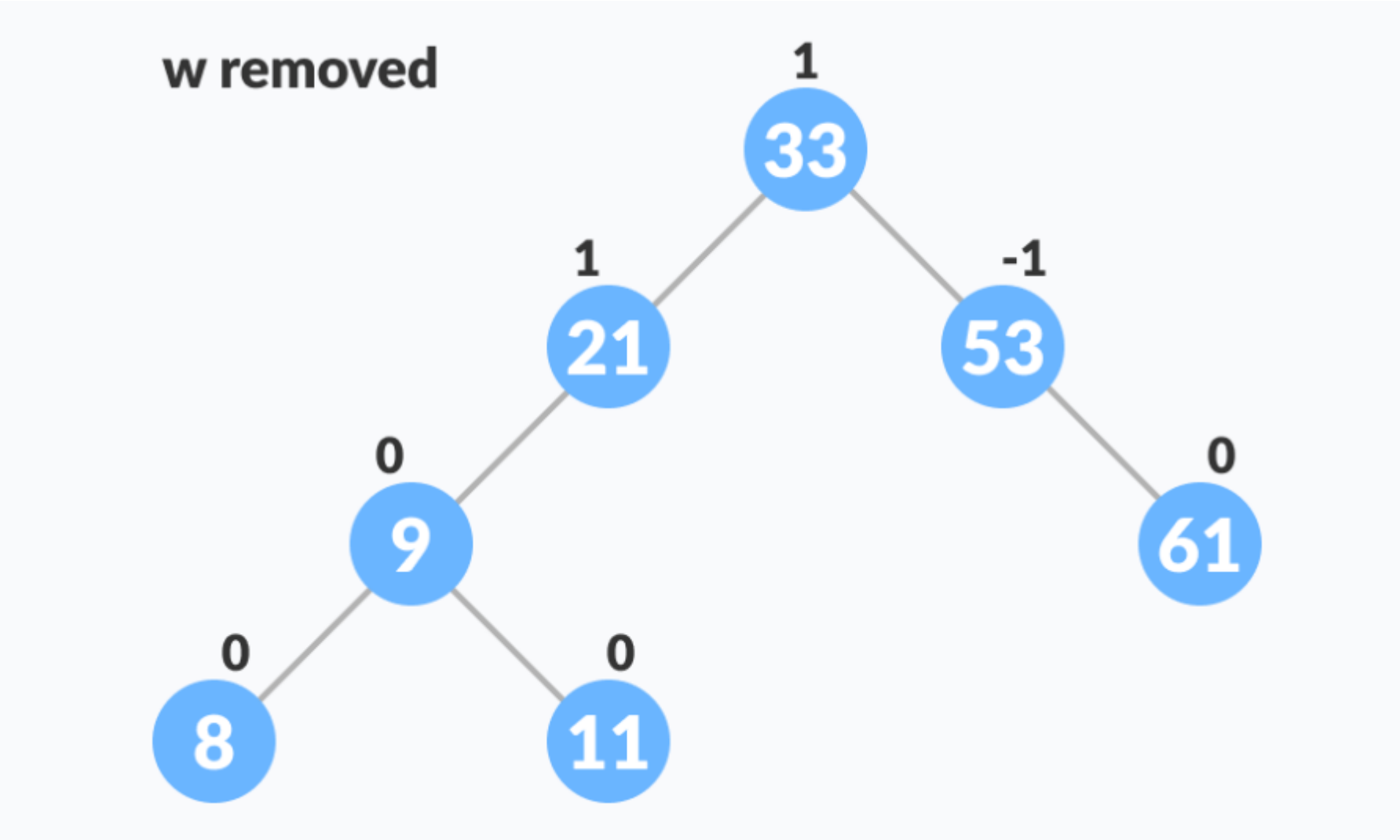

2
33
2
13
-1
53
2
11
0
21
0
61
-1
9
0
8
Right
Rotate
2
33
2
13
-1
53
2
9
0
21
0
61
0
8
-1
11

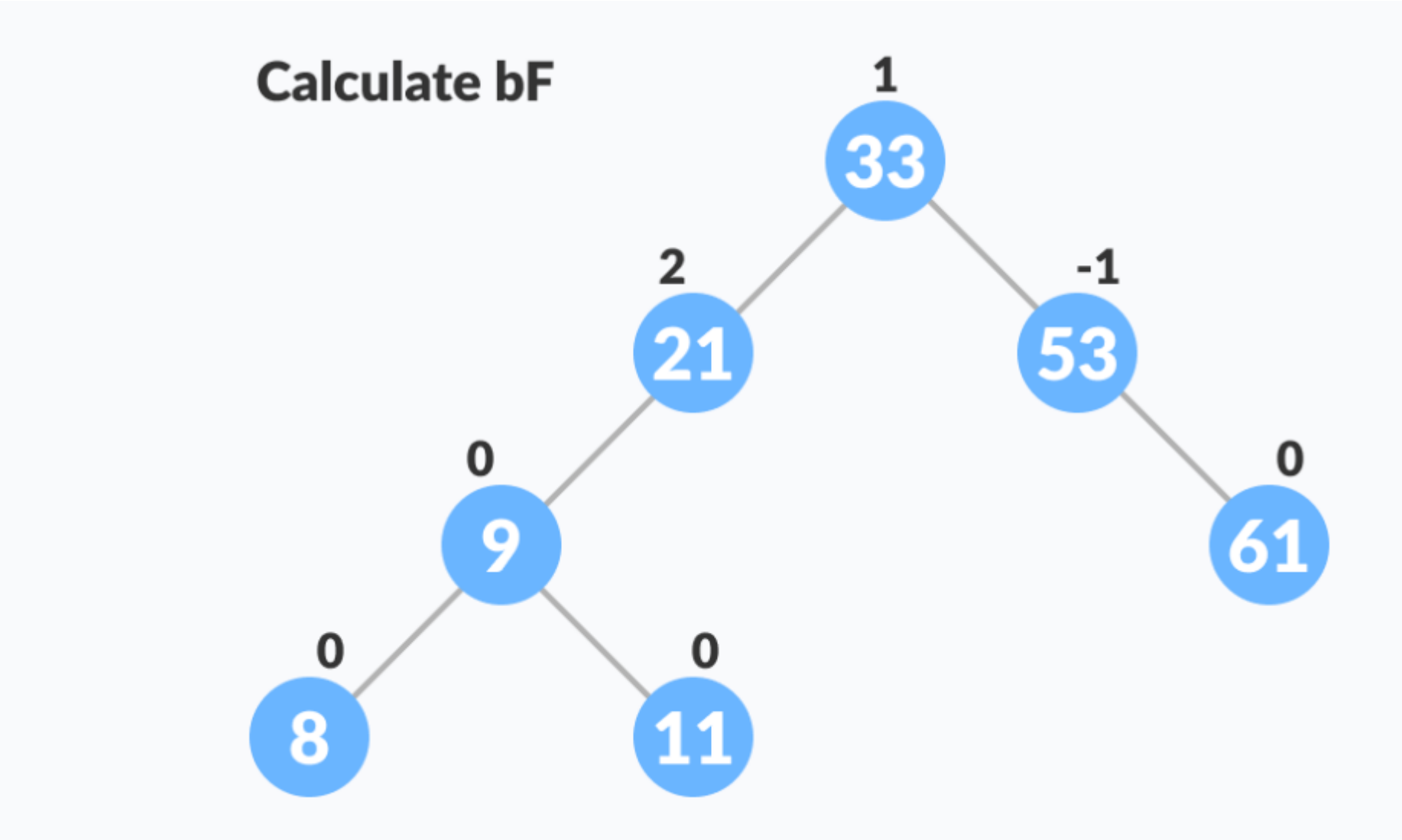

1
33
1
13
-1
53
0
9
0
21
0
61
0
8
0
11

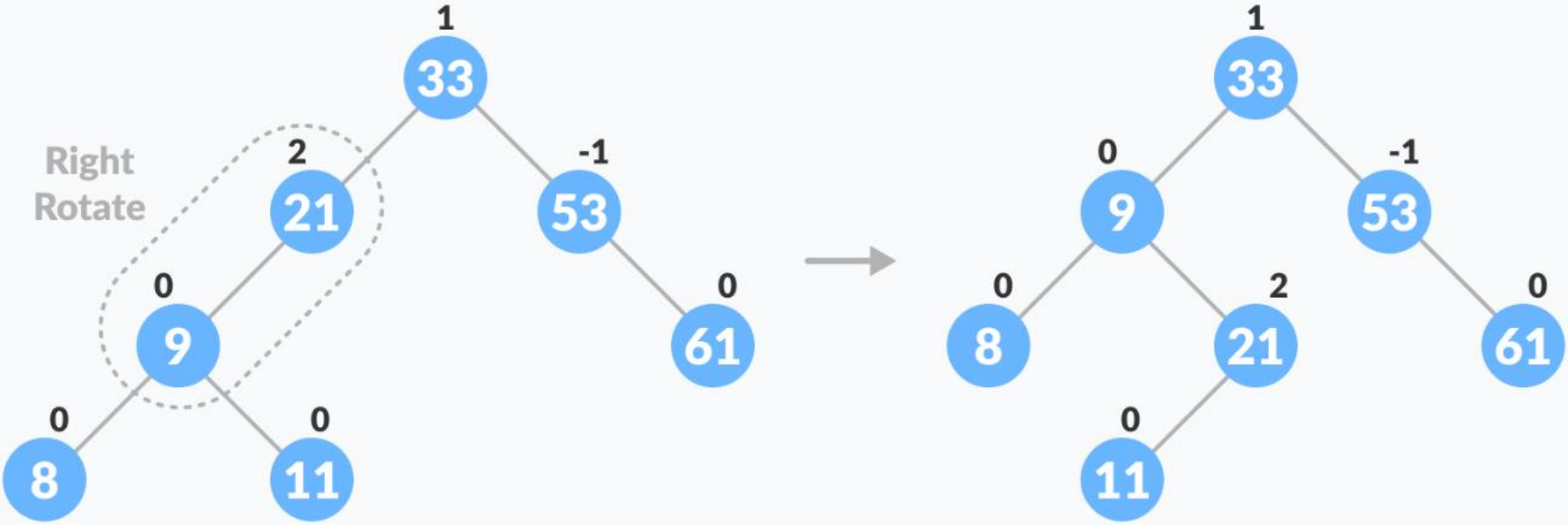

Node to be deleted
1
33
-1
13
-1
53
0
9
1
21
0
61
0
8
0
11

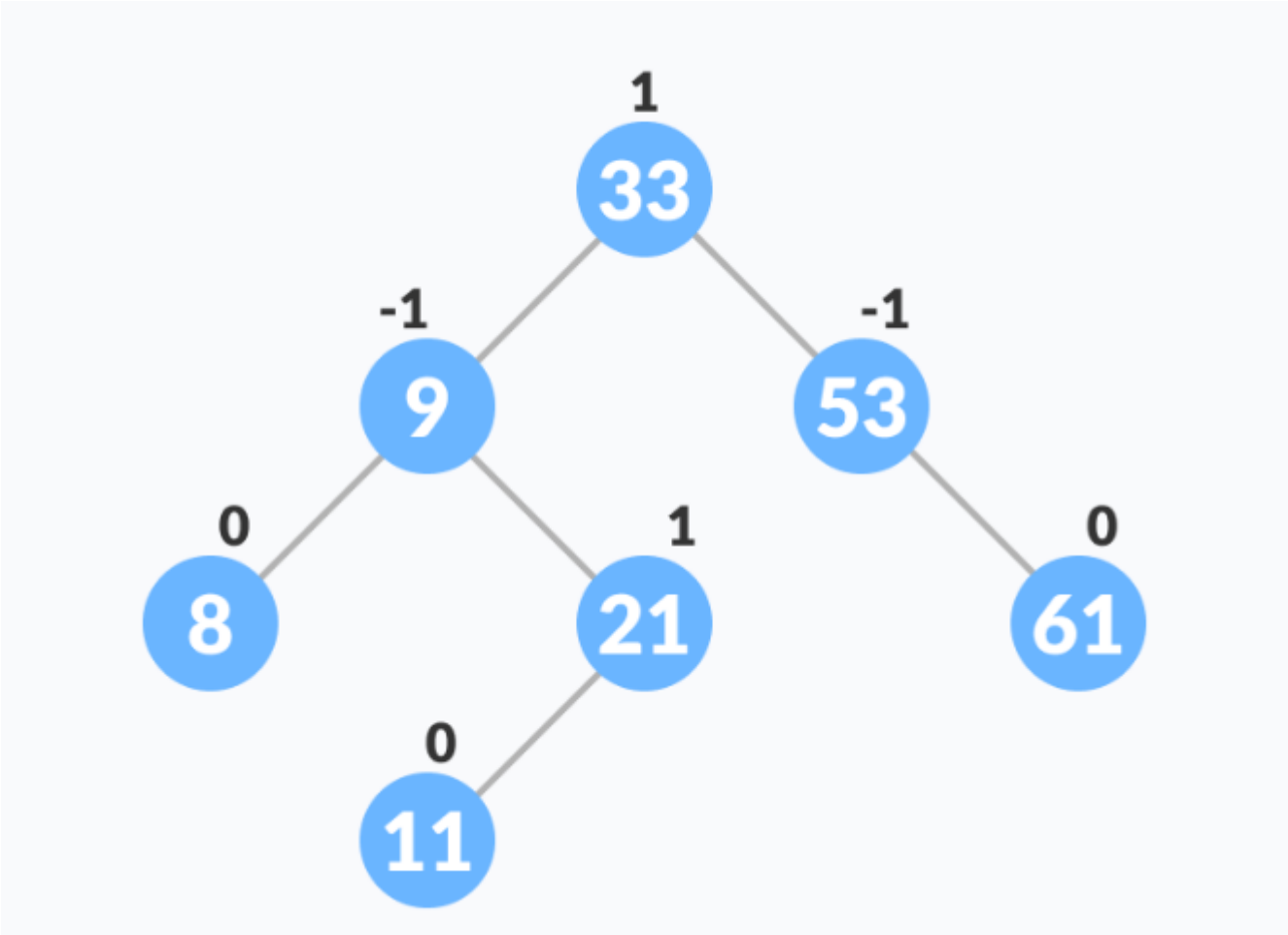

1
33
1
13
-1
53
0
9
0
21
0
61
w
0
8
0
11


Substitute the content
of nodeToBeDeleted
with that of w
1
33
1
21
-1
53
0
9
0
21
w
0
61
0
8
0
11


w removed
1
33
1
21
-1
53
0
9
0
61
0
8
0
11


Calculate bF
1
33
2
21
-1
53
0
9
0
61
0
8
0
11


Right
Rotate
1
33
2
21
-1
53
0
9
0
61
0
8
0
11
1
33
0
9
-1
53
0
8
2
21
0
61
0
11

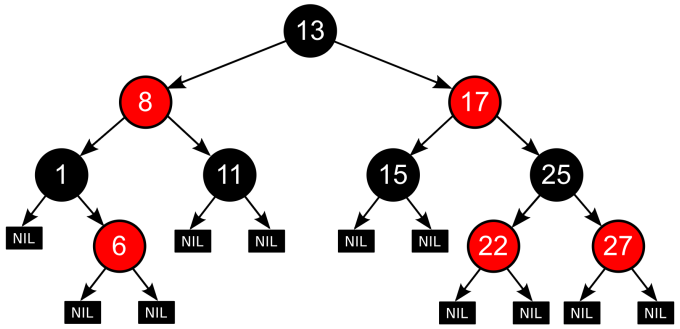

1
33
-1
9
-1
53
0
8
1
21
0
61
0
11

# Arbre(Tree)

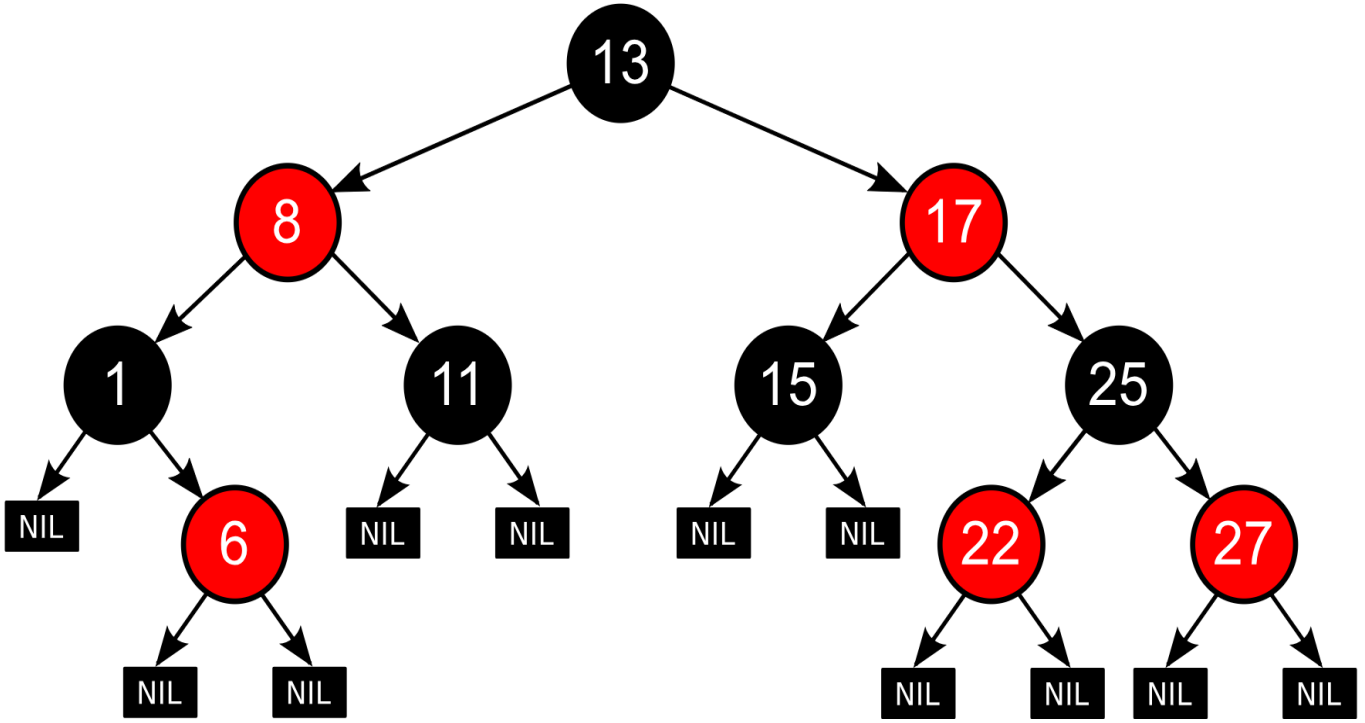
- Un nœud avec tout autre facteur est considéré comme déséquilibré et requiert un rééquilibrage.
- Chaque fois qu'un nœud est inséré ou supprimé d'un arbre AVL, le facteur d'équilibrage de chaque nœud le long du chemin depuis la racine jusqu'au nœud inséré (ou supprimé) doit être recalculé.
- Si l'arbre est resté équilibré, il n'y a rien à faire. Si ce n'est pas le cas, on effectuera des rotations d'équilibrage de manière à obtenir à nouveau un arbre AVL.

# RED BLACK TREE

Un arbre rouge-noir est un arbre binaire de recherche (BST) auto-équilibré qui utilise un bit supplémentaire par nœud, interprété comme une couleur (rouge ou noir), afin de garantir que l'arbre reste approximativement équilibré lors des insertions et des suppressions. Cet équilibrage assure que les opérations comme la recherche, l'insertion et la suppression s'exécutent en temps O(log n) dans le pire des cas, où n est le nombre de nœuds.



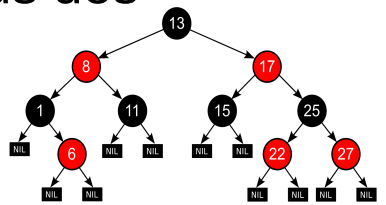
# RED BLACK TREE



# RED BLACK TREE

**Propriétés**

Un arbre rouge-noir doit satisfaire des propriétés spécifiques en plus des propriétés standard d'un BST :




**Couleur du nœud** : Chaque nœud est soit rouge, soit noir.

**Propriété de la racine** : La racine de l'arbre est toujours noire. (Certaines définitions omettent cela, car on peut toujours colorer la racine en noir sans violer d'autres propriétés.)

**Propriété des feuilles** : Toutes les feuilles (qui sont des nœuds null/NIL dans l'implémentation) sont considérées comme noires.

**Propriété rouge** : Un nœud rouge ne peut pas avoir un enfant rouge (pas deux nœuds rouges consécutifs sur un chemin).

**Propriété noire** : Tout chemin simple partant d'un nœud donné vers l'une de ses feuilles descendantes contient le même nombre de nœuds noirs (appelé la "profondeur noire" ou "hauteur noire").

# RED BLACK TREE

**Opérations**

Les opérations dans un arbre rouge-noir sont similaires à celles d'un BST standard, mais incluent des étapes supplémentaires impliquant des rotations et des recolorations pour restaurer les propriétés si elles sont violées :

**Recherche** : Effectuée comme dans un BST classique, avec une complexité de O(log n).

**Insertion** : Le nouveau nœud est d'abord inséré comme un nœud rouge. Si cela viole une propriété (par exemple un parent rouge avec un enfant rouge), une procédure de correction utilisant recoloration et/ou rotations (rotations gauche/droite) est effectuée pour rétablir l'équilibre, en O(log n) dans le pire des cas.

**Suppression** : Le nœud est d'abord supprimé selon les règles standards du BST. Si le nœud supprimé était noir, cela peut créer un déséquilibre (une situation de "double noir") nécessitant une procédure de rééquilibrage plus complexe utilisant rotations et recolorations, également en O(log n) dans le pire des cas.

# RED BLACK TREE

**Applications**

Les arbres rouge-noir sont largement utilisés dans les applications où les performances sont critiques, grâce à leurs garanties de temps d'exécution dans le pire des cas :

**Implémentations dans les bibliothèques standard :**
Utilisés pour implémenter les conteneurs associatifs tels que std::map, std::set, std::multimap et std::multiset dans la *Standard Template Library* (STL) de C++, ainsi que TreeMap et TreeSet en Java.

**Systèmes d'exploitation :**
Le noyau Linux utilise les arbres rouge-noir dans le *Completely Fair Scheduler* (CFS) pour gérer les processus en cours d'exécution, ainsi que dans la gestion de la mémoire.

**Systèmes de bases de données :**
Employés dans les moteurs de bases de données pour l'indexation efficace, permettant une récupération et une manipulation rapides des données.

**Géométrie computationnelle :**
Utilisés comme briques de base dans plusieurs structures de données destinées à résoudre des problèmes de géométrie algorithmique.

# RED BLACK TREE(**Insertion)**

Un nouveau nœud est toujours inséré initialement comme un nœud rouge. Le processus d'insertion suit généralement les étapes suivantes :

**1. Insertion BST standard** Le nouveau nœud est d'abord inséré à sa position correcte en utilisant l'algorithme d'insertion classique d'un arbre binaire de recherche (BST).

**2. Coloration du nouveau nœud en rouge** Le nouveau nœud est toujours coloré en rouge.
En effet, insérer un nœud noir modifierait la hauteur noire des chemins, ce qui créerait une violation beaucoup plus difficile à corriger qu'un simple conflit de type « double rouge ».

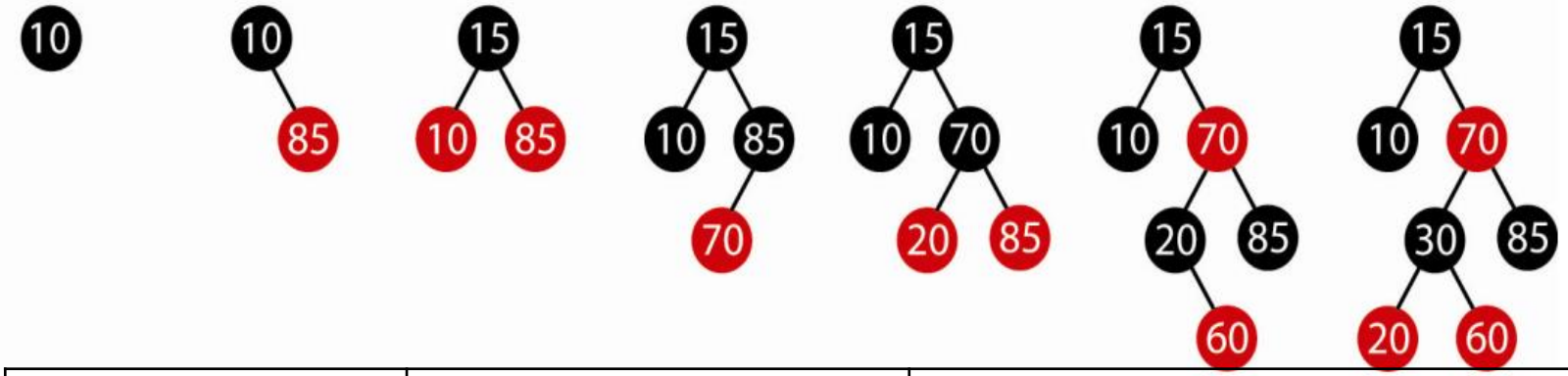
**3. Correction des violations (rééquilibrage)** Après l'insertion, l'arbre est vérifié pour détecter d'éventuelles violations des propriétés rouge-noir, en particulier la règle interdisant qu'un nœud rouge ait un enfant rouge.
En cas de violation, une procédure de correction (fix-up), qui remonte dans l'arbre, est appliquée et implique :

**Recoloration :** Modification des couleurs des nœuds (parent, oncle, grand-parent) pour résoudre les conflits.

**Rotations :** Utilisation de rotations gauche ou droite pour restructurer l'arbre et rétablir l'équilibre, en transformant les cas « en angle » en cas linéaires.

**4. Assurer que la racine est noire** Enfin, la racine de l'arbre est toujours recolorée en noir si elle est devenue rouge pendant le processus de rééquilibrage.

# RED BLACK TREE(**Insertion**)




| Step | Insert | Fix-up Result |
|---|---|---|
| 1 | 10 | Root black |
| 2 | 85 | No fix needed |
| 3 | 15 | Rotation + recoloring → new root 15 |
| 4 | 70 | Parent & uncle red → recoloring only |
| 5 | 20 | No violation |
| 6 | 60 | No violation |
| 7 | 30 | Parent & uncle red → recoloring only |

# RB TREE(Insertion CLRS )

**INSERT(T, z):**

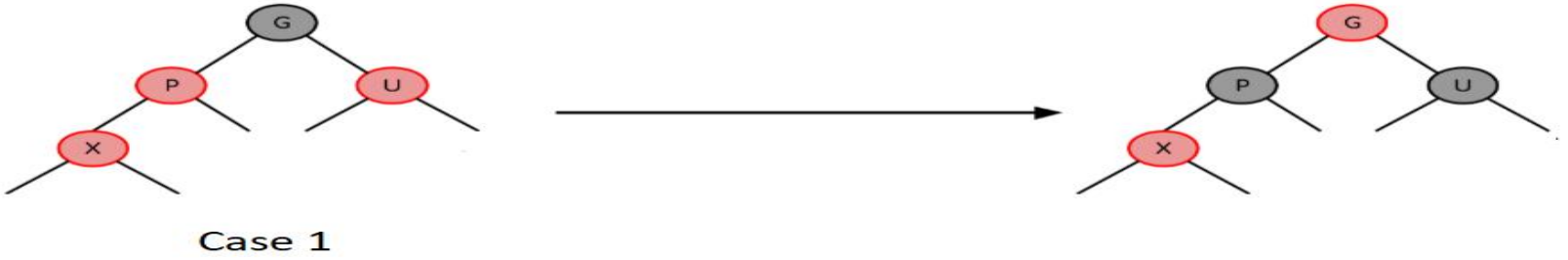
1. Insert z into the tree T using standard BST insertion rules.
2. Color z RED.
3. While z's parent is RED:

   a. Identify z's uncle (the sibling of z's parent).

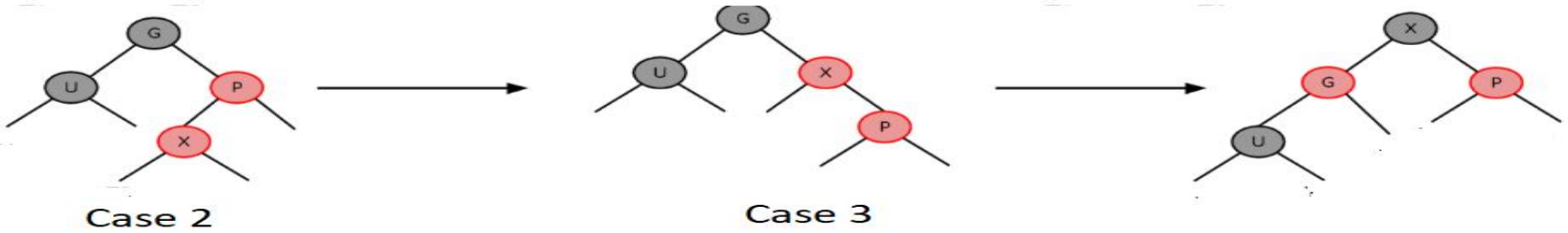
- **CASE 1**: The uncle is RED
- - Recolor parent and uncle to BLACK.
- - Recolor grandparent to RED.
- - Move z up to grandparent and continue.
- **CASE 2**: The uncle is BLACK and z is a "triangle" case
- - Rotate around parent to convert into a "line" case.
- - Update z to parent after rotation.
- **CASE 3**: The uncle is BLACK and z is a "line" case
- - Recolor parent BLACK.
- - Recolor grandparent RED.
- - Rotate around grandparent.

4. Set the root color to BLACK.

# RB TREE(Insertion CLRS )




Case 1




Case 2 Case 3

| Case | Condition | Shape | Fix |
|---|---|---|---|
| **1** | Parent = RED, Uncle = RED | any | Recolor (P,U→**BLACK**, G→**RED**) + move z up to **G** |
| **2** | P=RED, U=BLACK **triangle** | zig-zag | Rotate around **parent**, **update z = parent(z),** then Case 3 |
| **3** | P=RED, U=BLACK **line** | straight | Recolor P,G and rotate around **grandparent** |

# Insertion:  AVL VS. RB

| **Caractéristique** | **AVL** | **RED BLACK** |
|---|---|---|
| **Phase 1** | Insertion BST standard | Insertion BST standard (le nouveau nœud est **ROUGE**) |
| **Objectif principal** | Maintenir un équilibre **strict** (hauteur minimale) | Maintenir un équilibre **approximatif** (hauteur noire) |
| **Outil principal** | Rotations | Recolorations et rotations |
| **Nombre de cas** | 4 cas (LL, RR, LR, RL) | 3 cas principaux (selon la couleur de l'oncle) |
| **Philosophie** | « Nous devons être parfaitement équilibrés. » | « Nous sommes suffisamment équilibrés et nous privilégions une insertion rapide. » |

# Comparaison des performances : AVL vs. RB

| Aspect / Opération | Arbres AVL | Arbres Rouge-Noir |
|---|---|---|
| **Rigueur de l'équilibre** | Strict | Relâché |
| **Borne sur la hauteur** | 1,44 $\log_2(n)$ | 2 $\log_2(n)$ |
| **Fréquence des rotations** | Élevée | Faible |
| **Performance en recherche** | Meilleure | Bonne |
| **Performance en mise à jour** | Bonne | Meilleure |
| **Recherche** | O(log n) | O(log n) |
| **Insertion** | O(log n) | O(log n) *(plus rapide en pratique)* |
| **Suppression** | O(log n) | O(log n) *(plus rapide en pratique)* |
| **Rééquilibrage après mise à jour** | Nécessite souvent plusieurs rotations | Généralement peu de rotations |

# AVL VS RB

**Utilisez les arbres AVL lorsque :**

- Les opérations de recherche sont beaucoup plus nombreuses que les insertions/suppressions
- Les données sont relativement statiques
- Vous avez besoin d'une performance de recherche **garantie optimale**
- La simplicité d'implémentation est importante

**Utilisez les arbres Rouge-Noir lorsque :**

- Les insertions et suppressions sont fréquentes
- Vous avez besoin d'un bon équilibre entre performances de recherche et de modification
- Vous construisez une structure de données **généraliste** (usage polyvalent)

# AVL VS RB

**Exemples d'utilisation**

- Une application de dictionnaire où les utilisateurs consultent souvent des mots mais en ajoutent rarement : **Arbre AVL**
- Un index de base de données qui gère des insertions, des mises à jour et des requêtes fréquentes : **Arbre Rouge-Noir**

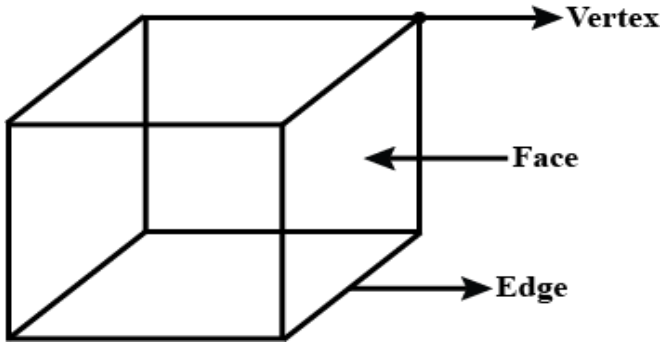
# Arbre(Tree)

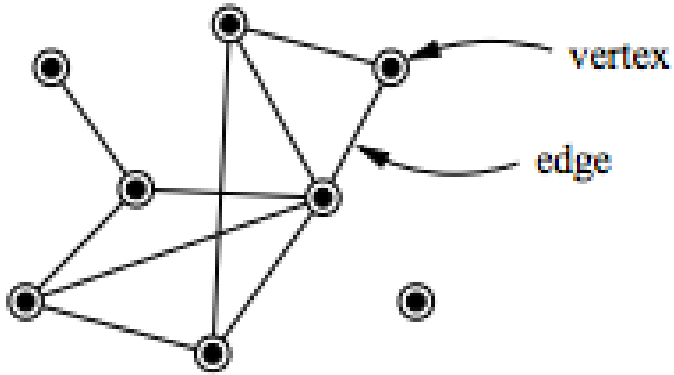
| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|---|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Array | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Queue | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Doubly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Skip List | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Hash Table | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Cartesian Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) |
| B-Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Red-Black Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Splay Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| AVL Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| KD Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |

# Algorithmique du graphe

# graphe

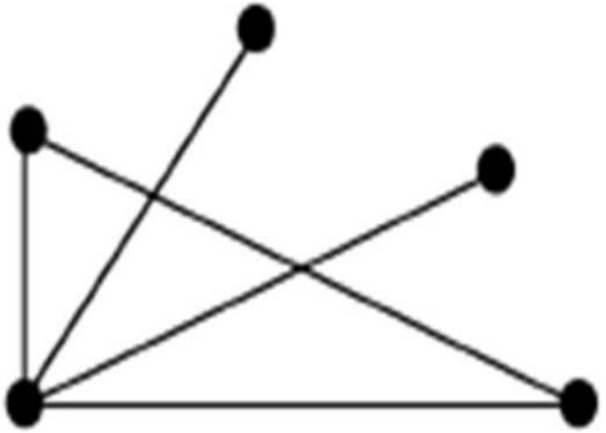
- Un graphe est la donnée d'un certain nombre de points du plan, appelés sommets(**vertices/nodes**), certains étant reliés par des segments de droites ou de courbes appelés arêtes( **edges**), la disposition des sommets et la forme choisie pour les arêtes n'intervenant pas



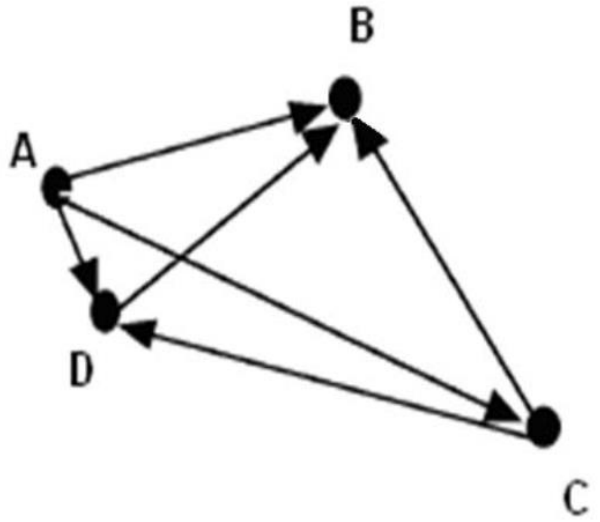



# graphe

**Graphe non orienté :**

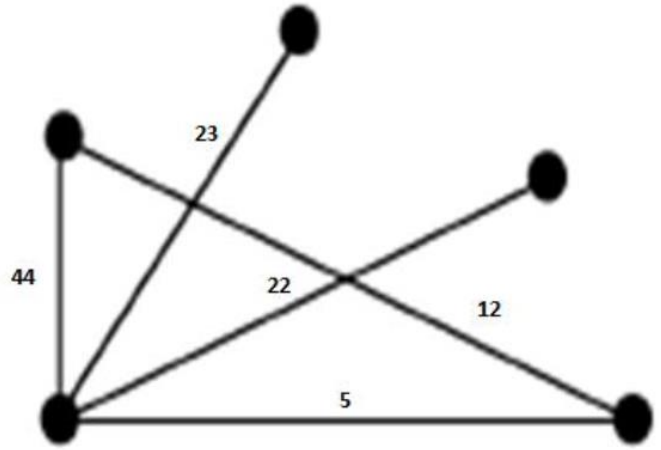


**Graphe orienté :**

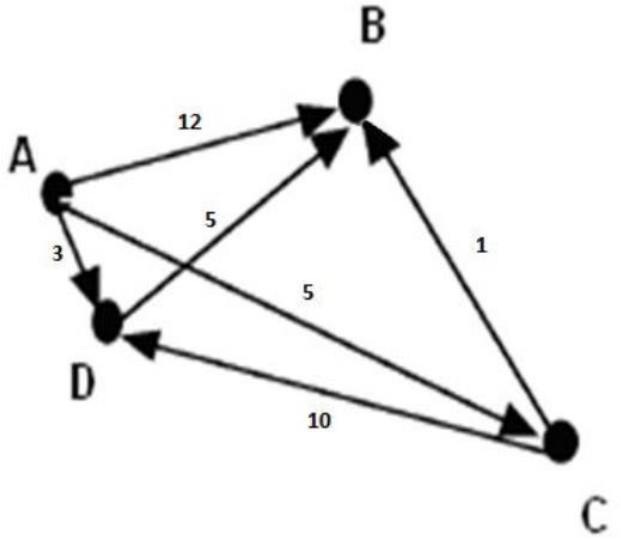


# graphe

**Graphe non orienté pondéré :**



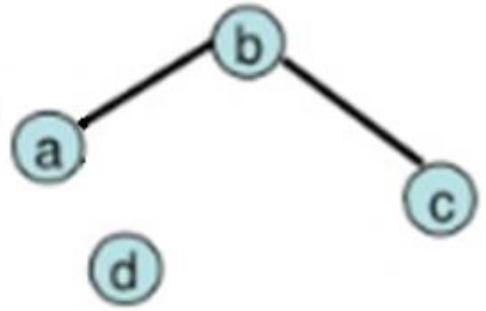

**Graphe orienté pondéré:**



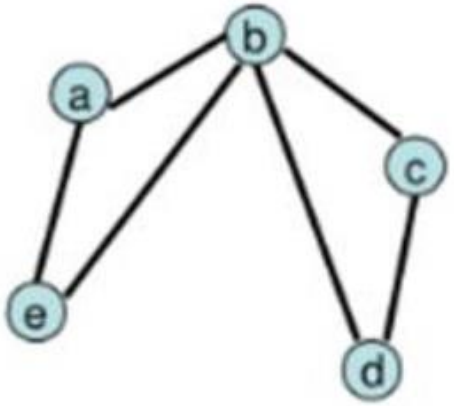
# graphe

- L’ordre d’un graphe est le nombre de ses sommets.

Graphe d’ordre 4

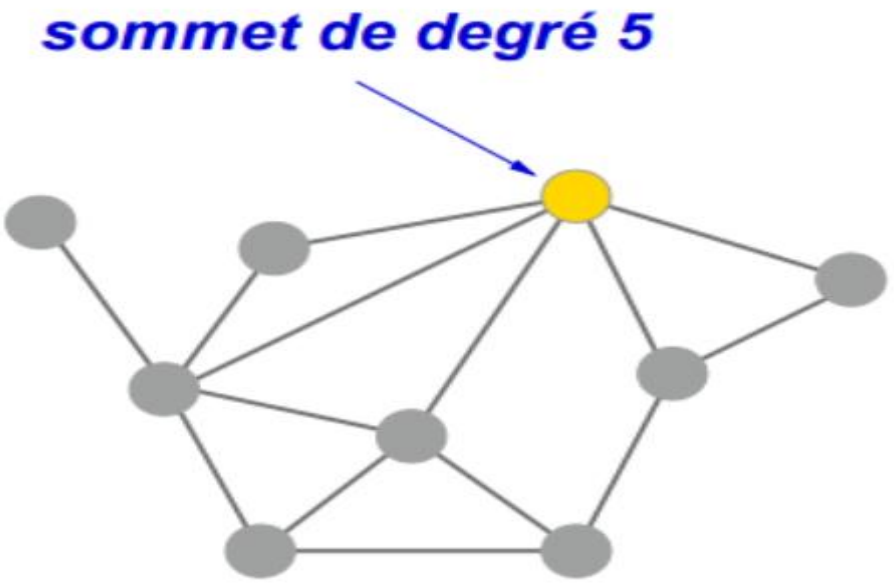



Graphe d’ordre 5



# graphe

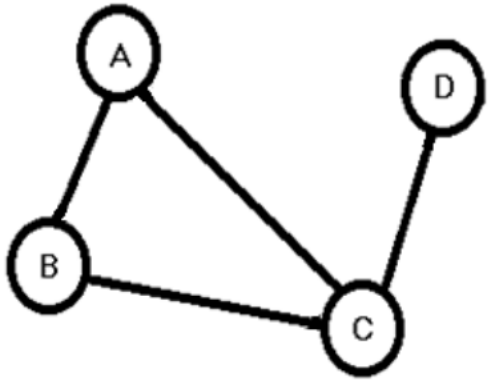
- Degré d’un sommet : nombre d’arêtes reliées à ce sommet.



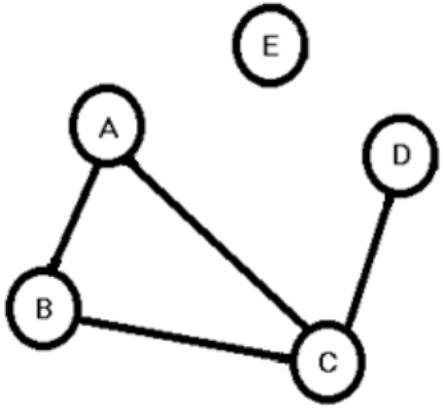
# graphe

- Graphe Connexe : Un graphe connexe est un graphe dont tout couple de sommets peut être relie par une chaine de longueur n>=1

**Graphe connexe :**

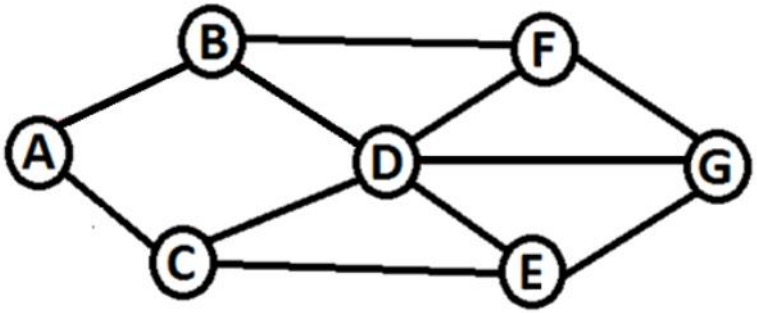



**Graphe non connexe:**

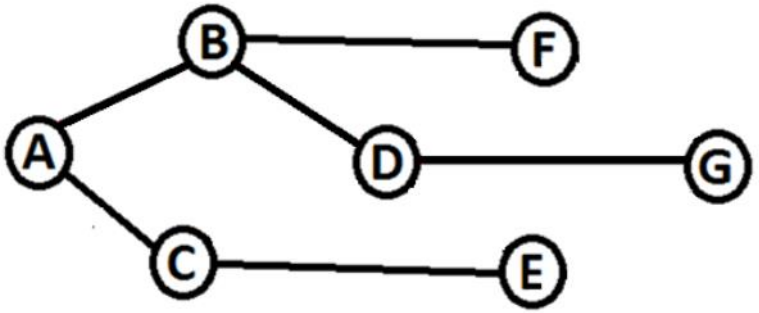


# graphe

- Un graphe est connexe s’il possède un arbre couvrant.
- Algorithme de recherche d’un arbre minimal d’un graphe :
- Algorithme de Kuskal
- Algorithme de Prime




Graphe G

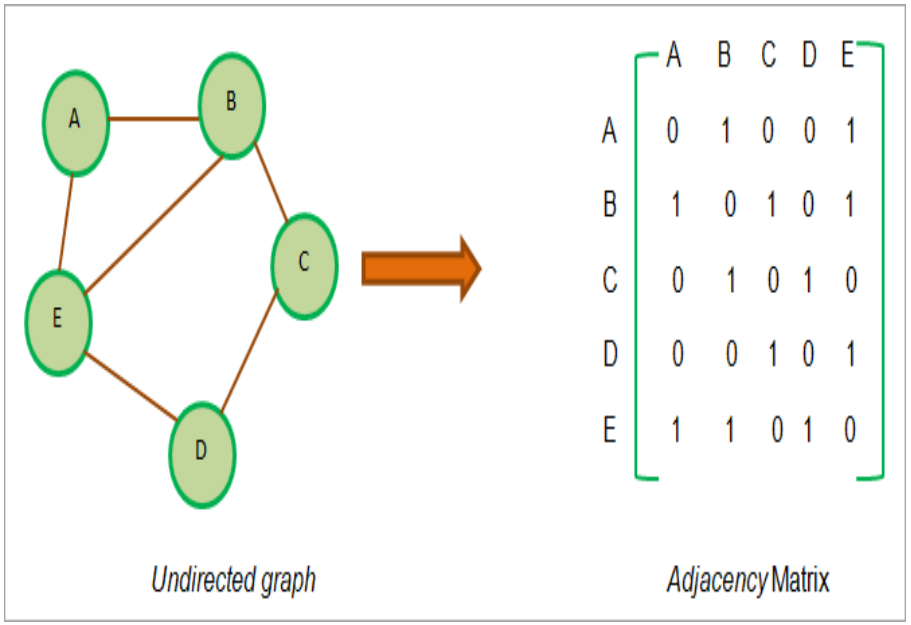



Un Arbre couvrant de G

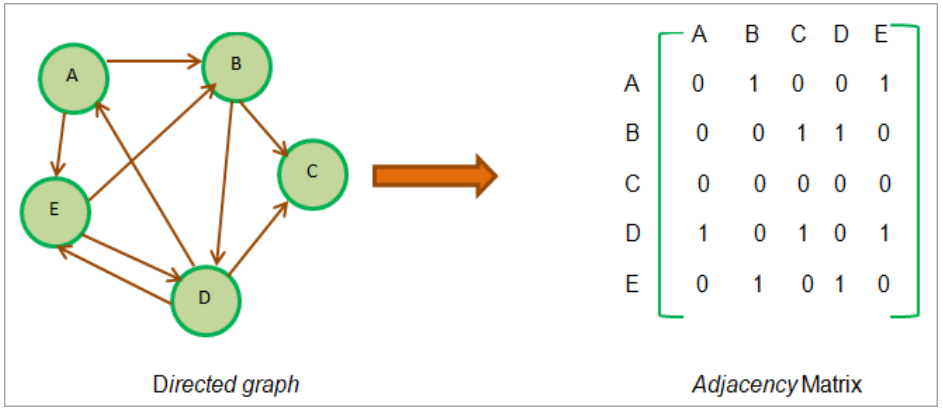
# graphe

- Implémentation des graphes
- Deux implémentations possibles :
- 1. Par les listes de précédence(**Adjacency list**)
- 2. Par la matrice d’adjacences(**Adjacency matrix**)

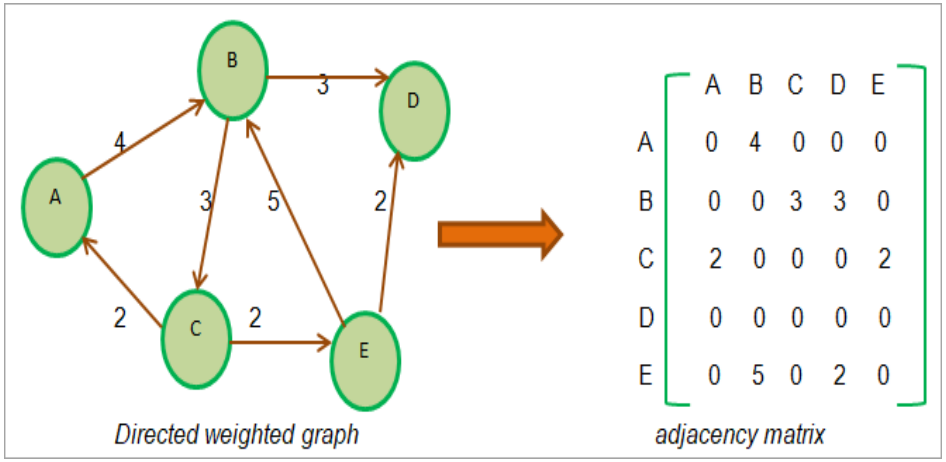
# graphe(Adjacency matrix)



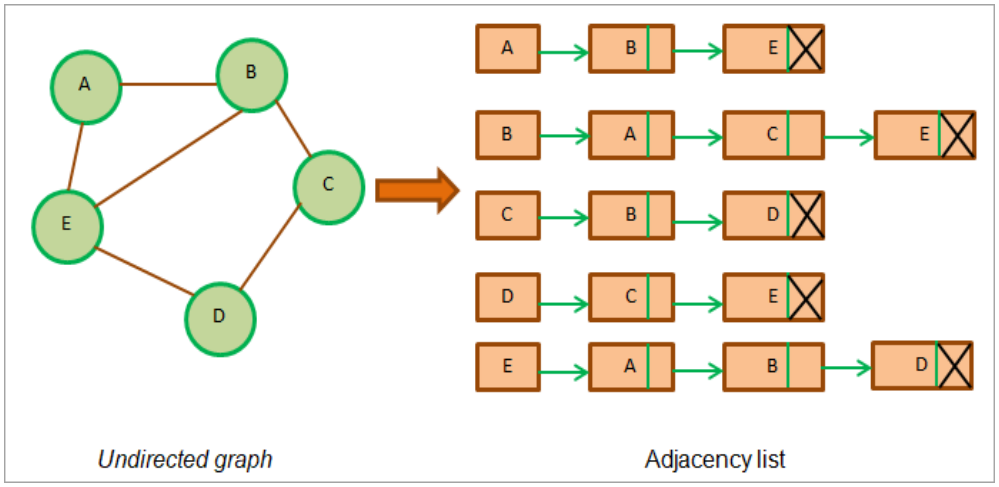
# graphe(Adjacency matrix)



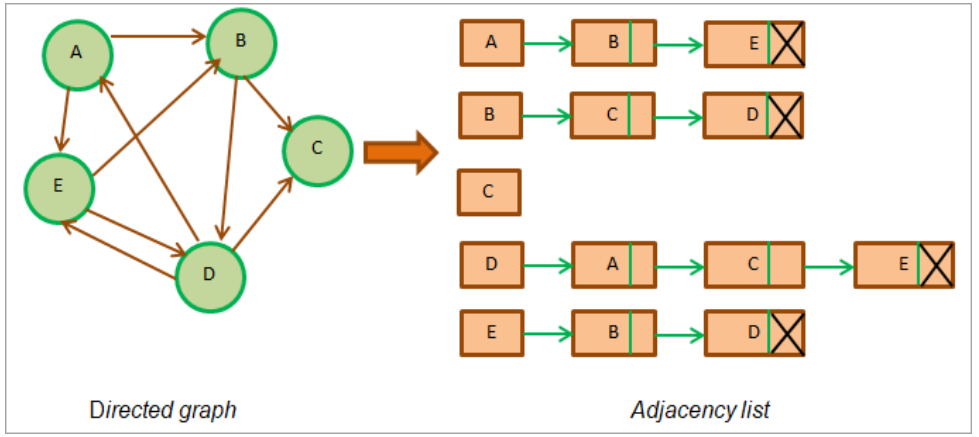
# graphe(Adjacency matrix)



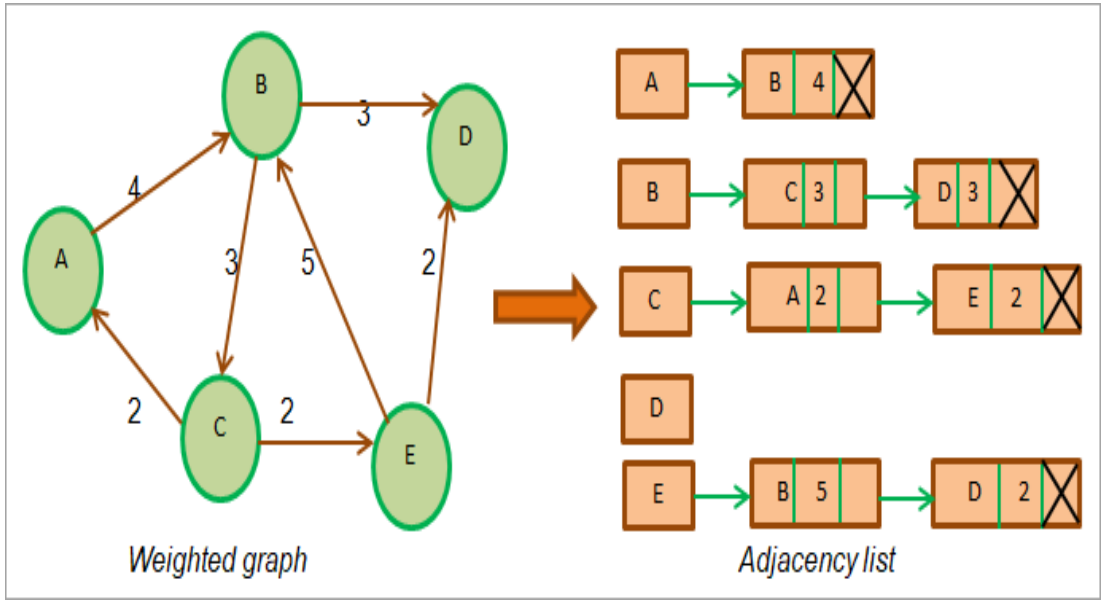
# graphe(Adjacency list)



# graphe(Adjacency list)



# graphe(Adjacency list)



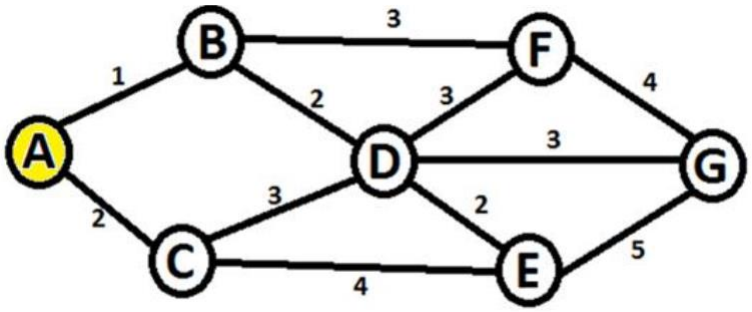
# graphe(BFS)

**Parcours en largeur d’un graphe(BFS):**

- Ce parcours peut êtr utilisé pour :
- Chemin plus court entre deux sommets
- Vérifier si un graphe est connexe
- **Algorithme** :
- 1- S est le sommet de départ
- 2- Cet algorithme liste d'abord les voisins de S pour
- ensuite les explorer un par un.
- 3- Répétez (1) pour chaque voisin.

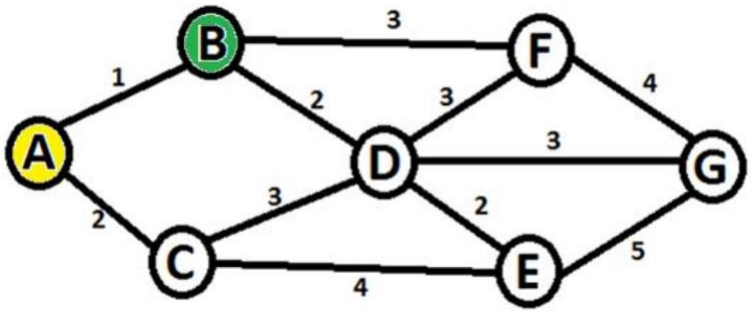
# graphe(BFS)

- Liste des sommets visités : A,



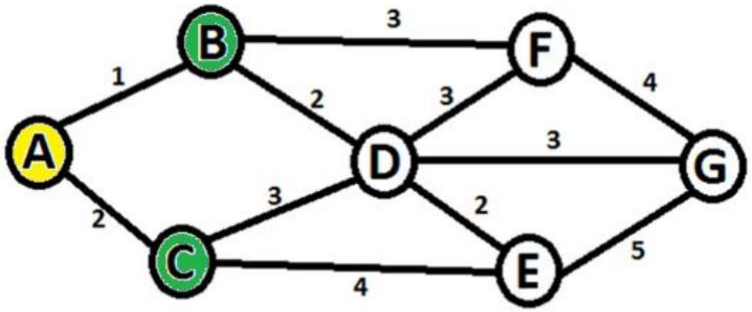
# graphe(BFS)

- Liste des sommets visités : A, B



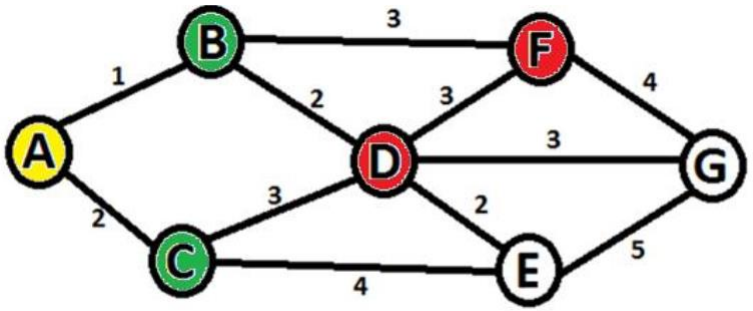
# graphe(BFS)

- Liste des sommets visités : A, B, C



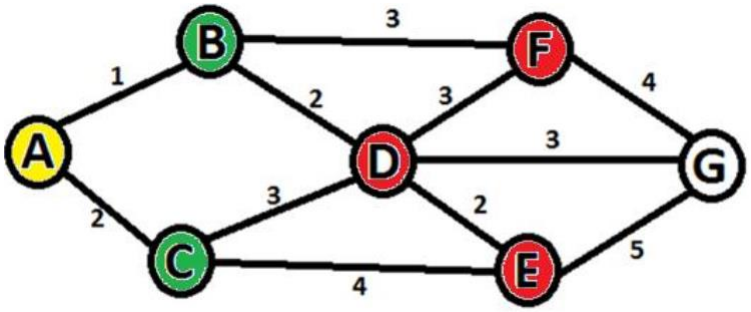
# graphe(BFS)

- Liste des sommets visités : A, B, C, F, D



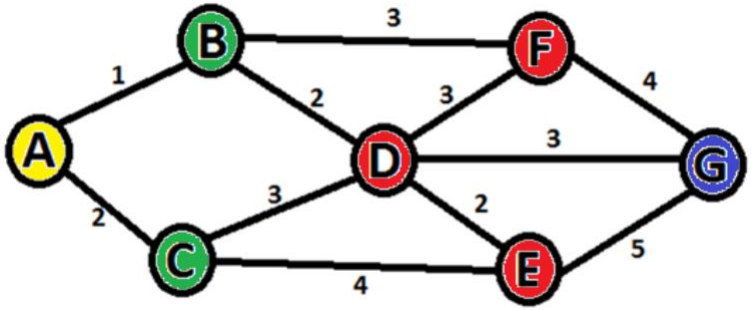
# graphe(BFS)

- Liste des sommets visités : A, B, C, F, D



# graphe(BFS)

- Liste des sommets visités : A, B, C, F, D, E, G



# graphe(DFS)

**.Parcours en profondeur d’un graphe(DFS)**

**.Ce parcours peut être utilisé pour :**

.Chemin plus court entre deux sommets

.Vérifier si un graphe est connexe

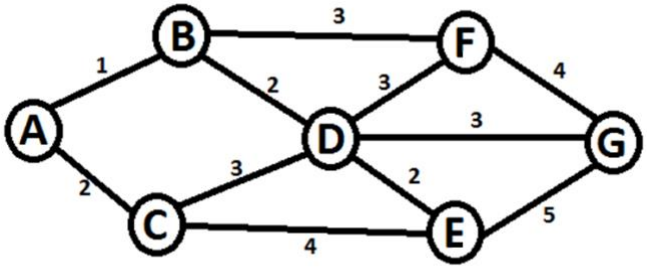
**.Algorithme :**

.Recherche qui progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S :

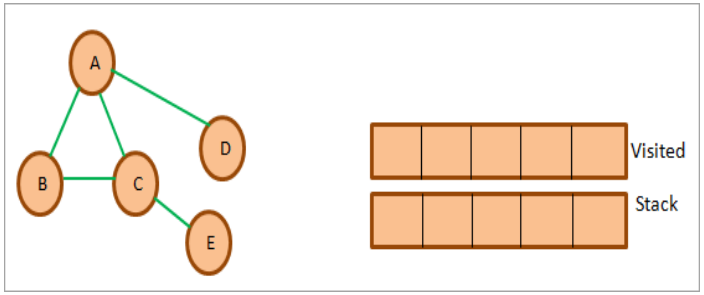
.Pour chaque sommet, il prend le premier sommet voisin jusqu'à ce qu'un sommet n'aie plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père.

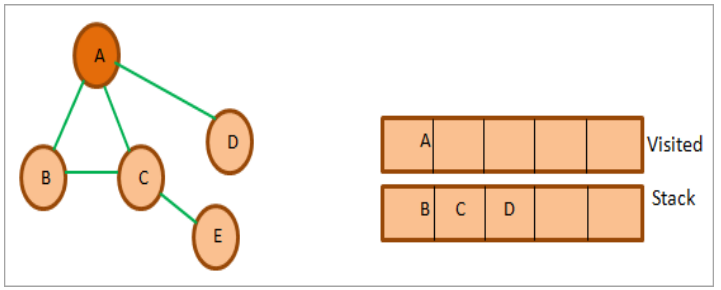
# graphe(DFS)

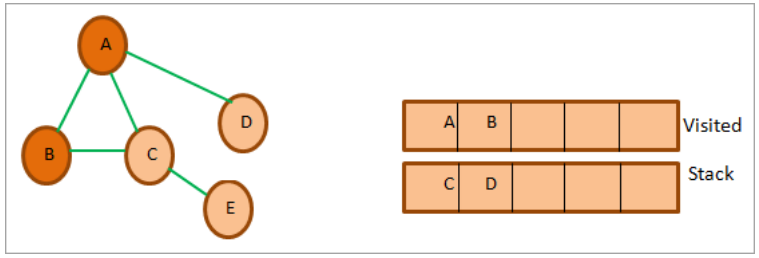
- Le résultat du parcours en profondeur du graphe en dessus à partir de A sera : A, B, F ,G ,D, E, C

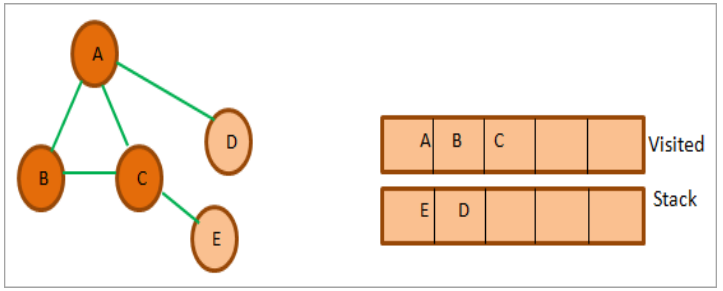


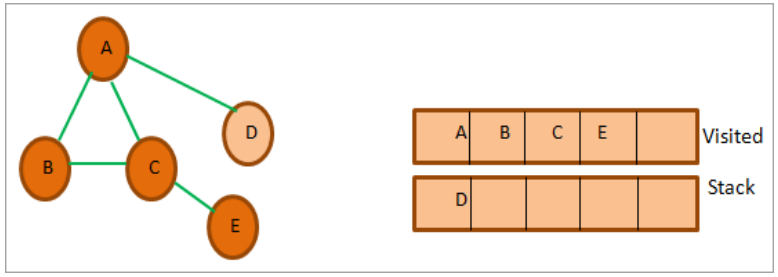
# graphe(DFS)

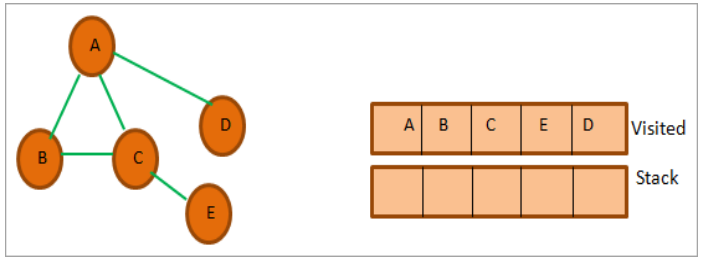

A
B
C
D
E
Visited
Stack


A
B
C
D
E
A
Visited
B C D
Stack

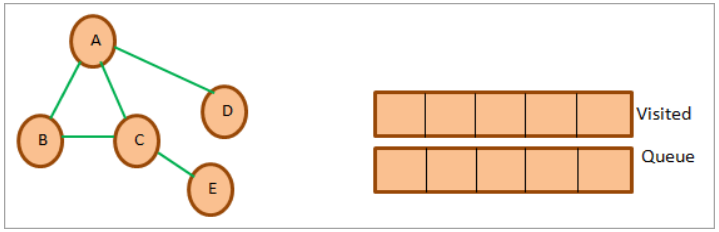

A
B
C
D
E
A B
Visited
C D
Stack

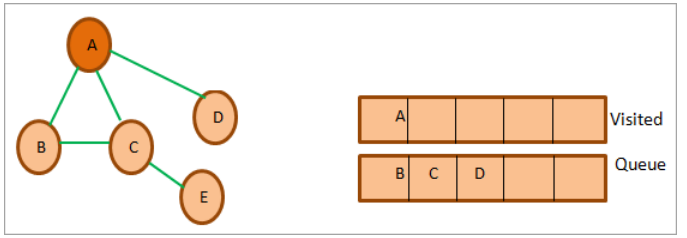

A
B
C
D
E
A B C
Visited
E D
Stack

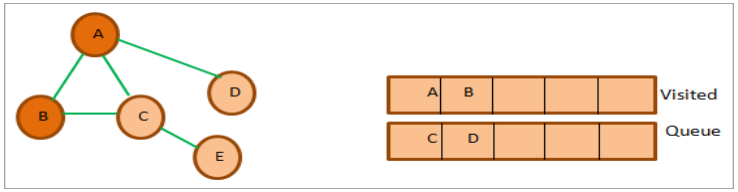

A
B
C
D
E
A B C E
Visited
D
Stack

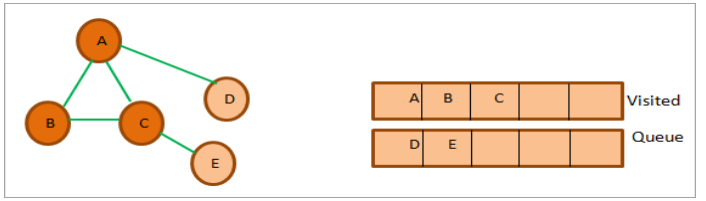

A
B
C
D
E
A B C E D
Visited
Stack

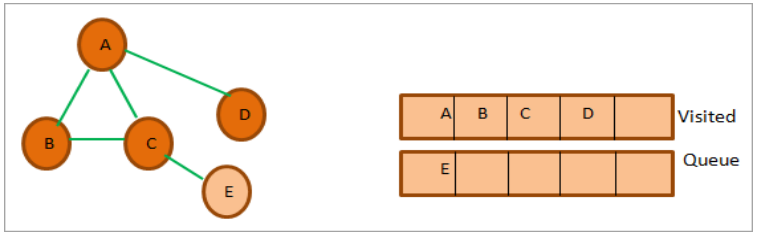
# graphe(BFS)

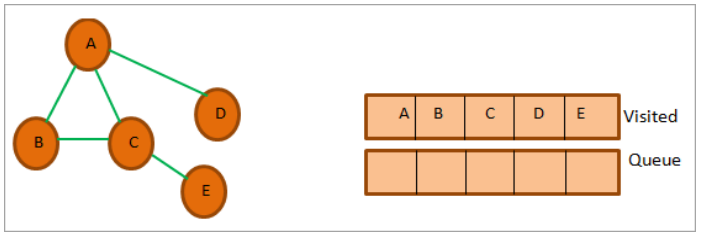

A
B
C
D
E
Visited
Queue


A
B
C
D
E
A
Visited
B C D
Queue


A
B
C
D
E
A B
Visited
C D
Queue


A
B
C
D
E
A B C
Visited
D E
Queue


A
B
C
D
E
A B C D
Visited
E
Queue


A
B
C
D
E
A B C D E
Visited
Queue

# Algorithmes de plus courts chemins

## Algorithmes de plus courts chemins

- Algorithme de Dijkstra
- Algorithme de Dantzig
- Algorithme de Bellman-Moore
- Algorithme de Ford
- Algorithme de Floyd-Warshall
- Algorithme de Ford-Bellman
- Algorithme de Johnson
- Algorithme A*

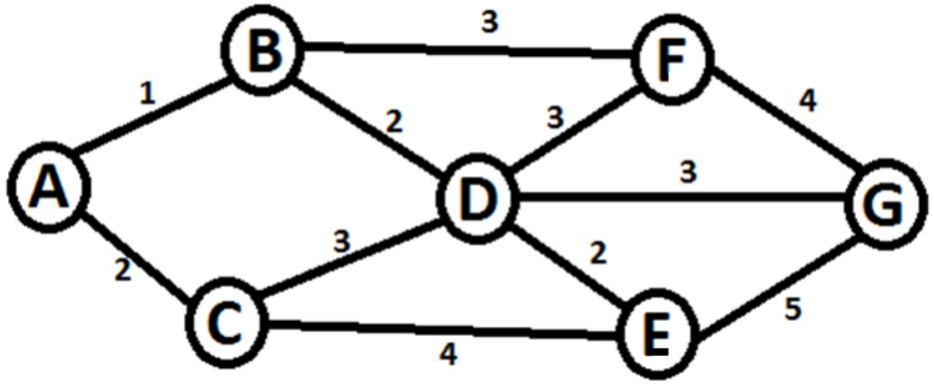
# graphe

## Algorithme de Dijkstra

- En théorie des graphes, l'algorithme de Dijkstra sert à résoudre le problème du plus court chemin.
- Il s'applique à un graphe connexe dont le poids lié aux arêtes est un réel positif.

# graphe

- Trouver le chemin optimal entre A et G



# Algorithme de Dijkstra

- Initialisation de l'algorithme :
- **Étape 1** : On affecte le poids 0 au sommet origine (E) et on attribue provisoirement un poids ∞ aux autres sommets.

# Algorithme de Dijkstra

- Répéter les opérations suivantes de l’étape 2 et 3 tantque le sommet de sortie (s) n'est pas affecté d'un poids définitif :

- **Étape 2** : Parmi les sommets dont le poids n'est pas définitivement fixé choisir le sommet X de poids p minimal. Marquer définitivement ce sommet X affecté du poids p(X).

# Algorithme de Dijkstra

- Étape 3 : Pour tous les sommets Y qui ne sont pas définitivement marqués, adjacents au dernier sommet fixé X :
- Calculer la somme s du poids de X et du poids de l'arête reliant X à Y.
- Si la somme s est inférieure au poids provisoirement affecté au sommet Y, affecter provisoirement à Y le nouveau poids s et indiquer entre parenthèses le sommet X pour se souvenir de sa provenance

# Algorithme de Dijkstra

- Quand le sommet s est définitivement marqué
- Le plus court chemin de E à S s'obtient en écrivant de gauche à droite le parcours en partant de la fin S.

# graphe

Etapes de l'algorithme :

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | | | | | | | |
| Etape 2 | | | | | | | |
| Etape 3 | | | | | | | |
| Etape 4 | | | | | | | |
| Etape 5 | | | | | | | |
| Etape 6 | | | | | | | |
| Etape 7 | | | | | | | |

# graphe

- Règles pour remplir les cases de chaque cellule:
- Soit **e** le sommet de départ.
- 1. La valeur de chaque cellule est un couple : (d(v), p(v))
Où : d(v) est la distance minimale du sommet de départ e au somment v et p(v) représente le sommet précédent du chemin optimal reliant e et v.
- 2. d(e)=0 (e est le sommet de départ)
- 3. d(v)=∞ si la distance est non encore calculé

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | | | | | | |
| Etape 3 | X | | | | | | |
| Etape 4 | X | | | | | | |
| Etape 5 | X | | | | | | |
| Etape 6 | X | | | | | | |
| Etape 7 | X | | | | | | |

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | | | | |
| Etape 3 | X | | | | | | |
| Etape 4 | X | | | | | | |
| Etape 5 | X | | | | | | |
| Etape 6 | X | | | | | | |
| Etape 7 | X | | | | | | |

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | ∞ | ∞ | ∞ | ∞ |
| Etape 3 | X | | | | | | |
| Etape 4 | X | | | | | | |
| Etape 5 | X | | | | | | |
| Etape 6 | X | | | | | | |
| Etape 7 | X | | | | | | |

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | ∞ | ∞ | ∞ | ∞ |
| Etape 3 | X | X | | (3,B) | | (4,B) | |
| Etape 4 | X | X | | | | | |
| Etape 5 | X | X | | | | | |
| Etape 6 | X | X | | | | | |
| Etape 7 | X | X | | | | | |

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | ∞ | ∞ | ∞ | ∞ |
| Etape 3 | X | X | (2,A) | (3,B) | ∞ | (4,B) | ∞ |
| Etape 4 | X | X | | | | | |
| Etape 5 | X | X | | | | | |
| Etape 6 | X | X | | | | | |
| Etape 7 | X | X | | | | | |

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | ∞ | ∞ | ∞ | ∞ |
| Etape 3 | X | X | (2,A) | (3,B) | ∞ | (4,B) | ∞ |
| Etape 4 | X | X | X | (3,B) | (6,C) | | |
| Etape 5 | X | X | X | | | | |
| Etape 6 | X | X | X | | | | |
| Etape 7 | X | X | X | | | | |

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | ∞ | ∞ | ∞ | ∞ |
| Etape 3 | X | X | (2,A) | (3,B) | ∞ | (4,B) | ∞ |
| Etape 4 | X | X | X | (3,B) | (6,C) | (4,B) | ∞ |
| Etape 5 | X | X | X | | | | |
| Etape 6 | X | X | X | | | | |
| Etape 7 | X | X | X | | | | |

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | ∞ | ∞ | ∞ | ∞ |
| Etape 3 | X | X | (2,A) | (3,B) | ∞ | (4,B) | ∞ |
| Etape 4 | X | X | X | (3,B) | (6,C) | (4,B) | ∞ |
| Etape 5 | X | X | X | X | (5,D) | (4,B) | (6,D) |
| Etape 6 | X | X | X | X | | | |
| Etape 7 | X | X | X | X | | | |

# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | ∞ | ∞ | ∞ | ∞ |
| Etape 3 | X | X | (2,A) | (3,B) | ∞ | (4,B) | ∞ |
| Etape 4 | X | X | X | (3,B) | (6,C) | (4,B) | ∞ |
| Etape 5 | X | X | X | X | (5,D) | (4,B) | (6,D) |
| Etape 6 | X | X | X | X | (5,D) | X | (6,D) |
| Etape 7 | X | X | X | X | | X | |

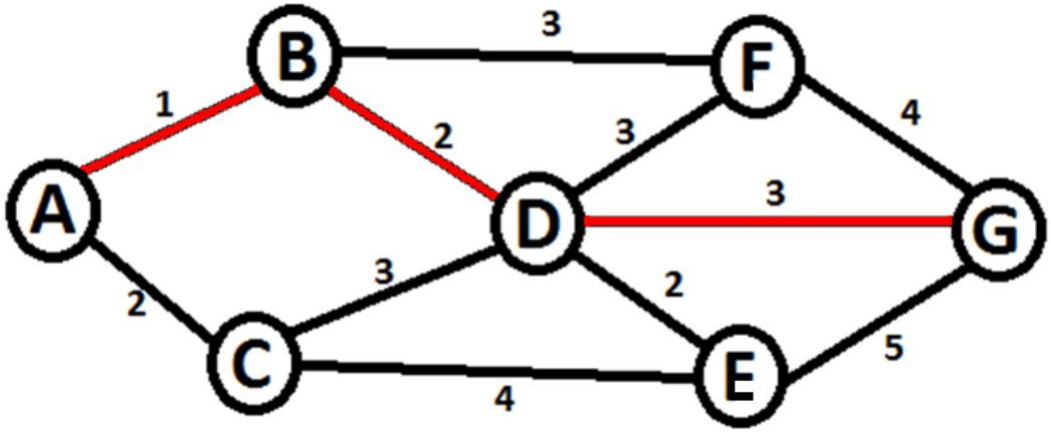
# graphe

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Etape 2 | X | (1,A) | (2,A) | ∞ | ∞ | ∞ | ∞ |
| Etape 3 | X | X | (2,A) | (3,B) | ∞ | (4,B) | ∞ |
| Etape 4 | X | X | X | (3,B) | (6,C) | (4,B) | ∞ |
| Etape 5 | X | X | X | X | (5,D) | (4,B) | (6,D) |
| Etape 6 | X | X | X | X | (5,D) | X | (6,D) |
| Etape 7 | X | X | X | X | X | X | (6,D) |

# graphe

- Le chemin optimal est : ABDG de coût égal à 6

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Etape 1 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| ... | | | | | | | |
| Etape 7 | (0,A) | (1,A) | (2,A) | (3,B) | (5,D) | (4,B) | (6,D) |



# MST

**Algorithmes d'arbres couvrants de poids minimum**

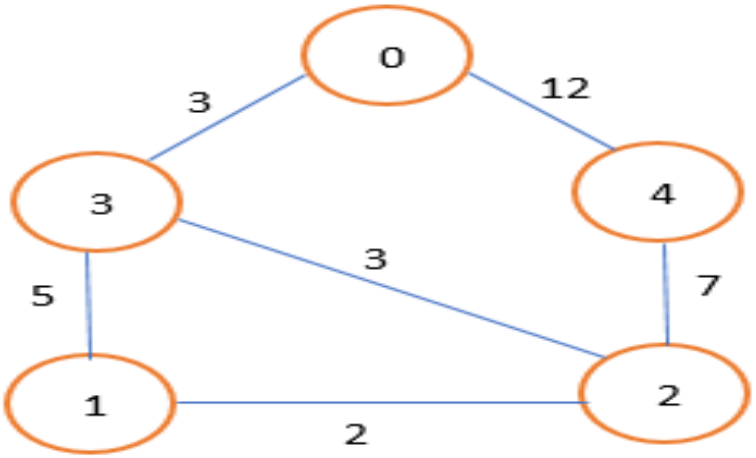
- Algorithme de Kruskal
- Algorithme de Prim
- Algorithme de Borůvka

# .graphe(MST:Prim)

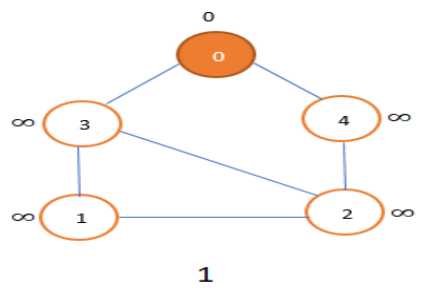
## Algorithme de Prim:

Le principe de l'algorithme de Prim est de partir d'un arbre initial réduit à un seul sommet, puis d'augmenter à chaque itération la taille de l'arbre en le connectant au plus proche voisin libre

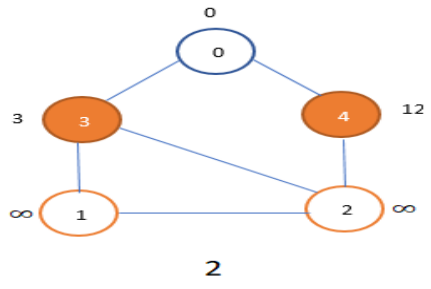
# graphe



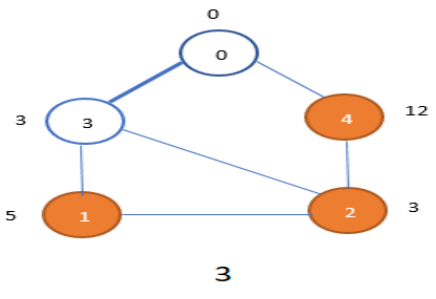
# graphe



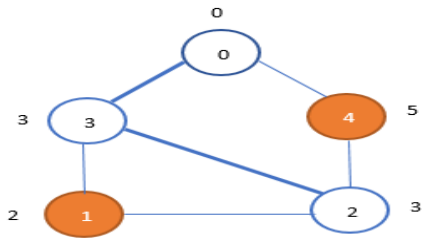

1



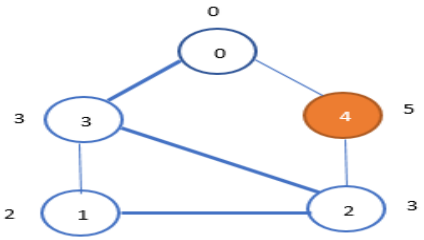

2



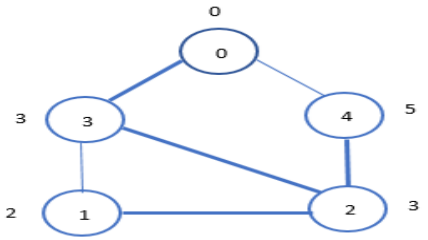

3



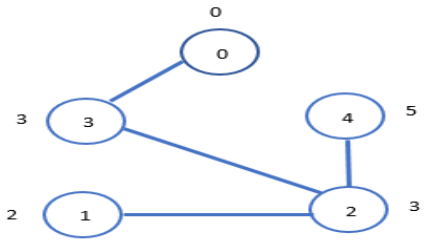

4



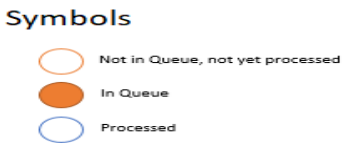

5




6




7



# graphe

- Prim’s Algorithm

T = {s}

enqueue edges connected to s in PQ (by inc weight)

while (!PQ.isEmpty)

  if (vertex v linked with e = PQ.remove ∉ T)

    T = T ∪ {v, e}, enqueue edges connected to v

  else ignore e

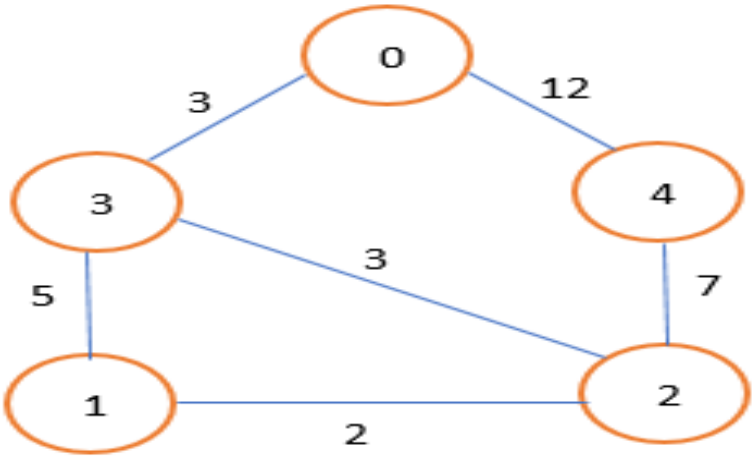
MST = T

# graphe(MST:Kruskal)

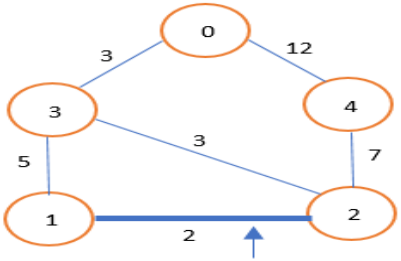
## Kruskal’s Algorithm

L'algorithme de Kruskal est conceptuellement différent de celui de Prim, puisqu'il part d'une forêt de  arbres réduits chacun à un sommet isolé de  , puis à chaque itération on relie deux arbres par l'arête de plus petit poids, jusqu'à ce qu'il n'y ait qu'un seul arbre dans la forêt, l'arbre recouvrant de poids minima

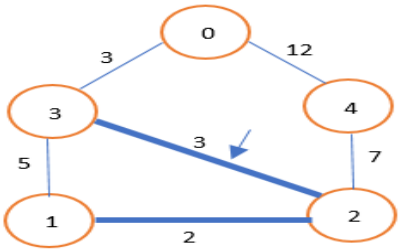
# graphe



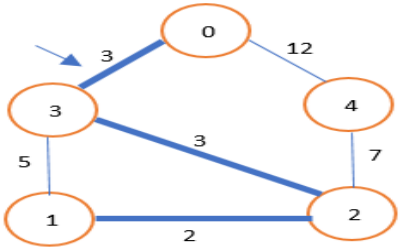
# graphe



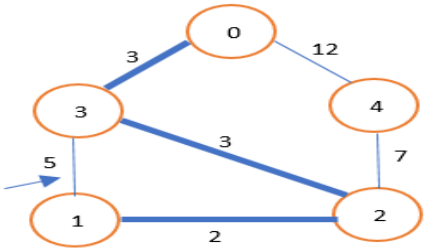

1



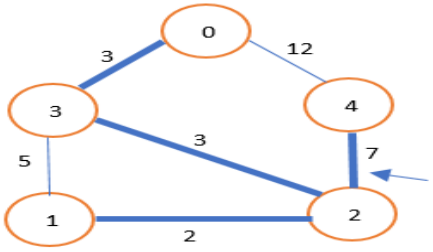

2



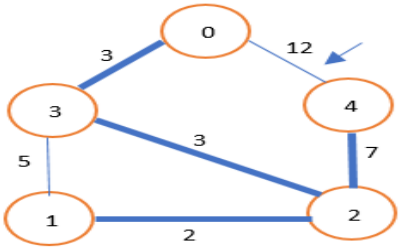

3



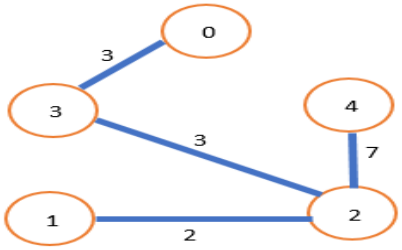

4




5




6




7

## Kruskal’s Algorithm

```
Sort E edges by increasing weight
T = {}
for (i = 0; i < edgeList.length; i++)
  if adding e = edgelist[i] does not form a cycle
    add e to T
  else ignore e
MST = T
```