

# Splay Tree

Splay trees are the self-balancing or self-adjusted binary search trees. In other words, we can say that the splay trees are the variants of the binary search trees. The prerequisite for the splay trees that we should know about the binary search trees.

the time complexity of a binary search tree in every case. The time complexity of a binary search tree in the average case is **O(logn)** and the time complexity in the worst case is O(n). In a binary search tree, the value of the left subtree is smaller than the root node, and the value of the right subtree is greater than the root node; in such case, the time complexity would be **O(logn)**. If the binary tree is left-skewed or right-skewed, then the time complexity would be O(n).

To limit the skewness, the [AVL and Red-Black tree](#) came into the picture, having **O(logn)** time complexity for all the operations in all the cases. We can also improve this time complexity by doing more practical implementations, so the new Tree [data structure](#) was designed, known as a Splay tree.

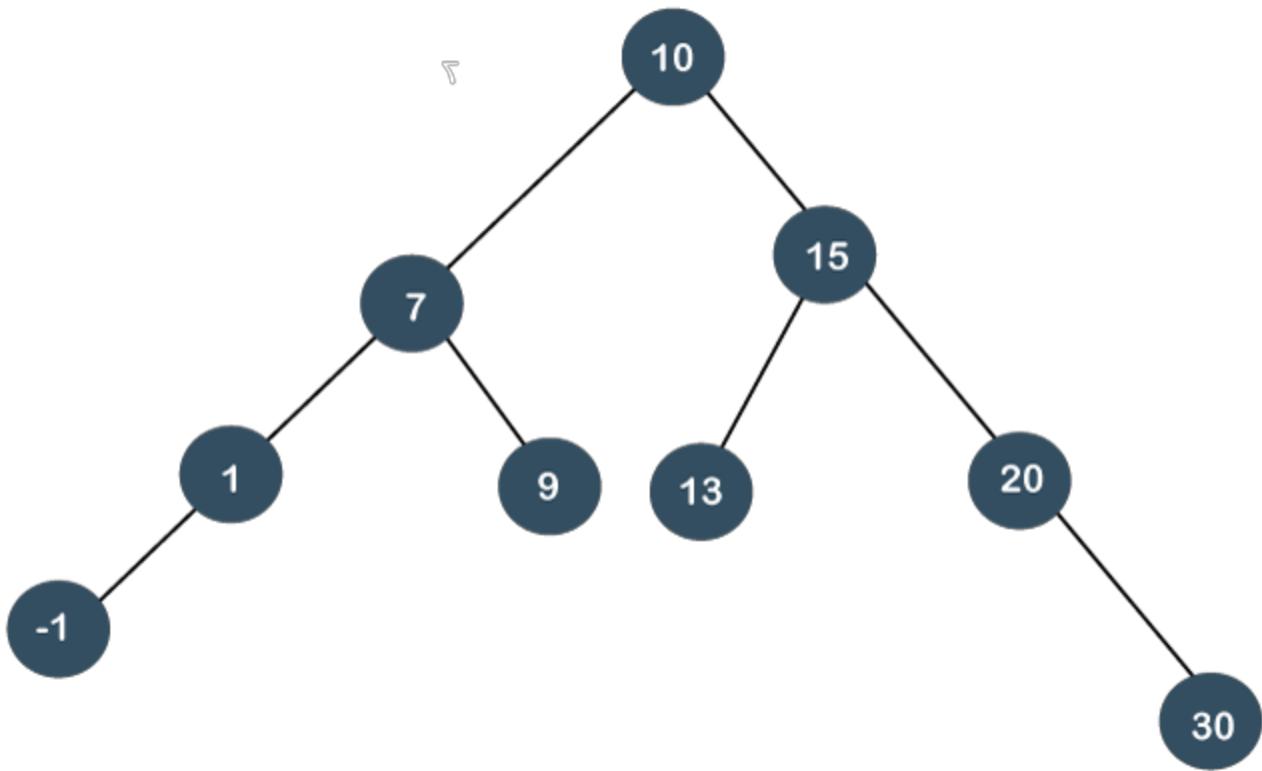
## What is a Splay Tree?

A splay tree is a self-balancing tree, but [AVL](#) and [Red-Black trees](#) are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is splaying.

A splay tree contains the same operations as a [Binary search tree](#), i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.

Splay trees are not strictly balanced trees, but they are roughly balanced trees. Let's understand the search operation in the splay-tree.

Suppose we want to search 7 element in the tree, which is shown below:



To search any element in the splay tree, first, we will perform the standard binary search tree operation. As 7 is less than 10 so we will come to the left of the root node. After performing the search operation, we need to perform splaying. Here splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements. The rearrangement of the tree will be done through the rotations.

## Rotations

**There are six types of rotations used for splaying:**

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

## **Factors required for selecting a type of rotation**

**The following are the factors used for selecting a type of rotation:**

- Does the node which we are trying to rotate have a grandparent?
- Is the node left or right child of the parent?
- Is the node left or right child of the grandparent?

## **Cases for the Rotations**

**Case 1:** If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

**Case 2:** If the node has a grandparent, then based on the following scenarios; the rotation would be performed:

**Scenario 1:** If the node is the right of the parent and the parent is also right of its parent, then ***zig zig right right rotation*** is performed.

**Scenario 2:** If the node is left of a parent, but the parent is right of its parent, then ***zig zag right left rotation*** is performed.

**Scenario 3:** If the node is right of the parent and the parent is right of its parent, then ***zig zig left left rotation*** is performed.

**Scenario 4:** If the node is right of a parent, but the parent is left of its parent, then ***zig zag right-left rotation*** is performed.

**Now, let's understand the above rotations with examples.**

To rearrange the tree, we need to perform some rotations. The following are the types of rotations in the splay tree:

- **Zig rotations**

The zig rotations are used when the item to be searched is either a root node or the child of a root node (i.e., left or the right child).

**The following are the cases that can exist in the splay tree while searching:**

**Case 1:** If the search item is a root node of the tree.

**Case 2:** If the search item is a child of the root node, then the two scenarios will be there:

1. If the child is a left child, the right rotation would be performed, known as a zig right rotation.
2. If the child is a right child, the left rotation would be performed, known as a zig left rotation.

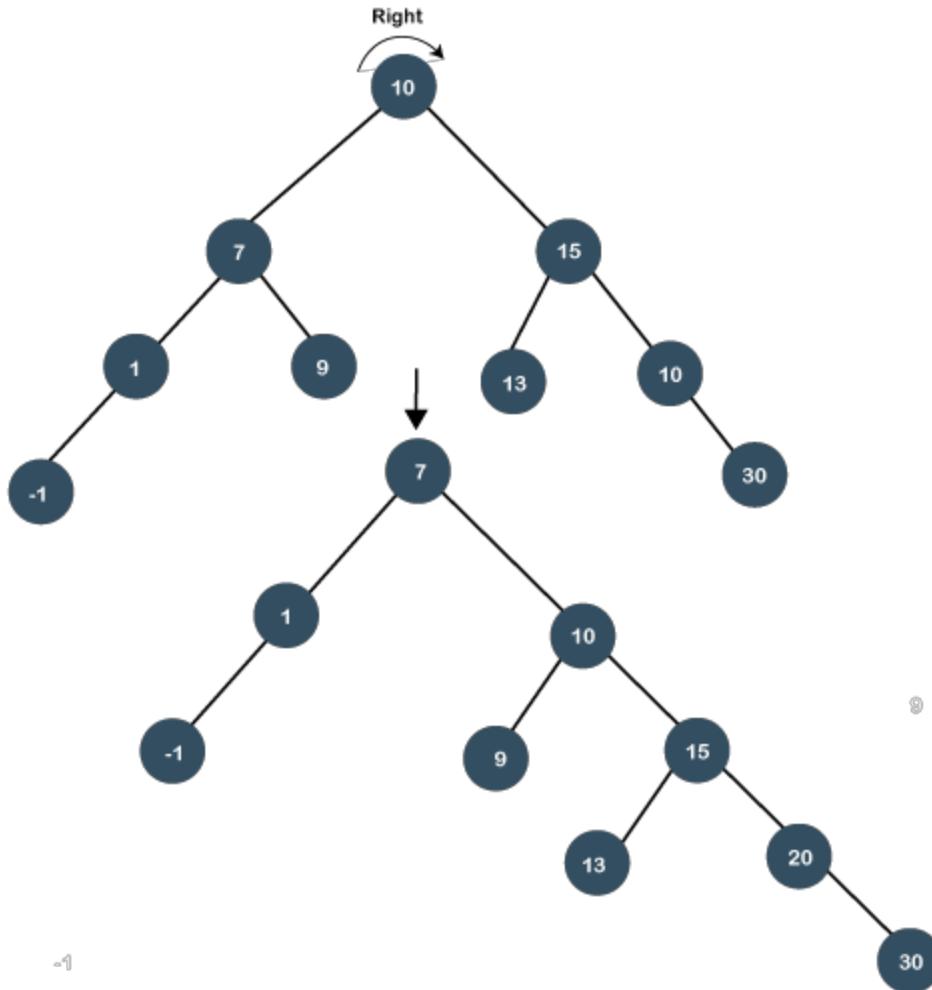
**Let's look at the above two scenarios through an example.**

**Consider the below example:**

In the above example, we have to search 7 element in the tree. We will follow the below steps:

**Step 1:** First, we compare 7 with a root node. As 7 is less than 10, so it is a left child of the root node.

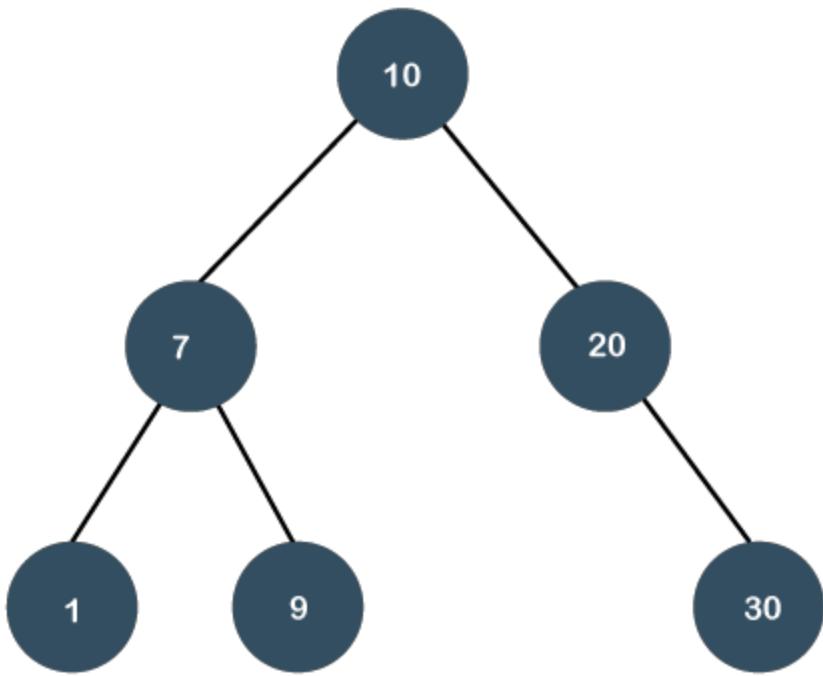
**Step 2:** Once the element is found, we will perform splaying. The right rotation is performed so that 7 becomes the root node of the tree, as shown below:



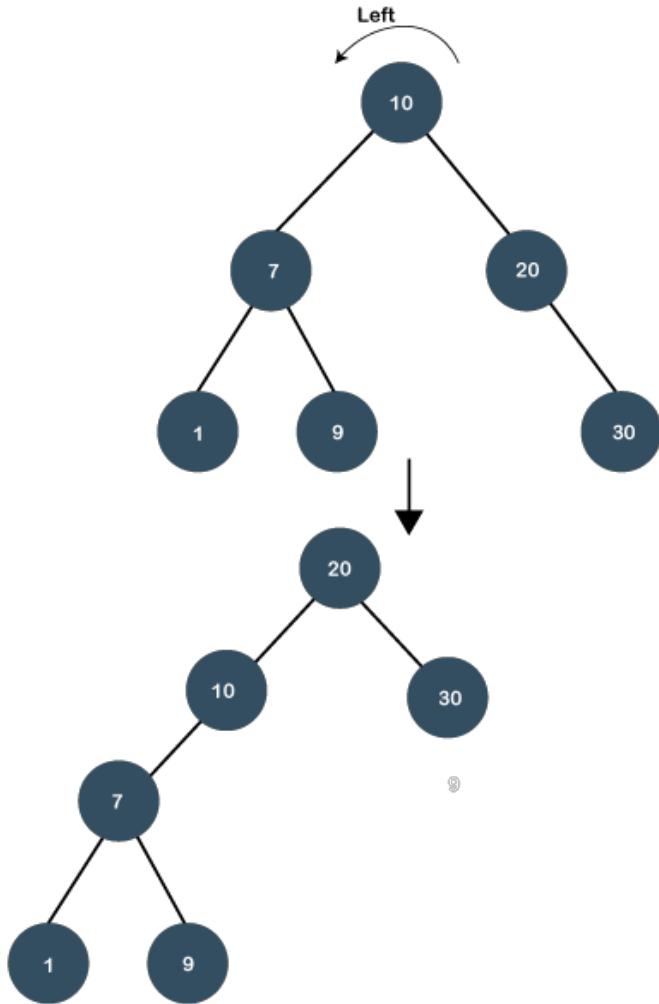
**Let's consider another example.**

In the above example, we have to search 20 element in the tree. We will follow the below steps:

**Step 1:** First, we compare 20 with a root node. As 20 is greater than the root node, so it is a right child of the root node.



**Step 2:** Once the element is found, we will perform splaying. The left rotation is performed so that 20 element becomes the root node of the tree.



- o **Zig zig rotations**

Sometimes the situation arises when the item to be searched is having a parent as well as a grandparent. In this case, we have to perform four rotations for splaying.

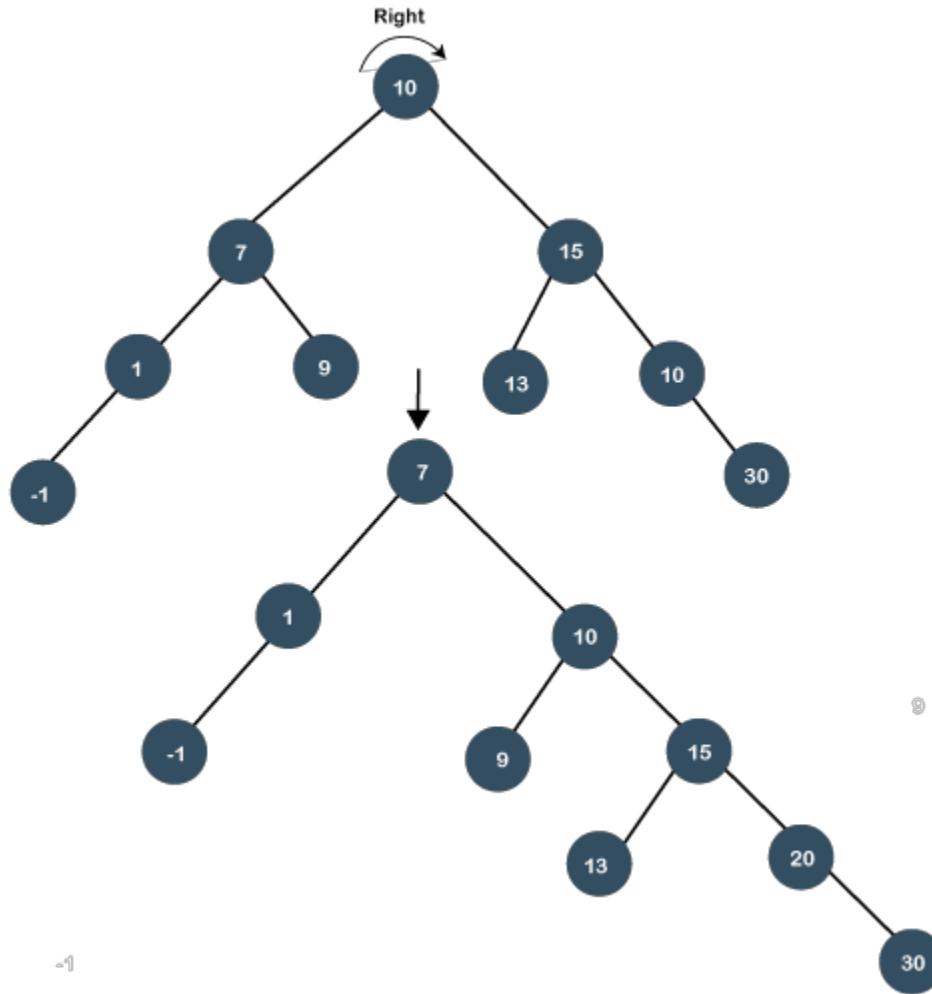
Let's understand this case through an example.

Suppose we have to search 1 element in the tree, which is shown below:

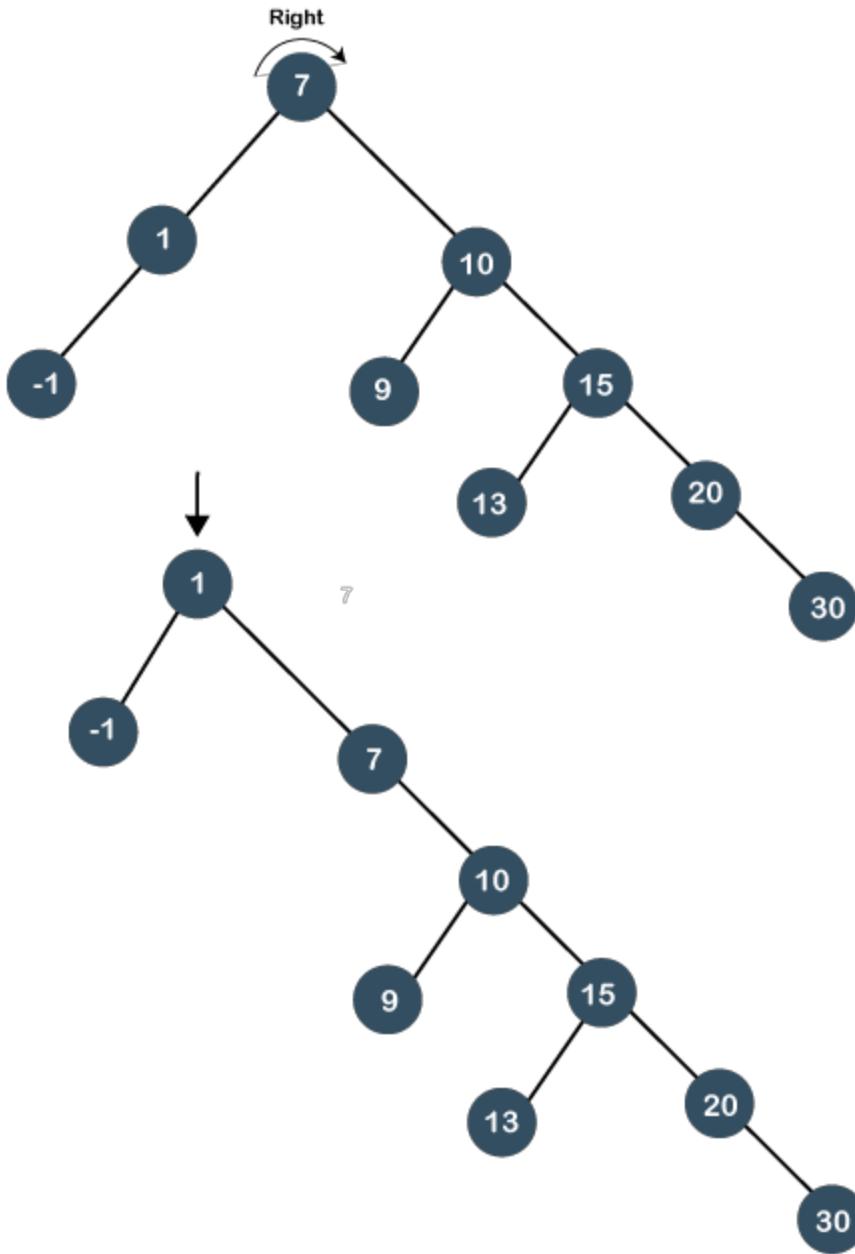
**Step 1:** First, we have to perform a standard BST searching operation in order to search the 1 element. As 1 is less than 10 and 7, so it will be at the left of the node 7. Therefore, element 1 is having a parent, i.e., 7 as well as a grandparent, i.e., 10.

**Step 2:** In this step, we have to perform splaying. We need to make node 1 as a root node with the help of some rotations. In this case, we cannot simply perform a zig or zag rotation; we have to implement zig zig rotation.

In order to make node 1 as a root node, we need to perform two right rotations known as zig zig rotations. When we perform the right rotation then 10 will move downwards, and node 7 will come upwards as shown in the below figure:



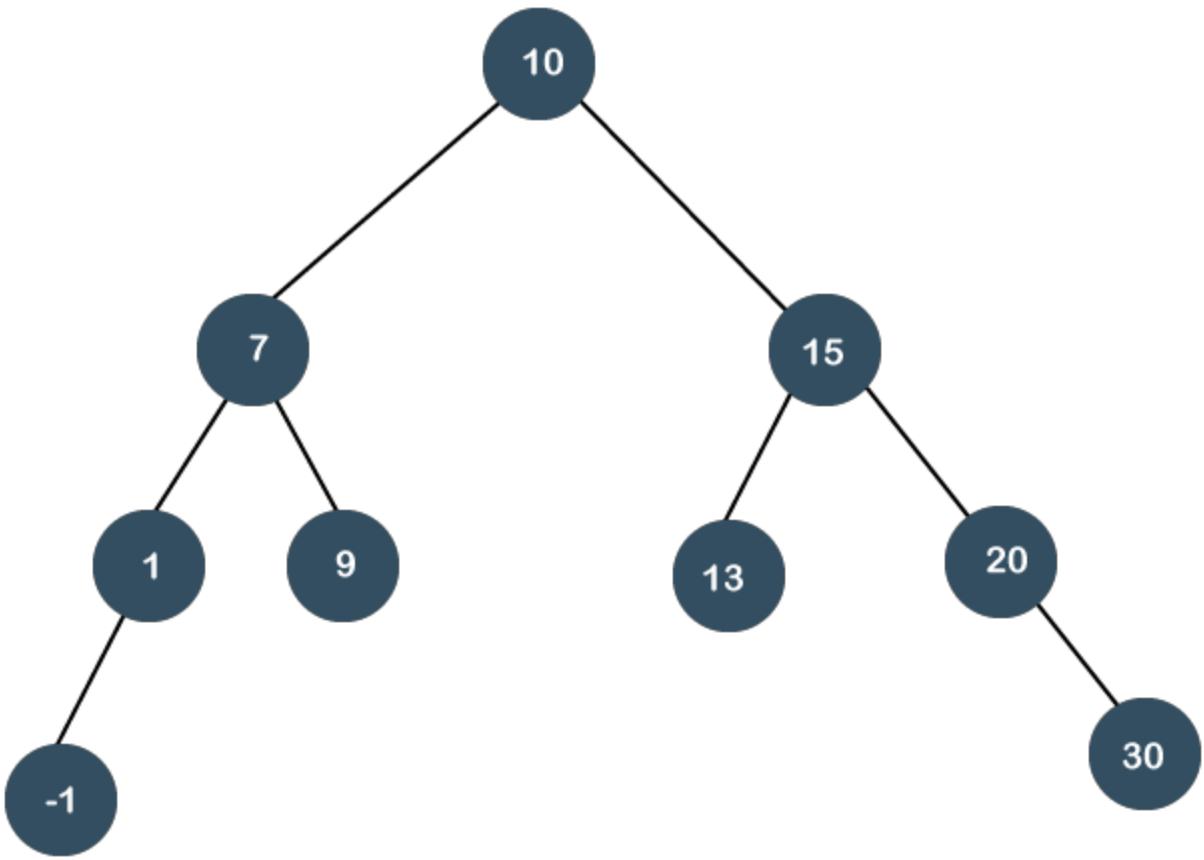
Again, we will perform zig right rotation, node 7 will move downwards, and node 1 will come upwards as shown below:



As we observe in the above figure that node 1 has become the root node of the tree; therefore, the searching is completed.

**Suppose we want to search 20 in the below tree.**

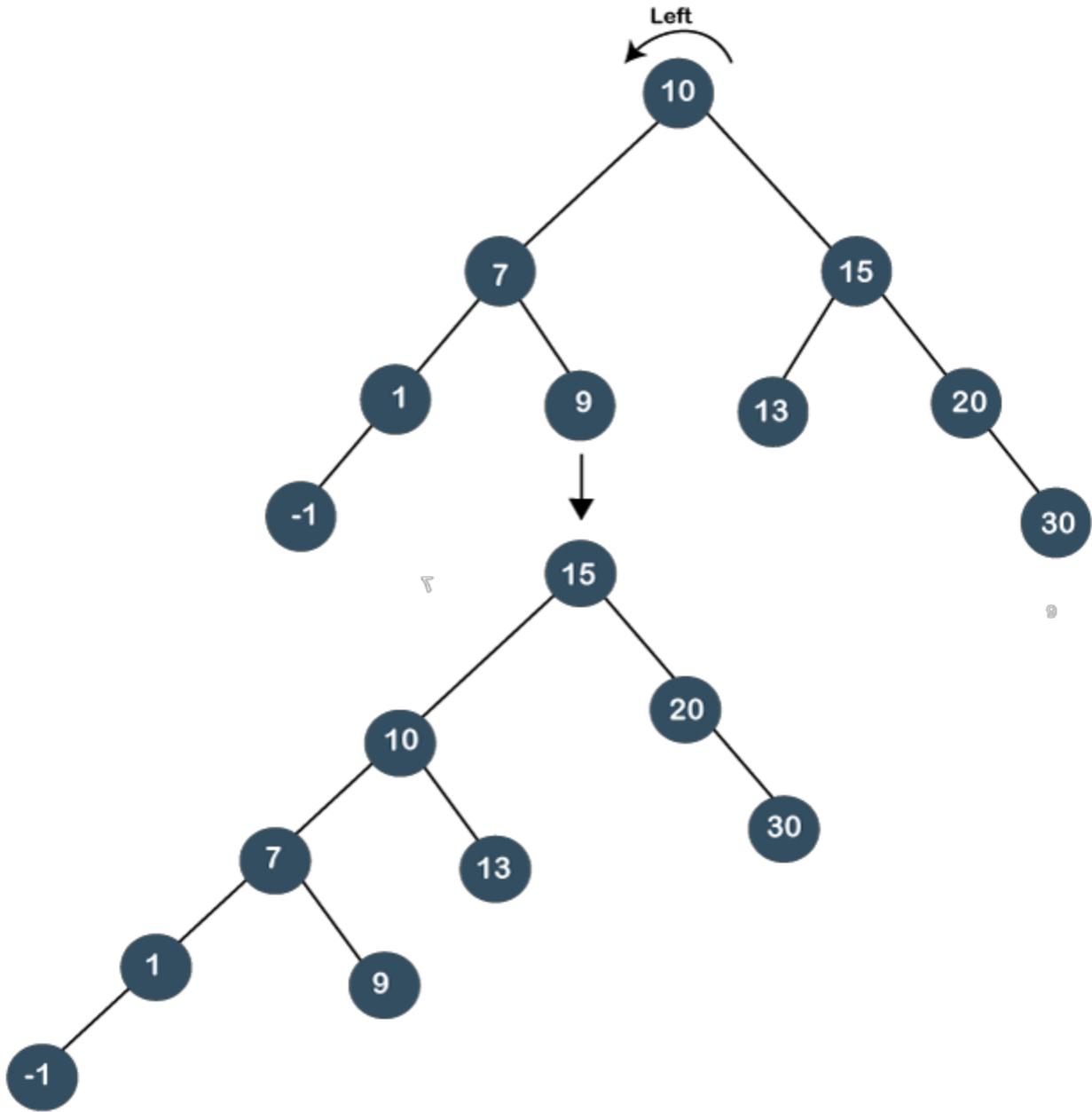
In order to search 20, we need to perform two left rotations. Following are the steps required to search 20 node:



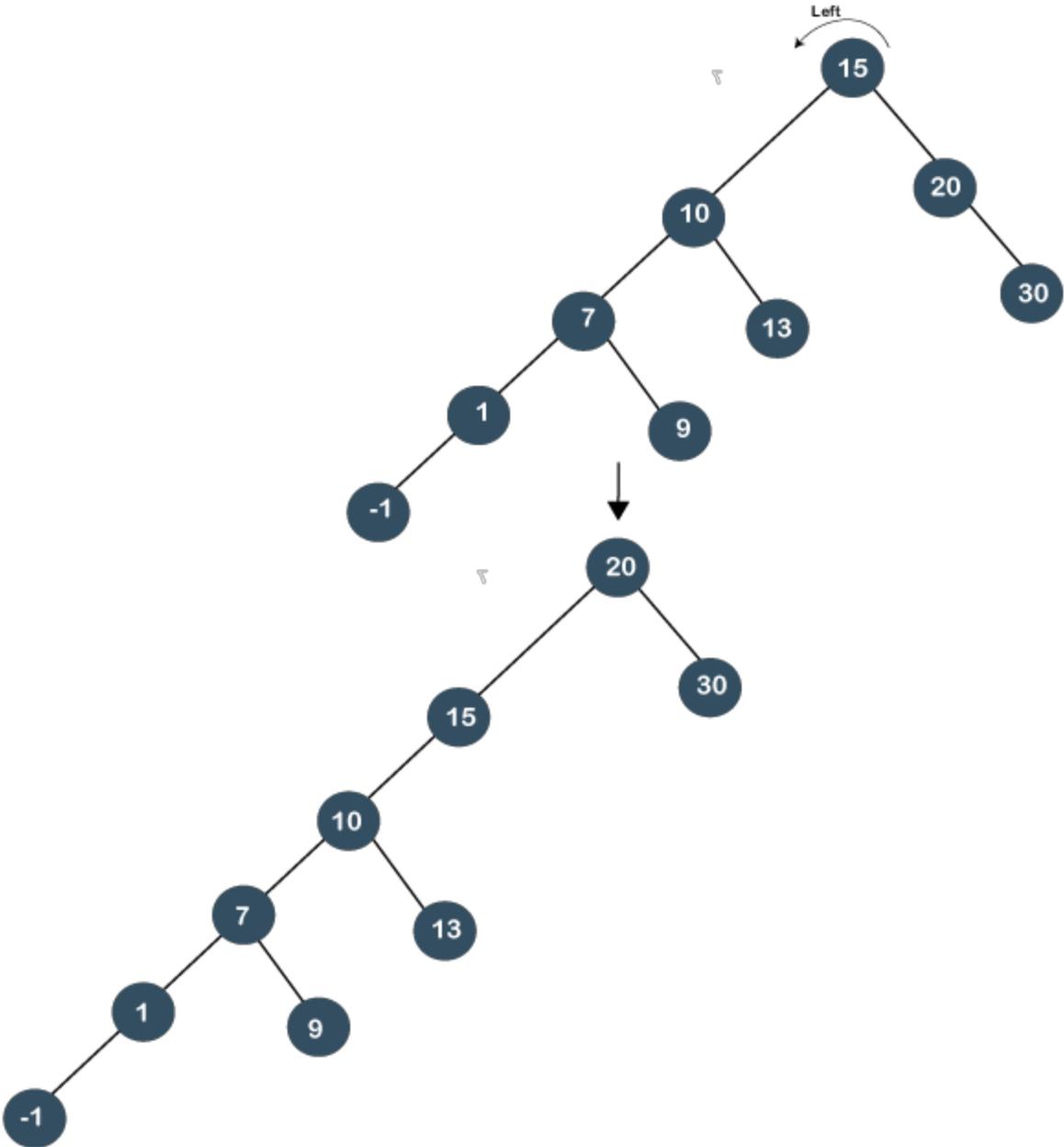
9

**Step 1:** First, we perform the standard BST searching operation. As 20 is greater than 10 and 15, so it will be at the right of node 15.

**Step 2:** The second step is to perform splaying. In this case, two left rotations would be performed. In the first rotation, node 10 will move downwards, and node 15 would move upwards as shown below:



In the second left rotation, node 15 will move downwards, and node 20 becomes the root node of the tree, as shown below:



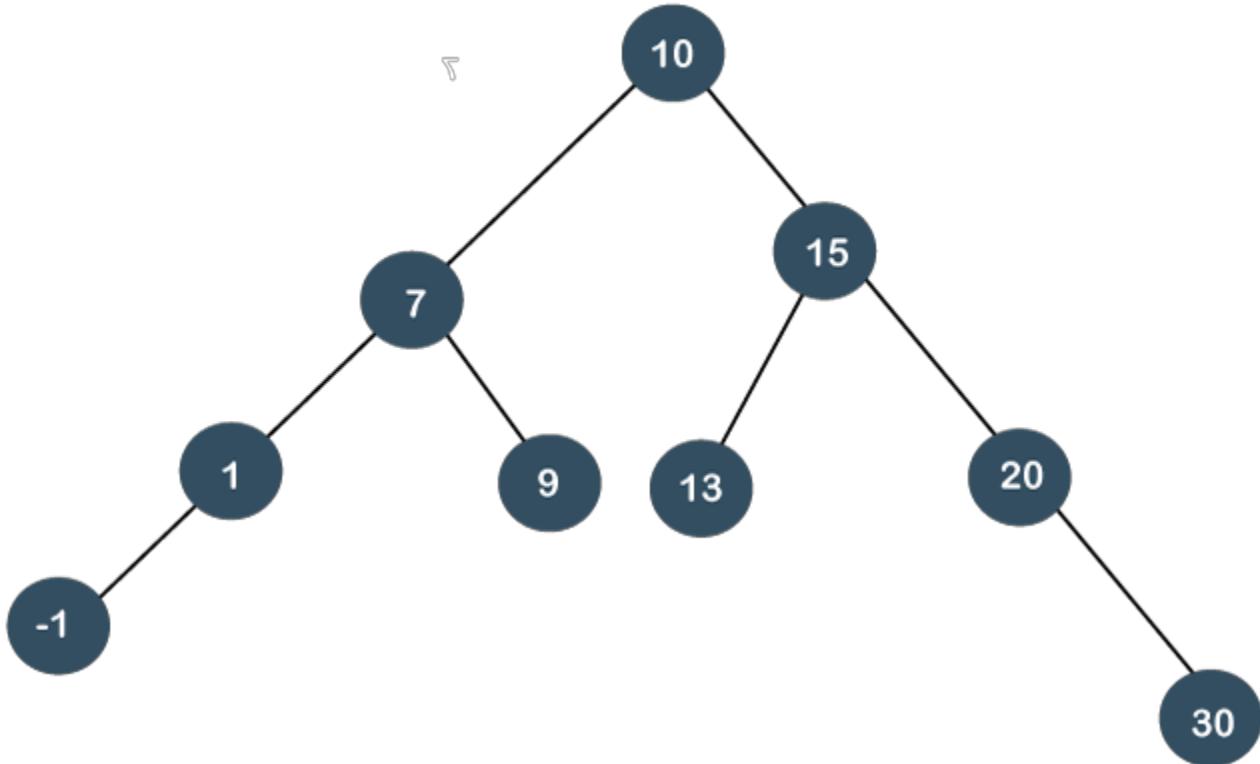
As we have observed that two left rotations are performed; so it is known as a zig zig left rotation.

- **Zig zag rotations**

Till now, we have read that both parent and grandparent are either in RR or LL relationship. Now, we will see the RL or LR relationship between the parent and the grandparent.

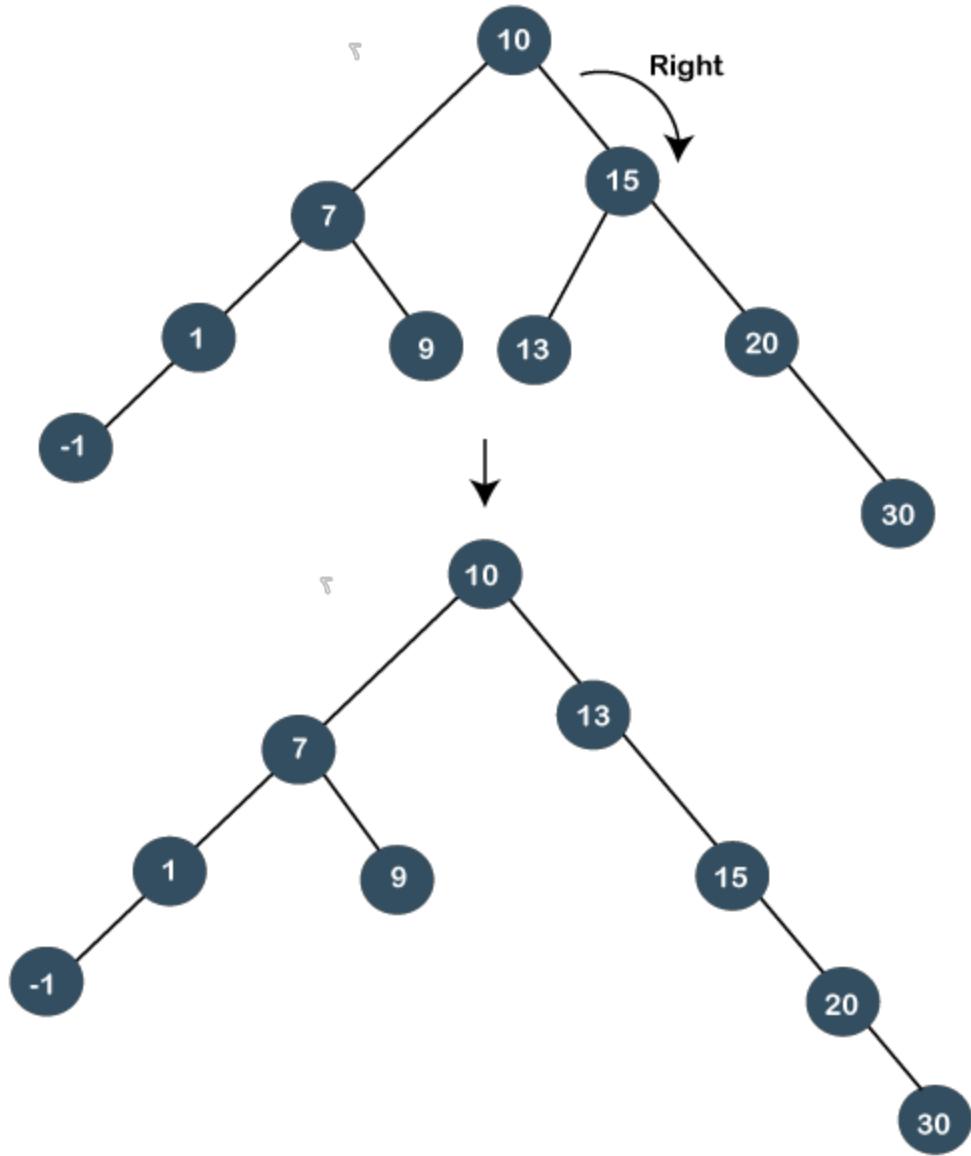
**Let's understand this case through an example.**

Suppose we want to search 13 element in the tree which is shown below:

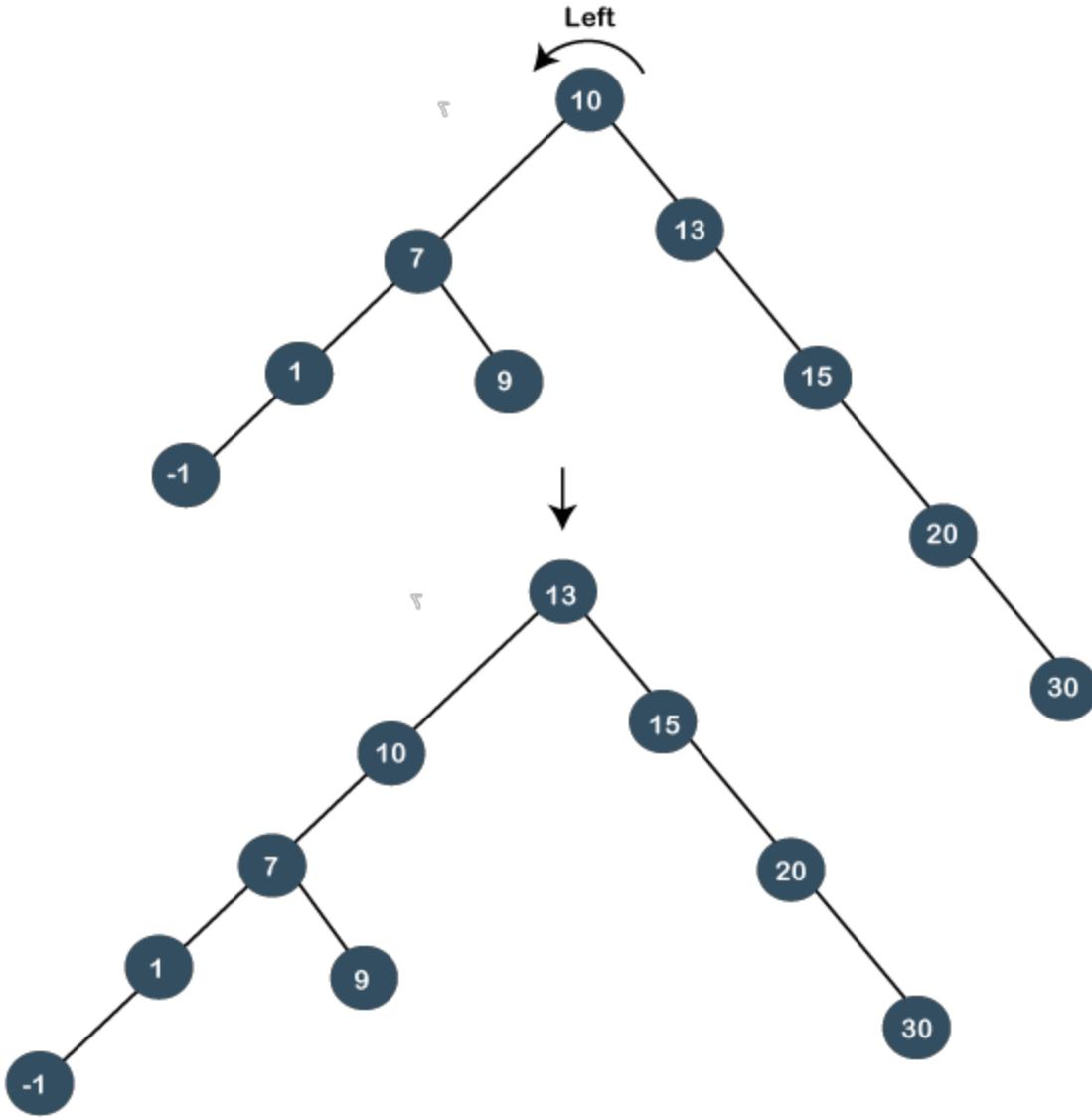


**Step 1:** First, we perform standard BST searching operation. As 13 is greater than 10 but less than 15, so node 13 will be the left child of node 15.

**Step 2:** Since node 13 is at the left of 15 and node 15 is at the right of node 10, so RL relationship exists. First, we perform the right rotation on node 15, and 15 will move downwards, and node 13 will come upwards, as shown below:



Still, node 13 is not the root node, and 13 is at the right of the root node, so we will perform left rotation known as a zig rotation. The node 10 will move downwards, and 13 becomes the root node as shown below:

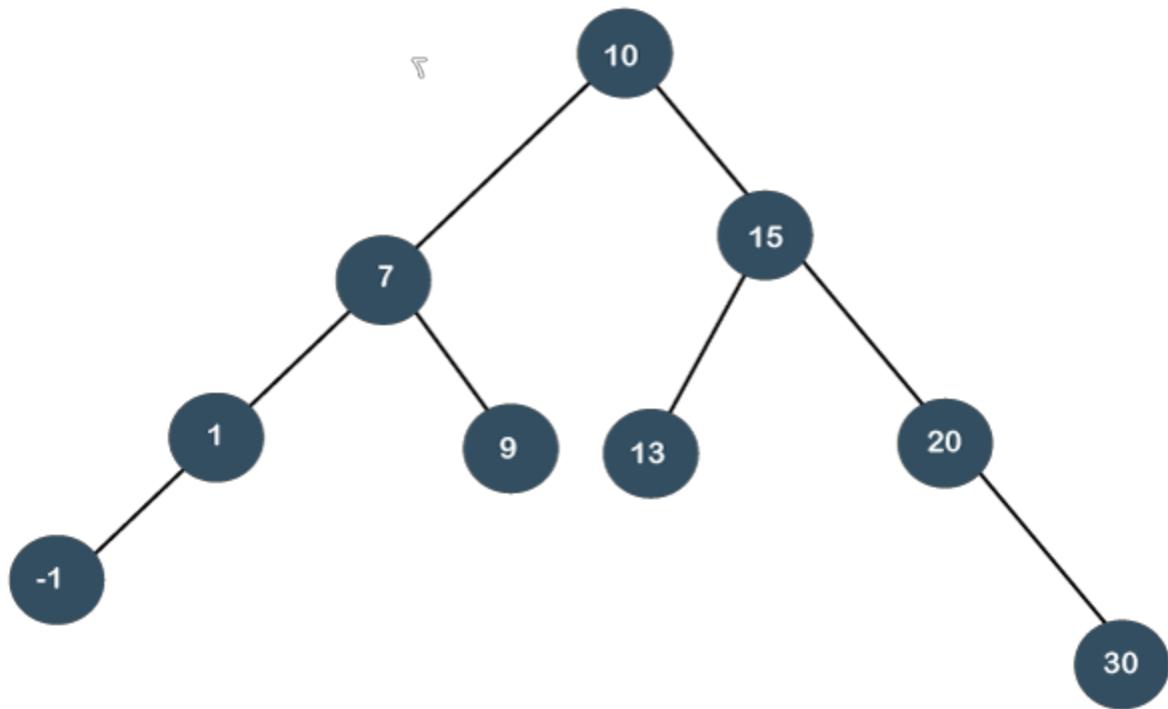


As we can observe in the above tree that node 13 has become the root node; therefore, the searching is completed. In this case, we have first performed the zig rotation and then zag rotation; so, it is known as a zig zag rotation.

- **Zag zig rotation**

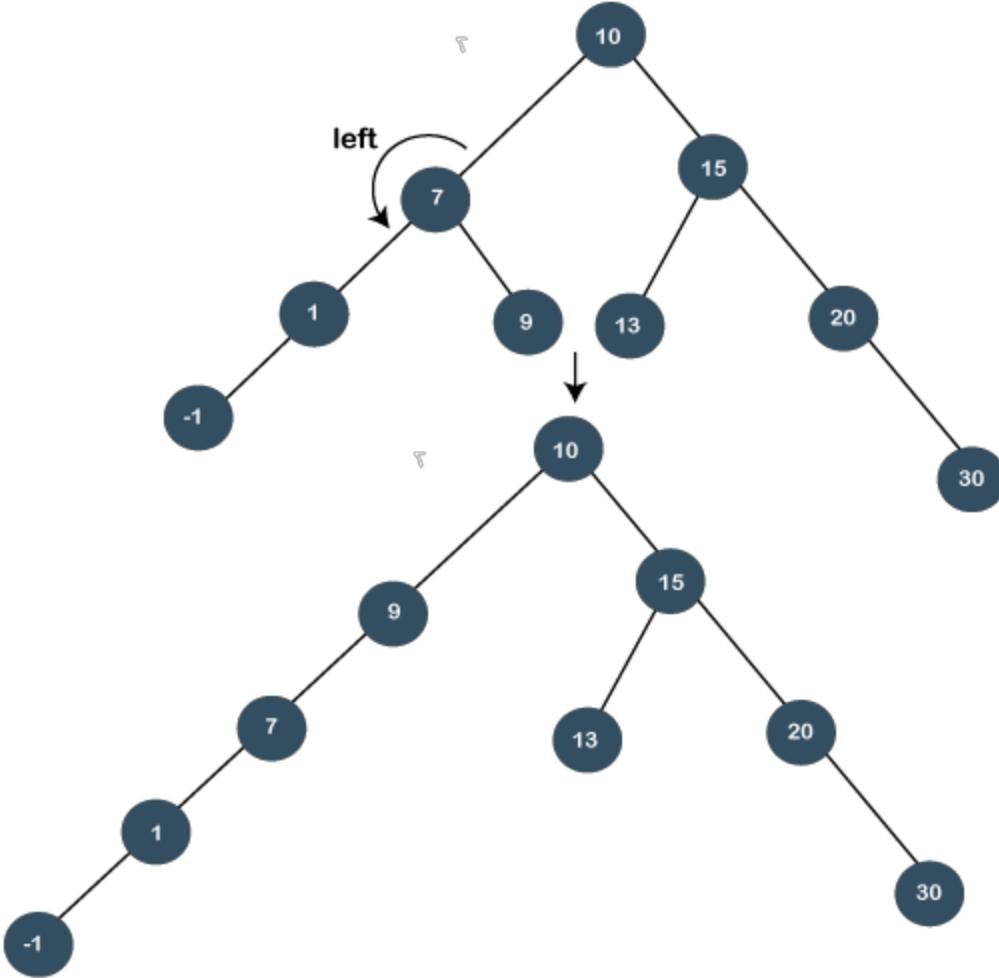
**Let's understand this case through an example.**

Suppose we want to search 9 element in the tree, which is shown below:

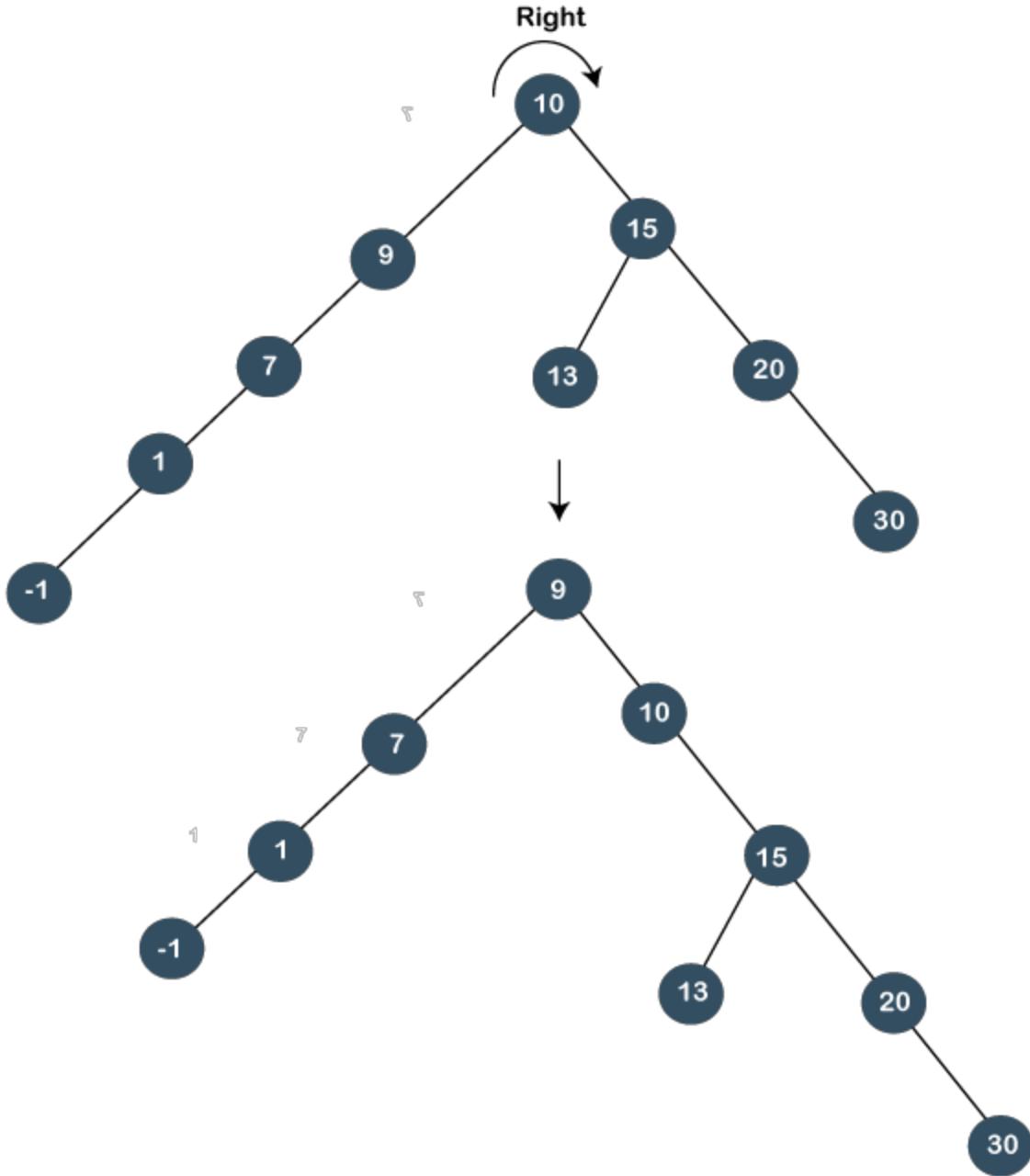


**Step 1:** First, we perform the standard BST searching operation. As 9 is less than 10 but greater than 7, so it will be the right child of node 7.

**Step 2:** Since node 9 is at the right of node 7, and node 7 is at the left of node 10, so LR relationship exists. First, we perform the left rotation on node 7. The node 7 will move downwards, and node 9 moves upwards as shown below:



Still the node 9 is not a root node, and 9 is at the left of the root node, so we will perform the right rotation known as zig rotation. After performing the right rotation, node 9 becomes the root node, as shown below:



As we can observe in the above tree that node 13 is a root node; therefore, the searching is completed. In this case, we have first performed the zig rotation (left rotation), and then zag rotation (right rotation) is performed, so it is known as a zig-zag rotation.

## Advantages of Splay tree

- In the splay tree, we do not need to store the extra information. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.
- It is the fastest type of Binary Search tree for various practical applications. It is used in **Windows NT and GCC compilers**.
- It provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees. It is used in the cache implementation as the recently accessed data is stored in the cache so that we do not need to go to the memory for accessing the data, and it takes less time.

## Drawback of Splay tree

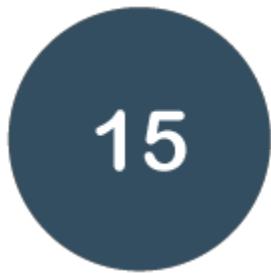
The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take O(n) time complexity.

### Insertion operation in Splay tree

In the **insertion** operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.

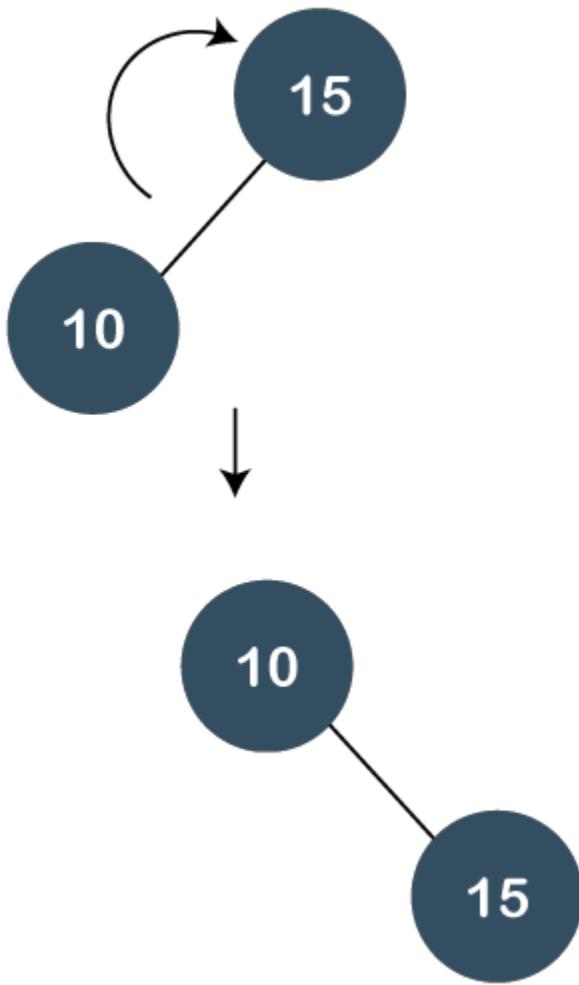
**15, 10, 17, 7**

**Step 1:** First, we insert node 15 in the tree. After insertion, we need to perform splaying. As 15 is a root node, so we do not need to perform splaying.



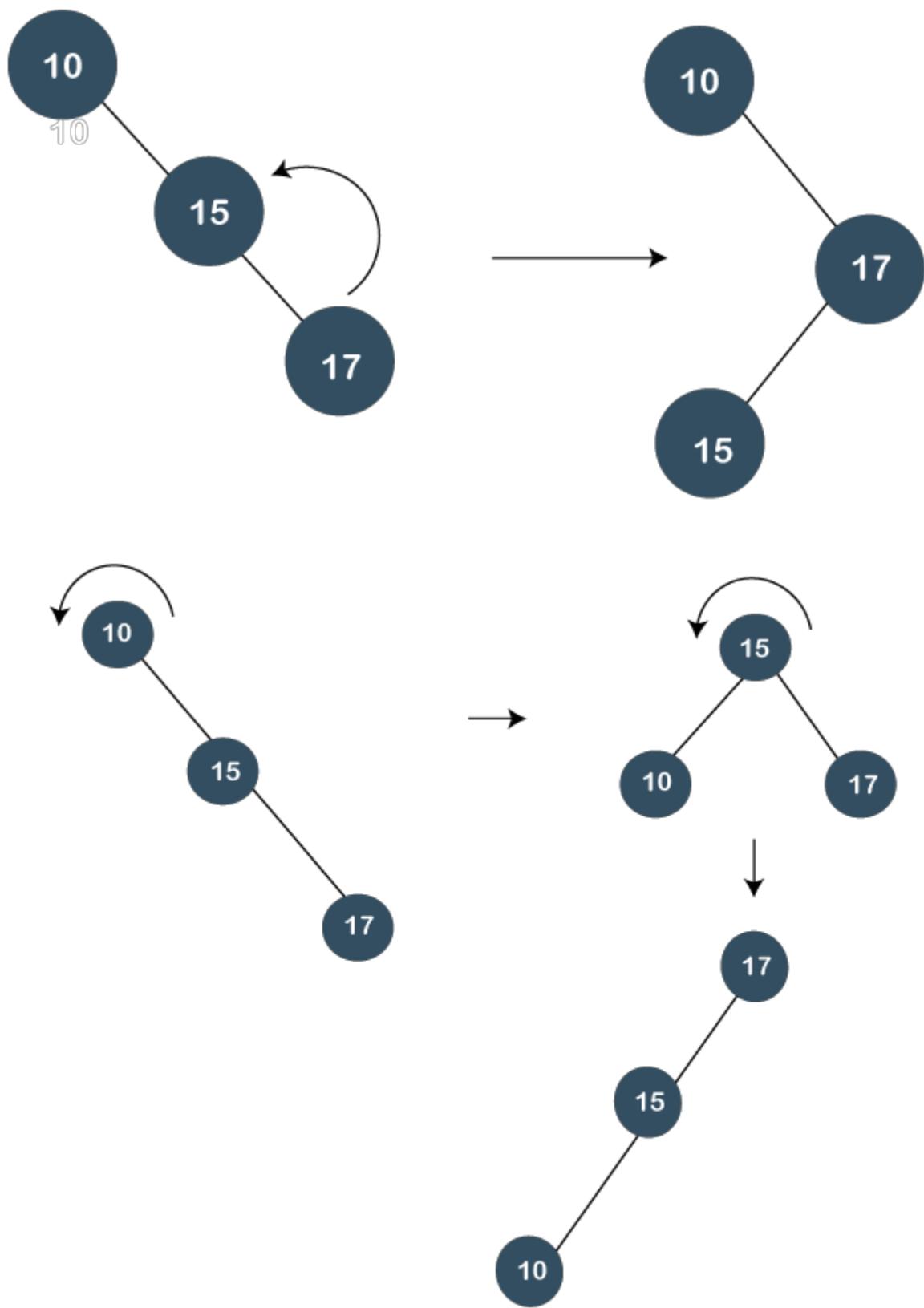
**Step 2:** The next element is 10. As 10 is less than 15, so node 10 will be the left child of node 15, as shown below:

Now, we perform **splaying**. To make 10 as a root node, we will perform the right rotation, as shown below:



**Step 3:** The next element is 17. As 17 is greater than 10 and 15 so it will become the right child of node 15.

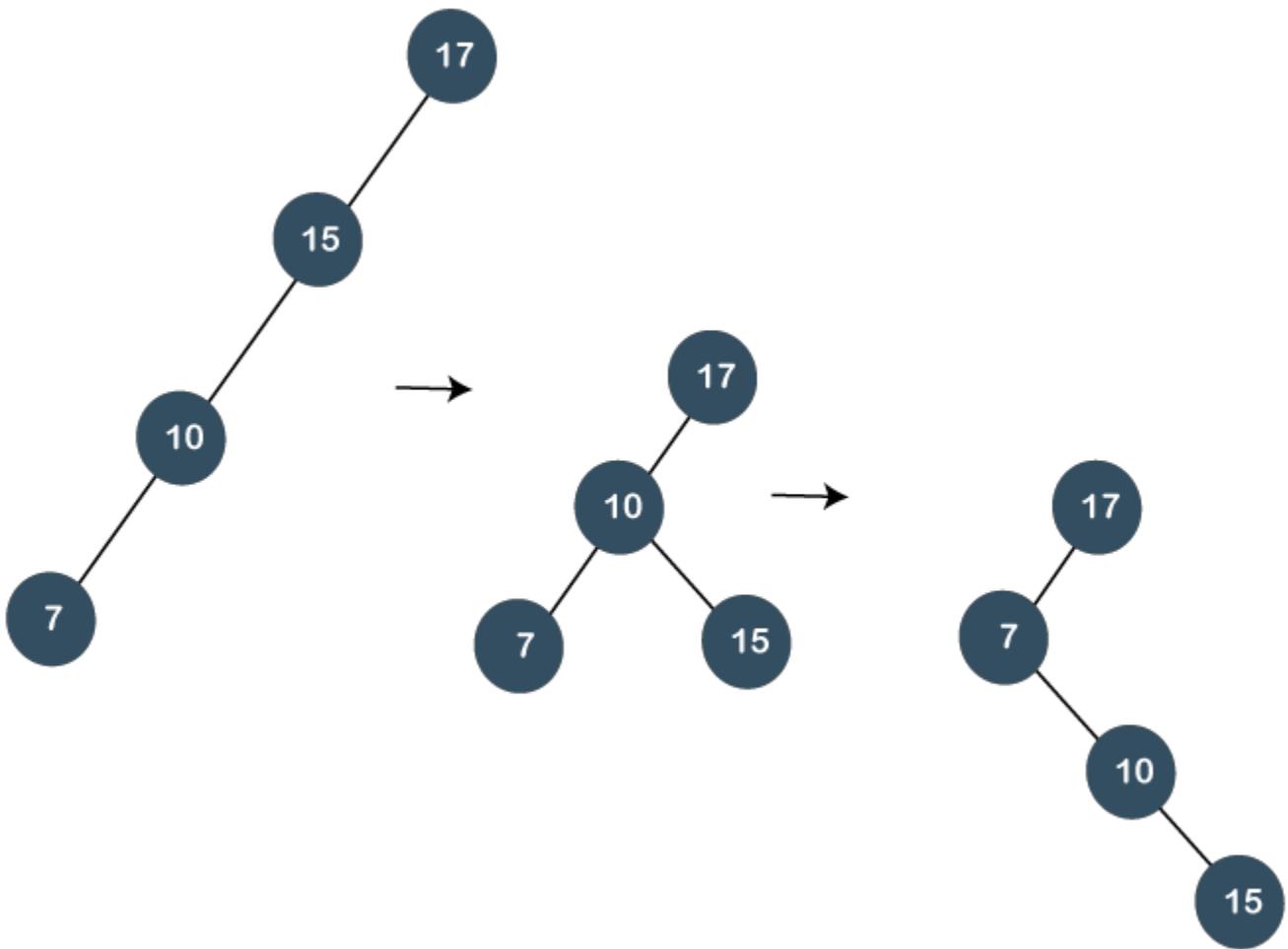
Now, we will perform splaying. As 17 is having a parent as well as a grandparent so we will perform zig-zig rotations.



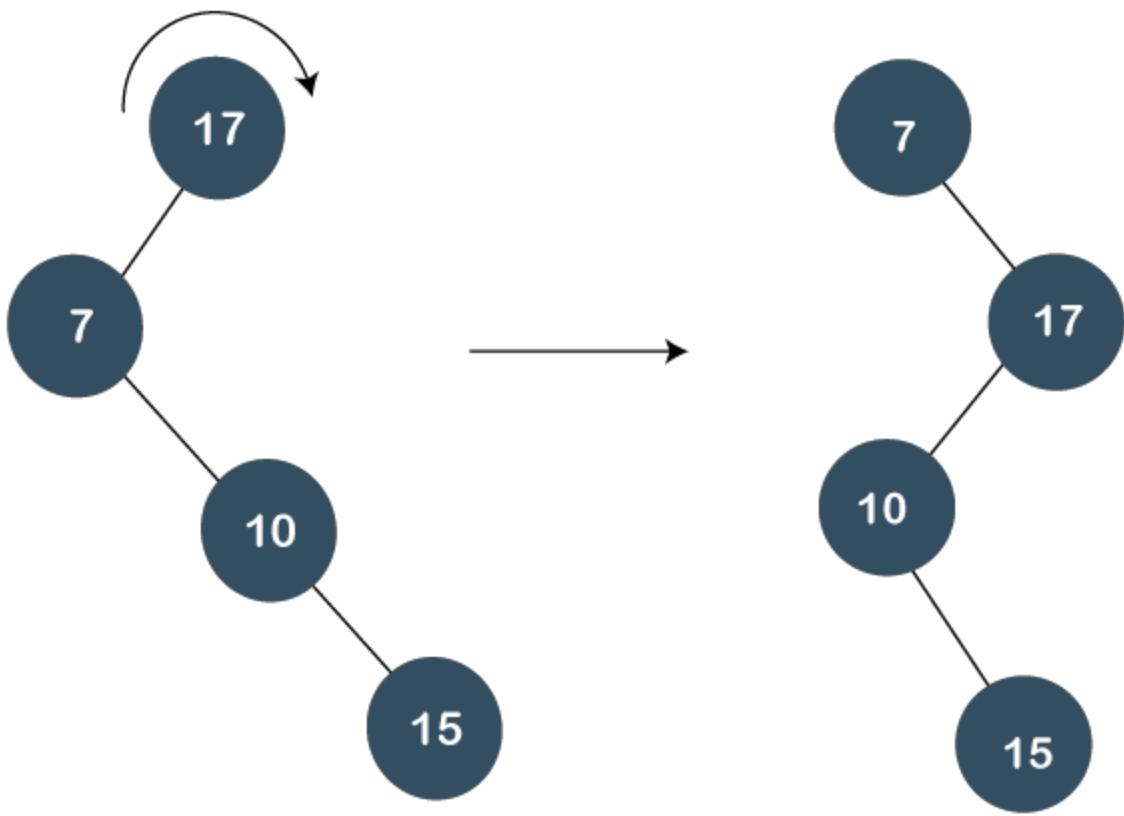
In the above figure, we can observe that 17 becomes the root node of the tree; therefore, the insertion is completed.

**Step 4:** The next element is 7. As 7 is less than 17, 15, and 10, so node 7 will be left child of 10.

Now, we have to splay the tree. As 7 is having a parent as well as a grandparent so we will perform two right rotations as shown below:



Still the node 7 is not a root node, it is a left child of the root node, i.e., 17. So, we need to perform one more right rotation to make node 7 as a root node as shown below:



### Algorithm for Insertion operation

1. Insert( $T, n$ )
2.  $\text{temp} = T_{\text{root}}$
3.  $y = \text{NULL}$
4. **while**( $\text{temp} \neq \text{NULL}$ )
5.  $y = \text{temp}$
6. **if**( $n \rightarrow \text{data} < \text{temp} \rightarrow \text{data}$ )
7.  $\text{temp} = \text{temp} \rightarrow \text{left}$
8. **else**
9.  $\text{temp} = \text{temp} \rightarrow \text{right}$
10.  $n.\text{parent} = y$
11. **if**( $y == \text{NULL}$ )
12.  $T_{\text{root}} = n$
13. **else if** ( $n \rightarrow \text{data} < y \rightarrow \text{data}$ )

14.  $y \rightarrow \text{left} = n$

15. **else**

16.  $y \rightarrow \text{right} = n$

17. Splay(T, n)

In the above algorithm, T is the tree and n is the node which we want to insert. We have created a temp variable that contains the address of the root node. We will run the while loop until the value of temp becomes NULL.

Once the insertion is completed, splaying would be performed

### Algorithm for Splaying operation

1. Splay(T, N)
2. **while**( $n \rightarrow \text{parent} \neq \text{Null}$ )
3. **if**( $n \rightarrow \text{parent} == T \rightarrow \text{root}$ )
4. **if**( $n == n \rightarrow \text{parent} \rightarrow \text{left}$ )
5. right\_rotation(T,  $n \rightarrow \text{parent}$ )
6. **else**
7. left\_rotation(T,  $n \rightarrow \text{parent}$ )
8. **else**
9.  $p = n \rightarrow \text{parent}$
10.  $g = p \rightarrow \text{parent}$
11. **if**( $n = n \rightarrow \text{parent} \rightarrow \text{left} \ \&\& \ p = p \rightarrow \text{parent} \rightarrow \text{left}$ )
12. right.rotation(T, g), right.rotation(T, p)
13. **else if**( $n = n \rightarrow \text{parent} \rightarrow \text{right} \ \&\& \ p = p \rightarrow \text{parent} \rightarrow \text{right}$ )
14. left.rotation(T, g), left.rotation(T, p)
15. **else if**( $n = n \rightarrow \text{parent} \rightarrow \text{left} \ \&\& \ p = p \rightarrow \text{parent} \rightarrow \text{right}$ )
16. right.rotation(T, p), left.rotation(T, g)
17. **else**
18. left.rotation(T, p), right.rotation(T, g)
- 19.
20. Implementation of right.rotation(T, x)
21. right.rotation(T, x)
22.  $y = x \rightarrow \text{left}$
23.  $x \rightarrow \text{left} = y \rightarrow \text{right}$

24.  $y->right=x$

25. **return** y

In the above implementation, x is the node on which the rotation is performed, whereas y is the left child of the node x.

### **Implementation of left.rotation(T, x)**

1. left.rotation(T, x)
2.  $y=x->right$
3.  $x->right = y->left$
4.  $y->left = x$
5. **return** y

In the above implementation, x is the node on which the rotation is performed and y is the right child of the node x.

## **Deletion in Splay tree**

As we know that splay trees are the variants of the Binary search tree, so deletion operation in the splay tree would be similar to the BST, but the only difference is that the delete operation is followed in splay trees by the splaying operation.

### **Types of Deletions:**

There are two types of deletions in the splay trees:

1. Bottom-up splaying
2. Top-down splaying

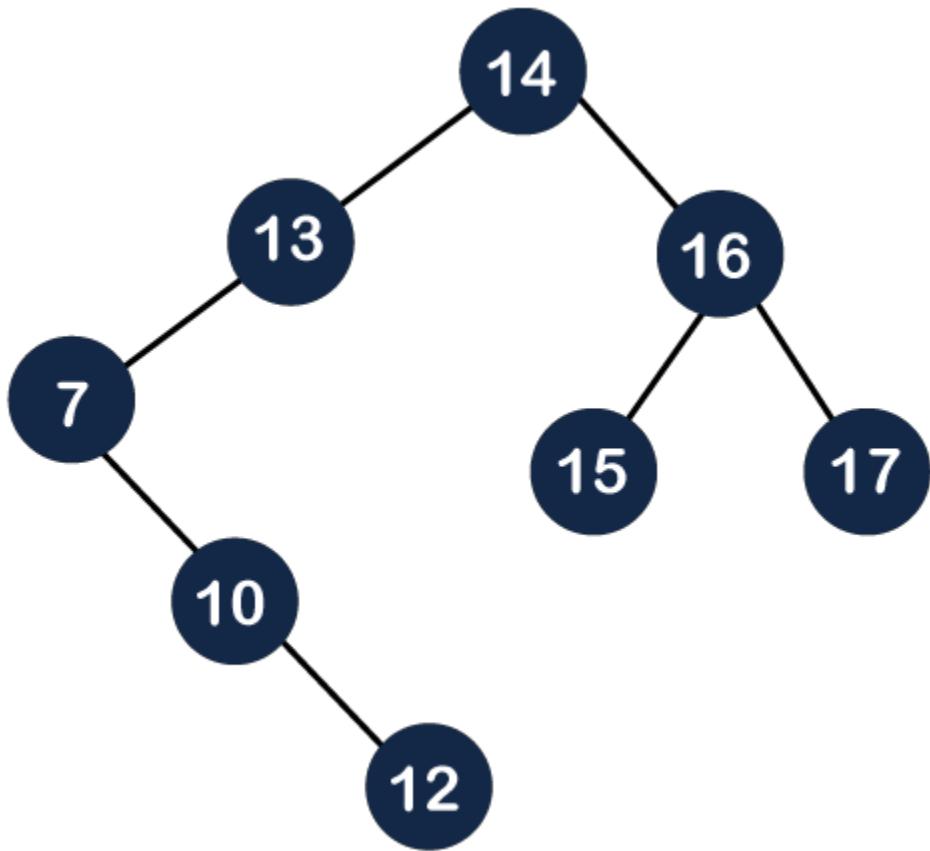
### **Bottom-up splaying**

In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the deleted node.

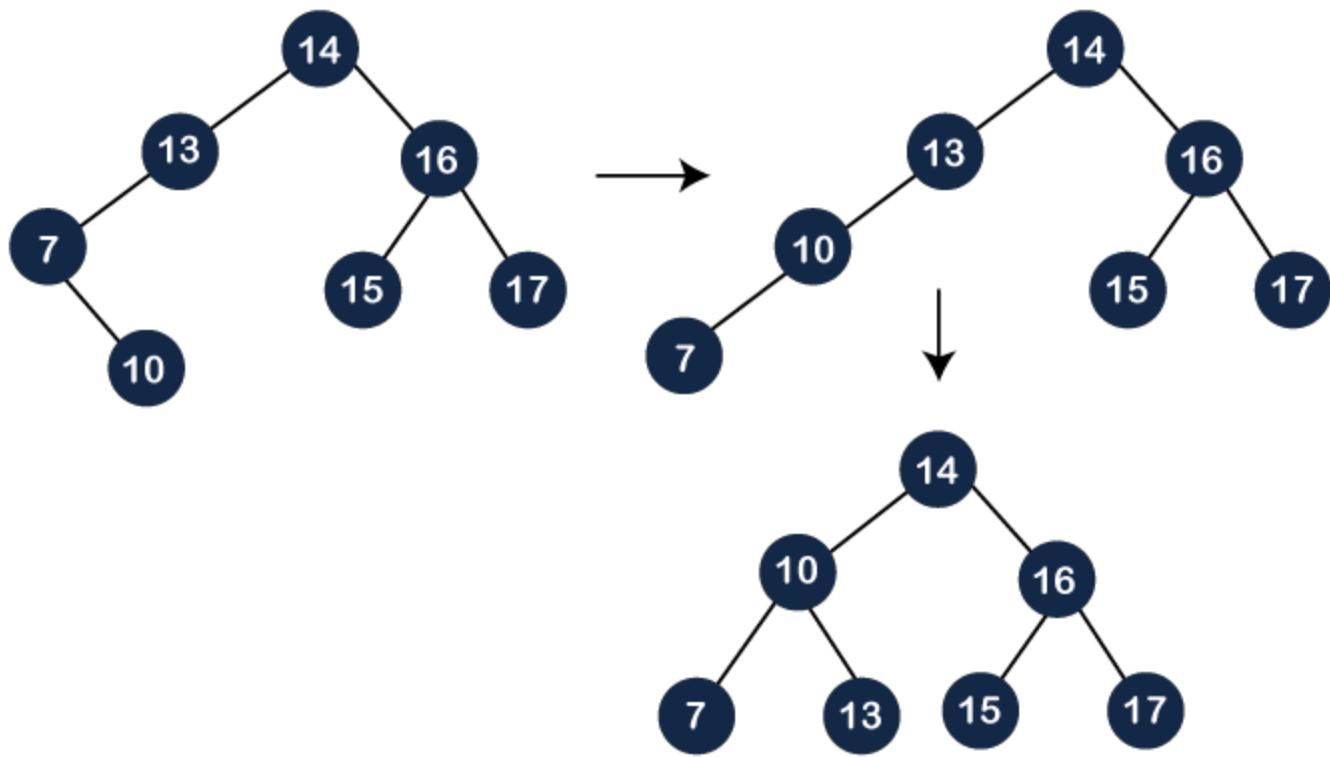
### **Let's understand the deletion in the Splay tree.**

Suppose we want to delete 12, 14 from the tree shown below:

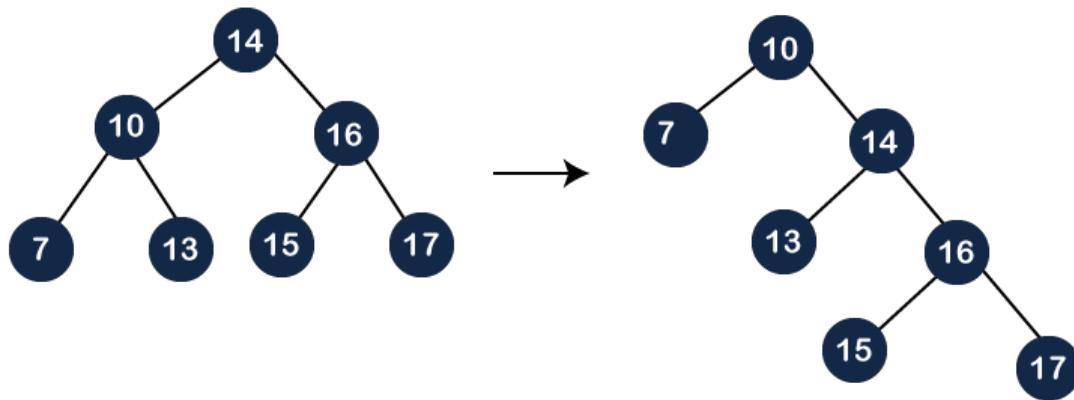
- First, we simply perform the standard BST deletion operation to delete 12 element. As 12 is a leaf node, so we simply delete the node from the tree.



The deletion is still not completed. We need to splay the parent of the deleted node, i.e., 10. We have to perform **Splay(10)** on the tree. As we can observe in the above tree that 10 is at the right of node 7, and node 7 is at the left of node 13. So, first, we perform the left rotation on node 7 and then we perform the right rotation on node 13, as shown below:



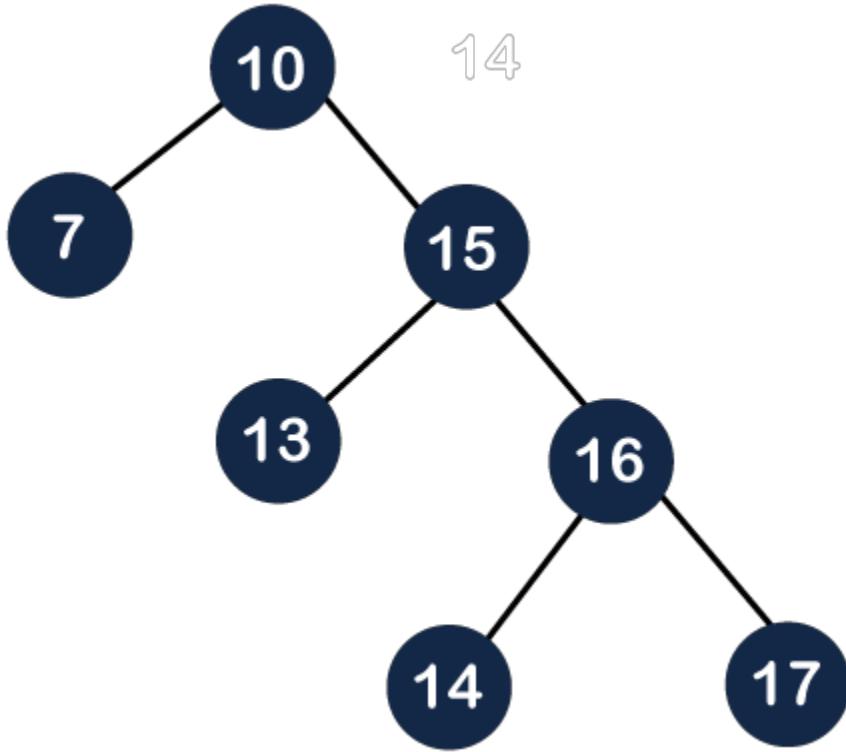
Still, node 10 is not a root node; node 10 is the left child of the root node. So, we need to perform the right rotation on the root node, i.e., 14 to make node 10 a root node as shown below:



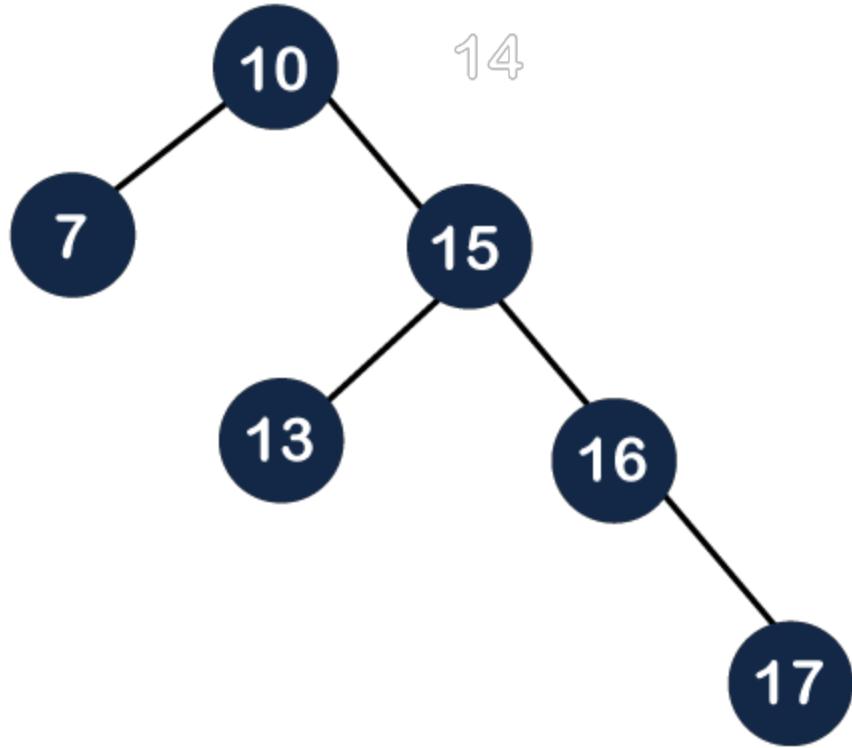
- Now, we have to delete the 14 element from the tree, which is shown below:

As we know that we cannot simply delete the internal node. We will replace the value of the node either using ***inorder predecessor*** or ***inorder successor***. Suppose we use

inorder successor in which we replace the value with the lowest value that exist in the right subtree. The lowest value in the right subtree of node 14 is 15, so we replace the value 14 with 15. Since node 14 becomes the leaf node, so we can simply delete it as shown below:



Still, the deletion is not completed. We need to perform one more operation, i.e., splaying in which we need to make the parent of the deleted node as the root node. Before deletion, the parent of node 14 was the root node, i.e., 10, so we do need to perform any splaying in this case.

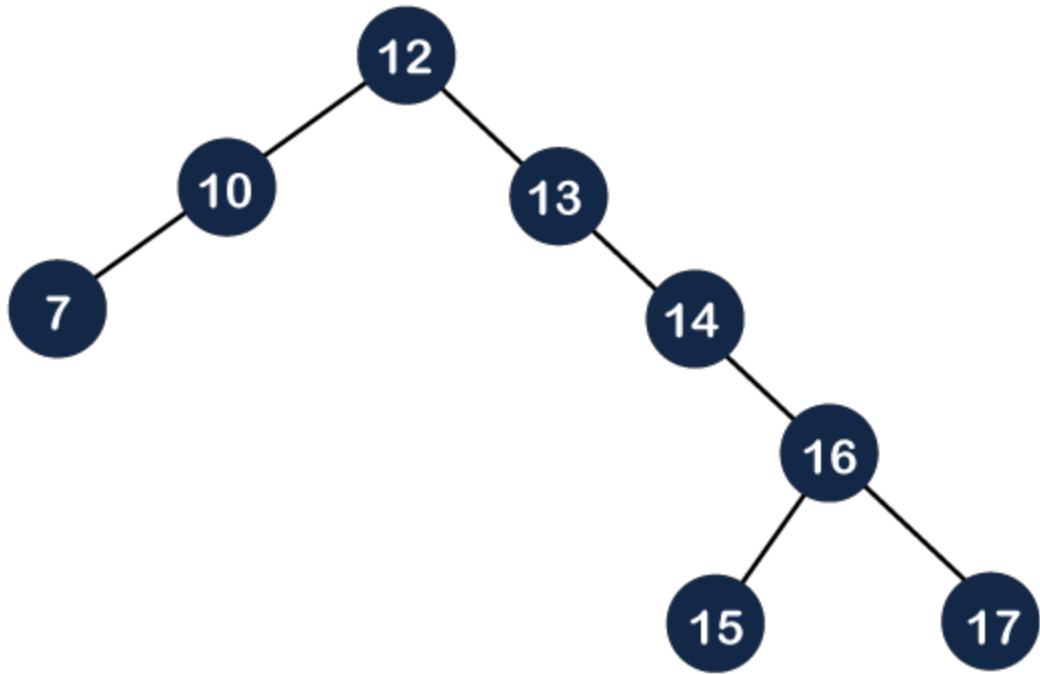


### Top-down splaying

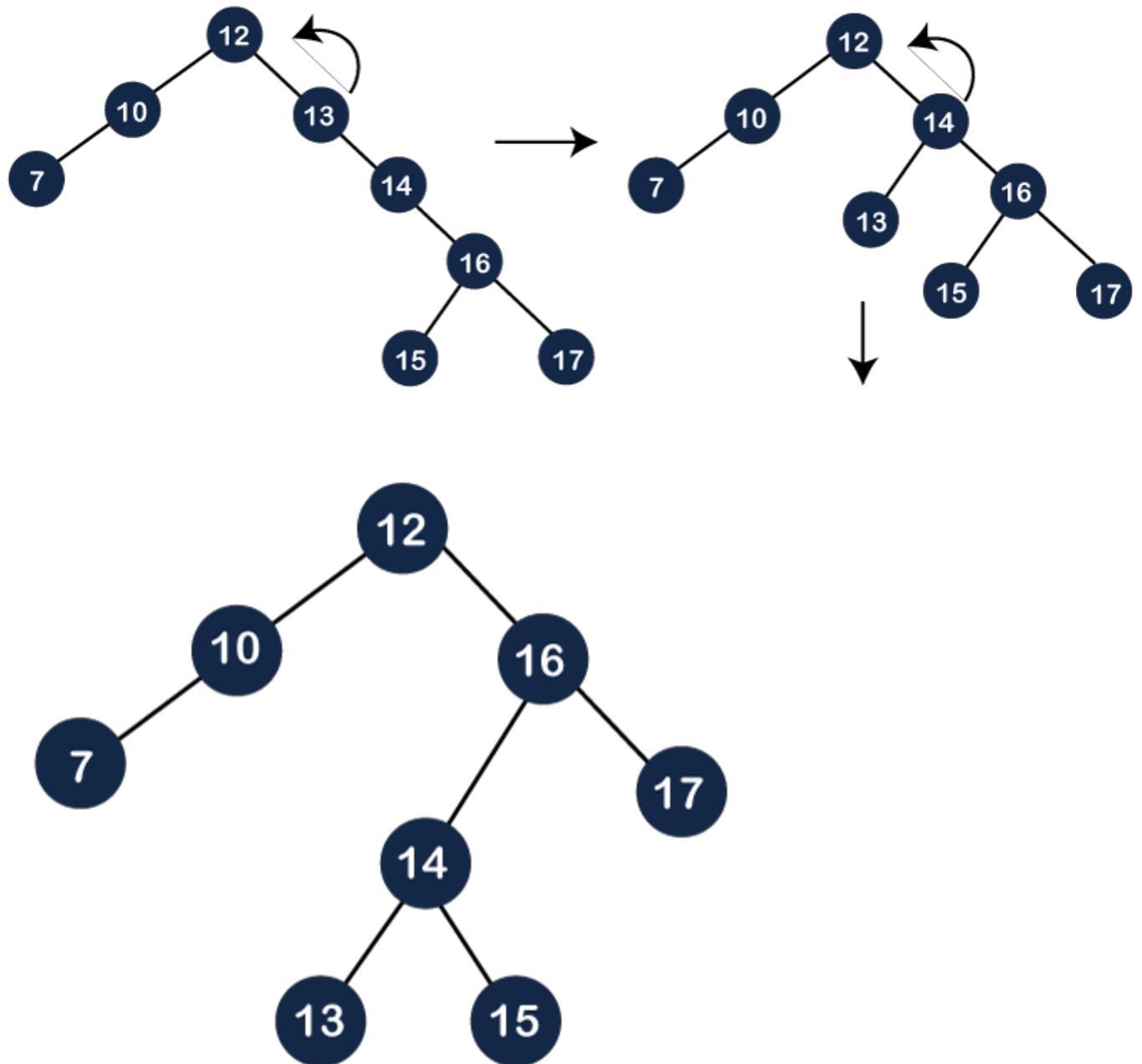
In top-down splaying, we first perform the splaying on which the deletion is to be performed and then delete the node from the tree. Once the element is deleted, we will perform the join operation.

**Let's understand the top-down splaying through an example.**

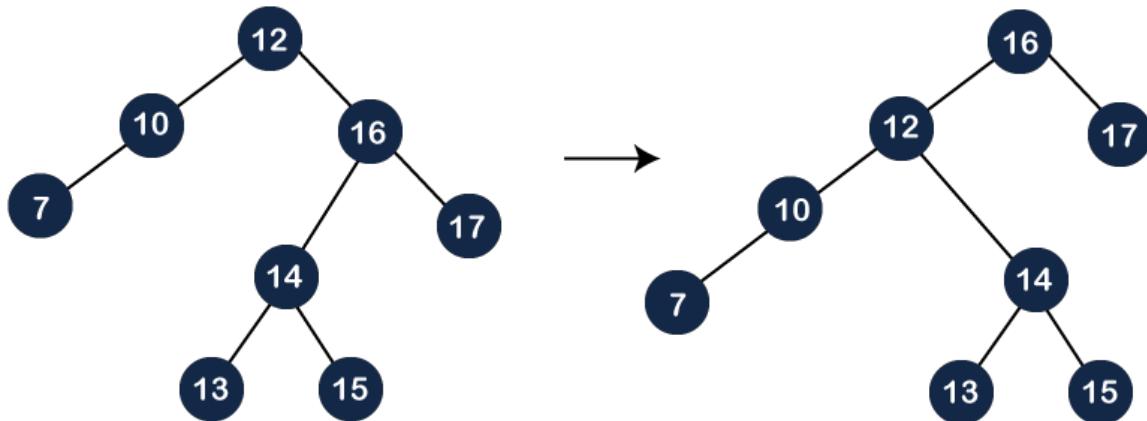
Suppose we want to delete 16 from the tree which is shown below:



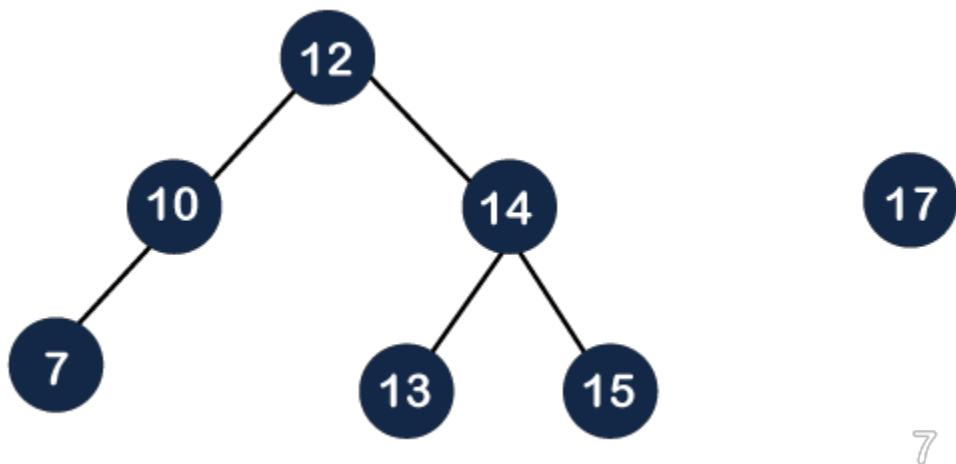
**Step 1:** In top-down splaying, first we perform splaying on the node 16. The node 16 has both parent as well as grandparent. The node 16 is at the right of its parent and the parent node is also at the right of its parent, so this is a zig-zag situation. In this case, first, we will perform the left rotation on node 13 and then 14 as shown below:



The node 16 is still not a root node, and it is a right child of the root node, so we need to perform left rotation on the node 12 to make node 16 as a root node.

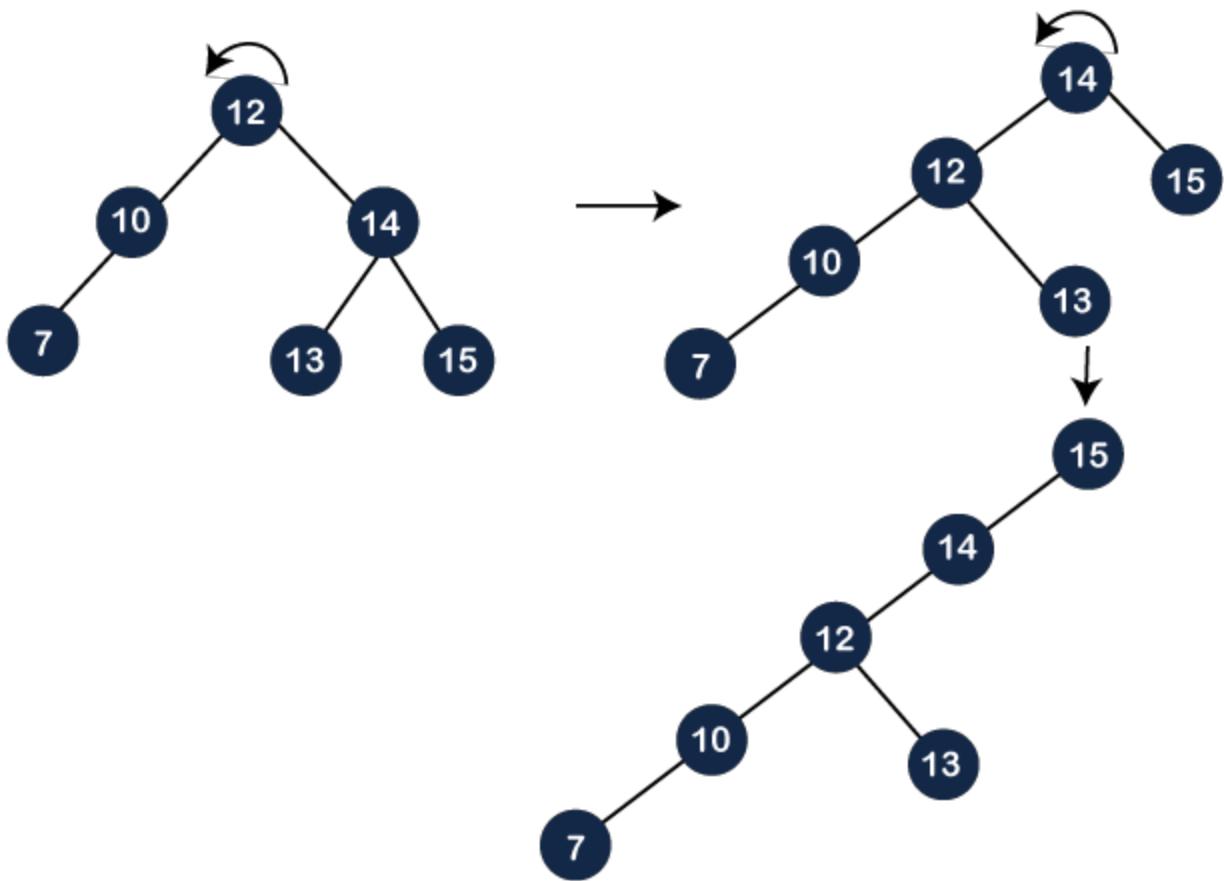


Once the node 16 becomes a root node, we will delete the node 16 and we will get two different trees, i.e., left subtree and right subtree as shown below:

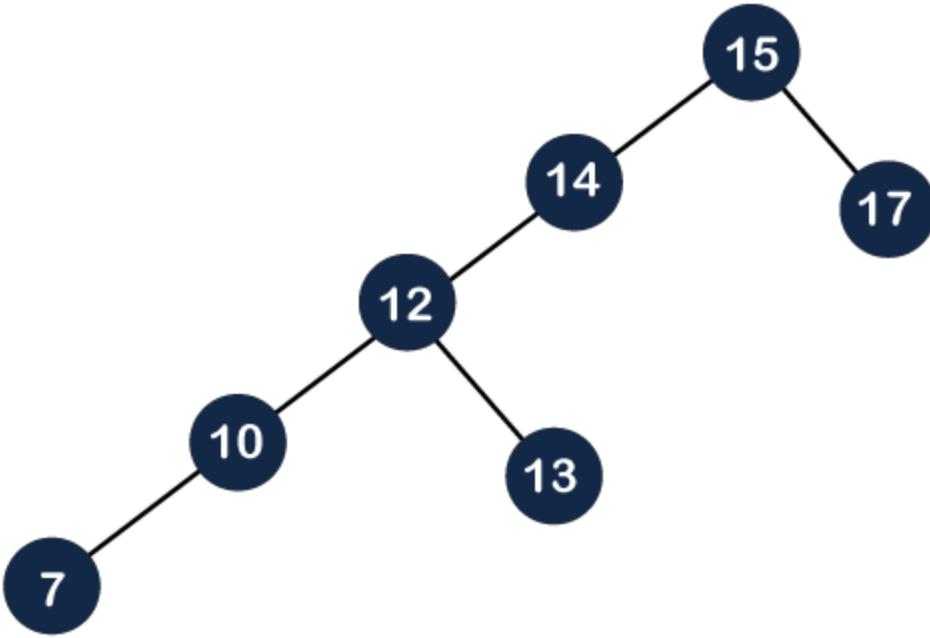


As we know that the values of the left subtree are always lesser than the values of the right subtree. The root of the left subtree is 12 and the root of the right subtree is 17. The first step is to find the maximum element in the left subtree. In the left subtree, the maximum element is 15, and then we need to perform splaying operation on 15.

As we can observe in the above tree that the element 15 is having a parent as well as a grandparent. A node is right of its parent, and the parent node is also right of its parent, so we need to perform two left rotations to make node 15 a root node as shown below:



After performing two rotations on the tree, node 15 becomes the root node. As we can see, the right child of the 15 is NULL, so we attach node 17 at the right part of the 15 as shown below, and this operation is known as a **join** operation.



*Note: If the element is not present in the splay tree, which is to be deleted, then splaying would be performed. The splaying would be performed on the last accessed element before reaching the NULL.*

### Algorithm of Delete operation

1. If(root==NULL)
2. **return** NULL
3. Splay(root, data)
4. If data!= root->data
5. Element is not present
6. If root->left==NULL
7. root=root->right
8. **else**
9. temp=root
10. Splay(root->left, data)
11. root1->right=root->right
12. free(temp)
13. **return** root

In the above algorithm, we first check whether the root is Null or not; if the root is NULL means that the tree is empty. If the tree is not empty, we will perform the splaying

operation on the element which is to be deleted. Once the splaying operation is completed, we will compare the root data with the element which is to be deleted; if both are not equal means that the element is not present in the tree. If they are equal, then the following cases can occur:

**Case 1:** The left of the root is NULL, the right of the root becomes the root node.

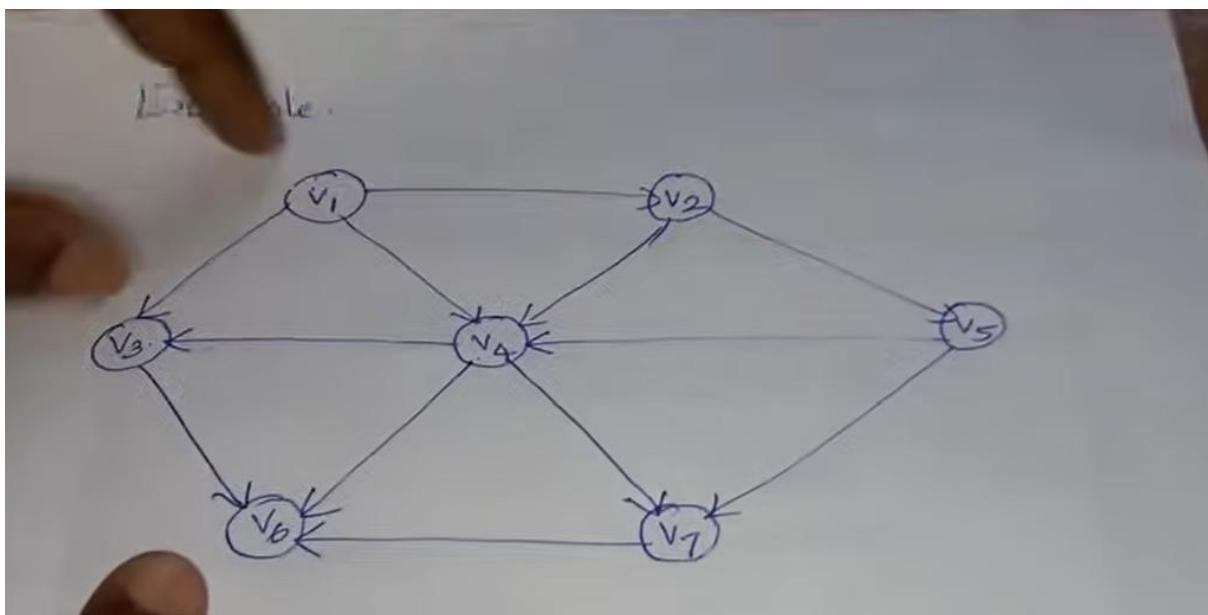
**Case 2:** If both left and right exist, then we splay the maximum element in the left subtree. When the splaying is completed, the maximum element becomes the root of the left subtree. The right subtree would become the right child of the root of the left subtree.

Nov/Dec 2018

### Topological Sort.

A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from  $v_i$  to  $v_j$  then  $v_j$  appears after  $v_i$  in the ordering.  $v_i \rightarrow v_j \quad v_i, v_j$

Topological ordering is not possible if the graph has a cycle. Because if two vertices  $v$  and  $w$  on the cycle,



Adjacency matrix:

	v <sub>1</sub>	v <sub>2</sub>	v <sub>3</sub>	v <sub>4</sub>	v <sub>5</sub>	v <sub>6</sub>	v <sub>7</sub>
v <sub>1</sub>	0	1	1	1	0	0	0
v <sub>2</sub>	0	0	0	1	1	0	0
v <sub>3</sub>	0	0	0	0	0	1	0
v <sub>4</sub>	0	0	1	0	0	1	1
v <sub>5</sub>	0	0	0	1	0	0	1
v <sub>6</sub>	0	0	0	0	0	0	0
v <sub>7</sub>	0	0	0	0	0	1	0

	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	1	1	1	1	0	0	0
4	2	1	0	0	0	0	0
5	1	0	0	0	0	0	0
6	3	3	3	2			
7	2	2	1	0			

	∞	2	2	1	∞	0	0
Enqueue	$v_1$	$v_2$	$v_5$	$v_4$	$v_3, v_7$		$v_6$
Dequeue	$v_1$	$v_2$	$v_5$	$v_4$	$v_3$	$v_7$	$v_6$

The Topological ordering are

$v_1, v_2, v_5, v_4, v_3, v_7, v_6$

WT

Step 1 : Find the indegree for every vertex.

Step 2 : Place the vertices whose indegree is '0' on the empty queue.

Step 3 : Dequeue the vertex  $v$  and decrement the indegree's of all its adjacent vertices.

Step 4 : Enqueue the vertex on the queue, if its indegree falls to zero.

Step 5 : Repeat from step 3 until the queue becomes empty.

### Routine to perform Topological Sort

```
/* Assume that the graph is read into an adjacency matrix and that the indegrees are computed for
every vertices and placed in an array (i.e. Indegree [ ] ) */

void Topsort (Graph G)

{

    Queue Q ;

    int counter = 0;

    Vertex V, W ;

    Q = CreateQueue (NumVertex);

    Makeempty (Q);

    for each vertex V

        if (indegree [V] == 0)

            Enqueue (V, Q);

    while (! IsEmpty (Q))

    {

        V = Dequeue (Q);

        TopNum [V] = ++ counter;

        for each W adjacent to V

            if (--Indegree [W] == 0)

                Enqueue (W, Q);

    }

    if (counter != NumVertex)

        Error (" Graph has a cycle");

    DisposeQueue (Q); /* Free the Memory */

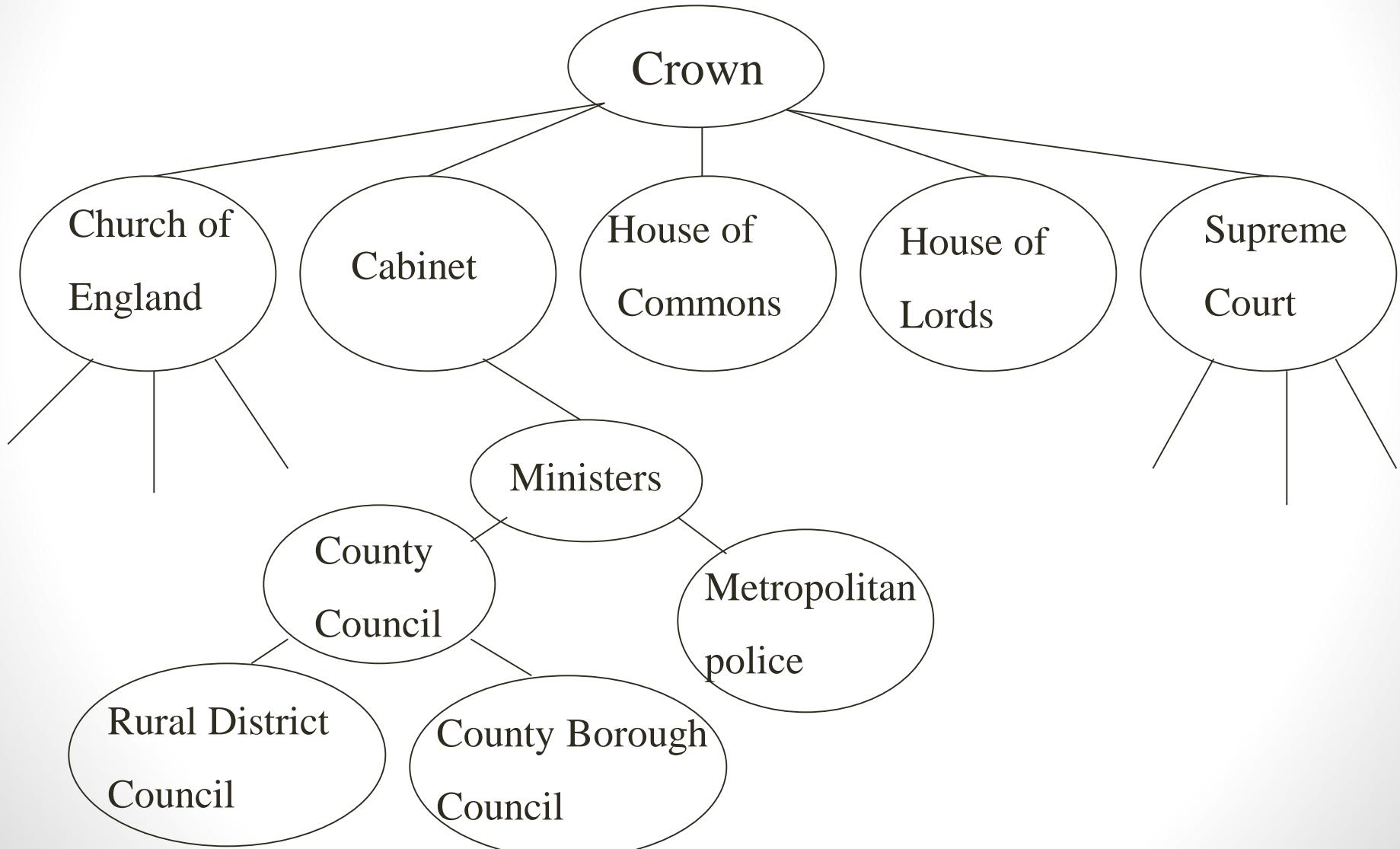
}
```



# **UNIT -III**

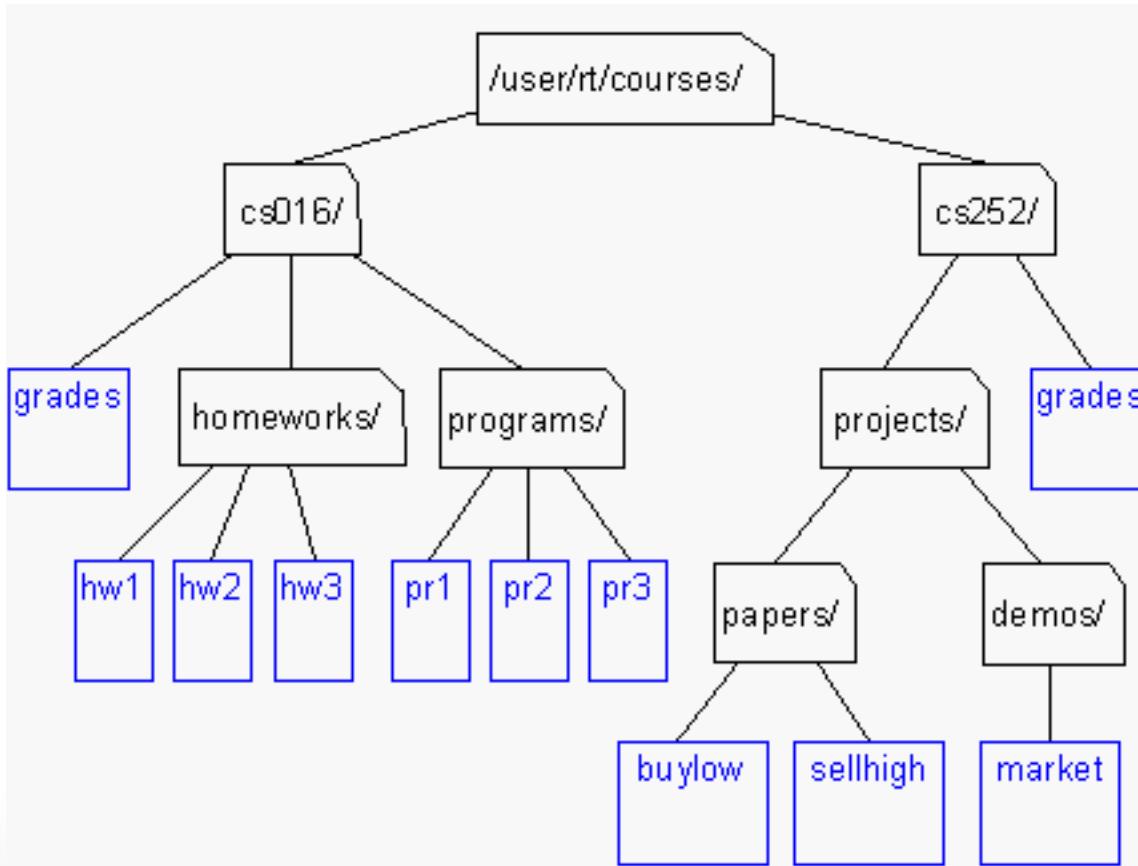
- **UNIT III NON-LINEAR DATA STRUCTURE**
- Trees - Binary Tree - Threaded Binary Tree  
- Binary Search Tree – B-Tree - B+ Tree -  
AVL Tree - Splay Tree.
- Graphs: Basic Terminologies - Directed –  
Undirected - Various Representations -  
Operations - Graph search and traversal  
algorithms - complexity analysis -  
Applications of Non-Linear Data  
Structures

# The British Constitution



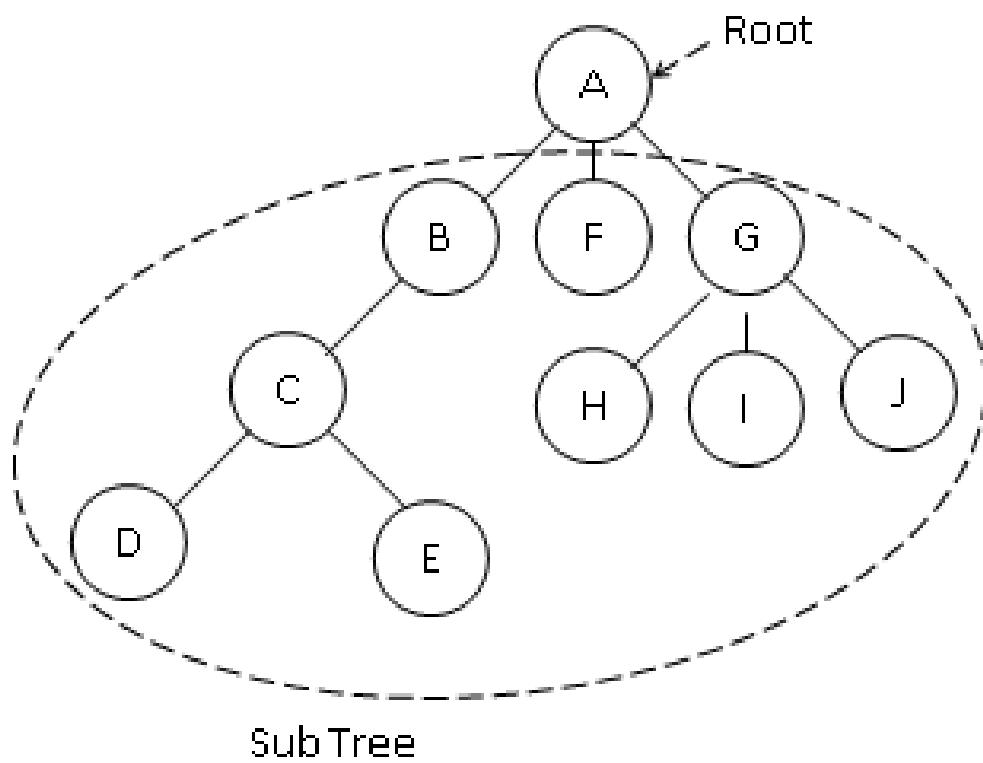
# More Trees Examples

- Unix / Windows file structure



# *Definition of Tree*

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root.
- The remaining nodes are partitioned into  $n \geq 0$  disjoint sets  $T_1, \dots, T_n$ , where each of these sets is a tree.
- We call  $T_1, \dots, T_n$  the subtrees of the root.

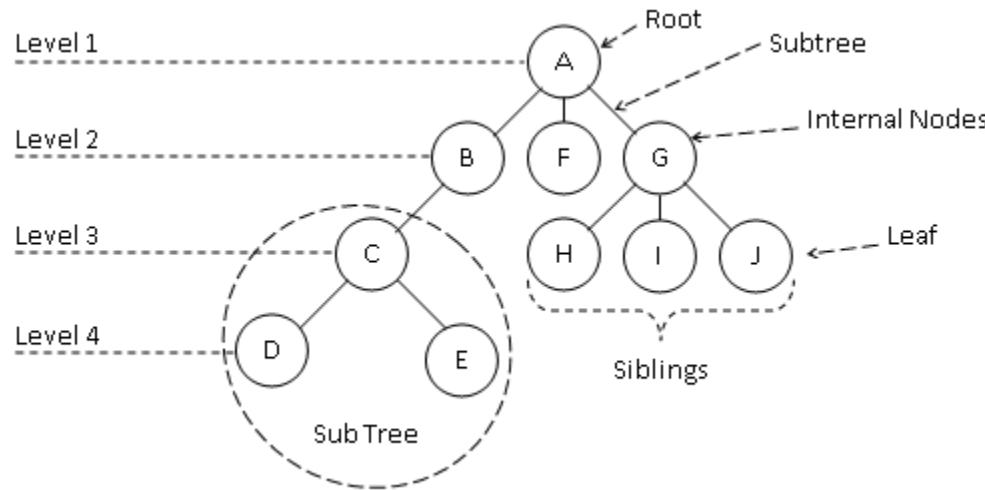


# *Terminology*

- ◆ **Node** stands for item of information.
- ◆ The nodes of a tree have a parent-child relationship. The root does not have a parent; but each one of the other nodes has a parent node associated to it.
- ◆ The number of sub-trees /child of a node is called its **degree**.
- ◆ A node with degree zero is called a leaf node or terminal nodes and other nodes are referred as non-terminals.
- ◆ Eg. The degree of A is 3, F and J is zero.
- ◆ The leaf node is F and J having degree zero.

- A line from a parent to a child node is called a sub tree. If a tree has  $n$  nodes, one of which is the root there would be  $n-1$  sub trees.
- The degree of a tree is the maximum degree of the nodes in the tree.
- Nodes with the same parent are called siblings. Here D & E are all **siblings**.
- The **level of a node** is defined by initially letting the root be at level one. If a node is at level 1, then its children are at level 1+1.

- The height or depth of a tree is defined to be the maximum level of any node in the tree.
- The **ancestors** of a node are all the nodes along the path from the root to the node
- A set of trees is called **forest**; if we remove the root of a tree we get a forest. In the above fig, if we remove A, we get a forest with three trees.



Root : A

Children  
of node A : B, F, G are child

Siblings : H, I, J

Leaf : D, E, H, I, J

Height of tree : 4

Degree of node A : 3

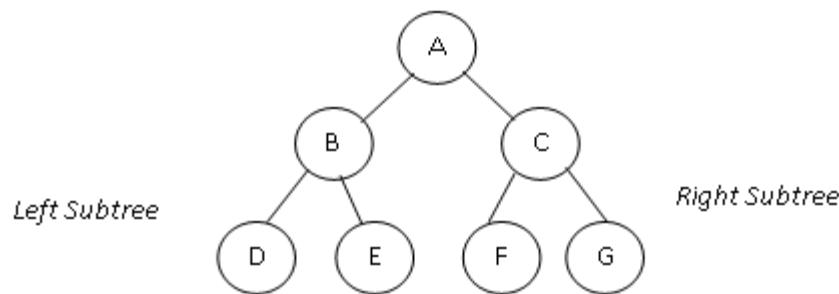
## Properties of a Tree:

- Any node can be the root of the tree and each node in a tree has the property that there is exactly one path connecting that node with every other node in the tree.
  - The tree in which the root is identified is called a rooted tree; a tree in which the root is not identified is called a free tree.
- Each node, except the root, has a unique parent.

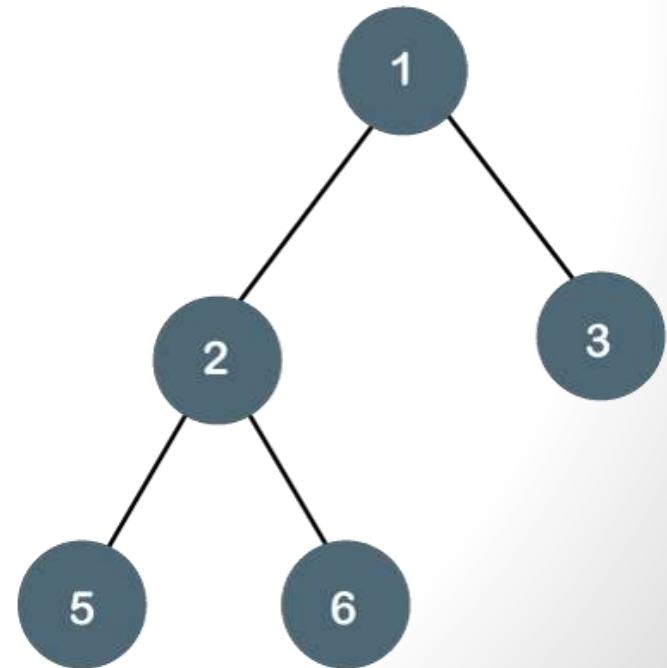
# *Binary Trees*

- ➊ A special class of trees: max degree for each node is 2
- ➋ Recursive definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- ➌ Any tree can be transformed into binary tree.
  - ▣ by left child-right sibling representation

A binary tree is a tree, which is, either empty or consists of a root node and atmost two disjoint binary trees called the left sub-tree and right sub-tree.



- Binary Tree is a tree in which no node can have more than two children.
- A binary tree is a tree which is either empty, or one in which every node:
  - has no children; or
  - has just a left child; or
  - has just a right child; or
  - has both a left and a right child.



## Properties of Binary Tree

- At each level of  $i$ , the maximum number of nodes is  $2^i$ .
- The height of the tree is defined as the longest path from the root node to the leaf node.
- The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to  $(1+2+4+8) = 15$ .
- In general, the maximum number of nodes possible at height  $h$  is  $(2^0 + 2^1 + 2^2 + \dots + 2^h) = 2^{h+1} - 1$ .
- The minimum number of nodes possible at height  $h$  is equal to  $h+1$ .
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.
- If there are ' $n$ ' number of nodes in the binary tree

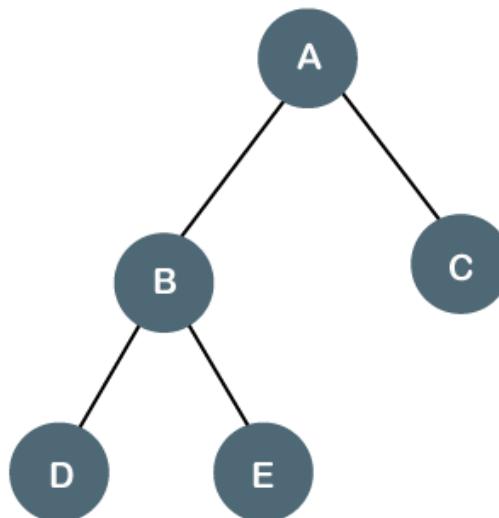
# Types of Binary Tree

Types of Binary tree:

- Full/ proper/ strict Binary tree
- Complete Binary tree
- Perfect Binary tree
- Degenerate Binary tree
- Balanced Binary tree

## Full/ proper/ strict Binary tree

- The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children.
- The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.



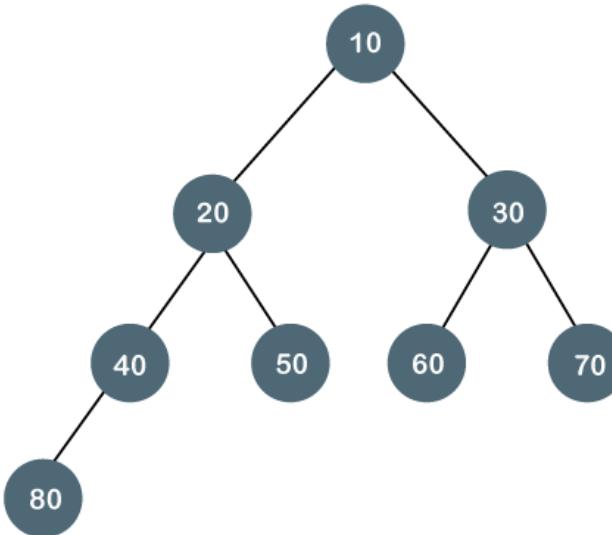
In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

## Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e.,  $2^{h+1} - 1$ .
- The minimum number of nodes in the full binary tree is  $2^h - 1$ .
- The minimum height of the full binary tree is  $\log_2(n+1) - 1$ .
- The maximum height of the full binary tree can be computed as:
  - $n = 2^h - 1$
  - $n + 1 = 2^{h+1} - 1$
  - $h = \log_2(n+1)$

## Complete Binary Tree

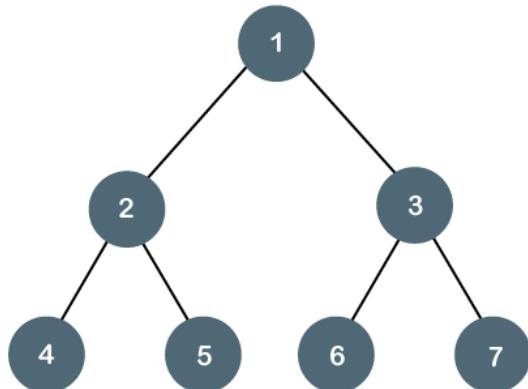
- A binary tree is called complete binary tree if each node has exactly two children and all leaf nodes need not to be at same level or depth.
- Total number of nodes in complete binary tree are  $2^{h+1} - 1$  where h is a height of the tree.
- The number of leaf nodes n in a complete binary tree:  $n = 2^h$  where n is total number of leaf nodes and h is the height of the binary tree
- The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.



- The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.
- **Properties of Complete Binary Tree**
  - The maximum number of nodes in complete binary tree is  $2^{h+1} - 1$ .
  - The minimum number of nodes in complete binary tree is  $2^h$ .
  - The minimum height of a complete binary tree is  $\log_2(n+1) - 1$ .

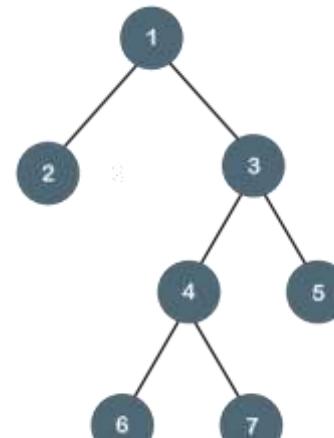
## Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



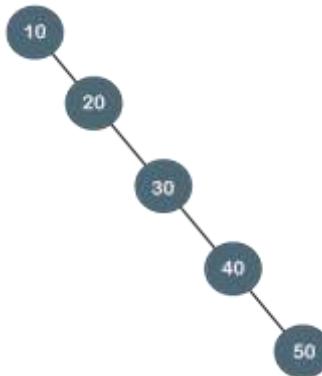
The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

**Note:** All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

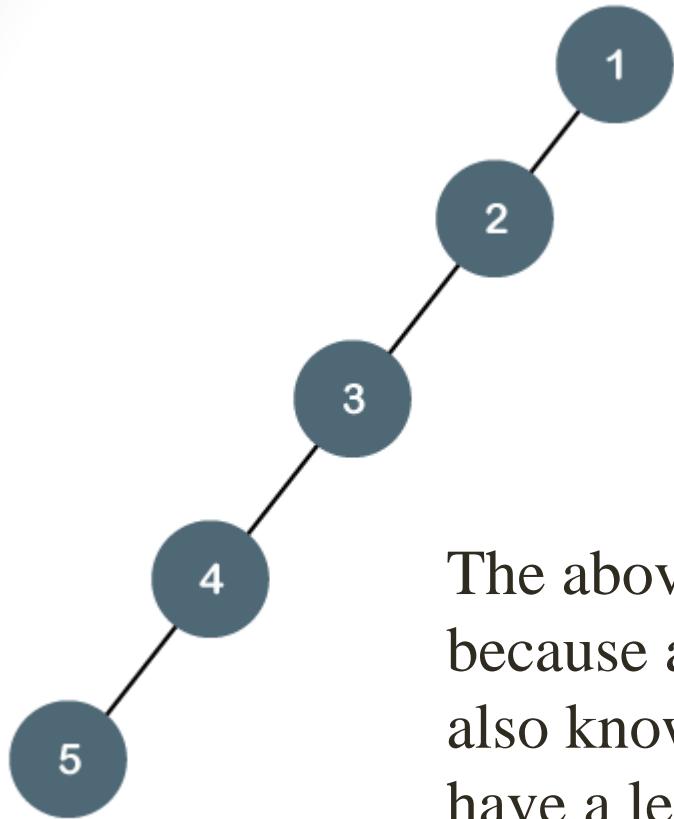


## Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

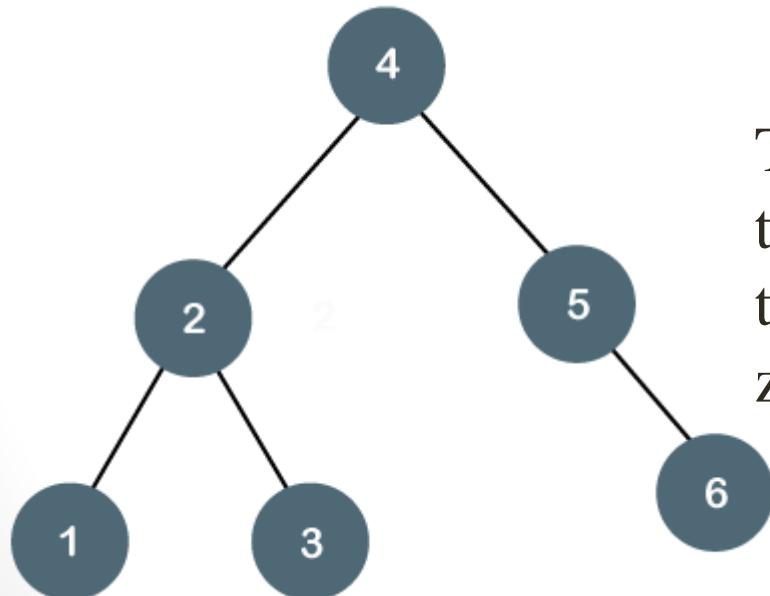


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

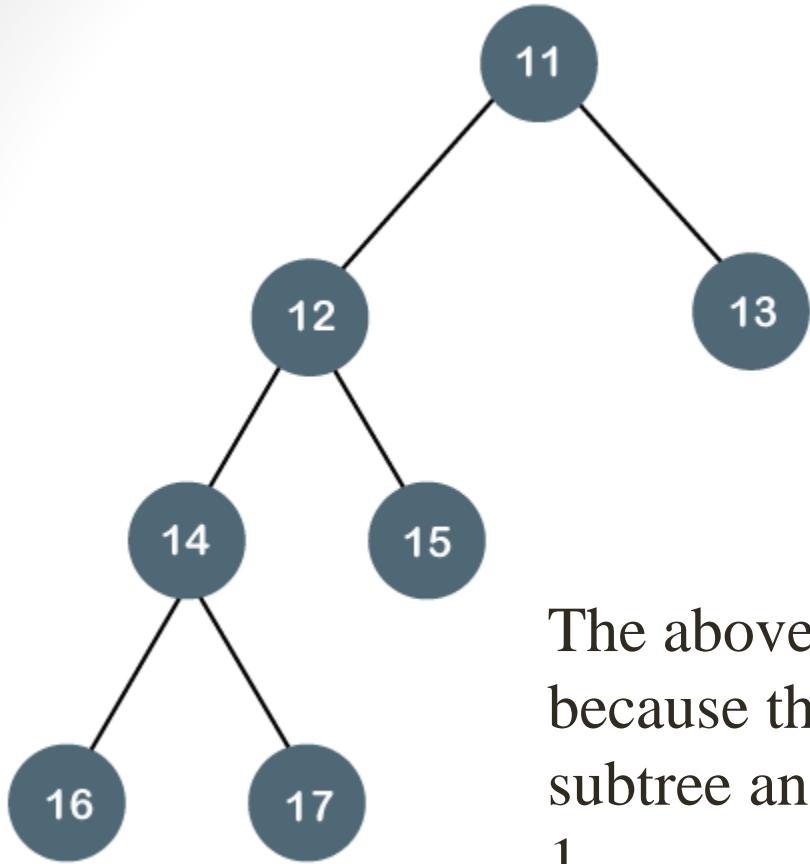
- **Left And Right Skewed Trees**
- The tree in which each node is attached as a left child of parent node then it is left skewed tree.
- The tree in which each node is attached as a right child of parent node then it is called right skewed tree.

## Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, *AVL* and *Red-Black trees* are balanced binary tree.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.

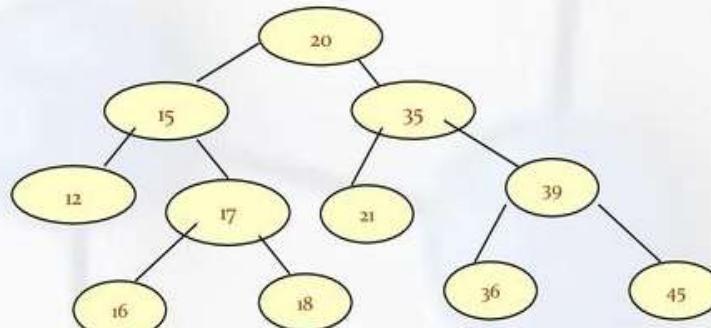


The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

# Tree Representation

- The **sequential representation** uses an array for the storage of tree elements.
- The number of nodes a binary tree has defines the size of the array being used. The root node of the binary tree lies at the array's first index.
- The index at which a particular node is stored will define the indices at which the right and left children of the node will be stored.
- An empty tree has null or 0 as its first index.
- For any element in position  $I$ , the left child is in position  $2i$ , the right child is in position  $(2i+1)$  and the parent is in position  $(i/2)$

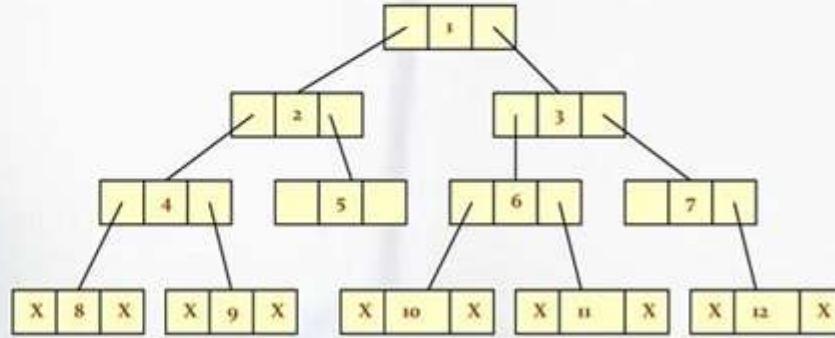
- Sequential representation of trees is done using a single or one dimensional array. Though, it is the simplest technique for memory representation, it is very inefficient as it requires a lot of memory space.
- A sequential binary tree follows the rules given below:
- One dimensional array called TREE is used.
- The root of the tree will be stored in the first location. That is, TREE[1] will store the data of the root element.
- The children of a node K will be stored in location  $(2 \cdot K)$  and  $(2 \cdot K + 1)$ .
- The maximum size of the array TREE is given as  $(2^h - 1)$ , where  $h$  is the height of the tree.
- An empty tree or sub-tree is specified using NULL. If TREE[1] = NULL, then the tree is empty.



# Linked Representation

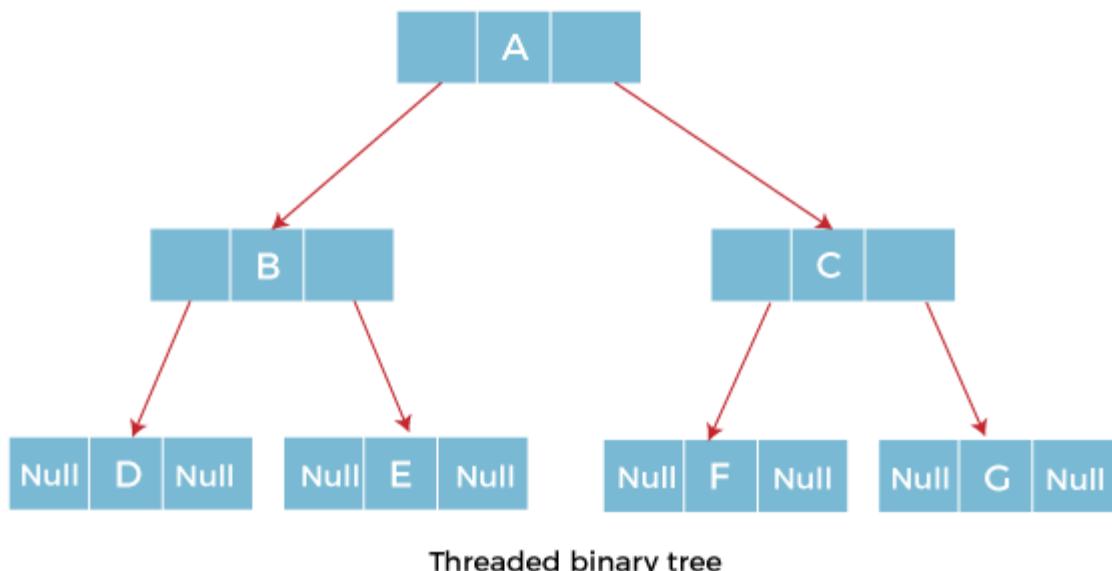
- In computer's memory, a binary tree can be maintained either using a linked representation or using sequential representation.
- In linked representation of binary tree, every node will have three parts: the data element, a pointer to the left node and a pointer to the right node. So in C, the binary tree is built with a node type given as below.

```
struct node
{
    struct node* left;
    int data;
    struct node* right;
};
```

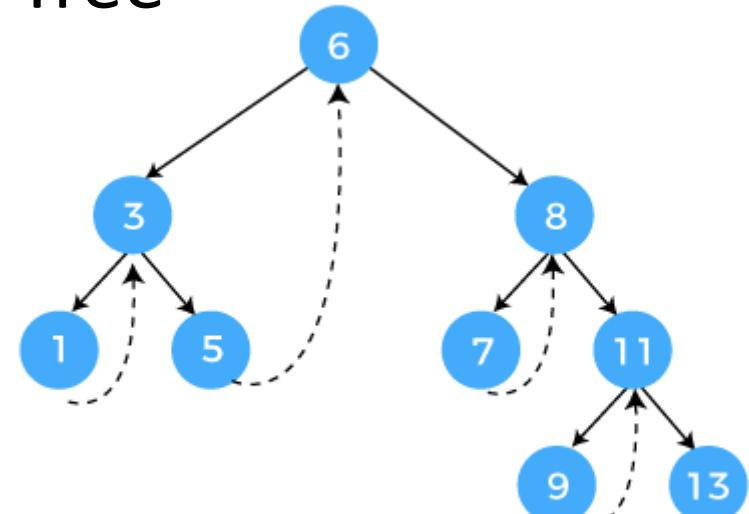


# Threaded Binary Trees

- In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of  $n$  nodes then  $n+1$  link fields contain NULL values.
- So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as **threaded binary trees**.
- Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.



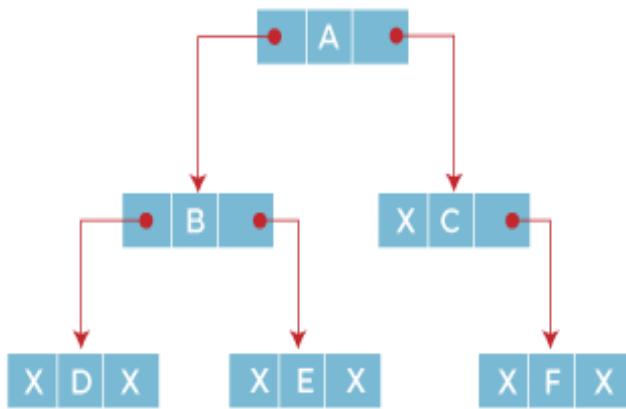
- Types of Threaded Binary Tree
- There are two types of threaded Binary Tree:
- One-way threaded Binary Tree
- Two-way threaded Binary Tree



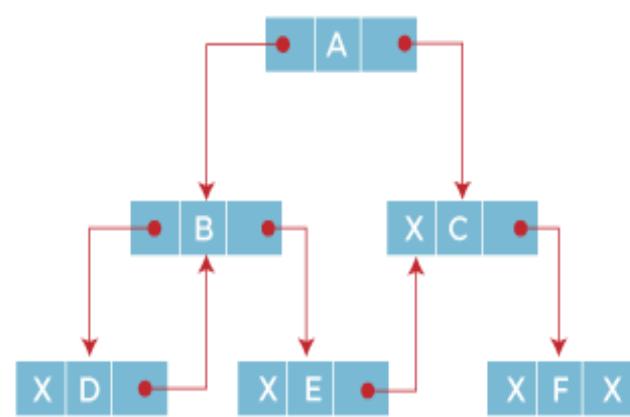
Single Threaded Binary Tree

- In one-way threaded binary trees, a thread will appear either in the right or left link field of a node.
- If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called **Right threaded binary trees**.
- If thread appears in the left field of a node then it will point to the nodes inorder predecessor. Such trees are called **Left threaded binary trees**.

- Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees.
- In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL. In order to distinguish threads from normal links they are represented by dotted lines.

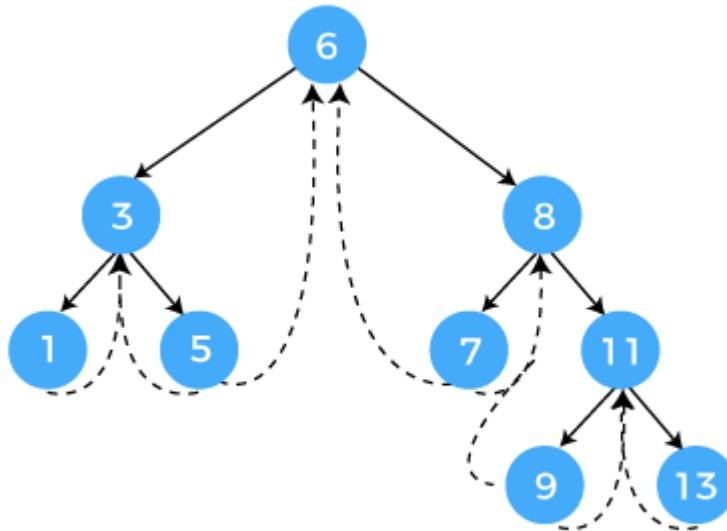


A binary tree ( Inorder traversal - D, B, E, A, C, F )

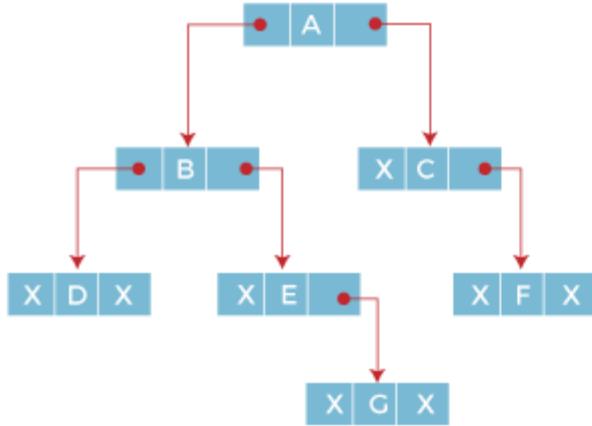


A right - threaded binary tree

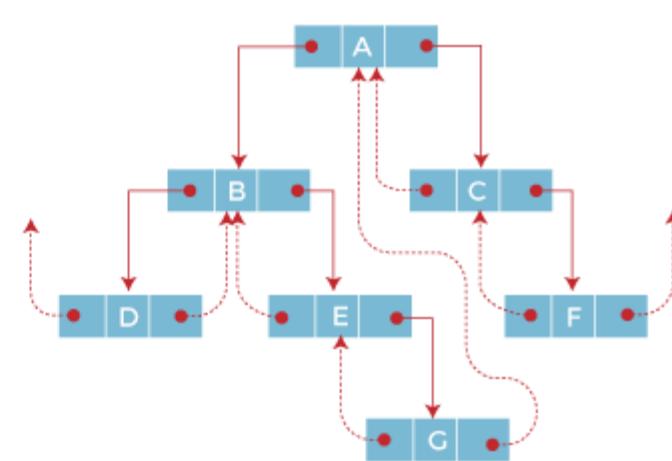
# Two-way threaded Binary Trees



In two-way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.

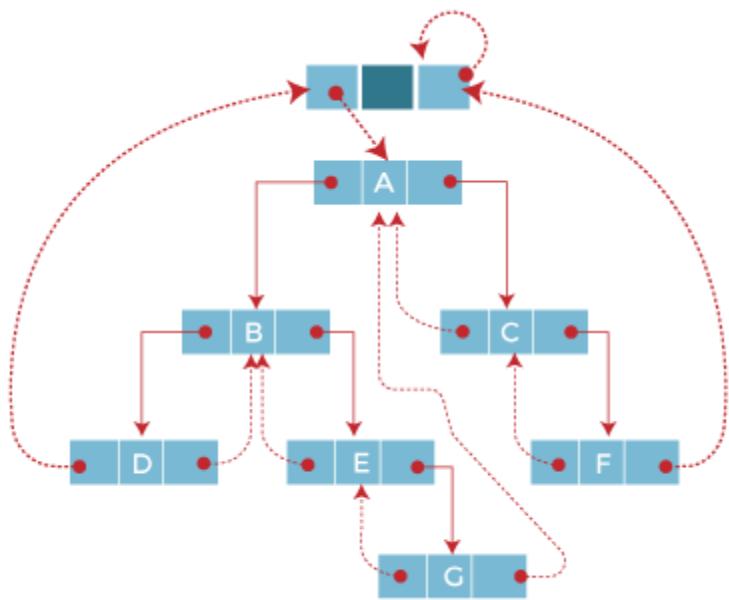


A binary tree ( Inorder traversal - D, B, E, G, A, C, F )



A two - way threaded binary tree

- The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F.
- If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B.
- Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor.
- In the same way, other nodes containing NULL values in their link fields are filled with threads.



Two-way threaded - tree with header node

- **Advantages of Threaded Binary Tree**
- In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack. If the stack is used then it consumes a lot of memory and time.
- It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links.
- It almost behaves like a circular linked list.

- **Disadvantages of Threaded Binary Tree**
- When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
- Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.

# Tree Traversal

- Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node.
- we cannot randomly access a node in a tree.

There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

## In-order Traversal

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

## Algorithm

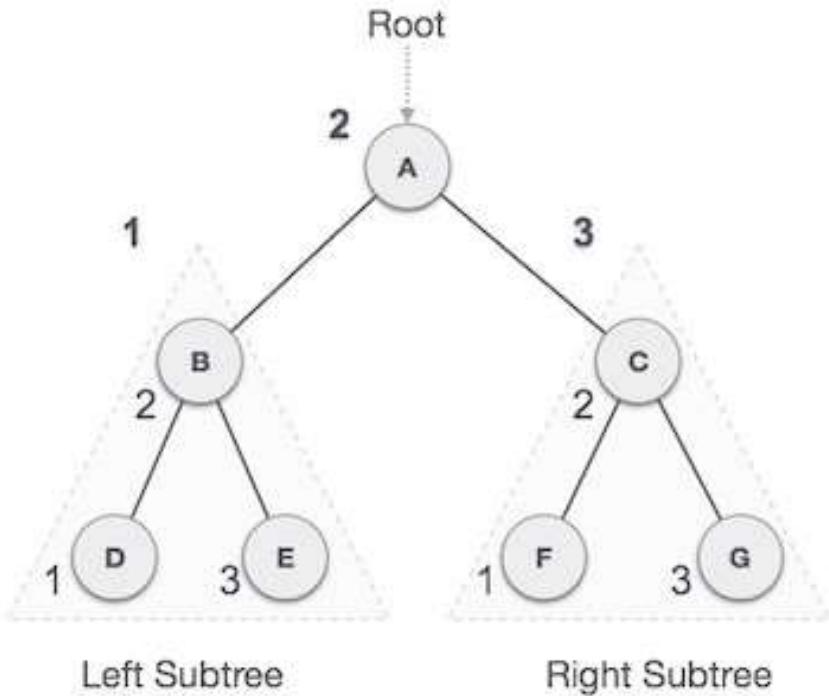
Until all nodes are traversed –

**Step 1** – Recursively traverse left subtree.

**Step 2** – Visit root node.

**Step 3** – Recursively traverse right subtree.

```
void inorder_traversal (Tree T)
{
    if( T != NULL)
    {
        inorder_traversal(T->lchild);
        printElement(T->Element);
        inorder_traversal(T->rchild);
    }
}
```



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

**D → B → E → A → F → C → G**

## Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

### Algorithm

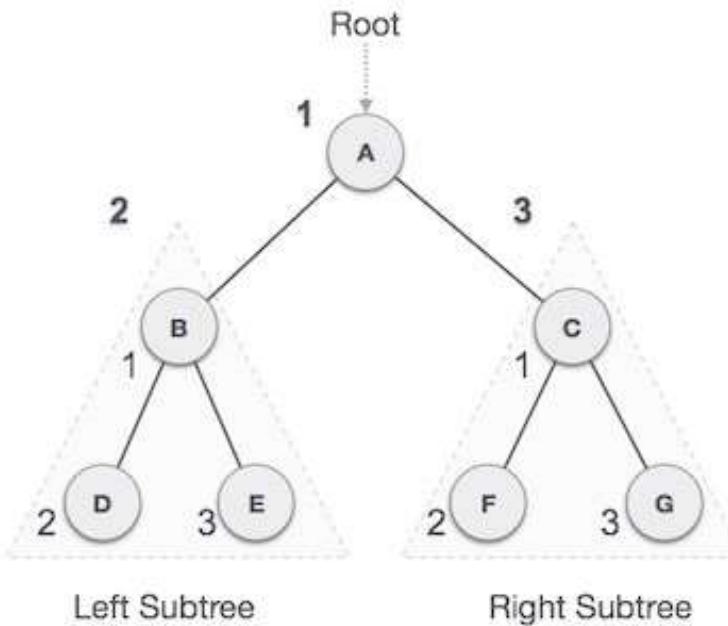
Until all nodes are traversed –

**Step 1** – Visit root node.

**Step 2** – Recursively traverse left subtree.

**Step 3** – Recursively traverse right subtree.

```
void preorder_traversal (Tree T)
{
    if( T != NULL)
    {
        printElement(T->Element);
        preorder_traversal(T->lchild);
        preorder_traversal(T->rchild);
    }
}
```



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited.

The output of pre-order traversal of this tree will be –

***A → B → D → E → C → F → G***

## Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

### Algorithm

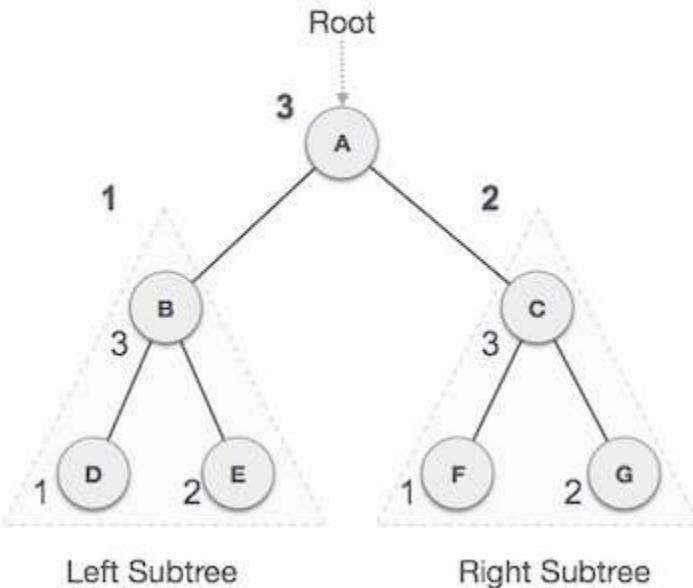
Until all nodes are traversed –

**Step 1** – Recursively traverse left subtree.

**Step 2** – Recursively traverse right subtree.

**Step 3** – Visit root node.

```
void postorder_traversal (Tree T)
{
    if( T != NULL)
    {
        postorder_traversal(T->lchild);
        postorder_traversal(T->rchild);
        printElement(T->Element);
    }
}
```



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited.

The output of post-order traversal of this tree will be –

**D → E → B → F → G → C → A**

# Expression Trees

It is a binary tree in which the leaf nodes are operands and the interior nodes are operators. Like binary tree, expression tree can also traversed by inorder, preoder and post order traversal.

- Expression Tree is a special kind of binary tree with the following properties:
- Each leaf is an operand. Examples: a, b, c, 6, 100
- The root and internal nodes are operators. Examples: +, -, \*, /, ^
- Subtrees are subexpressions with the root being an operator.

# Construction of Expression Tree

Let us consider a **postfix expression** is given as an input for constructing an expression tree. Following are the step to construct an expression tree:

- Read one symbol at a time from the postfix expression.
- Check if the symbol is an operand or operator.
- If the symbol is an operand, create a one node tree and push a pointer onto a stack
- If the symbol is an operator, pop two pointers from the stack namely  $T_1$  &  $T_2$  and form a new tree with root as the operator,  $T_1$  &  $T_2$  as a left and right child
- A pointer to this new tree is pushed onto the stack

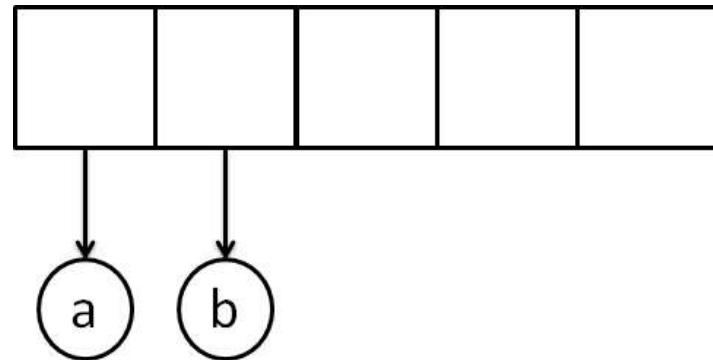
Thus, An expression is created or constructed by reading the symbols or numbers from the left. If operand, create a node. If operator, create a tree with operator as root and two pointers to left and right subtree

## Example - Postfix Expression Construction

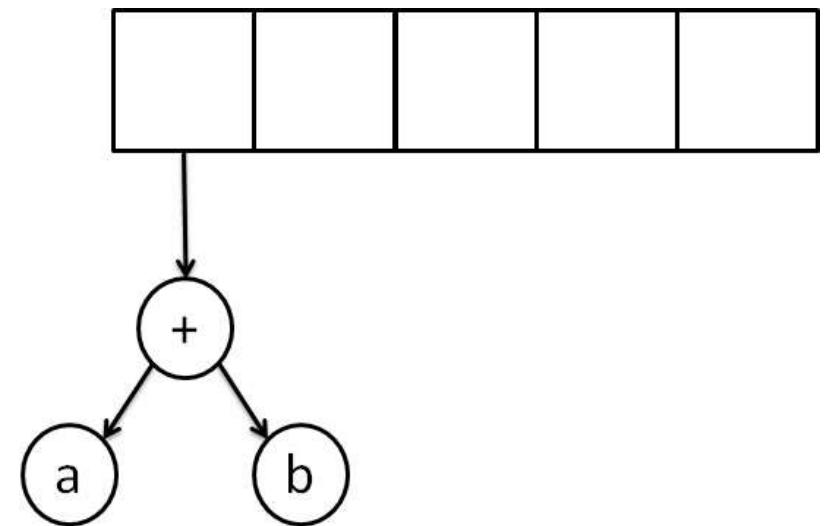
The input is:

a b + c \*

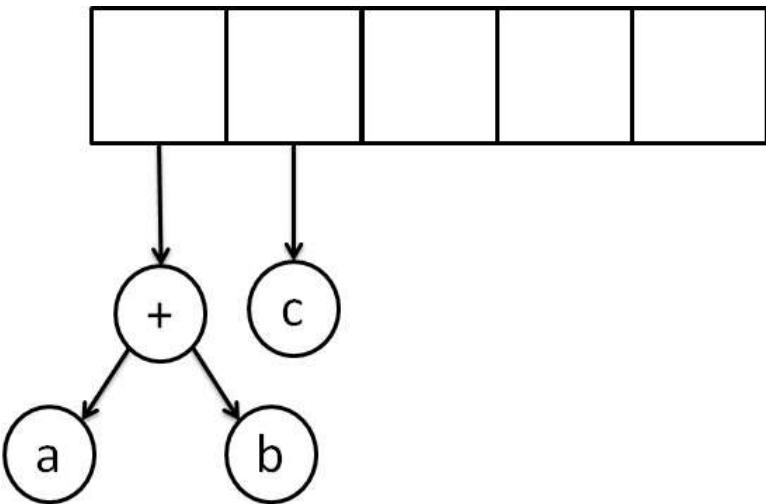
The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



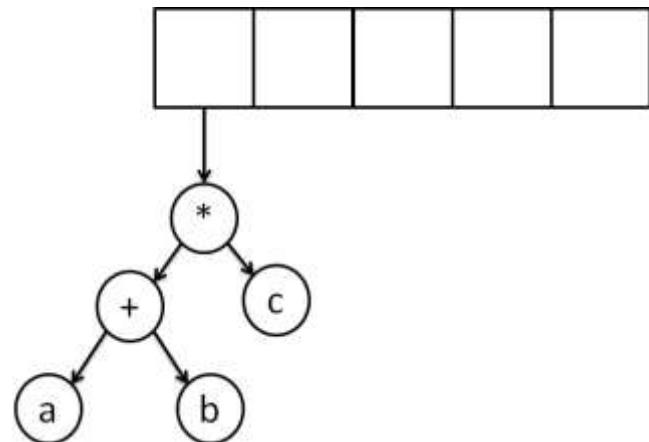
Next, read a '+' symbol, so two pointers to tree  
are popped,a new tree is formed and push a  
pointer to it onto the stack.



Next, 'c' is read, we create one node tree and push a pointer to it onto the stack.



Finally, the last symbol is read '\*', we pop two tree pointers and form a new tree with a, '\*' as root, and a pointer to the final tree remains on the stack.

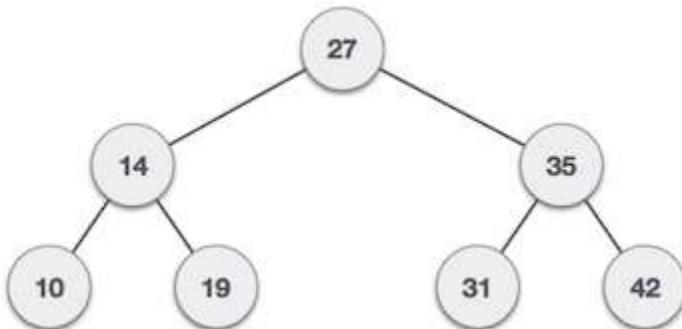


# Binary Search Tree (BST)

- A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –
- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.
- Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –
- $\text{left\_subtree(keys)} < \text{node(key)} \leq \text{right\_subtree(keys)}$

# Representation of BST

- BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.
- Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

## Basic Operations

Following are the basic operations of a tree

- **Search** – Searches an element in a tree.
- **Insert** – Inserts an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

## Node

Define a node having some data, references to its left and right child nodes.

```
struct node
```

```
{
```

```
    int data;
```

```
    struct node *leftChild;
```

```
    struct node *rightChild;
```

```
};
```

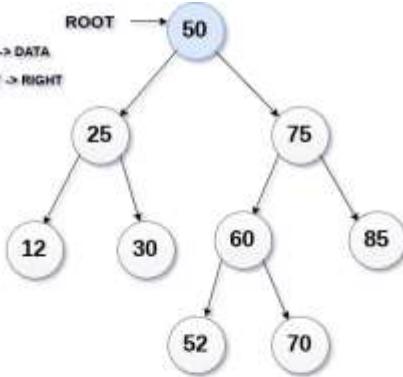
## Search Operation

- Whenever an element is to be searched, start searching from the root node.
- Then if the data is less than the key value, search for the element in the left subtree.
- Otherwise, search for the element in the right subtree.
- Follow the same algorithm for each node.

## **Algorithm:**

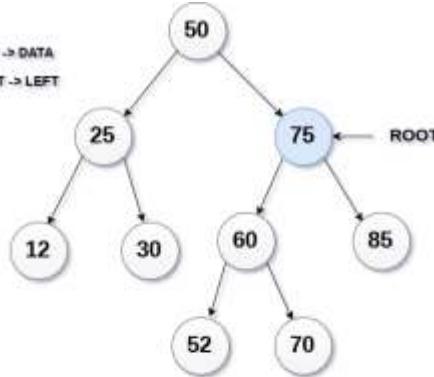
```
find(BSTnod *rroot, int x)
{
    If ((root == NULL))
        Return(NULL);
    If(root -> data == x)
        Return(root);
    If (x > root -> data)
        Return (find (root -> right), x));
    Else
        Return(find(root->left), x));
}
```

Item = 60  
ITEM > ROOT -> DATA  
ROOT = ROOT -> RIGHT



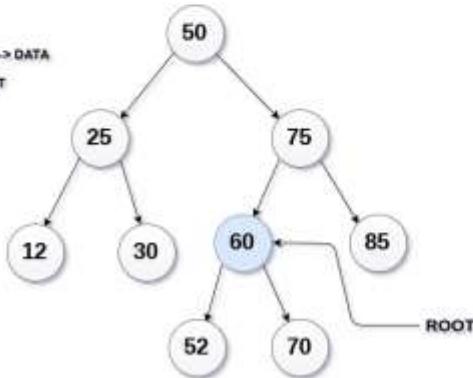
STEP 1

Item = 60  
ITEM < ROOT -> DATA  
ROOT = ROOT -> LEFT



STEP 2

Item = 60  
ITEM = ROOT -> DATA  
RETURN ROOT



STEP 3

## **Insertion:**

- Check whether the root node of the binary search tree is NULL.
- If the condition is true, it mean, that the binary search tree is empty hence consider the new node as the root node.
- Compare the new node data with root node data, for the following three condition
  - if the content of the new node is equal to the root node insertion operation is terminated.
  - If the content of the new node is lesser than the root node data. Check whether the left child of the root node is null. If it is true insert the new node.
  - If the content of the newnode data is greater than the root node data, check whether the right child of the root node is NULL. If the condition is true, insert the new data.

**Algorithm:**

```
BSTnode *insert(BST *T, int x)
{
    If(T==NULL)
    {
        T=(BSTnode*)malloc(sizeof(BSTnode));
        If(T!=NULL)
        {
            T->data=x;
            T->left=NULL;
            T->right=NULL;
        }
        Else
        if(x > T -> data)
        {
            T -> right = insert (T -> right, x);
            Return(T);
        }
        Else
        {
            T -> left = insert(T -> left, x);
            Return (T);
        }
    }
}
```

```

void insert(int data)
{
    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;

    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty
    if(root == NULL)
    {
        root = tempNode;
    } else
    {
        current = root;
        parent = NULL;

        while(1)
        {
            parent = current;

            //go to left of the tree
            if(data < parent->data)
            {
                current = current->leftChild;
                //insert to the left

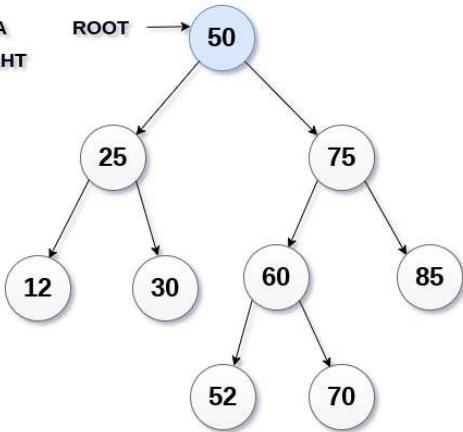
                if(current == NULL)
                {
                    parent->leftChild = tempNode;
                    return;
                }
            } //go to right of the tree
            else
            {
                current = current->rightChild;

                //insert to the right
                if(current == NULL)
                {
                    parent->rightChild = tempNode;
                    return;
                }
            }
        }
    }
}

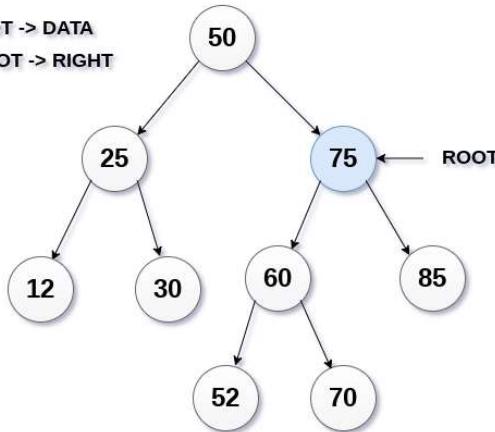
```

Item = 95

ITEM > ROOT -> DATA  
ROOT = ROOT -> RIGHT

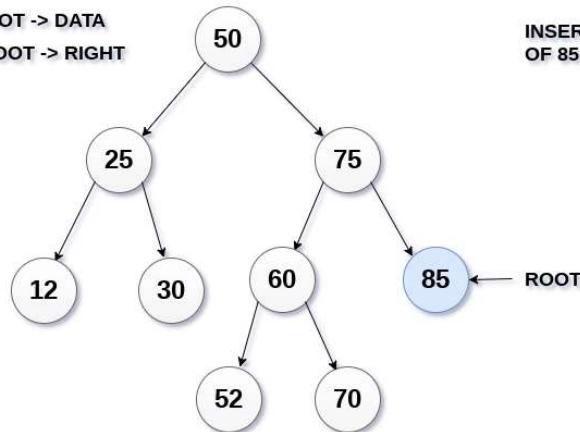


ITEM > ROOT -> DATA  
ROOT = ROOT -> RIGHT



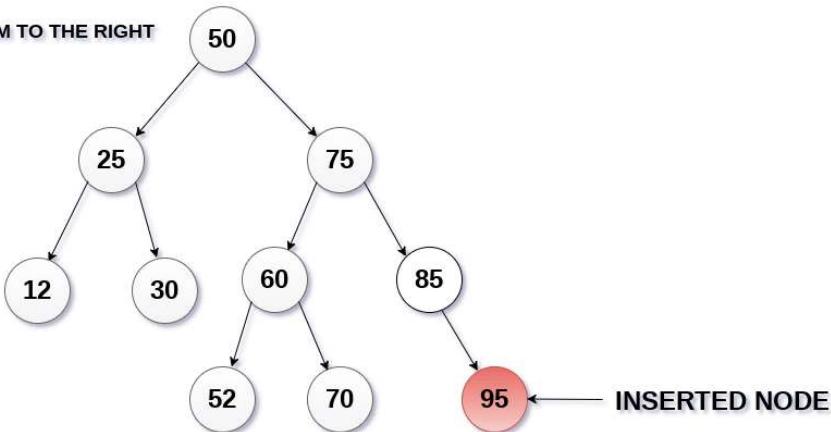
STEP 1

ITEM > ROOT -> DATA  
ROOT = ROOT -> RIGHT



STEP 2

INSERT ITEM TO THE RIGHT  
OF 85



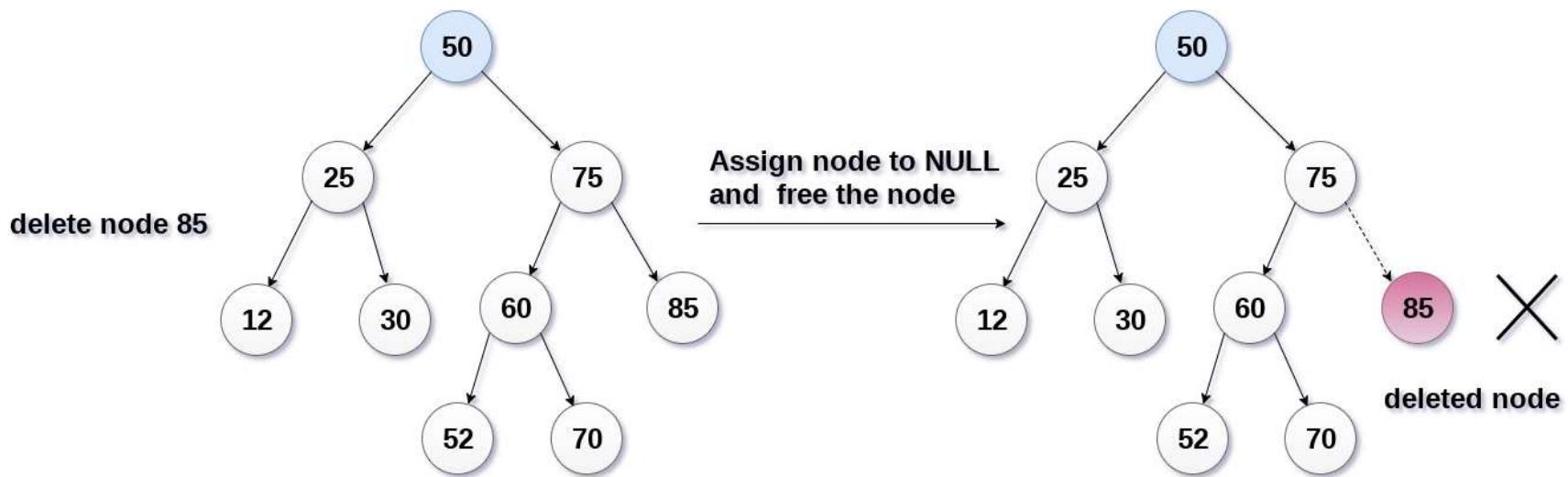
STEP 3

STEP 4

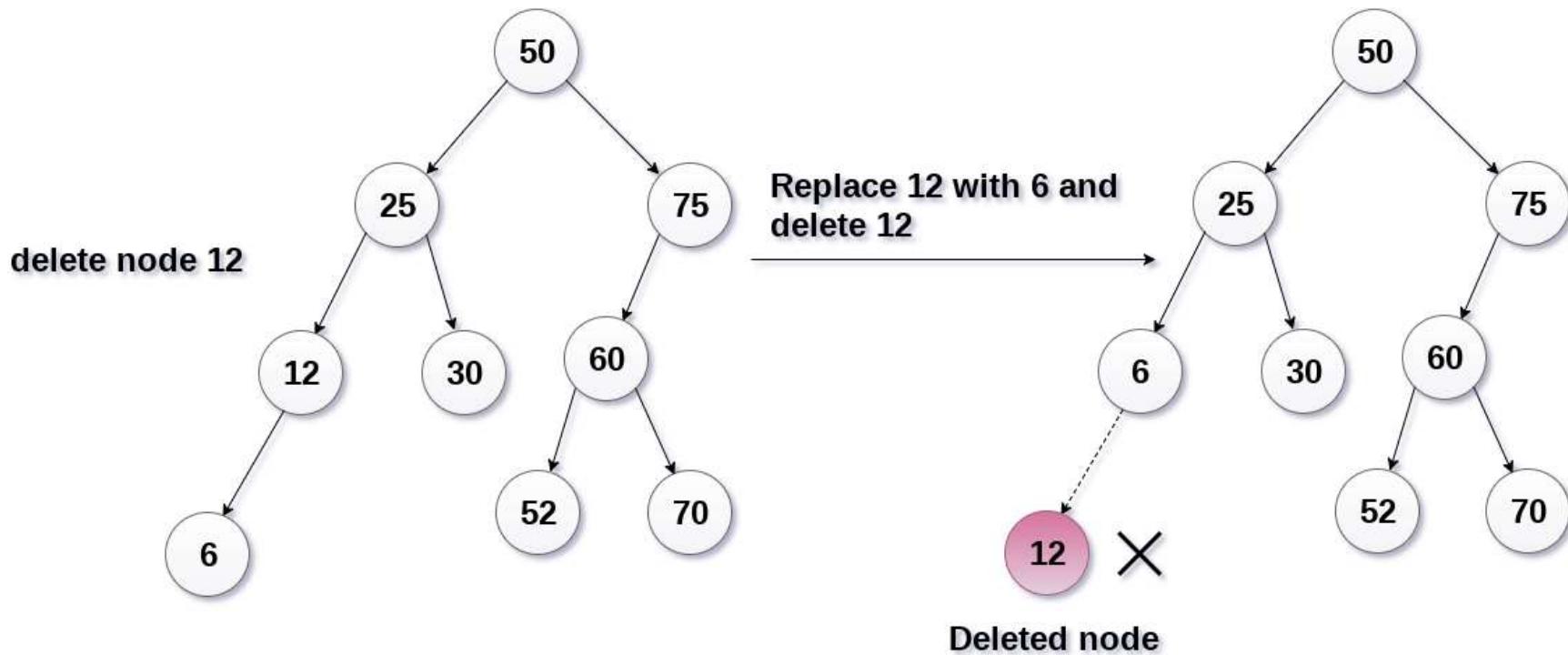
## **Deletion:**

- Memory is to be released for the node to be deleted. A node can be deleted from the tree from three different places namely,
- Deleting the leaf node
- Deleting the node with only one child
- Deleting the node with two children.

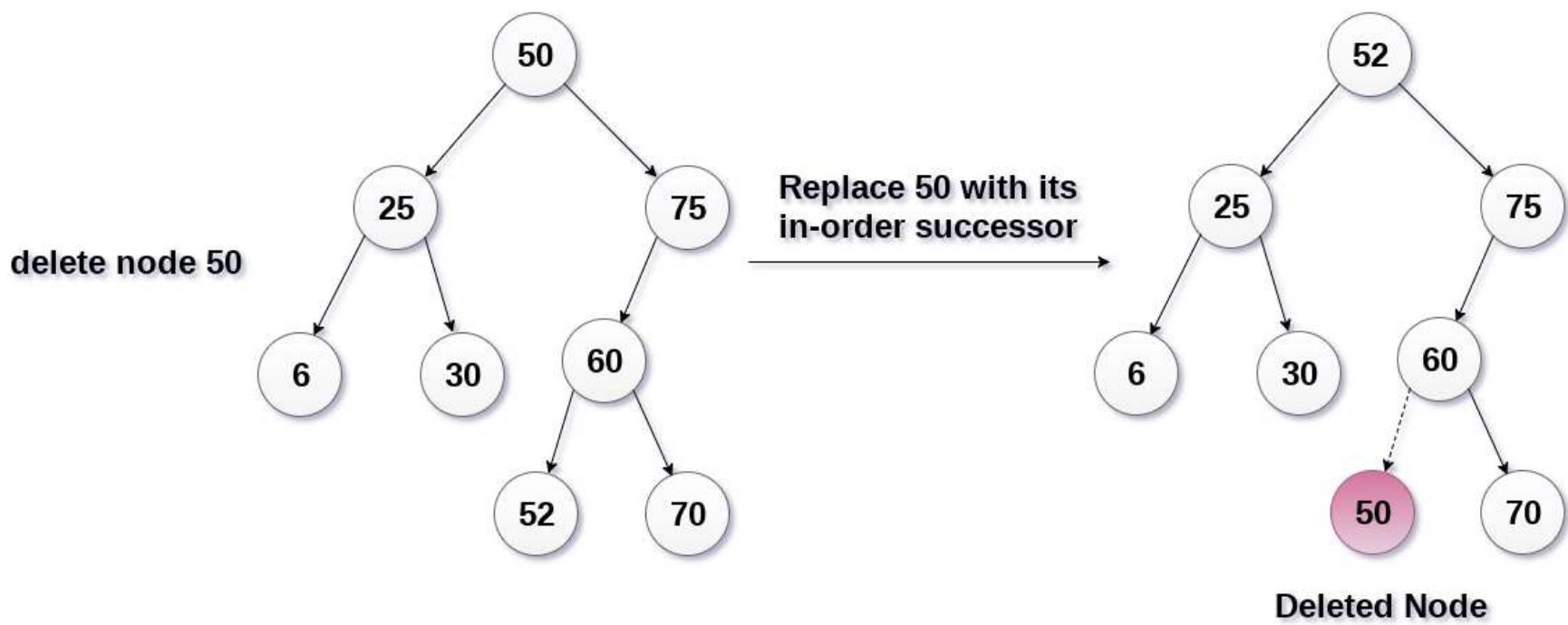
- The node to be deleted is a leaf node
- It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.
- In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



- The node to be deleted has only one child.
- In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.
- In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



- The node to be deleted has two children.
- It is a bit complexed case compare to other two cases.
- However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree.
- After the procedure, replace the node with NULL and free the allocated space.
- In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.
- 6, 25, 30, 50, 52, 60, 70, 75.
- replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



```

SearchTree Delete (int X, searchTree T)
{
    int Tmpcell ;

    if (T == NULL)
        Error ("Element not found");
    else
        if (X < T → Element) // Traverse towards left
            T → Left = Delete (X, T → Left);
        else
            if (X > T → Element) // Traverse towards right
                T → Right = Delete (X, T → Right);
                // Found Element to be deleted
            else
                // Two children
                if (T → Left && T → Right)
                {
                    // Replace with smallest data in right subtree
                    Tmpcell = FindMin (T → Right);
                    T → Element = Tmpcell → Element ;
                    T → Right = Delete (T → Element;
                                         T → Right);
                }
            else // one or zero children
            {
                Tmpcell = T;
                if (T → Left == NULL)
                    T = T → Right;
                else if (T → Right == NULL)
                    T = T → Left ;
                free (TmpCell);
            }
        return T;
}

```

## Find Min:

This function returns to address of the node with smallest value in the tree.

### Algorithm:

```
BSTnode *find min (BSTnode *T)
```

```
{
```

```
    While (T -> left! = NULL)
```

```
        T = T -> left;
```

```
    Return(T);
```

```
}
```

## Find Max:

This function returns the address of the node with largest value in the tree.

### Algorithm:

```
BSTnode find max(BSTnode *T)
```

```
{
```

```
    While (T -> right! = NULL)
```

```
        T = T -> right;
```

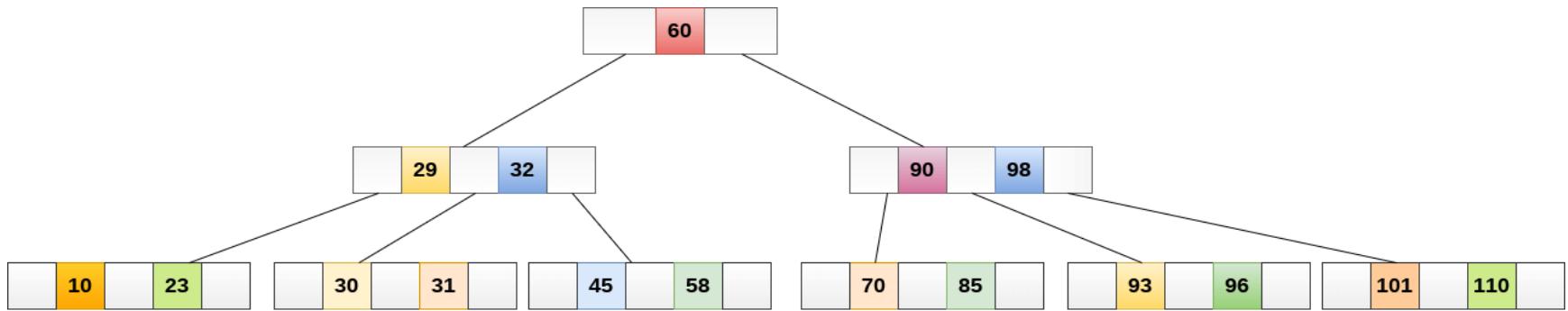
```
    Return(T);
```

```
}
```

# B Tree

- B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most  $m-1$  keys and m children.
- One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.
- A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.
  - Every node in a B-Tree contains at most m children.
  - Every node in a B-Tree except the root node and the leaf node contain at least  $m/2$  children.
  - The root nodes must have at least 2 nodes.
  - All leaf nodes must be at the same level.
  - It is not necessary that, all the nodes contain the same number of children but, each node must have  $m/2$  number of nodes.

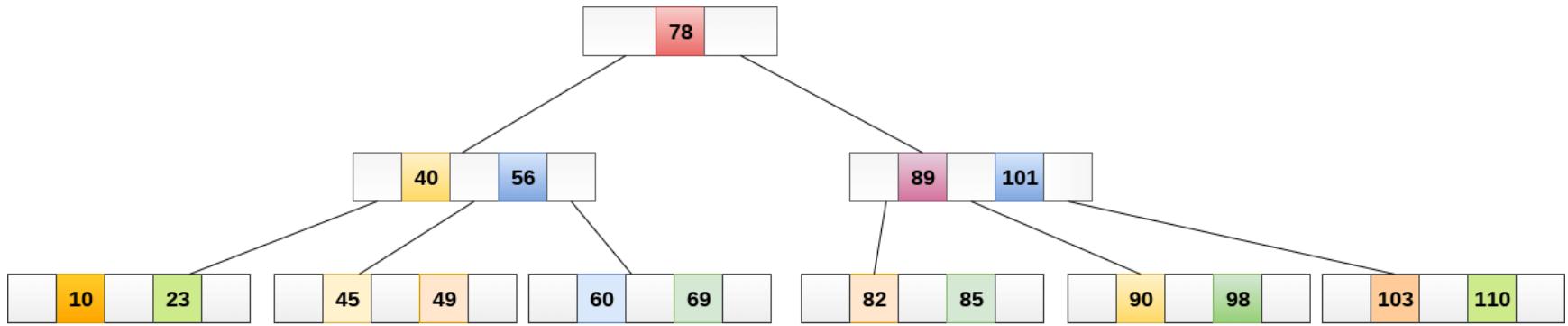
# A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

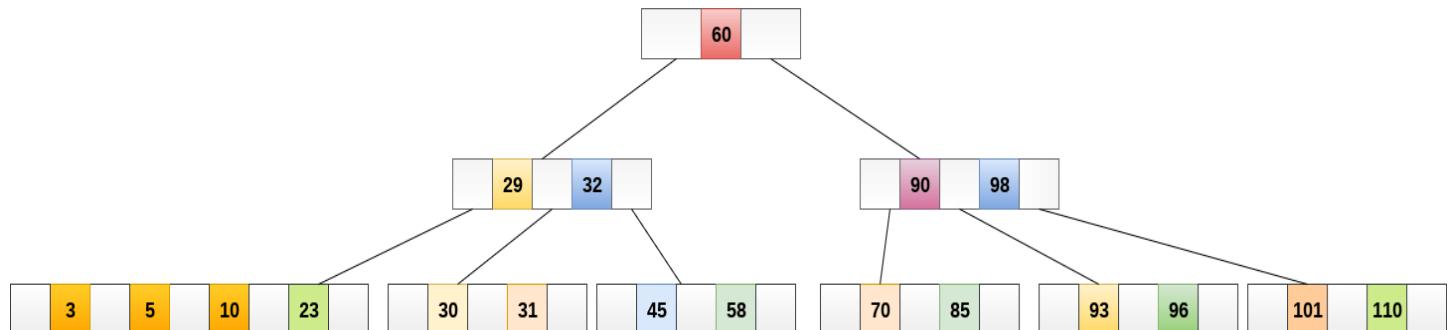
# Operations

- **Searching :**
- Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :
  - Compare item 49 with root node 78. since  $49 < 78$  hence, move to its left sub-tree.
  - Since,  $40 < 49 < 56$ , traverse right sub-tree of 40.
  - $49 > 45$ , move to right. Compare 49.
  - match found, return.
- Searching in a B tree depends upon the height of the tree. The search algorithm takes  $O(\log n)$  time to search any element in a B tree.

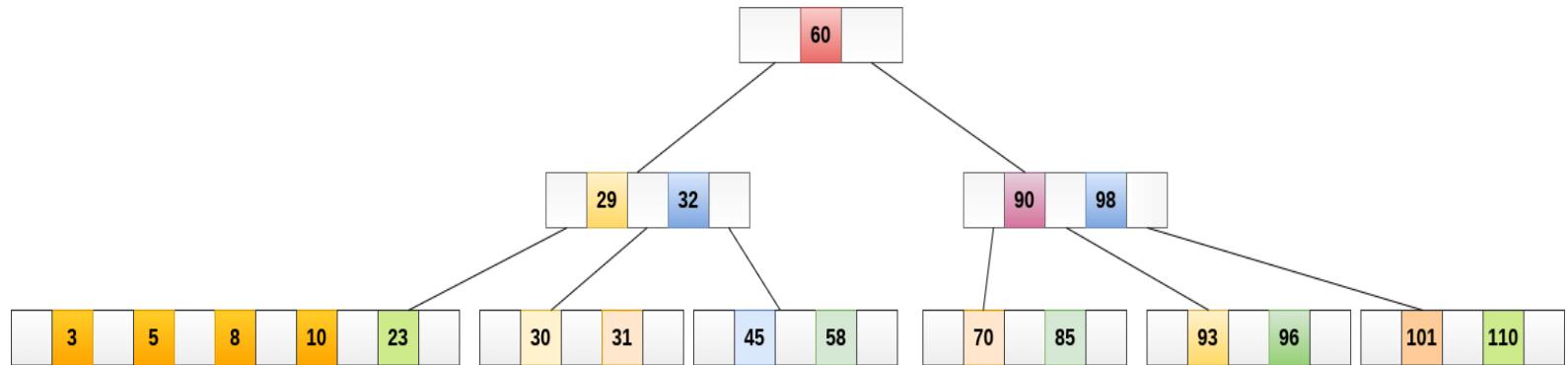


- Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.
- Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
- If the leaf node contain less than  $m-1$  keys then insert the element in the increasing order.
- Else, if the leaf node contains  $m-1$  keys, then follow the following steps.
  - Insert the new element in the increasing order of elements.
  - Split the node into the two nodes at the median.
  - Push the median element upto its parent node.
  - If the parent node also contain  $m-1$  number of keys, then split it too by following the same steps.

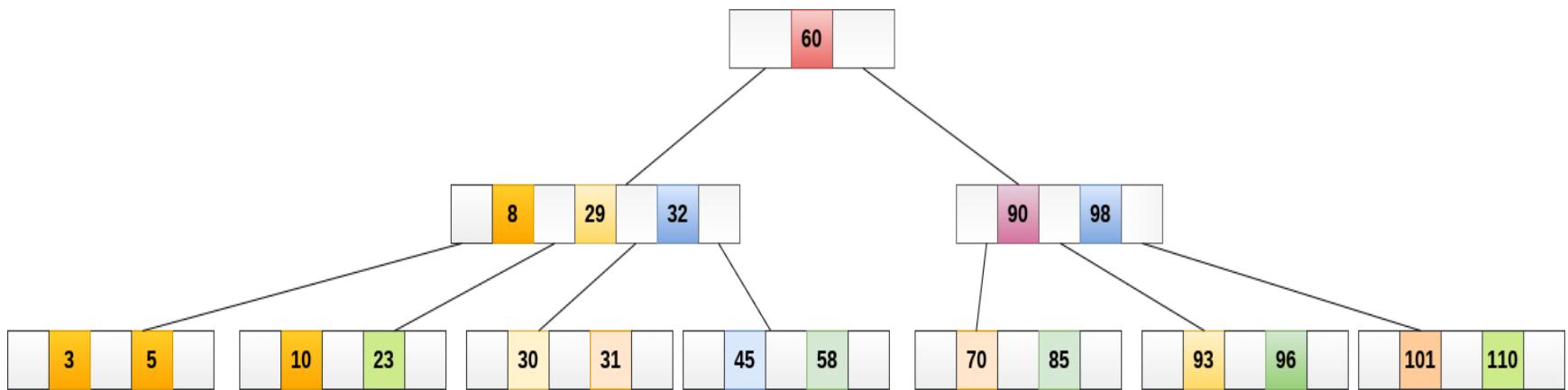
- **Example:**
- Insert the node 8 into the B Tree of order 5 shown in the following image.



- 8 will be inserted to the right of 5, therefore insert 8.



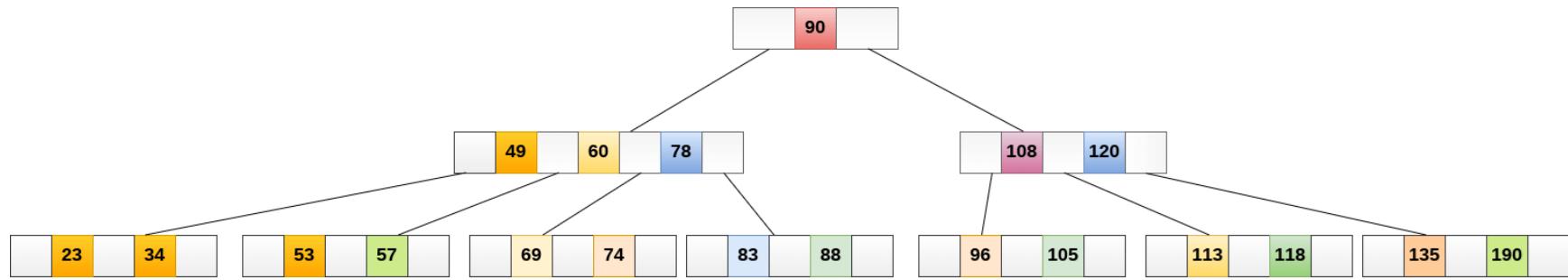
The node, now contain 5 keys which is greater than ( $5 - 1 = 4$ ) keys.  
 Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



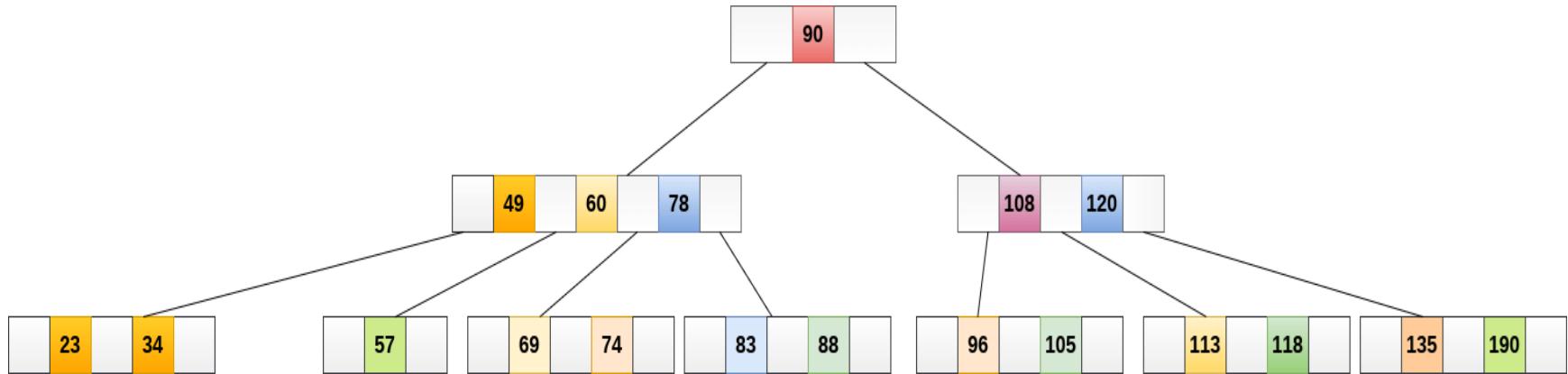
- Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node.
- Following algorithm needs to be followed in order to delete a node from a B tree.
- Locate the leaf node.
- If there are more than  $m/2$  keys in the leaf node then delete the desired key from the node.

- If the leaf node doesn't contain  $m/2$  keys then complete the keys by taking the element from right or left sibling.
  - If the left sibling contains more than  $m/2$  elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
  - If the right sibling contains more than  $m/2$  elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
- If neither of the sibling contain more than  $m/2$  elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
- If parent is left with less than  $m/2$  nodes then, apply the above process on the parent too.

- If the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor.
- Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.
- **Example 1**
- Delete the node 53 from the B Tree of order 5 shown in the following figure.

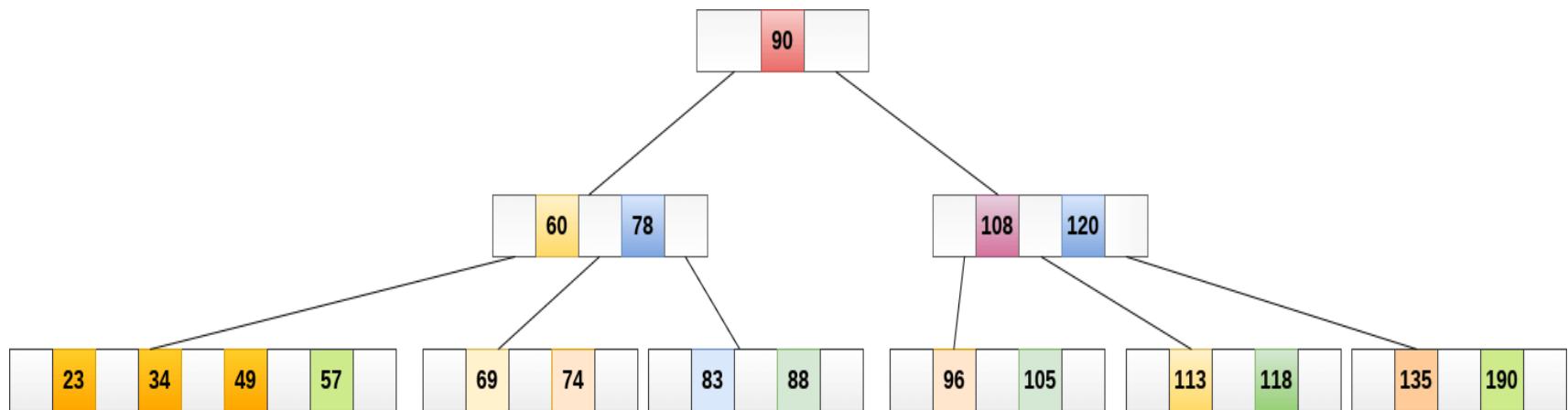


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

- The final B tree is shown as follows.

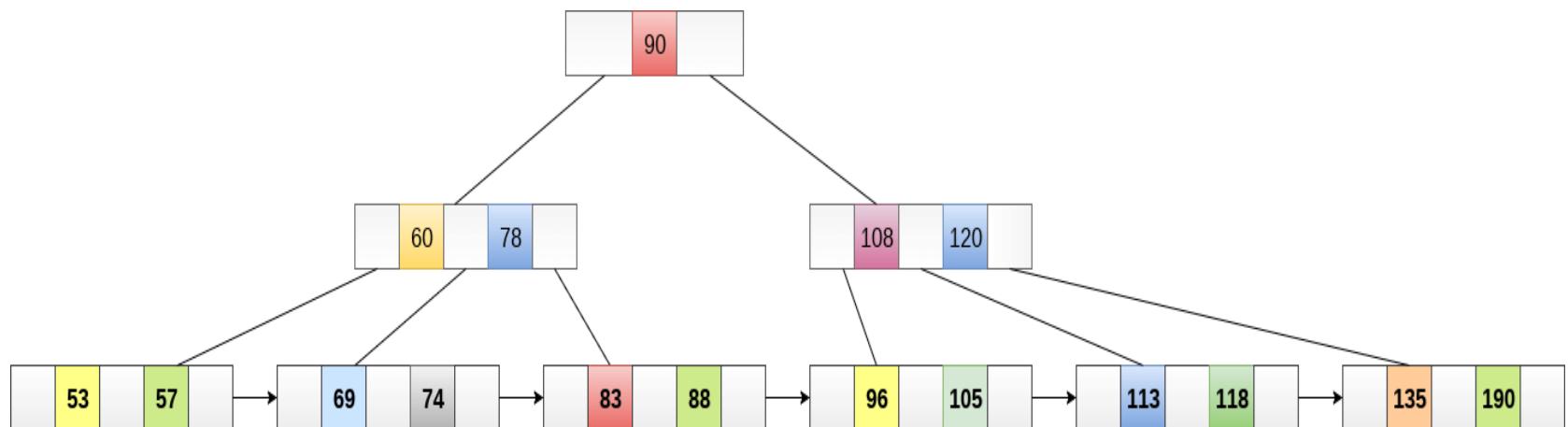


- Application of B tree
- B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.
- Searching an un-indexed and unsorted database containing  $n$  key values needs  $O(n)$  running time in worst case. However, if we use B Tree to index this database, it will be searched in  $O(\log n)$  time in worst case.

# B+ Tree

- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.
- B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

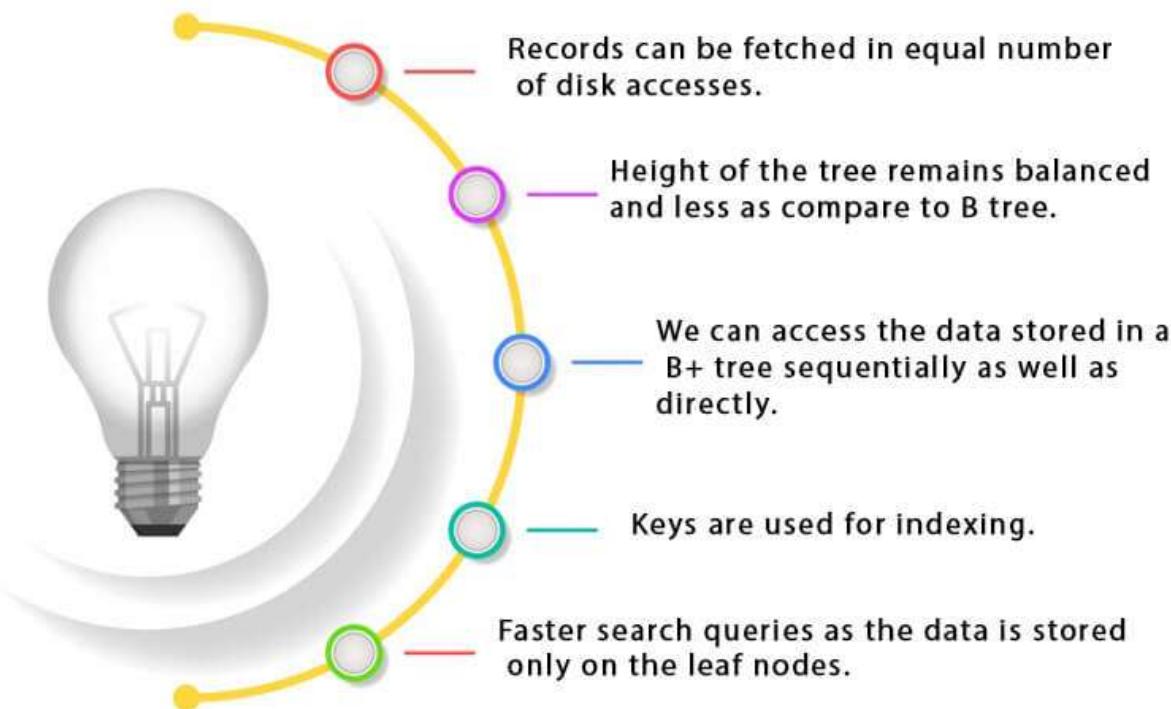
- The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



## Advantages of B+ Tree

- Records can be fetched in equal number of disk accesses.
- Height of the tree remains balanced and less as compare to B tree.
- We can access the data stored in a B+ tree sequentially as well as directly.
- Keys are used for indexing.
- Faster search queries as the data is stored only on the leaf nodes.

## Advantages of B+ Tree

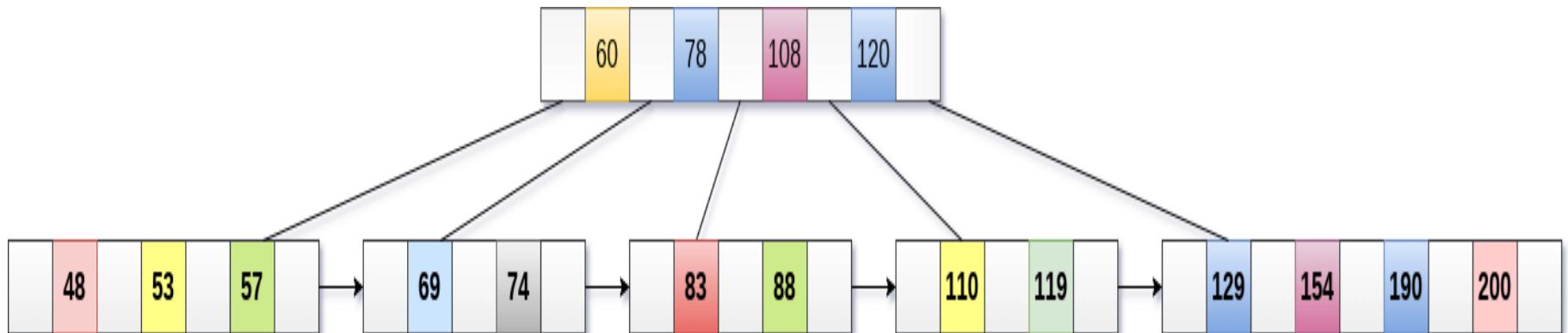


<b>SN</b>	<b>B Tree</b>	<b>B+ Tree</b>
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

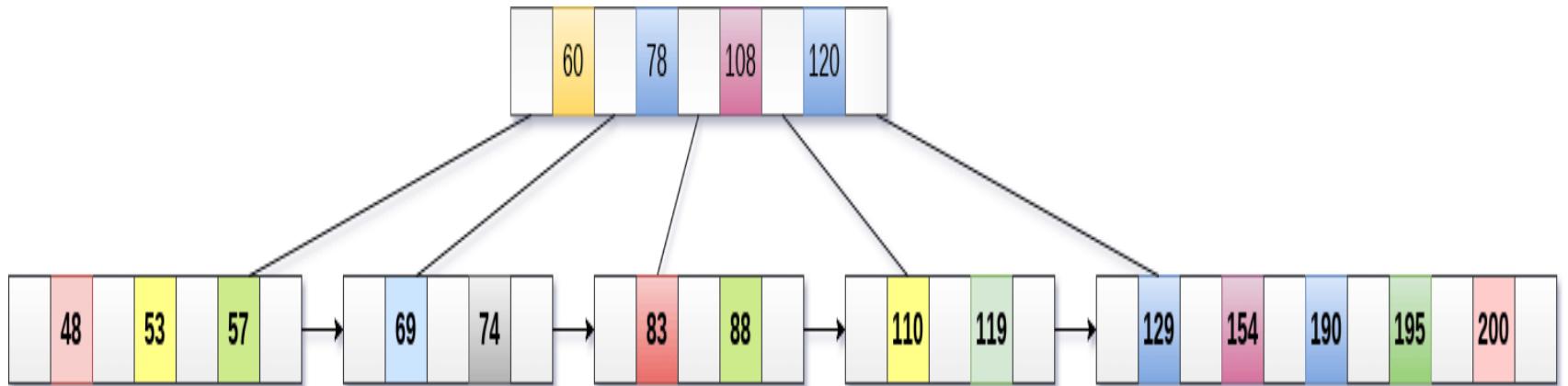
- **Insertion in B+ Tree**
- Step 1: Insert the new node as a leaf node
- Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.
- Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.



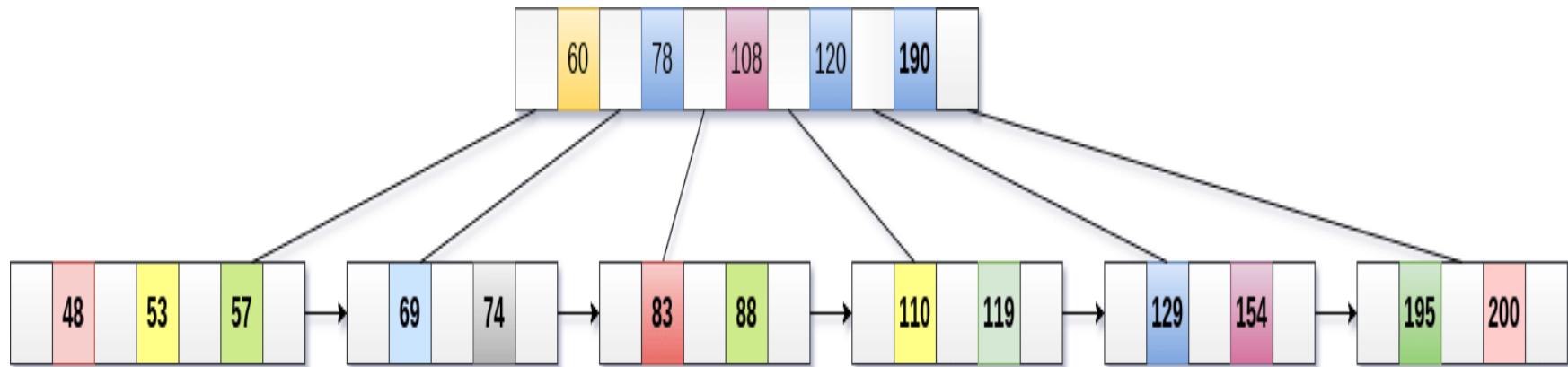
- Example :
- Insert the value 195 into the B+ tree of order 5 shown in the following figure.



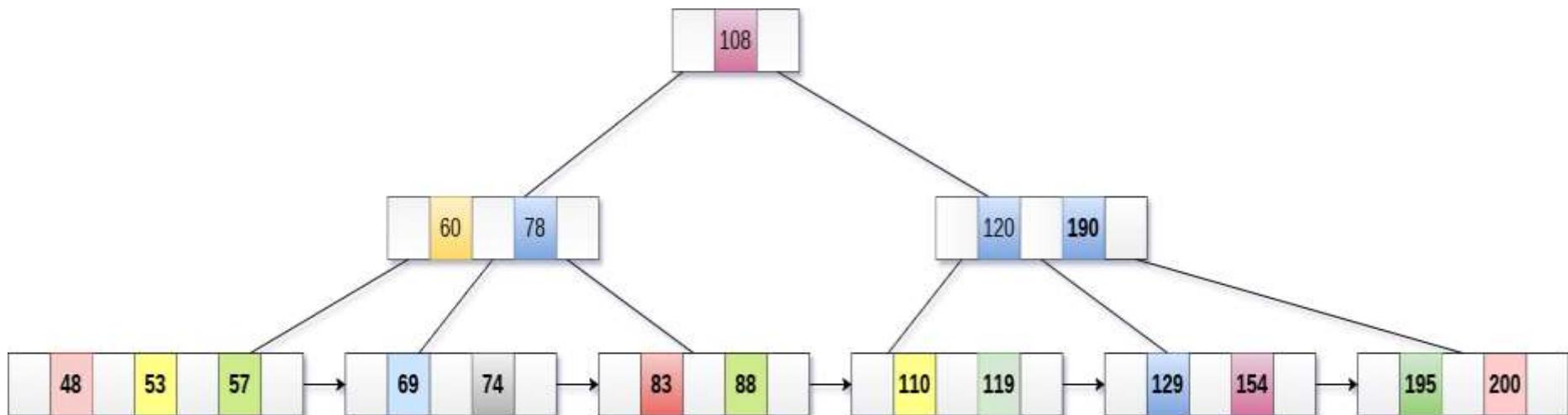
- 195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



- The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



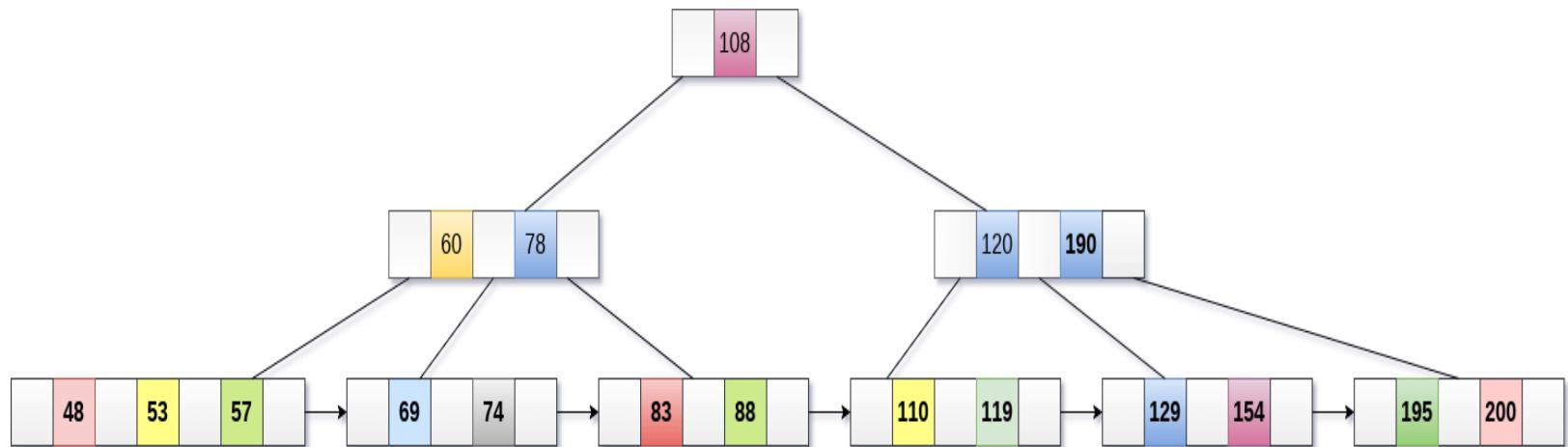
- Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



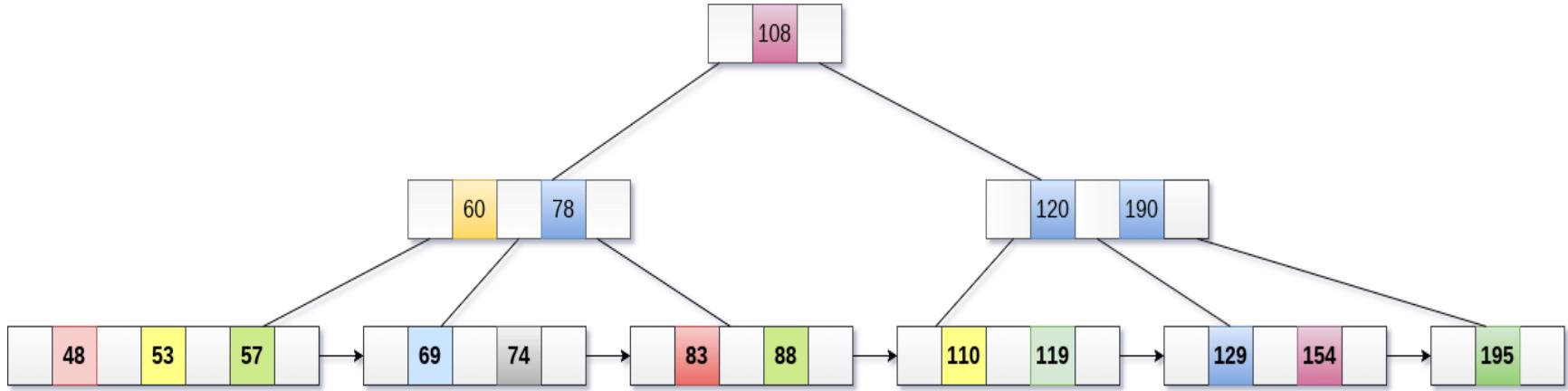
## Deletion in B+ Tree

- Step 1: Delete the key and data from the leaves.
- Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.
- Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

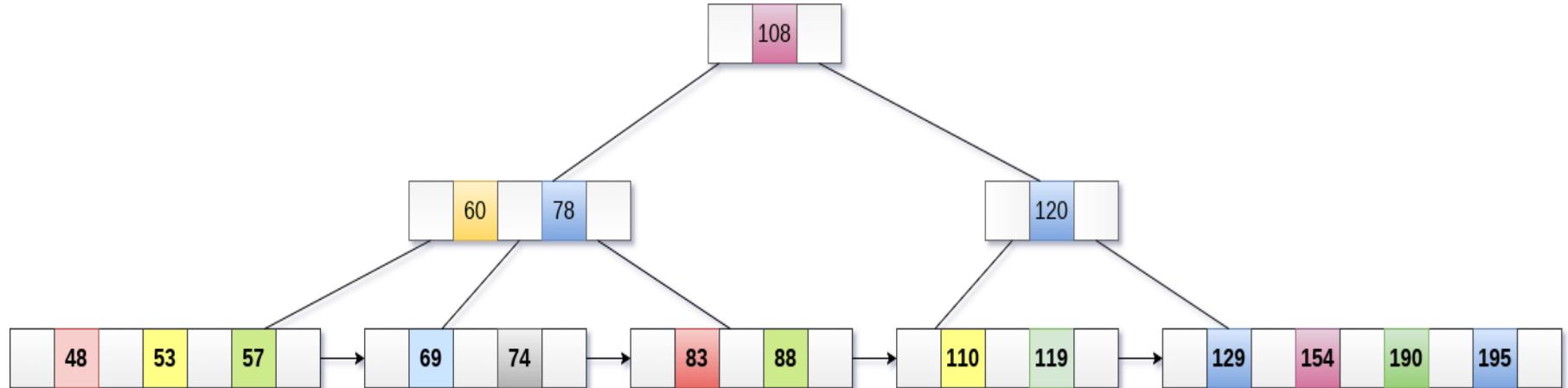
- Example
- Delete the key 200 from the B+ Tree shown in the following figure.



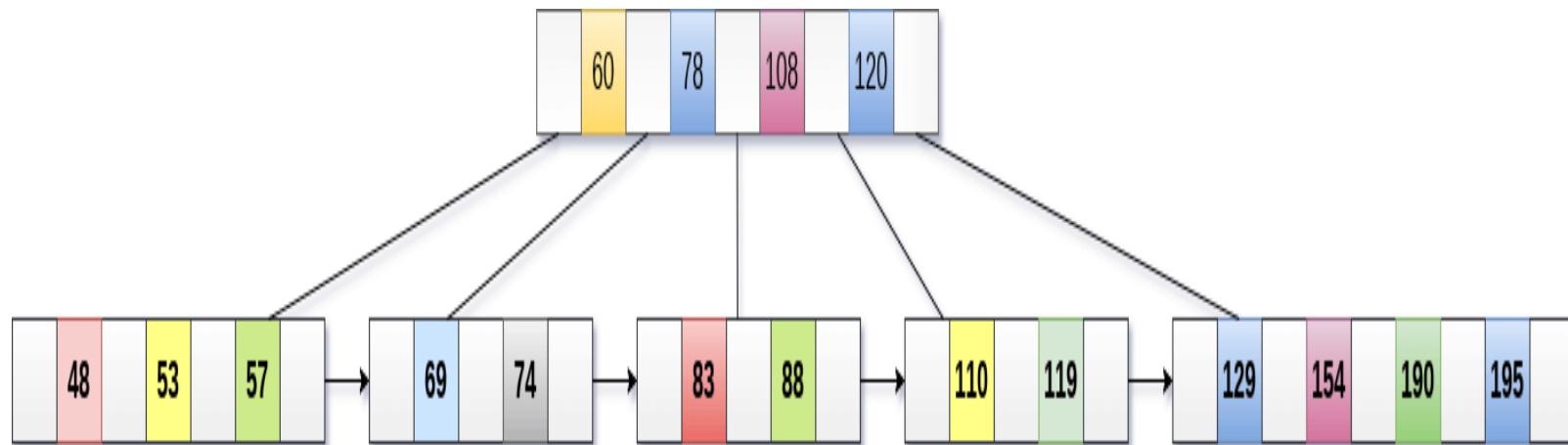
200 is present in the right sub-tree of 190, after 195. delete it.



- Merge the two nodes by using 195, 190, 154 and 129.



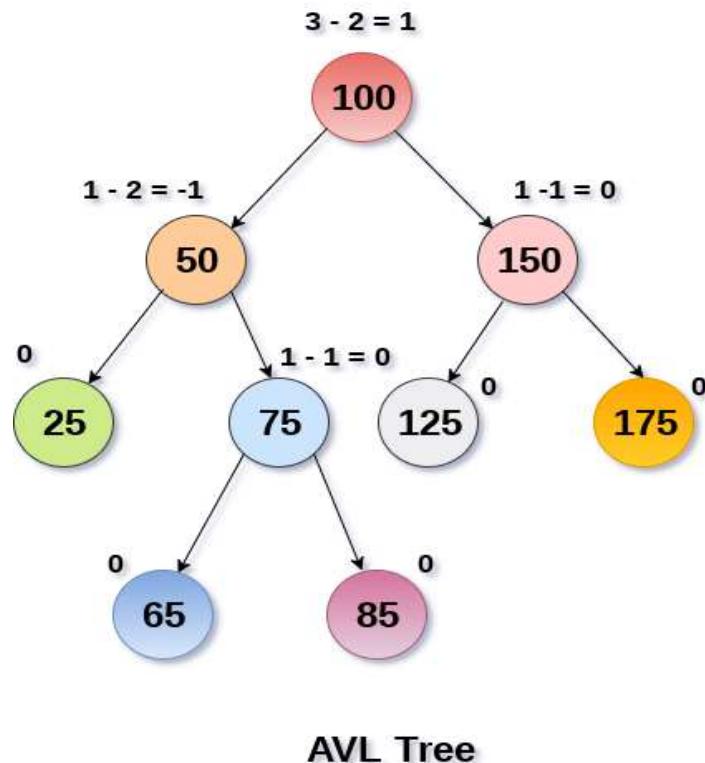
- Now, element 120 is the single element present in the node which is violating the B+ Tree properties.
- Therefore, we need to merge it by using 60, 78, 108 and 120.
- Now, the height of B+ tree will be decreased by 1.



# AVL Tree

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- Balance Factor ( $k$ ) =  $\text{height}(\text{left}(k)) - \text{height}(\text{right}(k))$
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.
- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



## Complexity

Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

# Operations on AVL tree

- Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree.
- Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

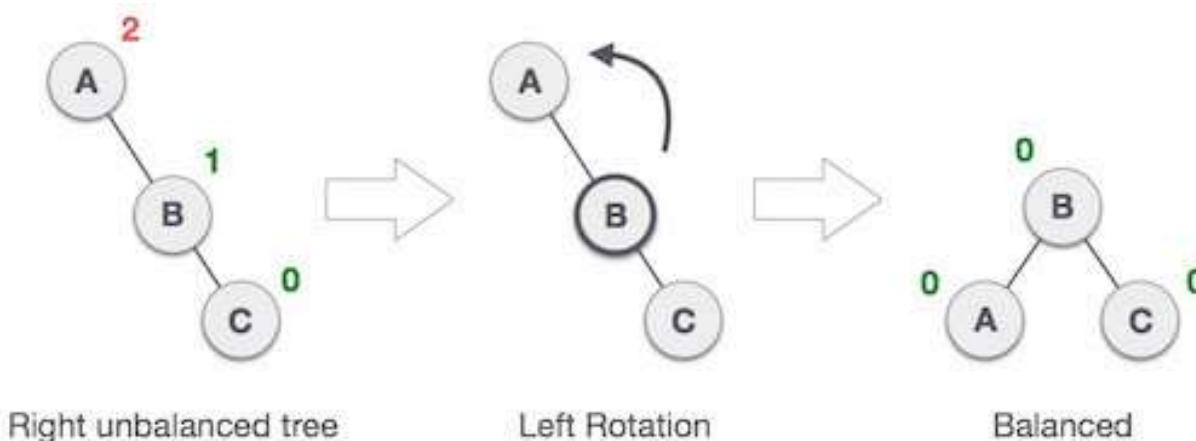
SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

# AVL Rotations

- We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:
- L L rotation: Inserted node is in the left subtree of left subtree of A
- R R rotation : Inserted node is in the right subtree of right subtree of A
- L R rotation : Inserted node is in the right subtree of left subtree of A
- R L rotation : Inserted node is in the left subtree of right subtree of A
- Where node A is the node whose balance Factor is other than -1, 0, 1.
- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

# RR Rotation

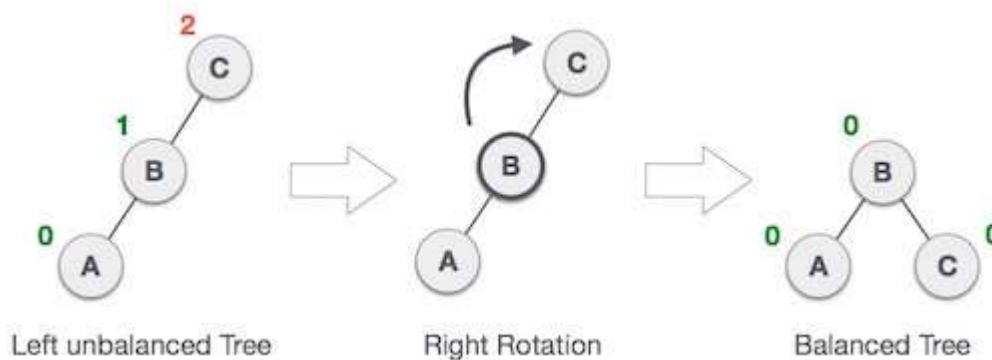
- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

# LL Rotation

- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

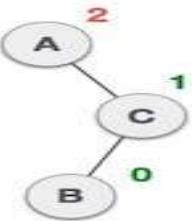
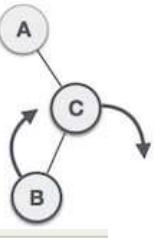
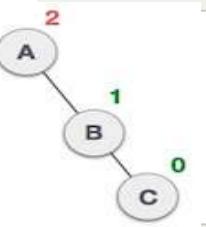
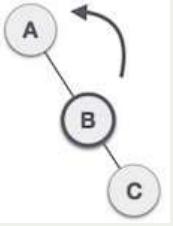
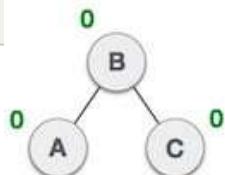
# LR Rotation

- Double rotations are bit tougher than single rotation which has already explained above.  
LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	<p>A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C</p>
	<p>As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.</p>
	<p>After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C</p>
	<p>Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.</p>

# RL Rotation

- double rotations are bit tougher than single rotation which has already explained above. RL rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	 <p>A node <b>B</b> has been inserted into the left subtree of <b>C</b> the right subtree of <b>A</b>, because of which <b>A</b> has become an unbalanced node having balance factor -2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of <b>A</b></p>
	<p>As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at <b>C</b> is performed first. By doing RR rotation, node <b>C</b> has become the right subtree of <b>B</b>.</p>
	<p>After performing LL rotation, node <b>A</b> is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node <b>A</b>.</p>
	<p>Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node <b>A</b>. node <b>C</b> has now become the right subtree of node <b>B</b>, and node <b>A</b> has become the left subtree of <b>B</b>.</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.</p>

# Definition Of B-Tree

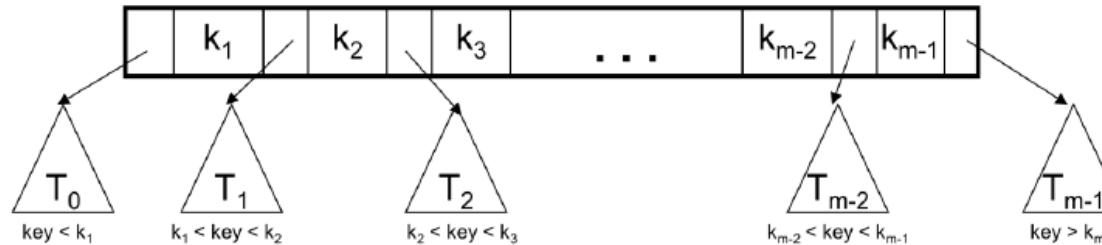
A B-tree of order  $m$  is an  $m$ -way tree (i.e., a tree where each node may have up to  $m$  children) in which:

1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least  $\lceil m / 2 \rceil$  children
  4. the root is either a leaf node, or it has from two to  $m$  children
  5. a leaf node contains no more than  $m - 1$  keys
- The number  $m$  should always be odd

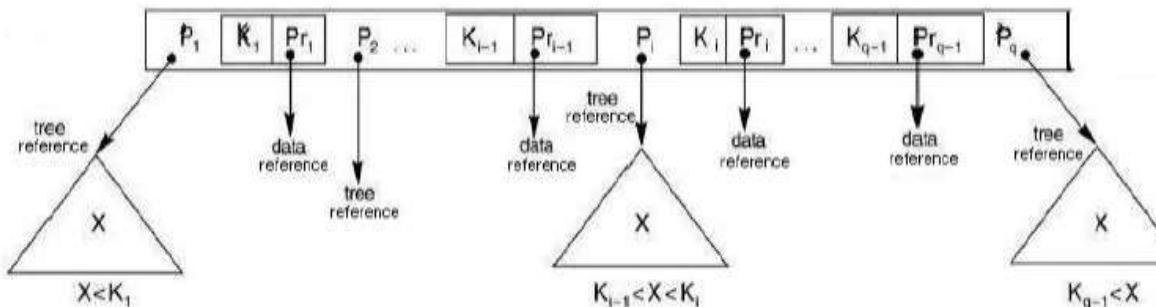
## Multi-way Tree

A multi-way (or m-way) search tree of order m is a tree in which

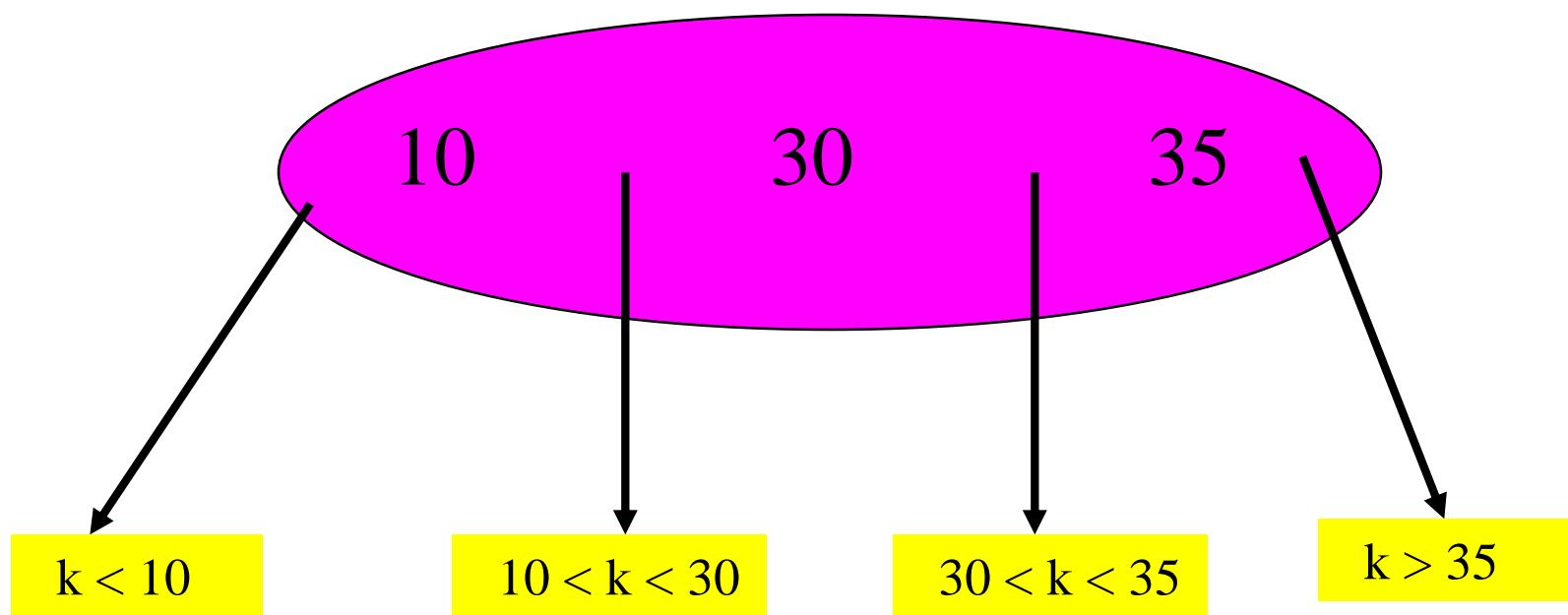
- Each node has at-most m subtrees, where the subtrees may be empty.
- Each node consists of at least 1 and at most m-1 distinct keys
- The keys in each node are sorted.



- The keys and subtrees of a non-leaf node are ordered as:
  - $T_0, k_1, T_1, k_2, T_2, \dots, k_{m-1}, T_{m-1}$  such that:
- All keys in subtree  $T_0$  are less than  $k_1$ .
- All keys in subtree  $T_i$ ,  $1 \leq i \leq m - 2$ , are greater than  $k_i$  but less than  $k_{i+1}$ .
- All keys in subtree  $T_{m-1}$  are greater than  $k_{m-1}$ .



# 4-Way Search Tree



# Definition Of B-Tree

- Definition assumes external nodes (extended **m**-way search tree).
- B-tree of order **m**.
  - **m**-way search tree.
  - Not empty => root has at least **2** children.
  - Remaining internal nodes (if any) have at least  **$\text{ceil}(m/2)$**  children.
  - External (or failure) nodes on same level.

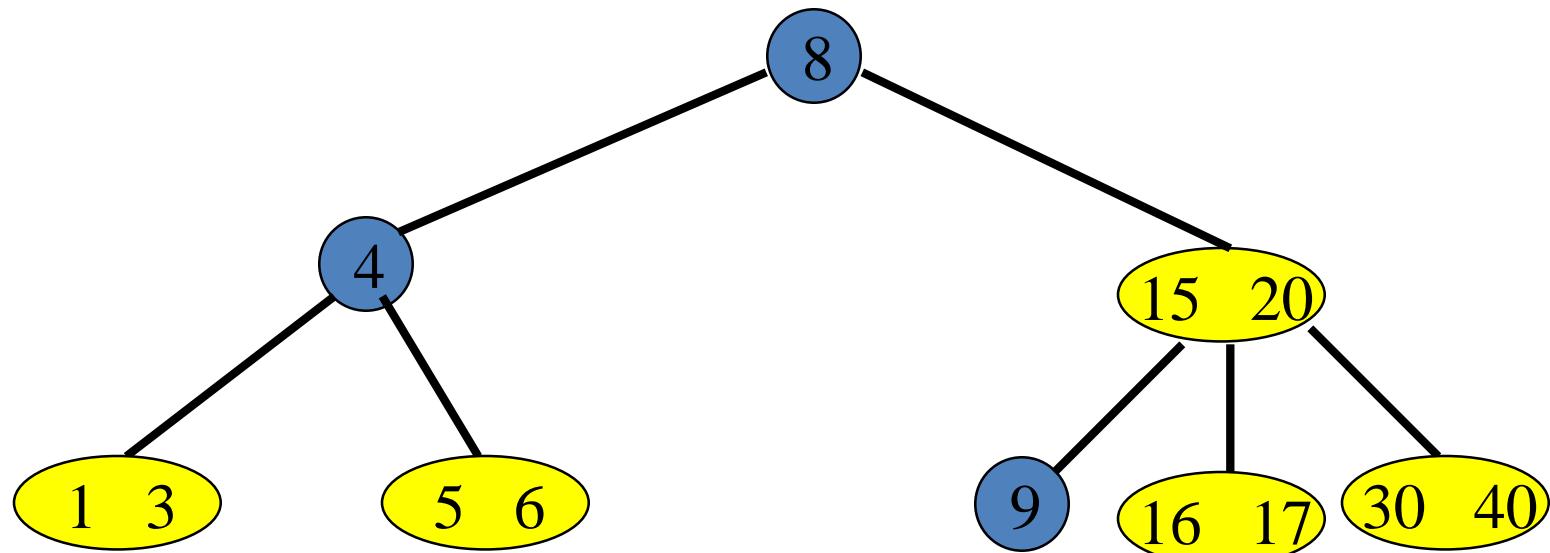
# 2-3 And 2-3-4 Trees

- B-tree of order  $m$ .
  - $m$ -way search tree.
  - Not empty => root has at least  $2$  children.
  - Remaining internal nodes (if any) have at least  $\text{ceil}(m/2)$  children.
  - External (or failure) nodes on same level.
- 2-3 tree is B-tree of order  $3$ .
- 2-3-4 tree is B-tree of order  $4$ .

# B-Trees Of Order 5 And 2

- B-tree of order  $m$ .
  - $m$ -way search tree.
  - Not empty => root has at least  $2$  children.
  - Remaining internal nodes (if any) have at least  $\text{ceil}(m/2)$  children.
  - External (or failure) nodes on same level.
- B-tree of order  $5$  is  $3\text{-}4\text{-}5$  tree (root may be  $2$ -node though).
- B-tree of order  $2$  is full binary tree.

# Insert



Insertion into a full leaf triggers bottom-up node splitting pass.

# Split An Overfull Node

$m \ a_0 \ p_1 \ a_1 \ p_2 \ a_2 \dots p_m \ a_m$

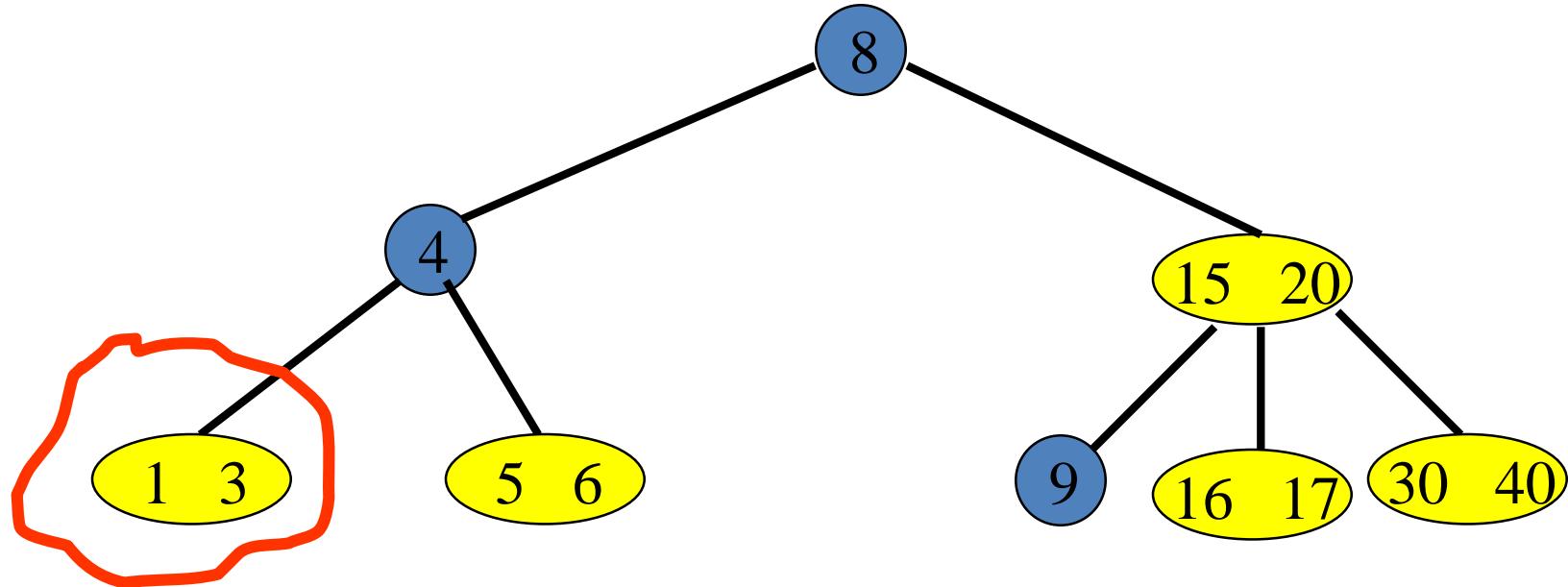
- $a_i$  is a pointer to a subtree.
- $p_i$  is a dictionary pair.

$\text{ceil}(m/2)-1 \ a_0 \ p_1 \ a_1 \ p_2 \ a_2 \dots p_{\text{ceil}(m/2)-1} \ a_{\text{ceil}(m/2)-1}$

$m - \text{ceil}(m/2) \ a_{\text{ceil}(m/2)} \ p_{\text{ceil}(m/2)+1} \ a_{\text{ceil}(m/2)+1} \dots p_m \ a_m$

- $p_{\text{ceil}(m/2)}$  plus pointer to new node is inserted in parent.

# Insert



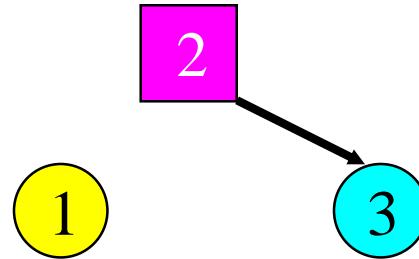
- Insert a pair with key = 2.
- New pair goes into a 3-node.

# Insert Into A Leaf 3-node

- Insert new pair so that the **3** keys are in ascending order.

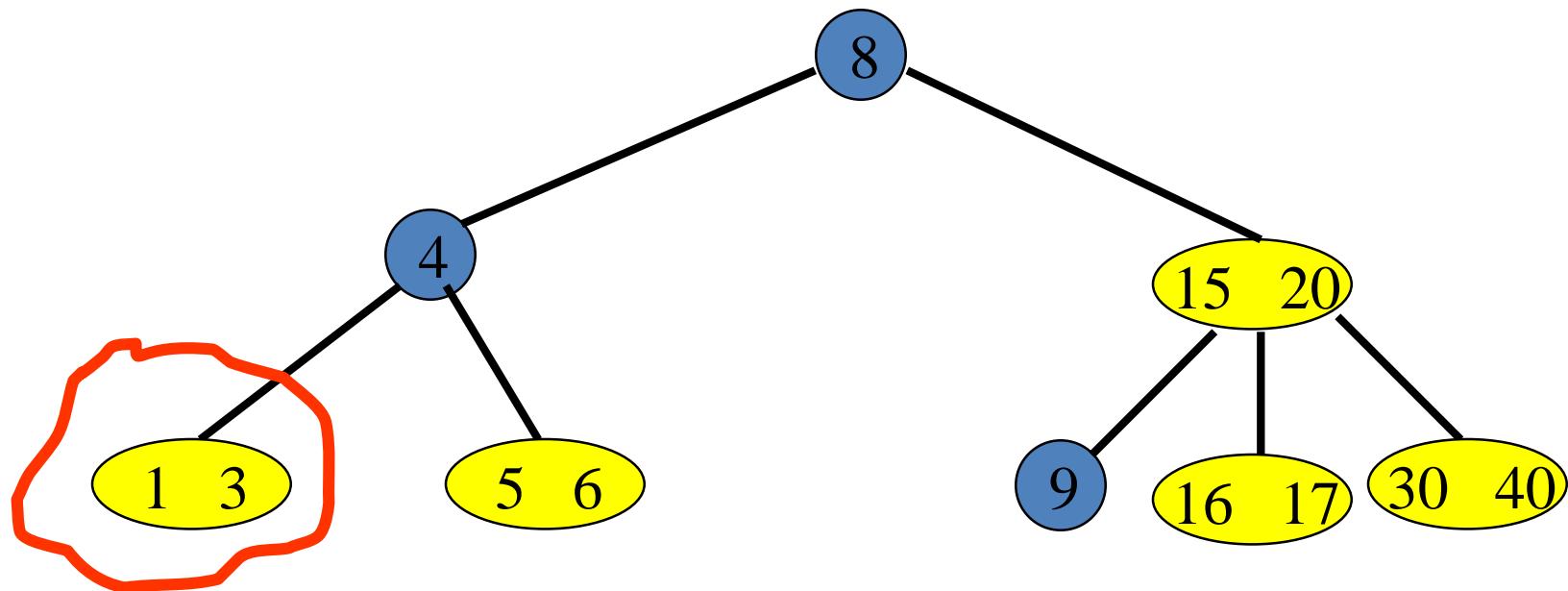


- Split overflowed node around middle key.



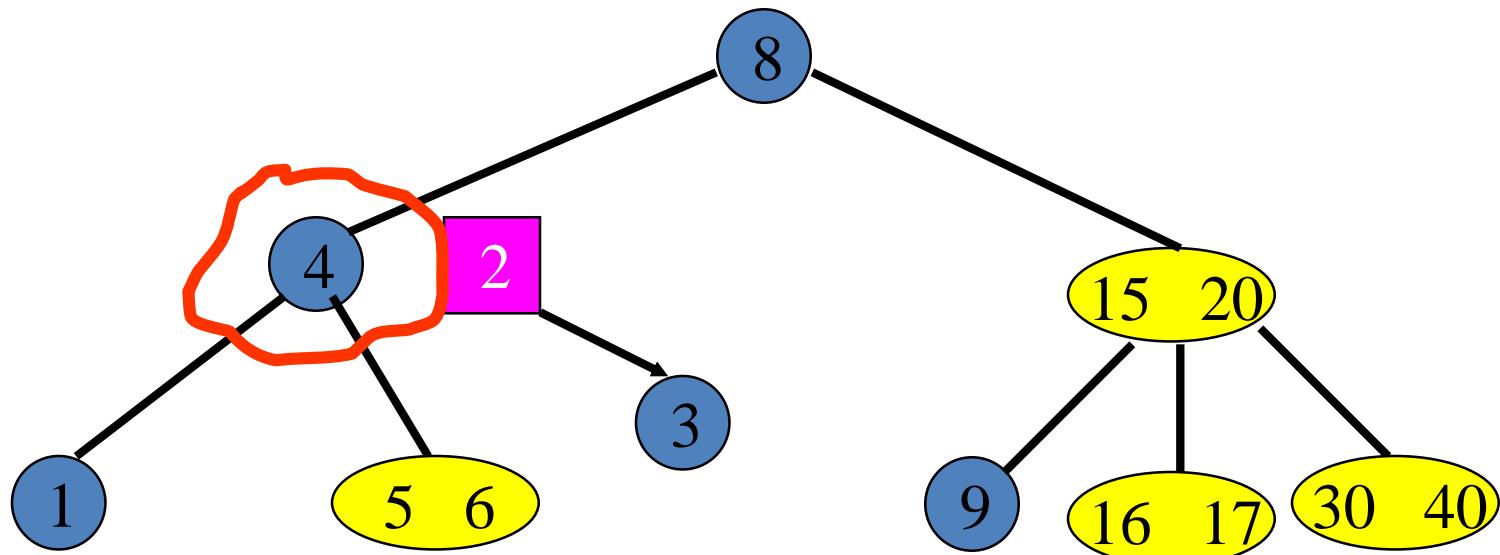
- Insert middle key and pointer to new node into parent.

# Insert



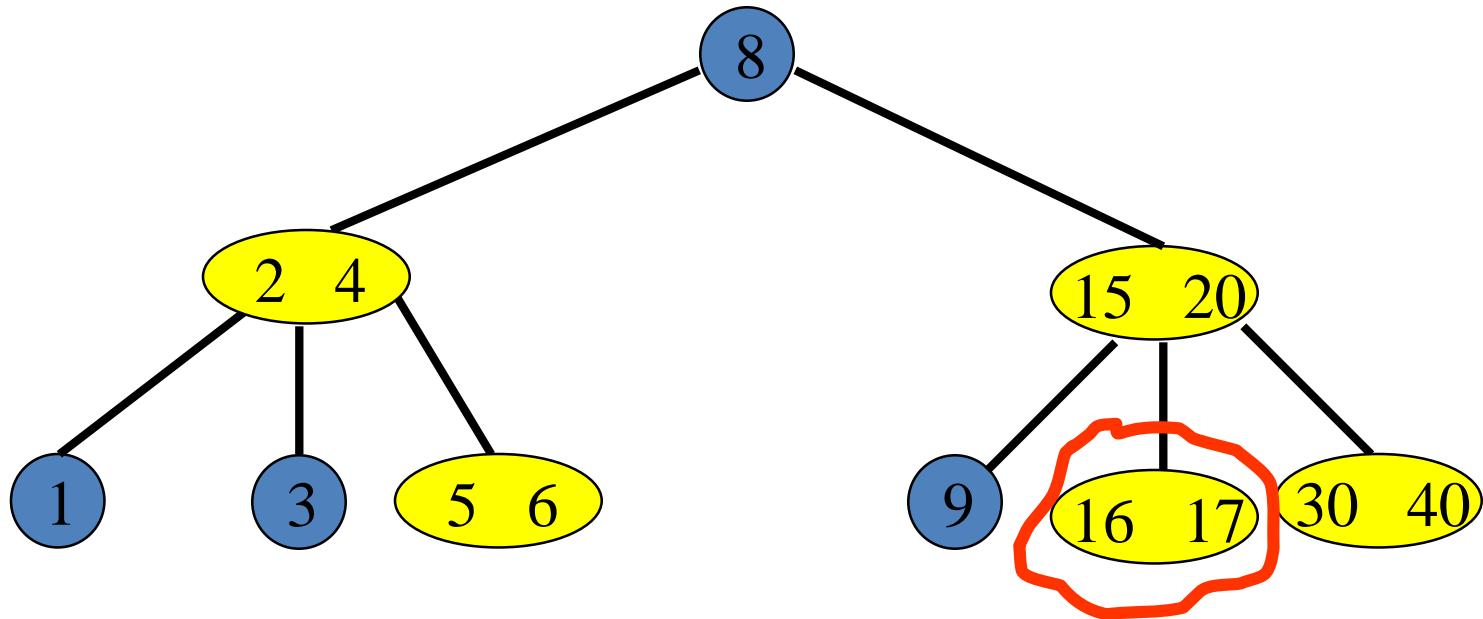
- Insert a pair with key = 2.

# Insert



- Insert a pair with key = 2 plus a pointer into parent.

# Insert



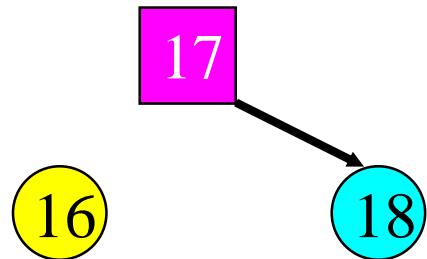
- Now, insert a pair with key = 18.

# Insert Into A Leaf 3-node

- Insert new pair so that the **3** keys are in ascending order.

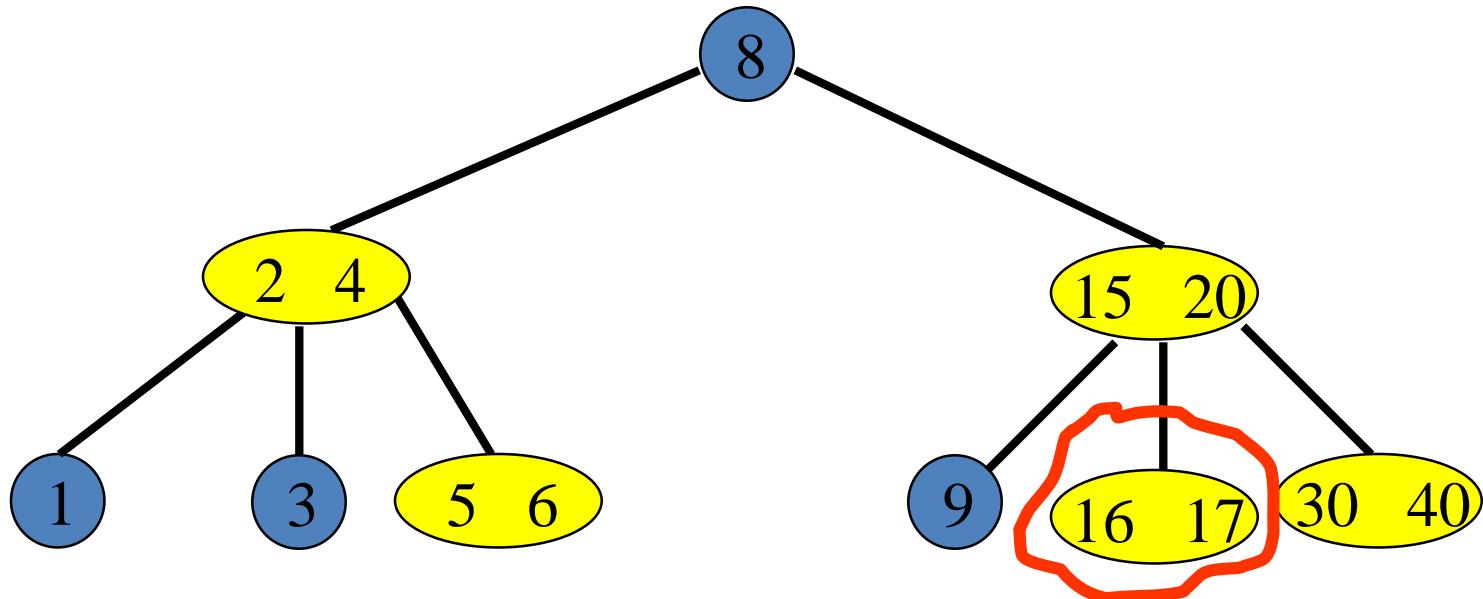


- Split the overflowed node.



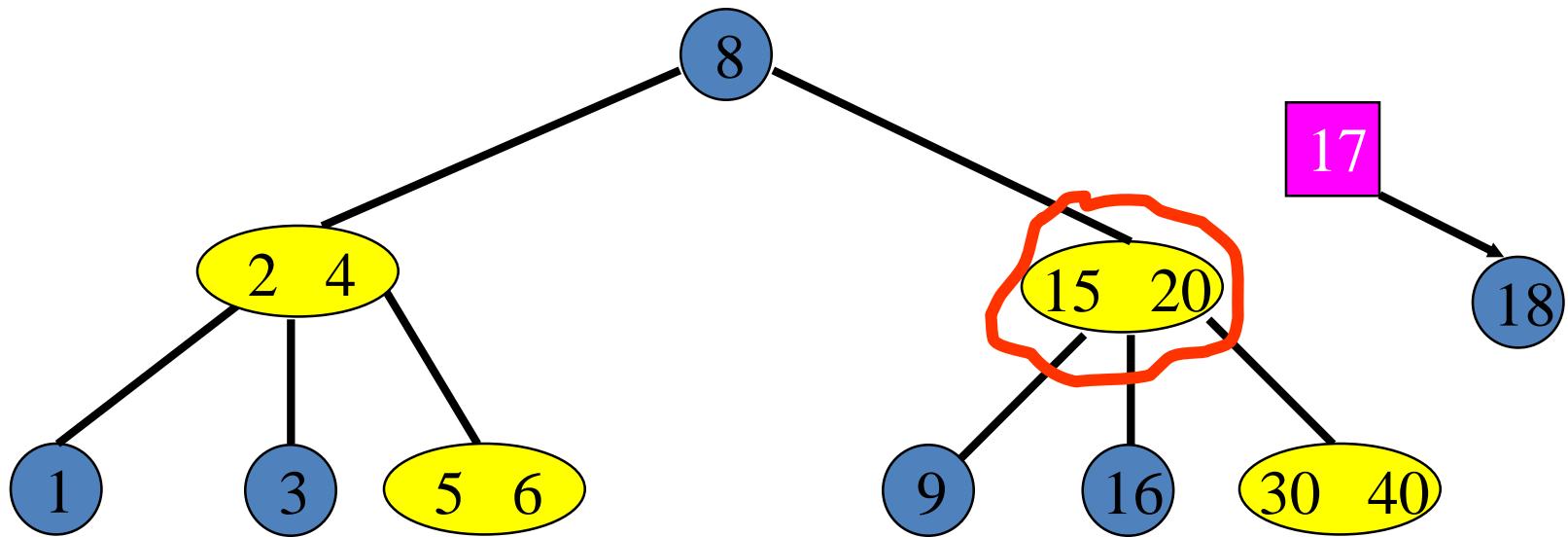
- Insert middle key and pointer to new node into parent.

# Insert



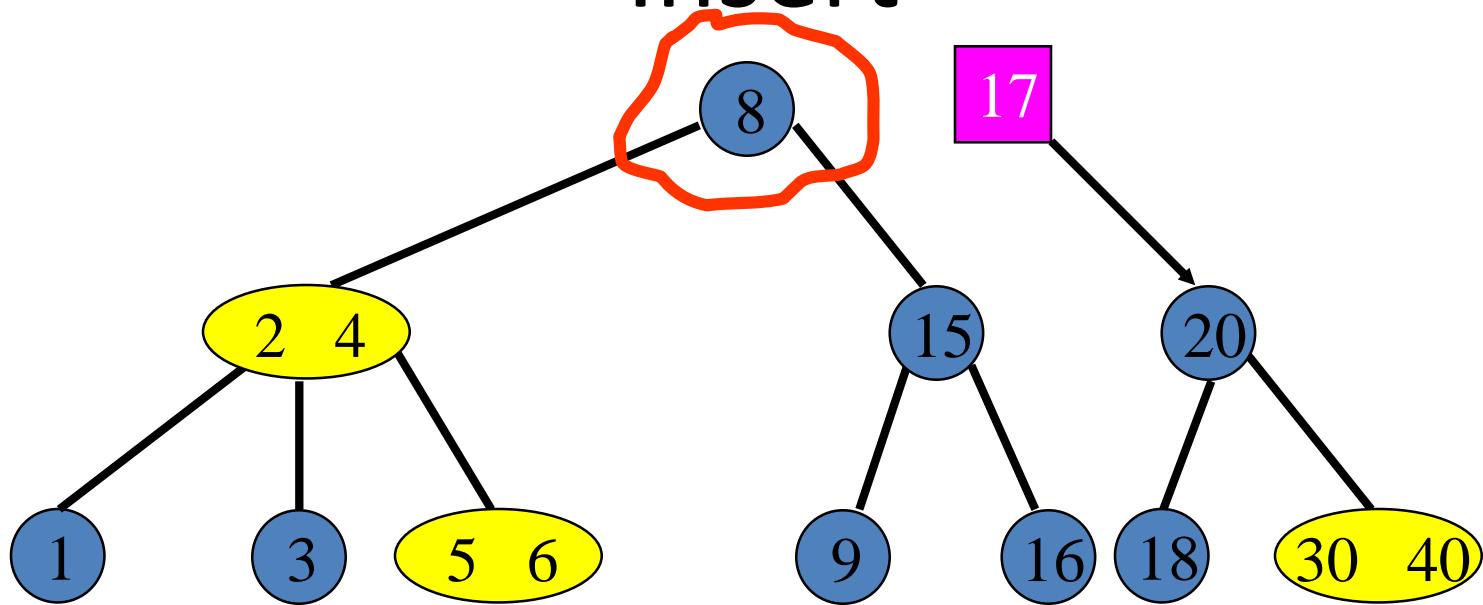
- Insert a pair with key = 18.

# Insert



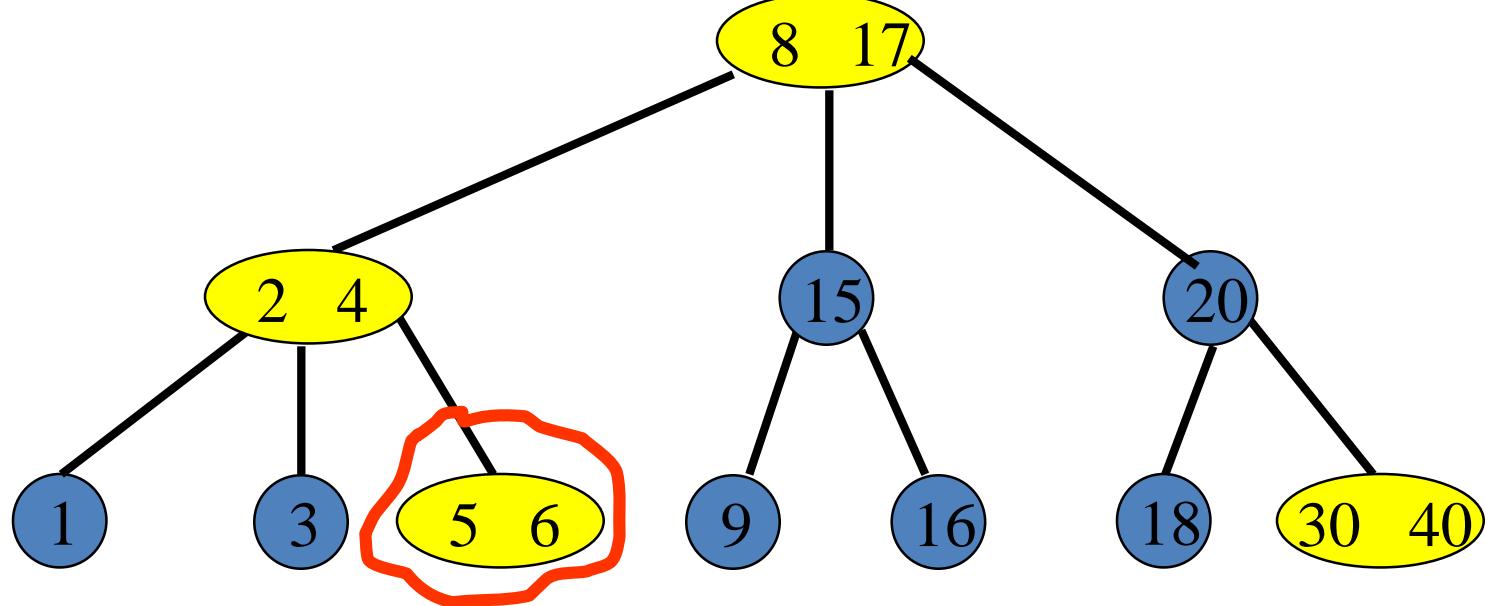
- Insert a pair with key = 17 plus a pointer into parent.

# Insert



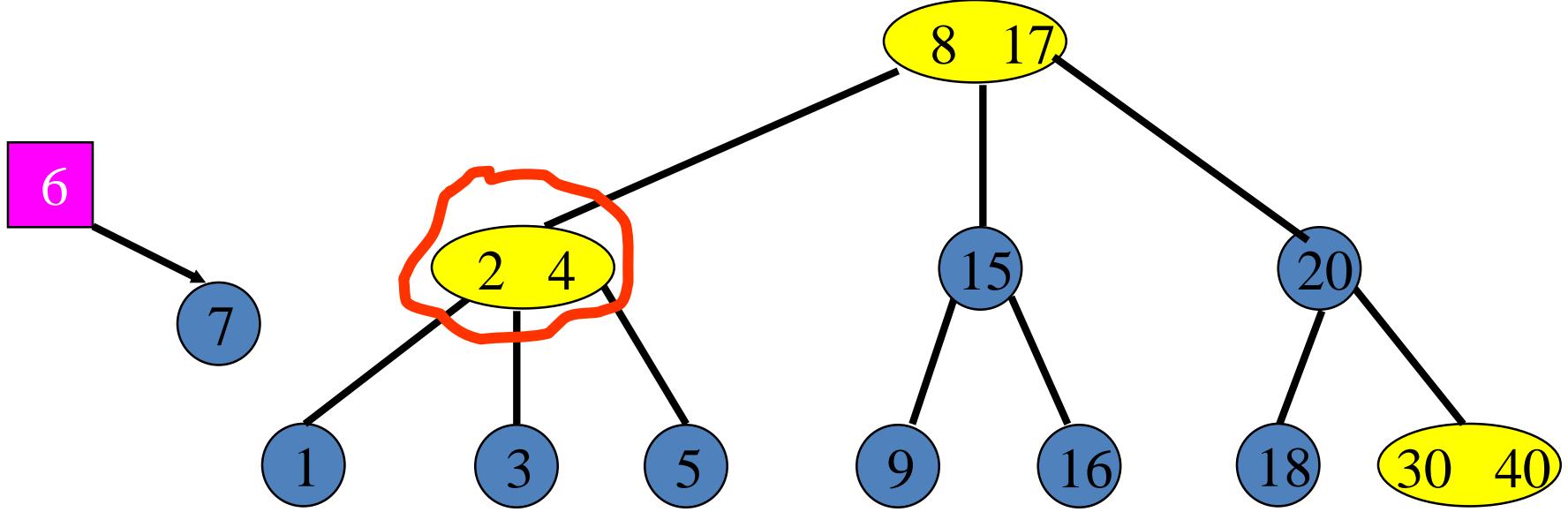
- Insert a pair with key = 17 plus a pointer into parent.

# Insert



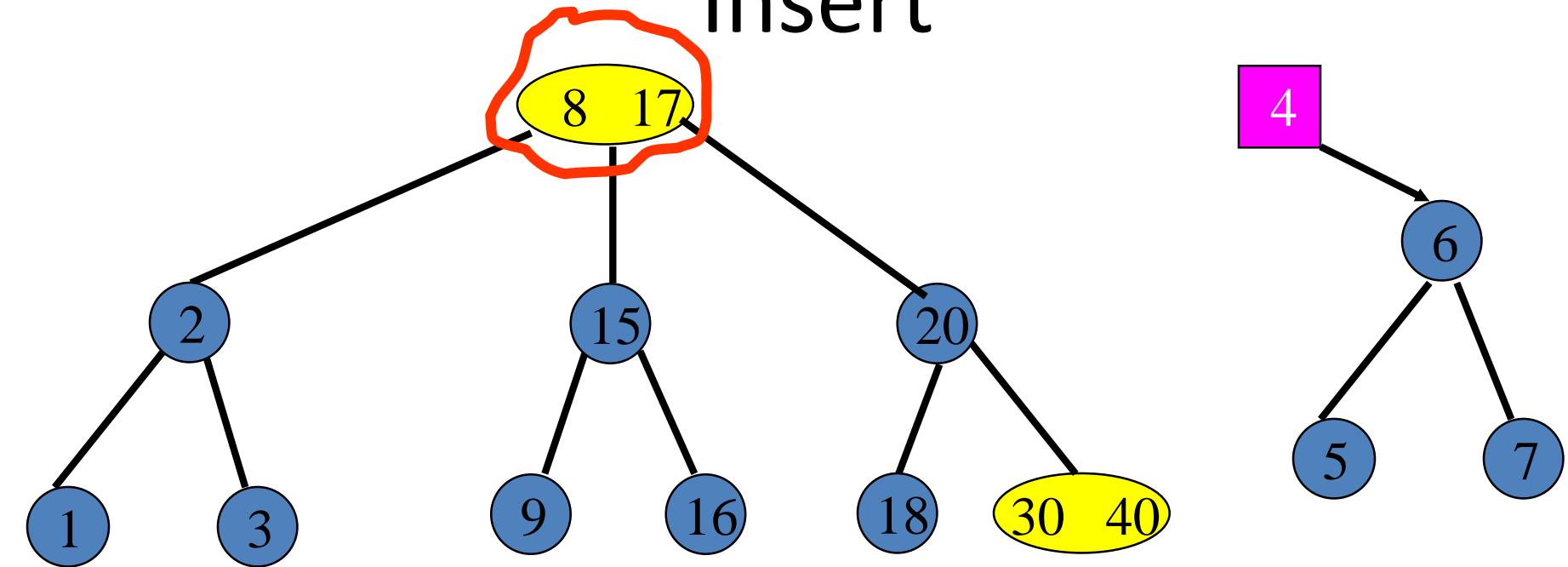
- Now, insert a pair with key = 7.

# Insert



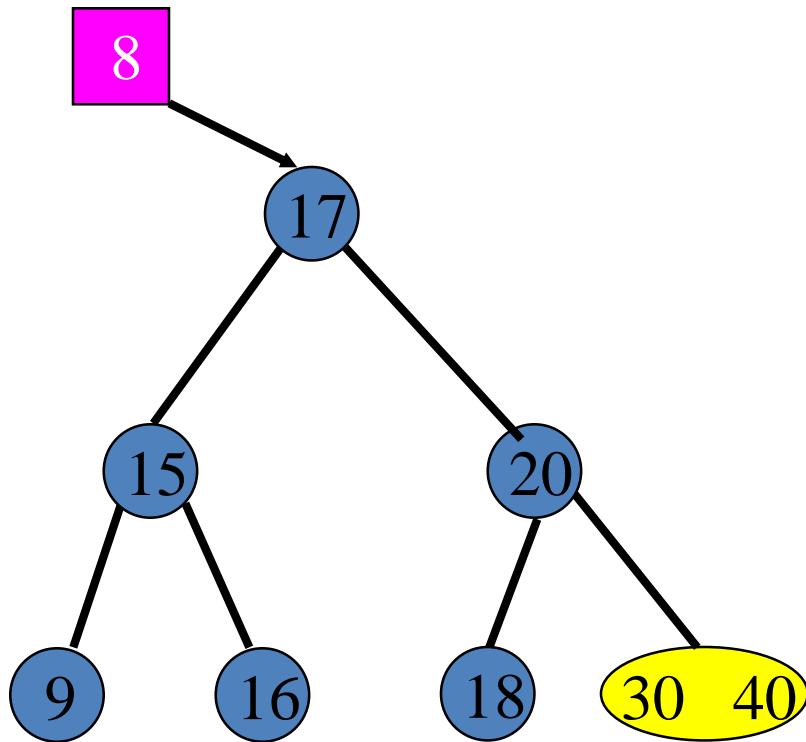
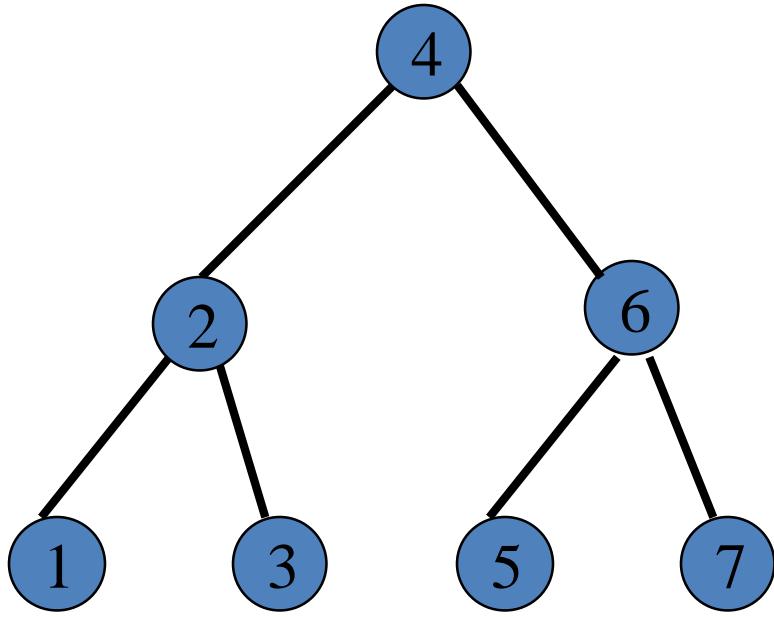
- Insert a pair with key = 6 plus a pointer into parent.

# Insert



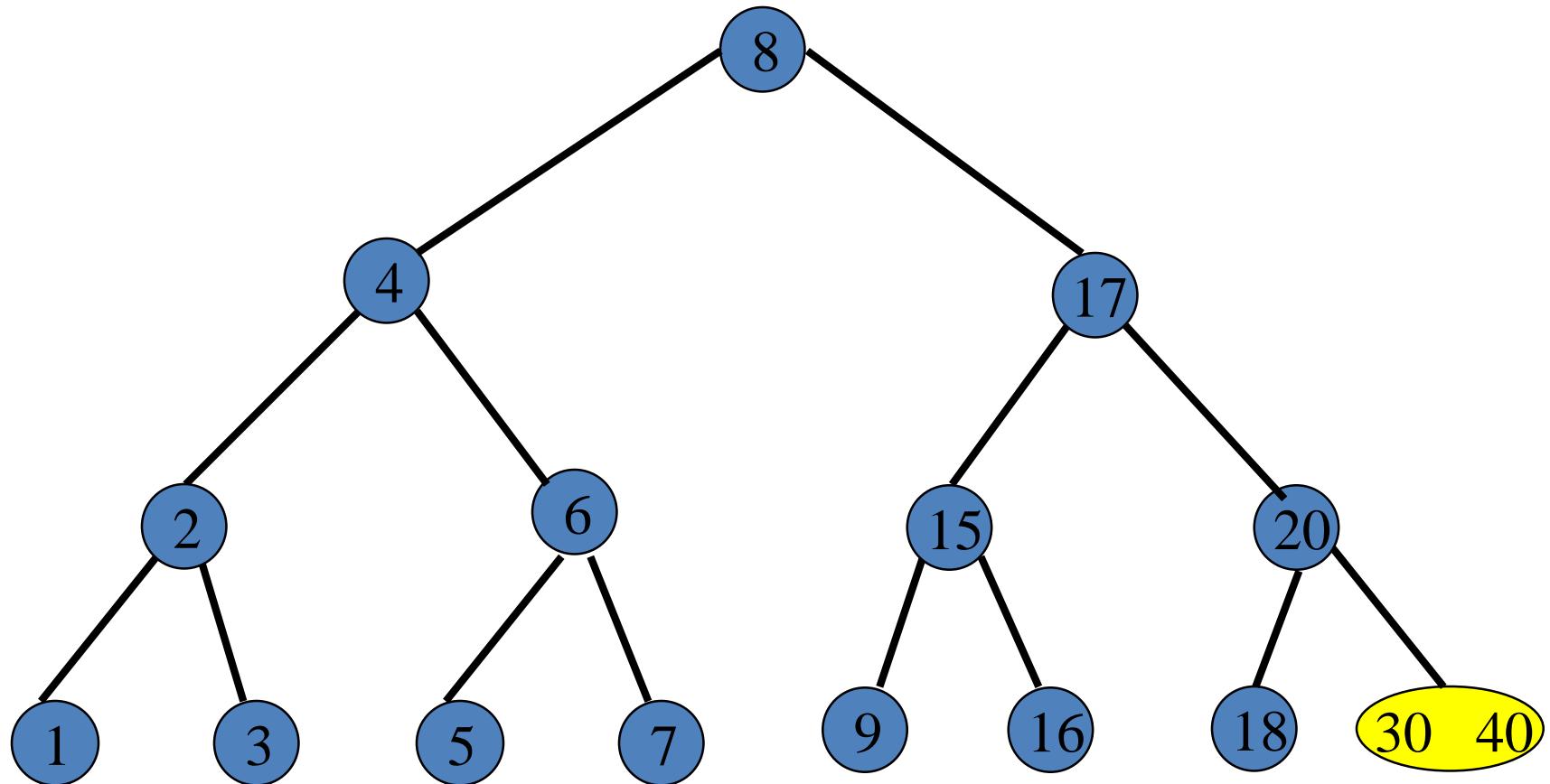
- Insert a pair with key = 4 plus a pointer into parent.

# Insert



- Insert a pair with key = 8 plus a pointer into parent.
- There is no parent. So, create a new root.

# Insert



- Height increases by 1.

## Inserting in a B-Tree

- Find the node where the item is to be inserted by following the search procedure.
- If the node is not full, insert the item into the node in order.
- If the node is full, it has to be split.

## Example of Insertion

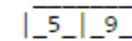
Let's look at an example of inserting into a B-tree. For preservation of sanity, let  $t = 2$ . So a node is full if it has  $2(2)-1 = 3$  keys in it, and each node

Step 1: Insert 5



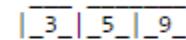
Step 2: Insert 9

B-Tree-Insert simply calls B-Tree-Insert-Nonfull, putting 9 to the right of 5:



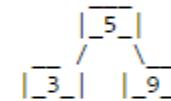
Step 3: Insert 3

Again, B-Tree-Insert-Nonfull is called

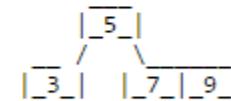


Step 4: Insert 7

Tree is full. We allocate a new (empty) node, make it the root, split the former root, then pull 5 into the new root:

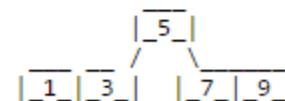


Then insert we insert 7; it goes in with 9



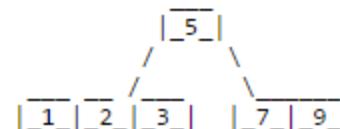
Step 5: Insert 1

It goes in with 3

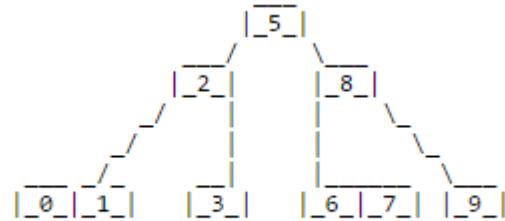
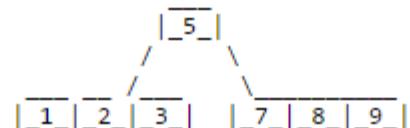


Step 6: Insert 2

It goes in with 3

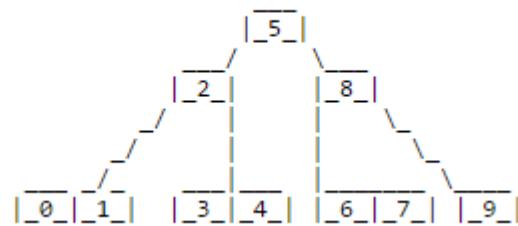
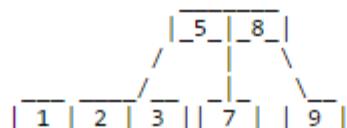


Step 7: Insert 8  
It goes in with 9

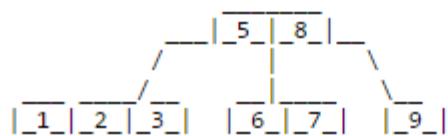


Now we can insert 4, assured that future insertions will work

Step 8: Insert 6  
It would go in with |7|8|9|, but that node is full. So we split it bringing its middle child into the root:

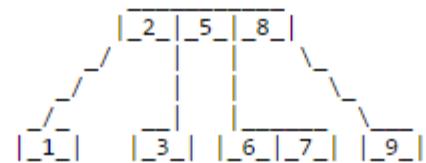


Then insert 6, which goes in with 7:

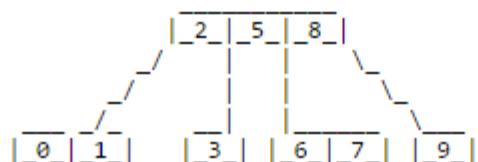


Step 9: Insert 0

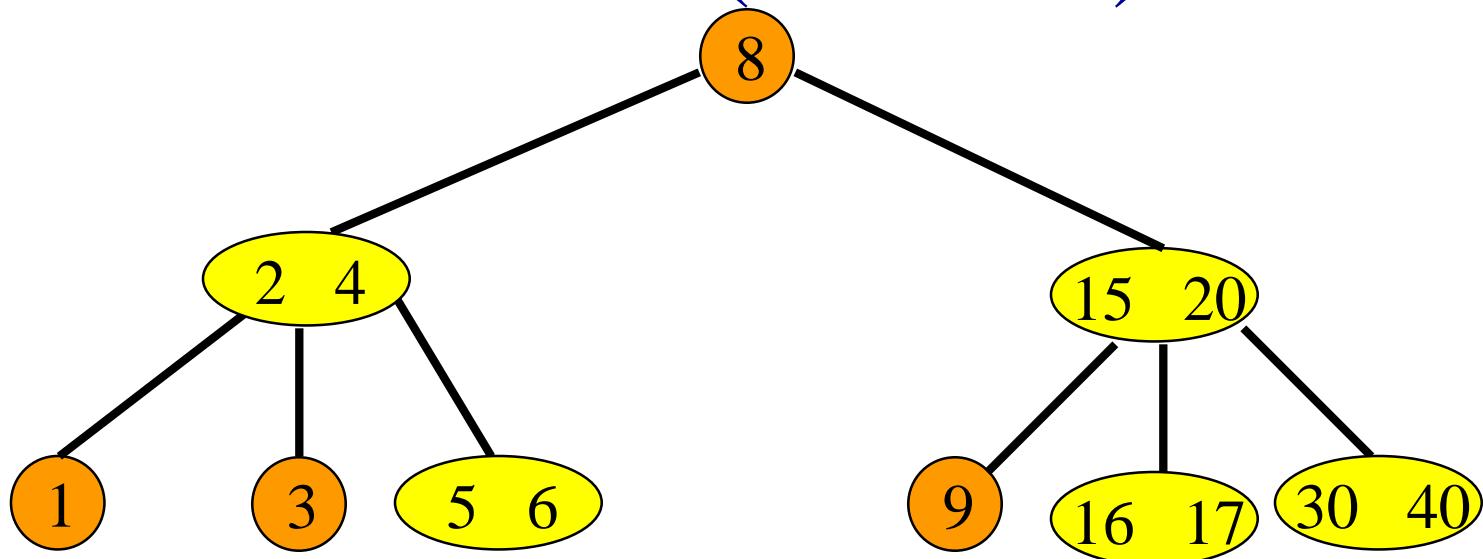
0 would go in with |1|2|3|, which is full, so we split it, sending the middle child up to the root:



Now we can put 0 in with 1

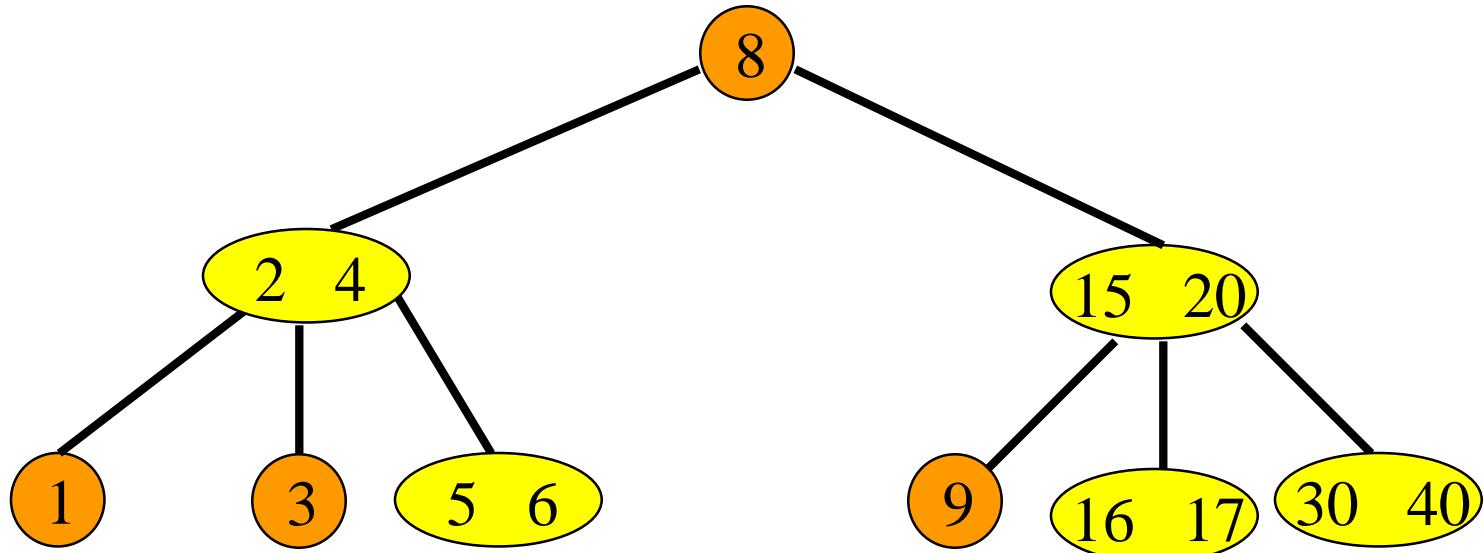


# Delete (2-3 tree)



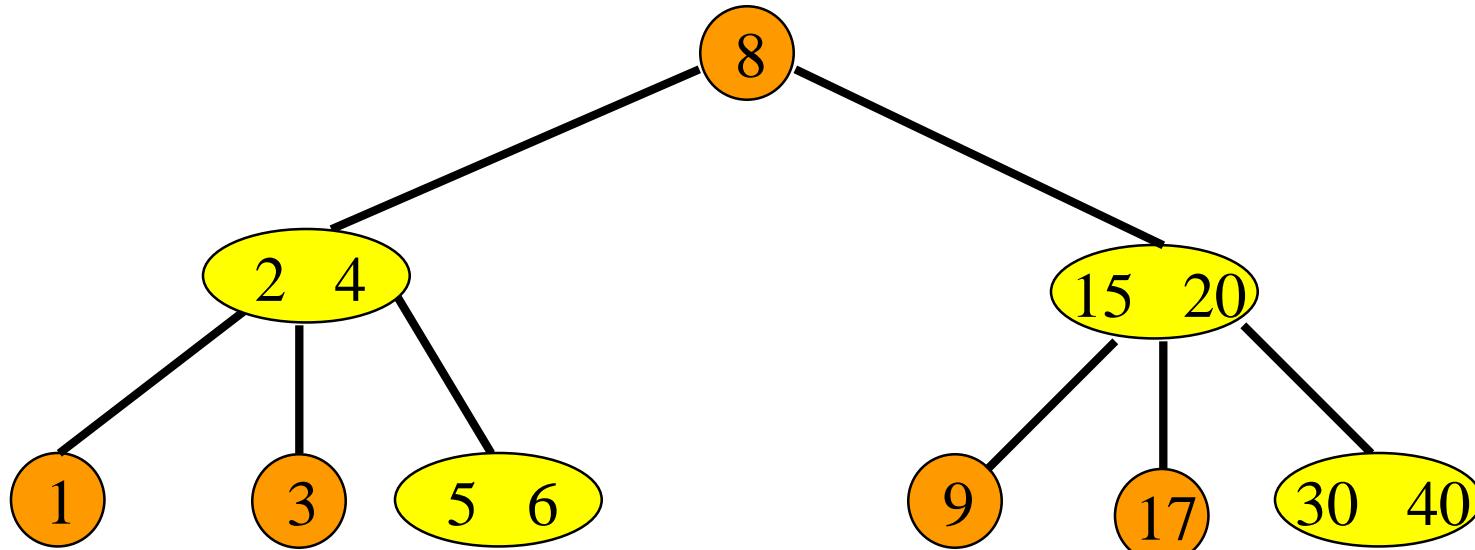
- Delete the pair with key = 8.
- Transform deletion from interior into deletion from a leaf.
- Replace by largest in left subtree.

# Delete From A Leaf



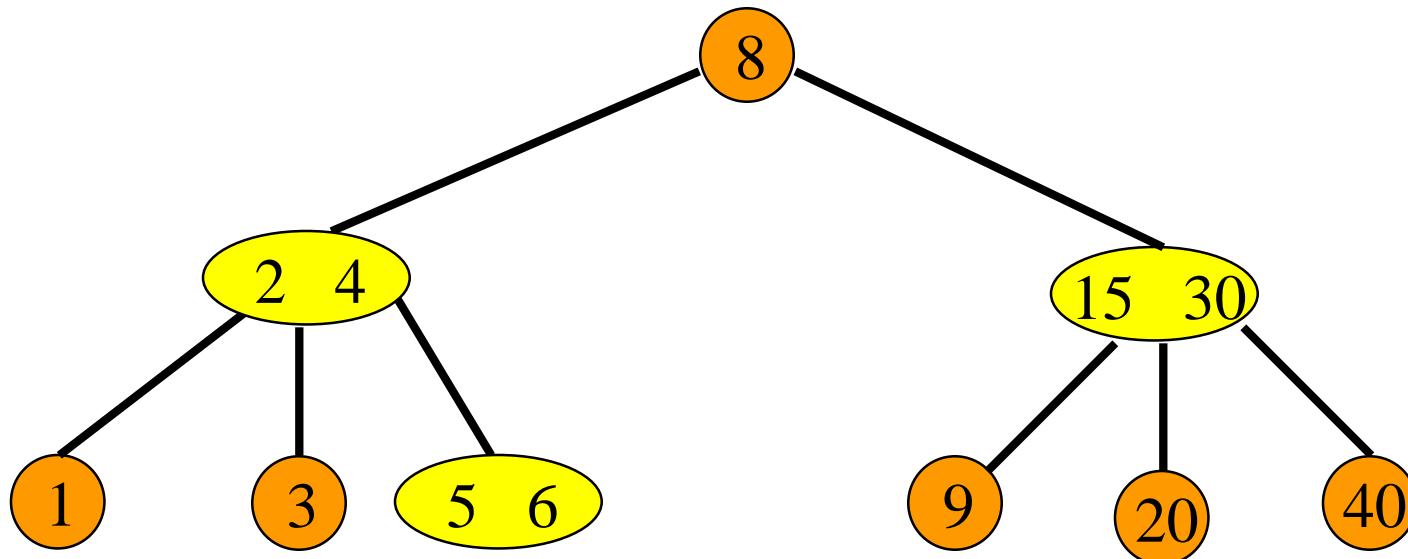
- Delete the pair with key = 16.
- 3-node becomes 2-node.

# Delete From A Leaf



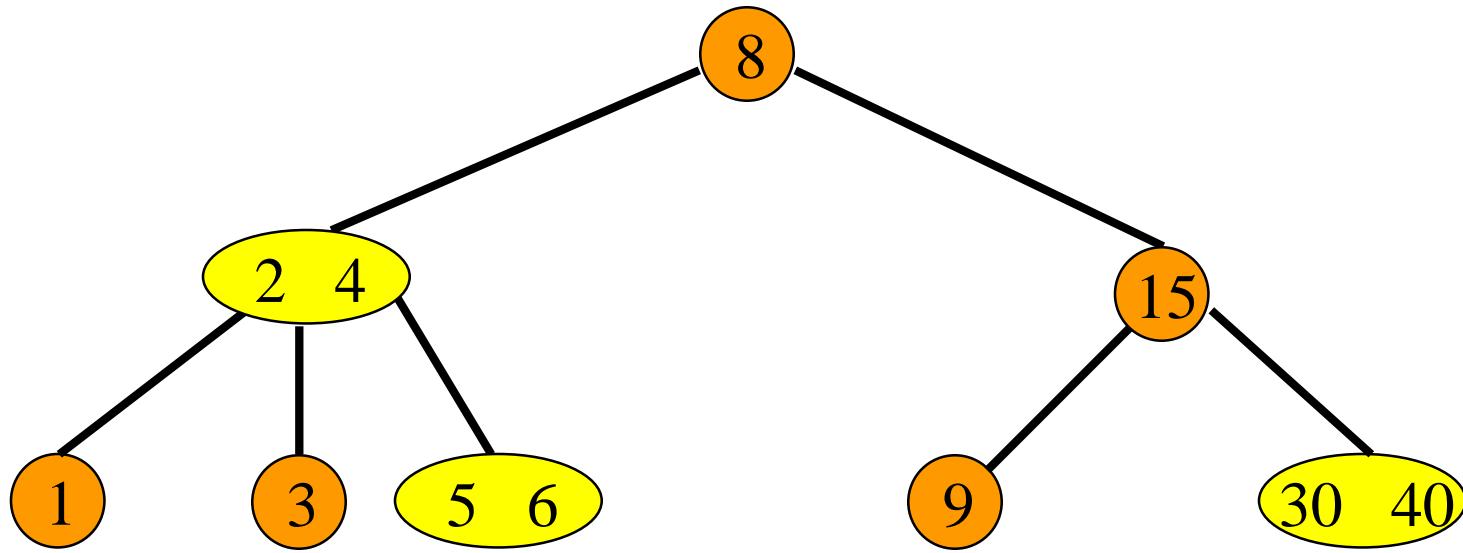
- Delete the pair with key = 17.
- Deletion from a 2-node.
- Check an adjacent sibling and determine if it is a 3-node.
- If so borrow a pair and a subtree via parent node.

# Delete From A Leaf



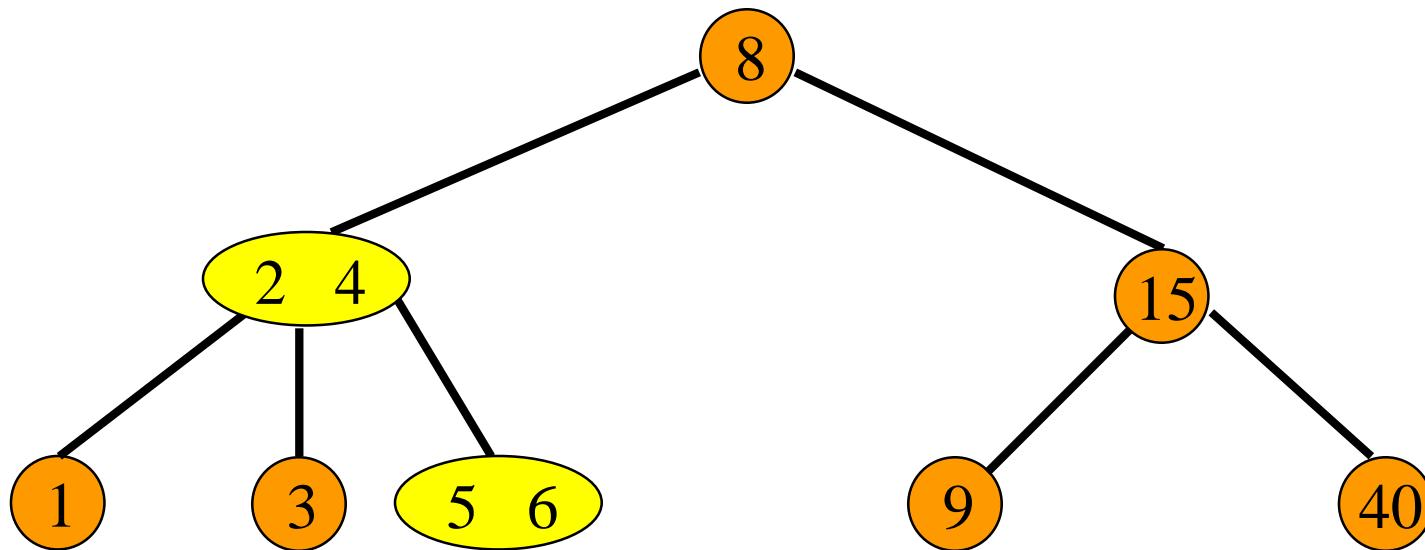
- Delete the pair with key = 20.
- Deletion from a 2-node.
- Check an adjacent sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

# Delete From A Leaf



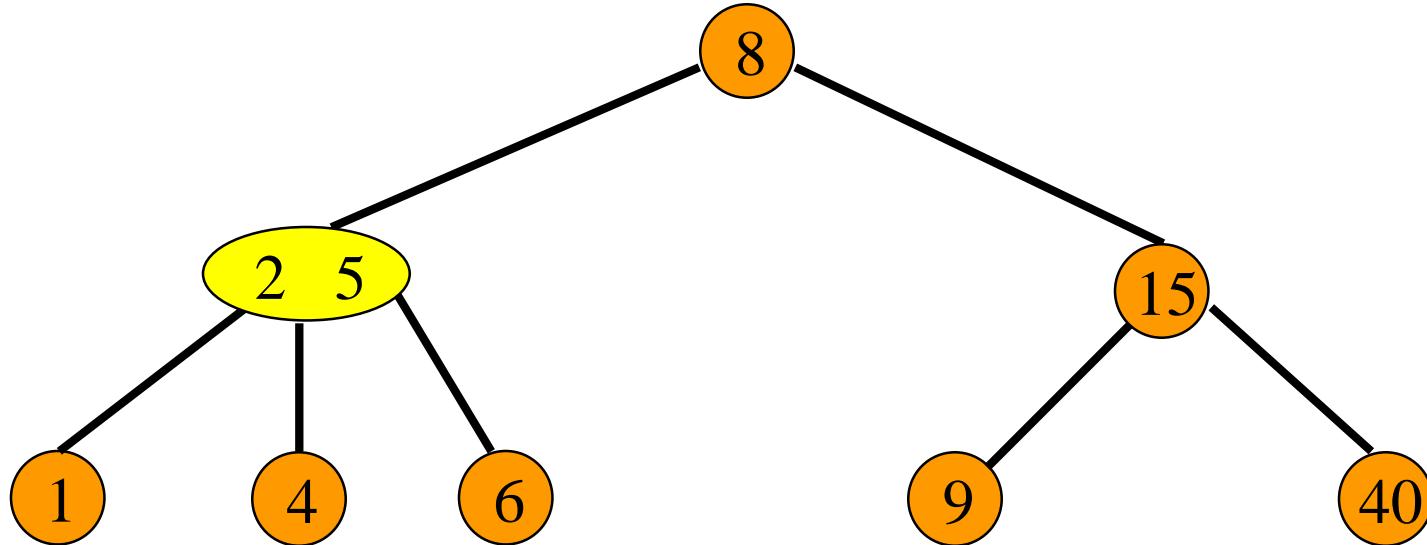
- Delete the pair with key = 30.
- Deletion from a 3-node.
- 3-node becomes 2-node.

# Delete From A Leaf



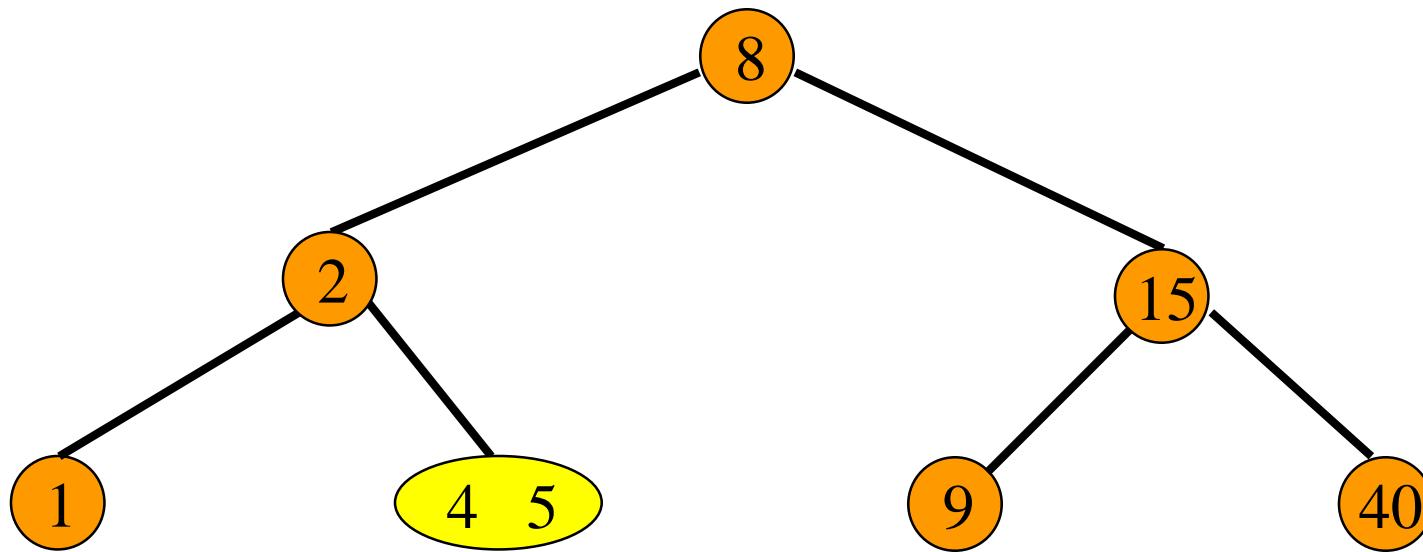
- Delete the pair with key = 3.
- Deletion from a 2-node.
- Check an adjacent sibling and determine if it is a 3-node.
- If so borrow a pair and a subtree via parent node.

# Delete From A Leaf



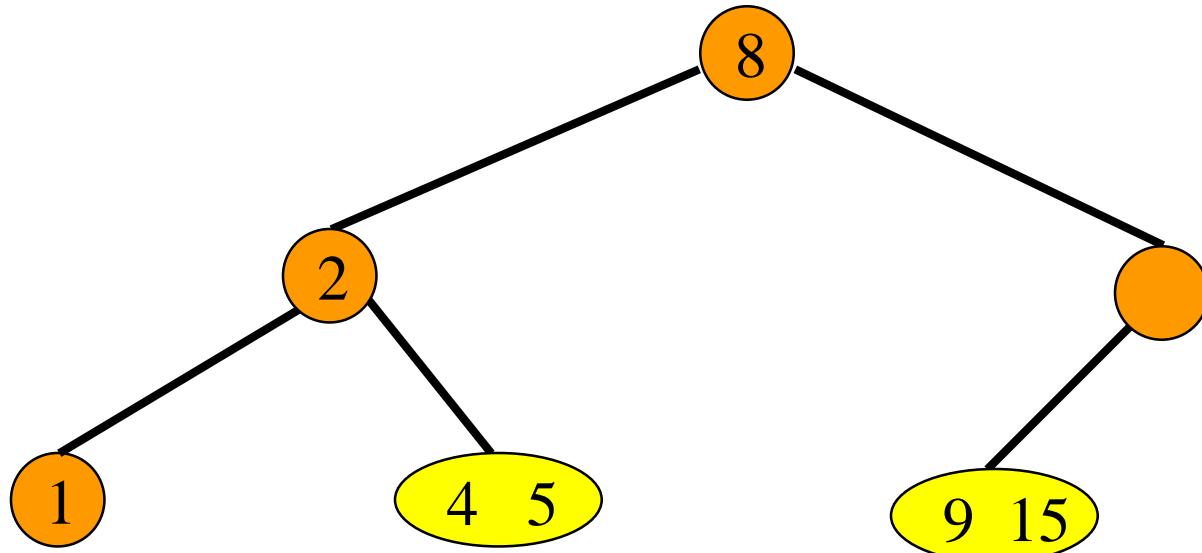
- Delete the pair with key = 6.
- Deletion from a 2-node.
- Check an adjacent sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

# Delete From A Leaf



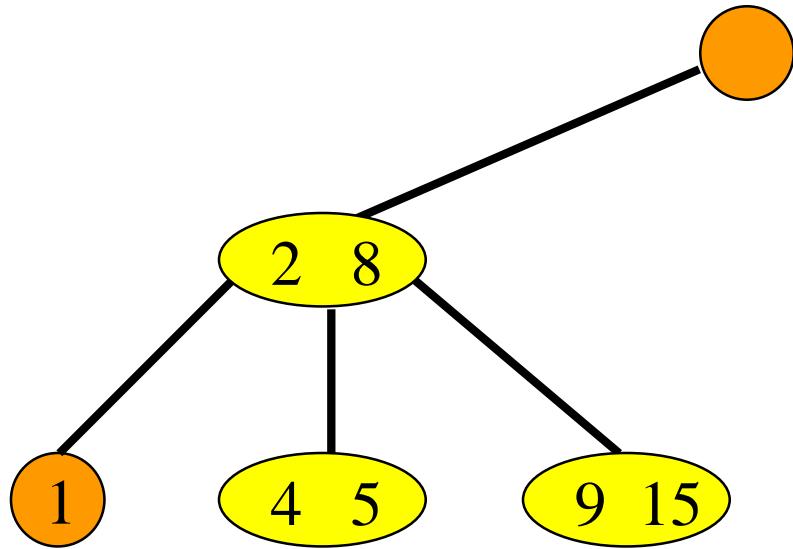
- Delete the pair with key = 40.
- Deletion from a 2-node.
- Check an adjacent sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

# Delete From A Leaf



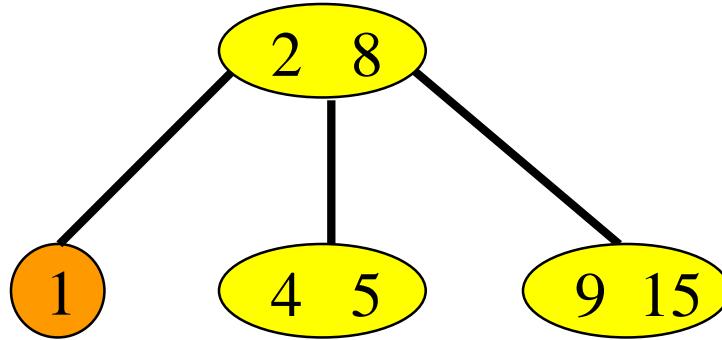
- Parent pair was from a **2-node**.
- Check an adjacent sibling and determine if it is a **3-node**.
- If not, combine with sibling and parent pair.

# Delete From A Leaf



- Parent pair was from a **2**-node.
- Check an adjacent sibling and determine if it is a **3**-node.
- No sibling, so must be the root.
- Discard root. Left child becomes new root.

# Delete From A Leaf

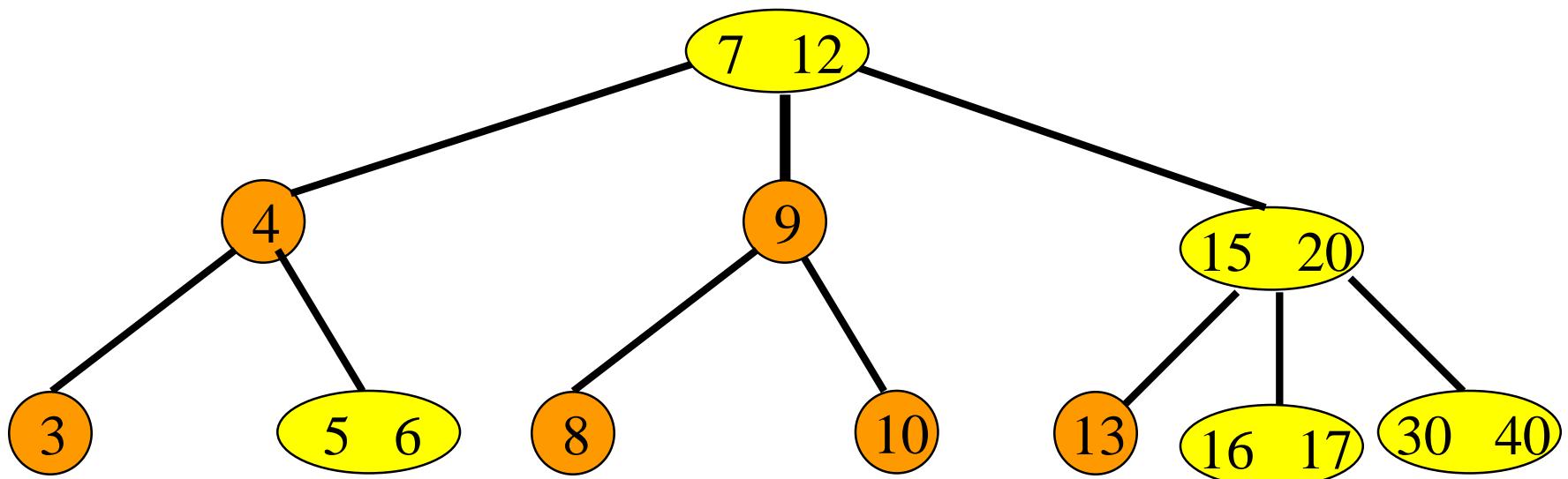


- Height reduces by 1.

# Delete A Pair

- Deletion from interior node is transformed into a deletion from a leaf node.
- Deficient leaf triggers bottom-up borrowing and node combining pass.
- Deficient node is combined with an adjacent sibling who has exactly  $\text{ceil}(m/2) - 1$  pairs.
- After combining, the node has  $[\text{ceil}(m/2) - 2]$  (original pairs) +  $[\text{ceil}(m/2) - 1]$  (sibling pairs) + 1 (from parent)  $\leq m - 1$  pairs.

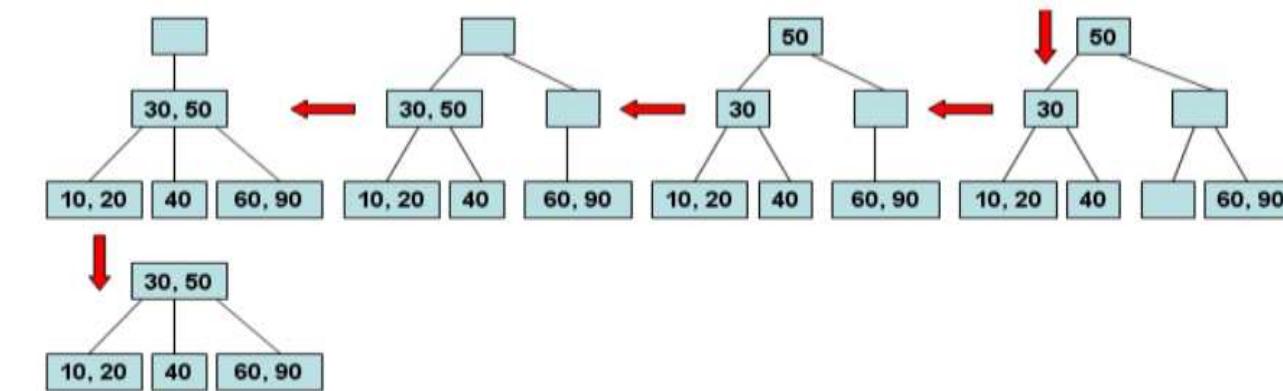
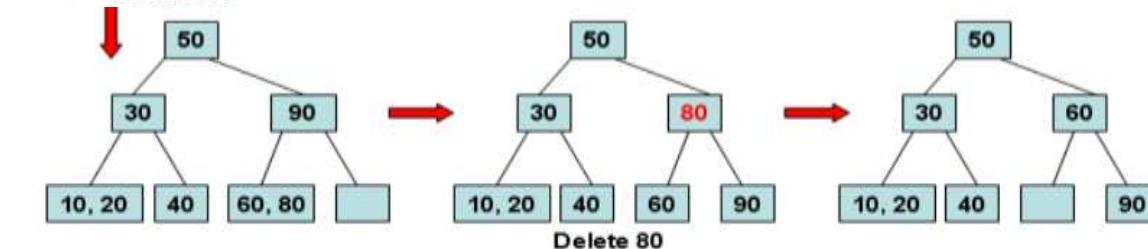
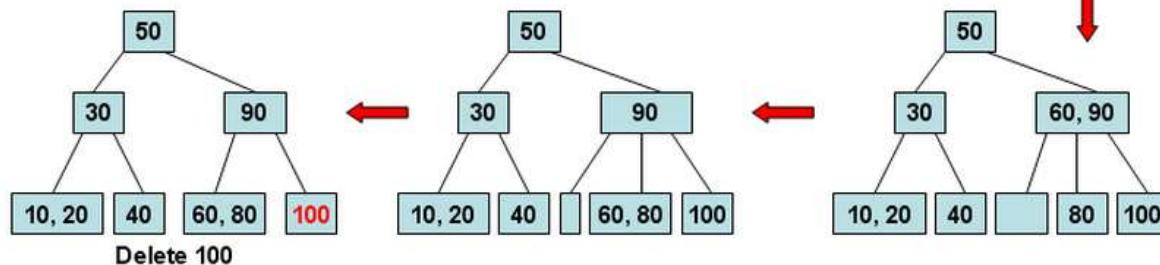
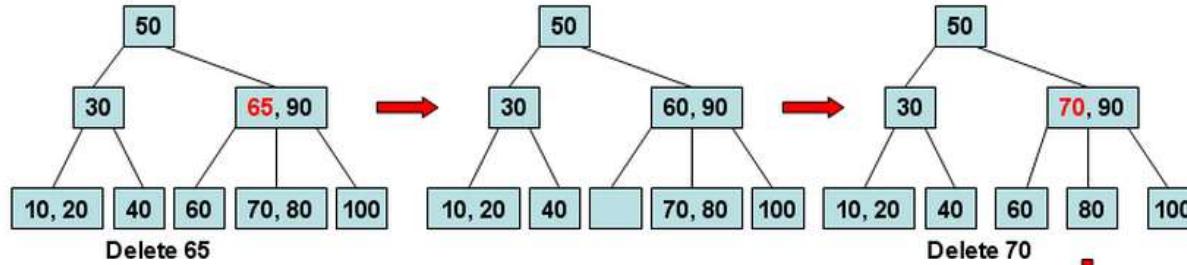
# Disk Accesses



Minimum.  
Borrow.  
Combine.

## **Deletion in B-trees**

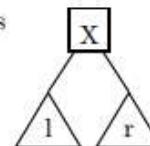
- As in trees, replace the item to be removed by its in-order successor (which is bound to be in a leaf node).
- Remove successor from its leaf node.
- This may cause underflow (fewer than  $d/2$  records in that leaf node  $u$ ).
- Depending on how many records the sibling of  $u$  has, this is fixed either by fusion or by transfer.



## 2-3 Trees

A 2-3 tree has three different kinds of nodes:

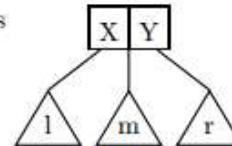
1. A leaf, written as  $\bullet$ .
2. A 2-node, written as



$X$  is called the **value** of the 2-node;  $l$  is its **left subtree**; and  $r$  is its **right subtree**. Every 2-node must satisfy the following invariants:

- (a) Every value  $v$  appearing in subtree  $l$  must be  $\leq X$ .
- (b) Every value  $v$  appearing in subtree  $r$  must be  $\geq X$ .
- (c) The length of the path from the 2-node to *every* leaf in its subtrees must be the same.

3. A 3-node, written as



$X$  is called the **left value** of the 3-node;  $Y$  is called the **right value** of the 3-node;  $l$  is its **left subtree**;  $m$  is its **middle subtree**; and  $r$  is its **right subtree**.

Every 3-node must satisfy the following invariants:

- (a) Every value  $v$  appearing in subtree  $l$  must be  $\leq X$ .
- (b) Every value  $v$  appearing in subtree  $m$  must be  $\geq X$  and  $\leq Y$ .
- (c) Every value  $v$  appearing in subtree  $r$  must be  $\geq Y$ .
- (d) The length of a path from the 3-node to *every* leaf in its subtrees must be the same.

The rules for 2-3 search trees are rather elementary.

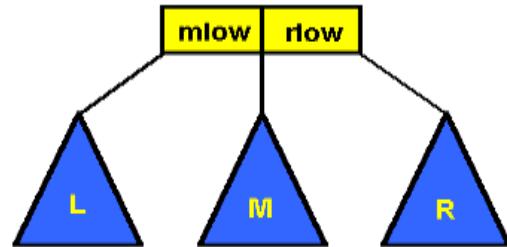
*All data appears at the leaves.*

*Data elements are ordered from left (minimum) to right (maximum).*

*Every path through the tree is the same length.*

*Interior nodes have two or three subtrees.*

Thus if there are n records (or data items) stored in the leaves of a 2-3 search tree, then the tree is between  $\log_3 n$  and  $\log_2 n$  in height. Interior nodes hold no records, but contain some information about the keys of elements stored in their subtrees. Consider figure below.



**Figure A 2-3 Tree Interior Node**

The interior node may contain two numbers or keys:

*mlow = smallest key in the M (middle) subtree*

*rlow = smallest key in the R (right) subtree*

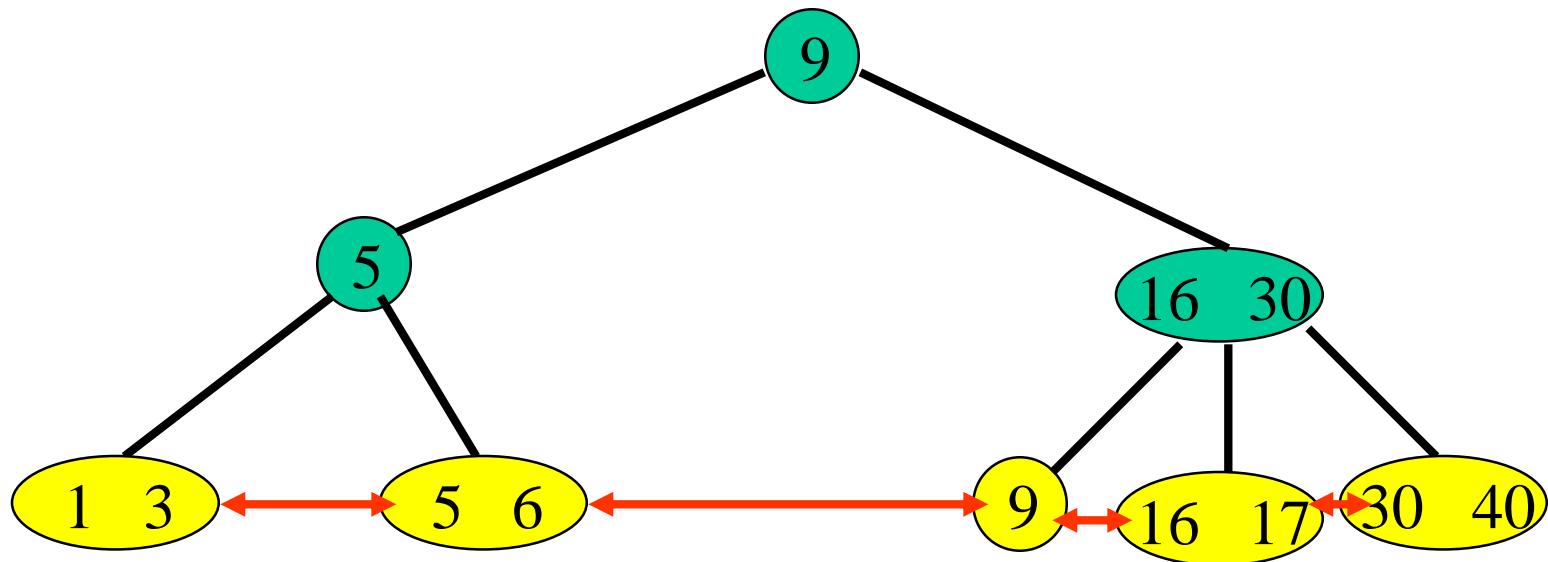
# B<sup>+</sup>-Trees

- Same structure as B-trees.
- Dictionary pairs are in leaves only. Leaves form a doubly-linked list.
- Remaining nodes have following structure:

$j \ a_0 \ k_1 \ a_1 \ k_2 \ a_2 \dots k_j \ a_j$

- $j$  = number of keys in node.
- $a_i$  is a pointer to a subtree.
- $k_i \leq$  smallest key in subtree  $a_i$  and  $>$  largest in  $a_{i-1}$ .

# Example B+-tree

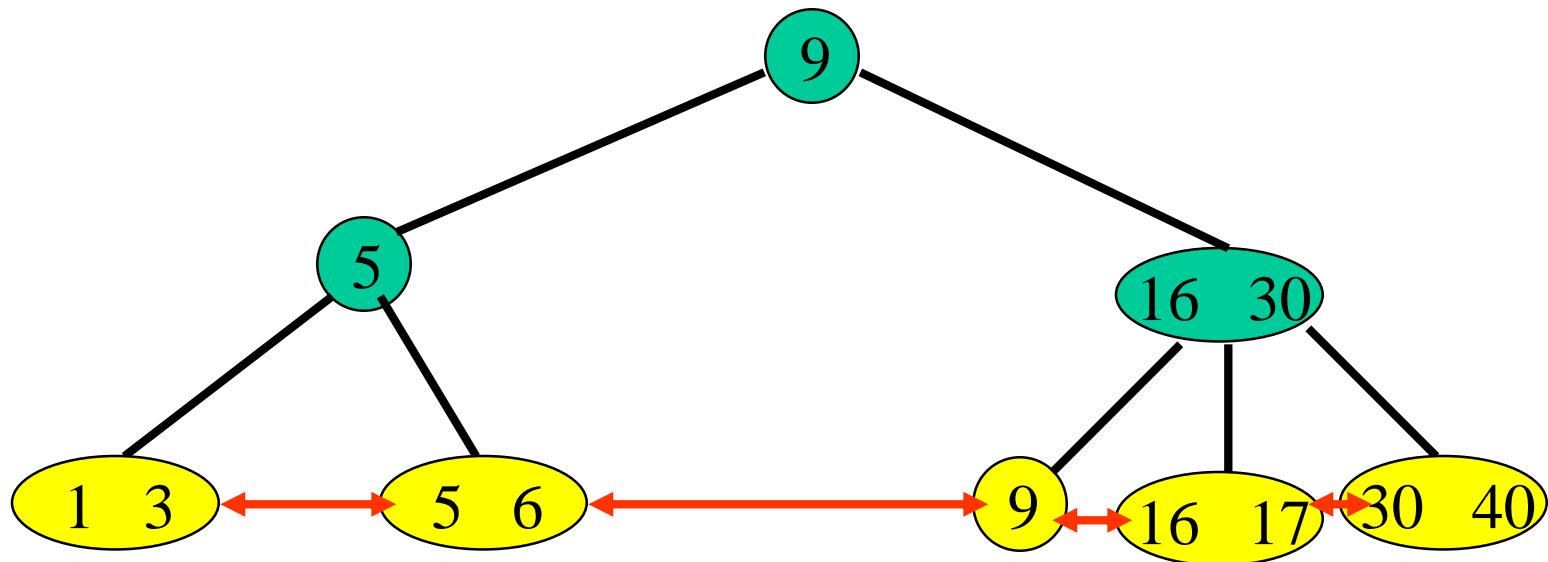


→ index node



→ leaf/data node

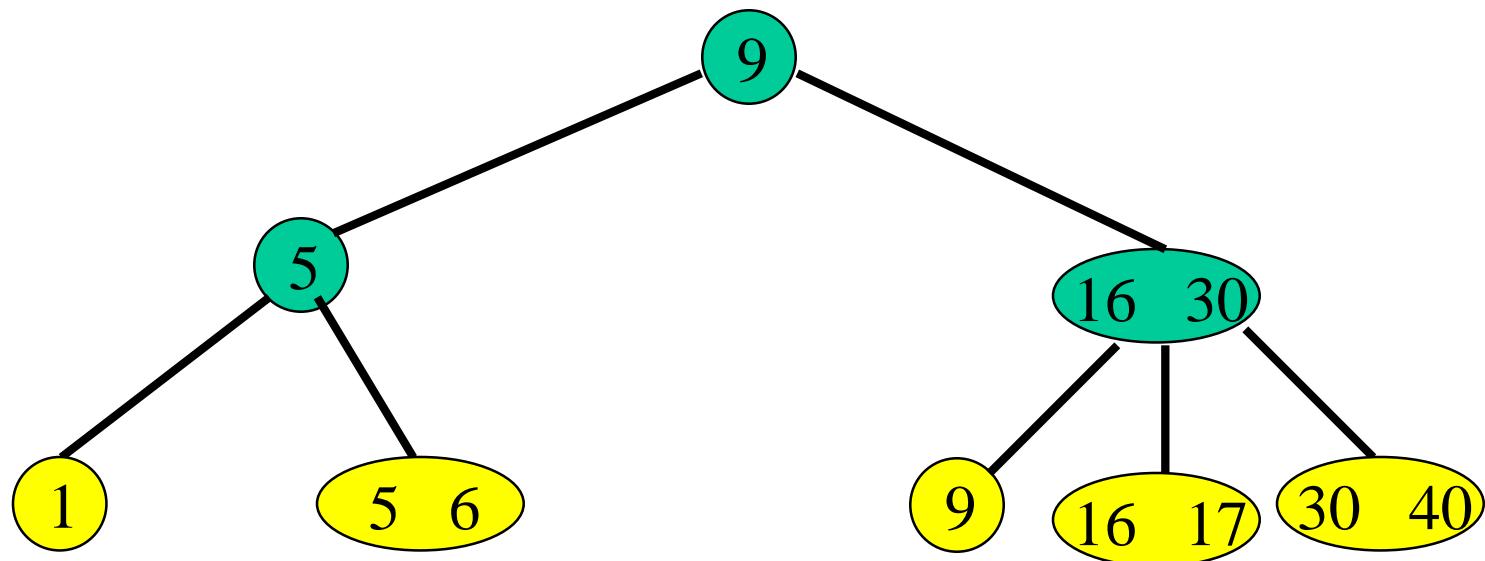
# B+-tree—Search



key = 5

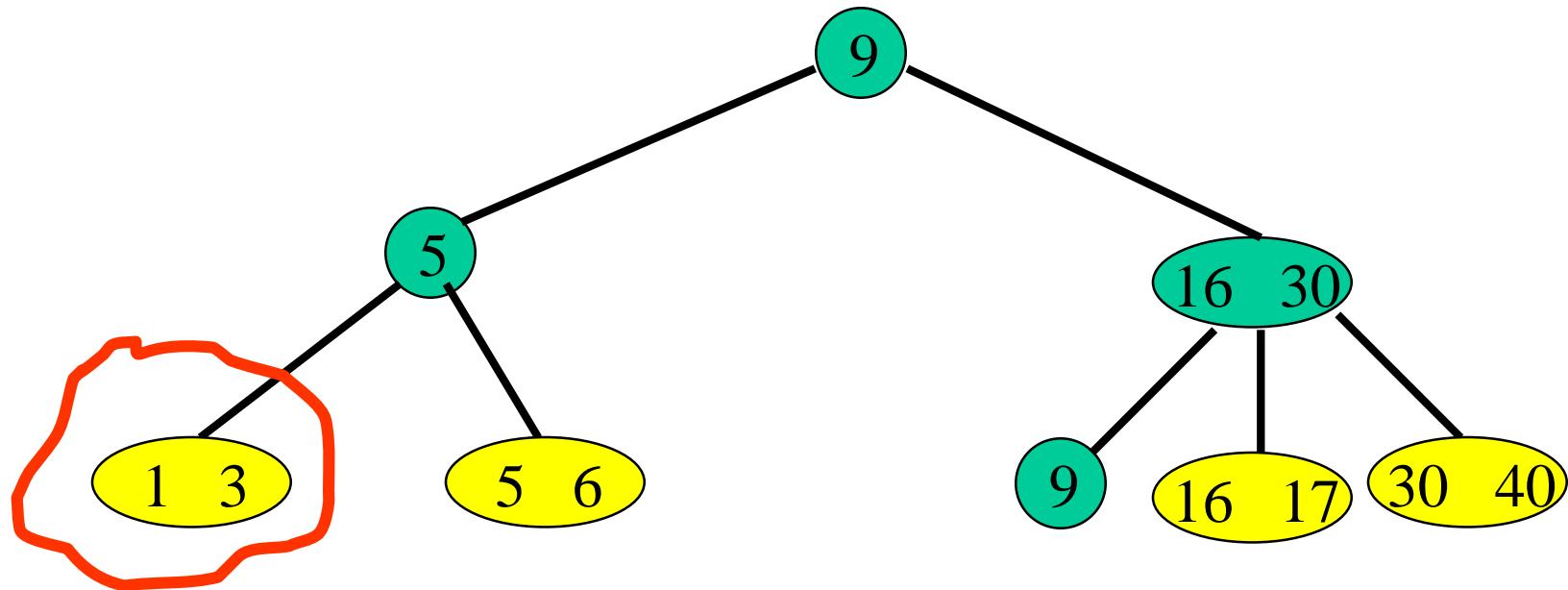
$6 \leq \text{key} \leq 20$

# B+-tree—Insert



Insert 10

# Insert



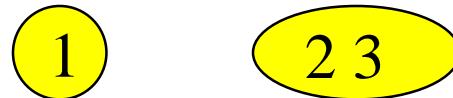
- Insert a pair with key = 2.
- New pair goes into a 3-node.

# Insert Into A 3-node

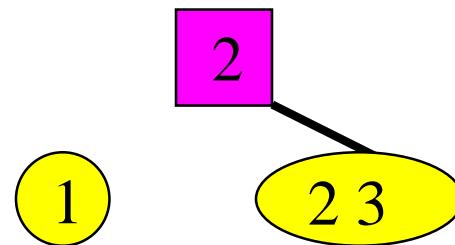
- Insert new pair so that the keys are in ascending order.



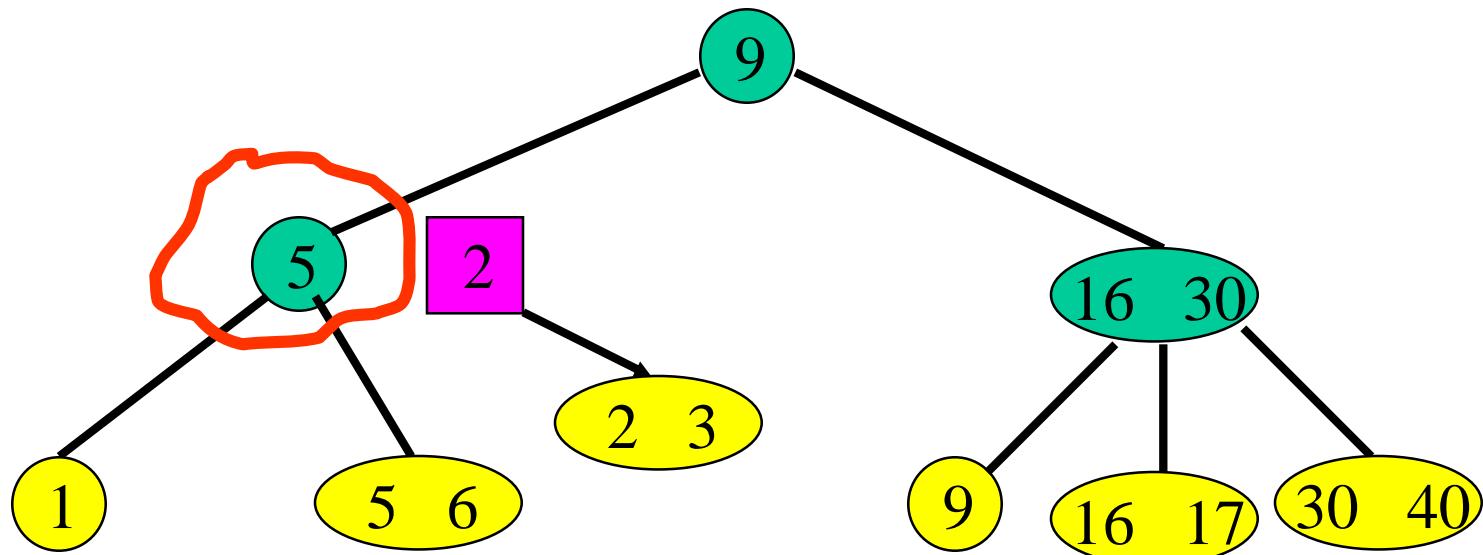
- Split into two nodes.



- Insert smallest key in new node and pointer to this new node into parent.

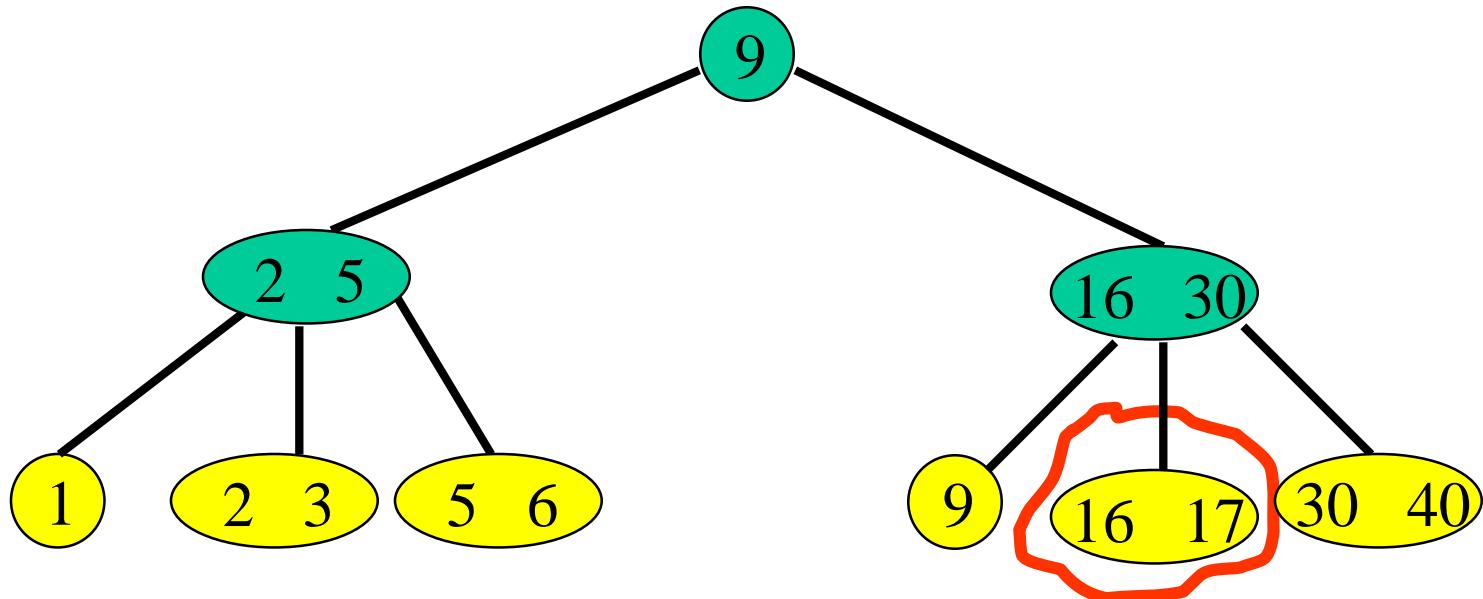


# Insert



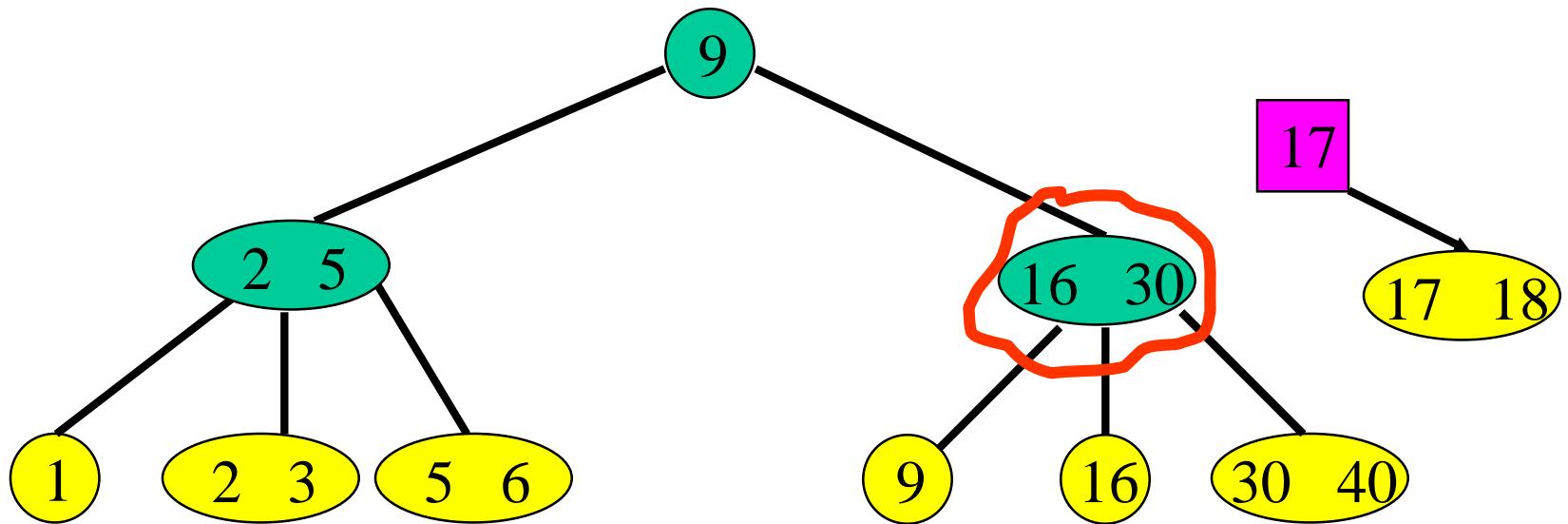
- Insert an index entry 2 plus a pointer into parent.

# Insert



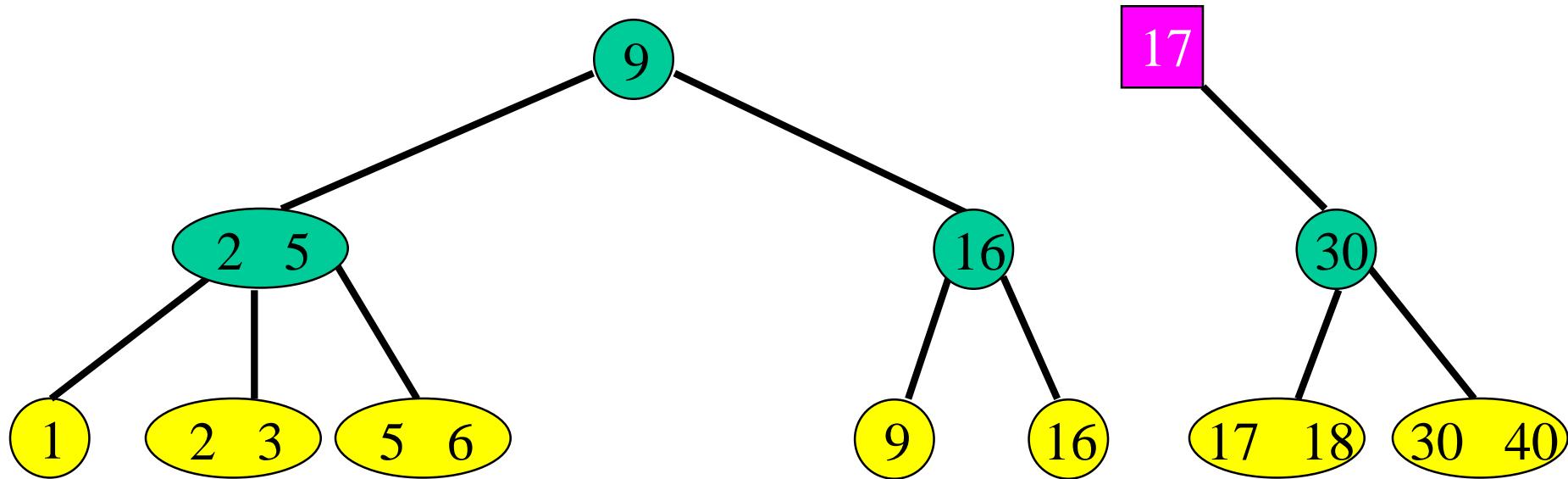
- Now, insert a pair with key = 18.

# Insert



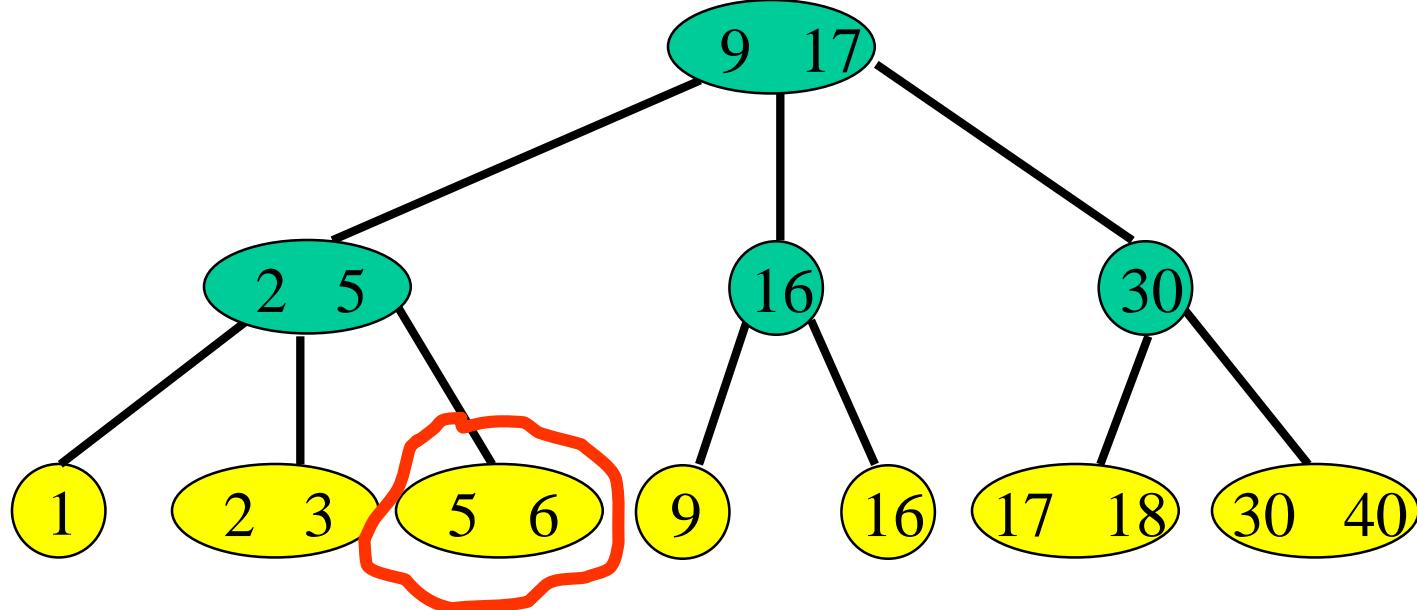
- Now, insert a pair with key = 18.
- Insert an index entry 17 plus a pointer into parent.

# Insert



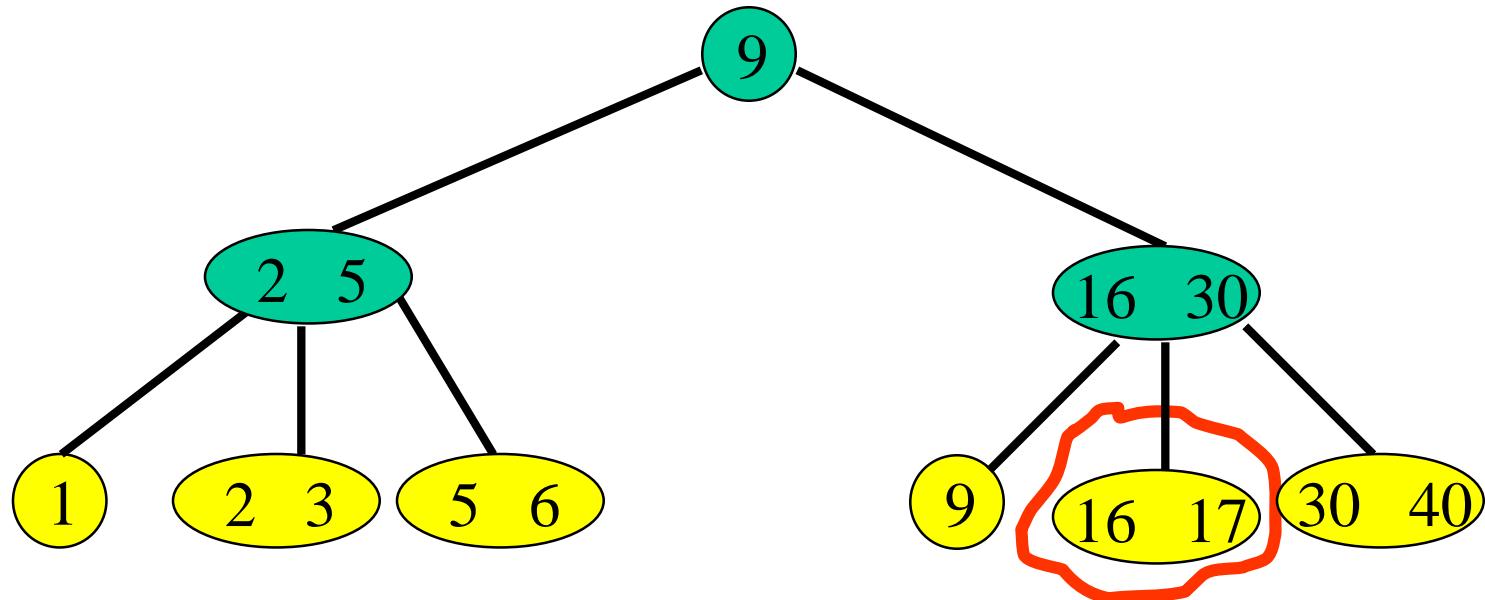
- Now, insert a pair with key = 18.
- Insert an index entry 17 plus a pointer into parent.

# Insert



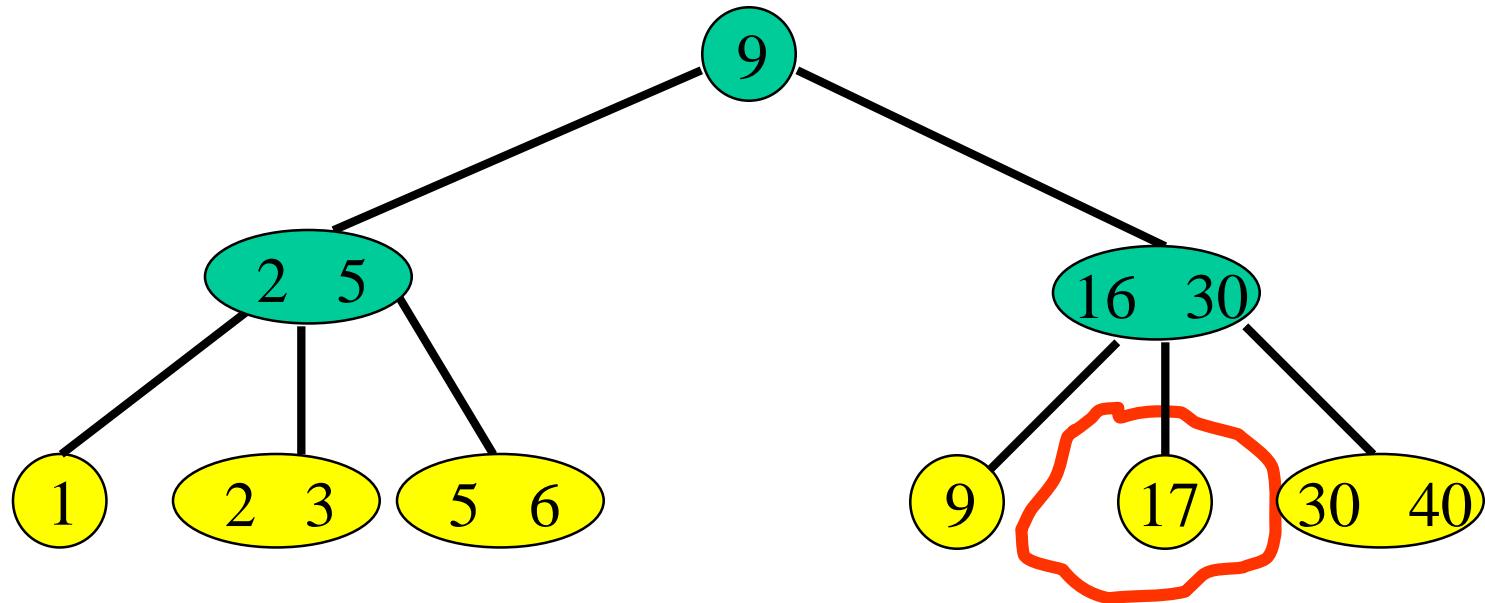
- Now, insert a pair with key = 7.

# Delete



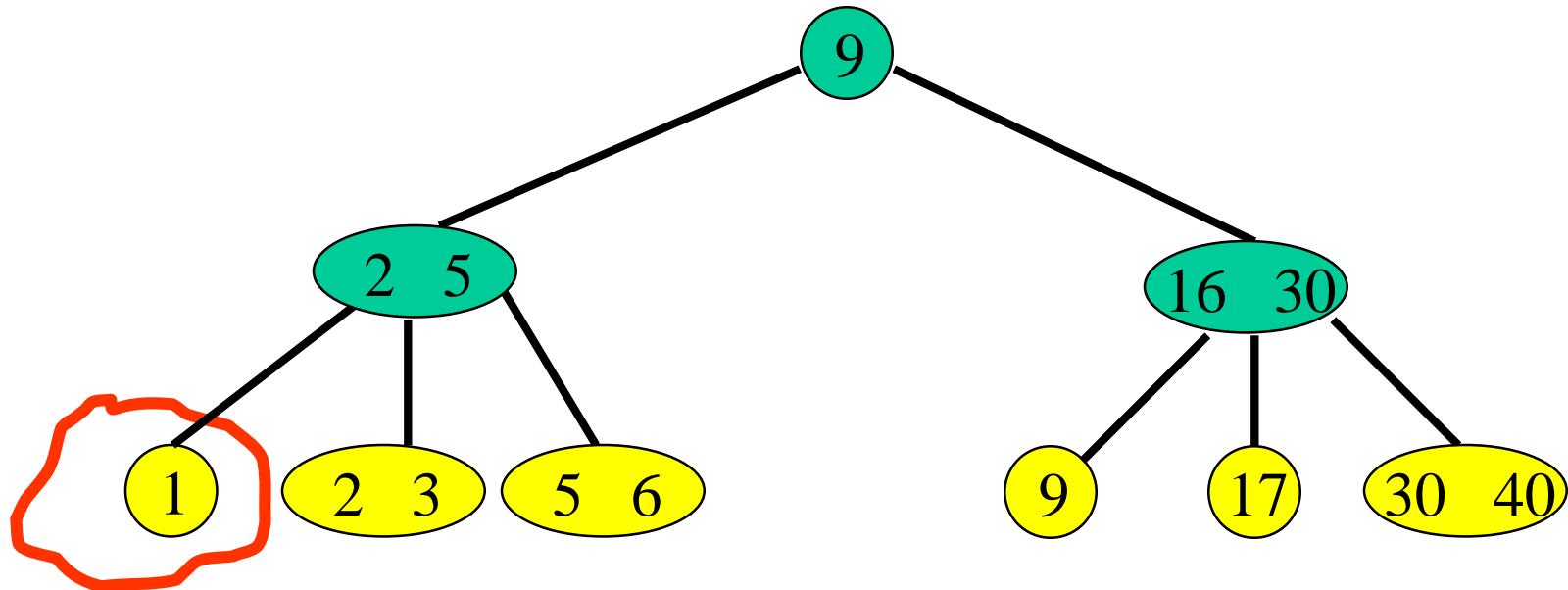
- Delete pair with key = 16.
- Note: delete pair is always in a leaf.

# Delete



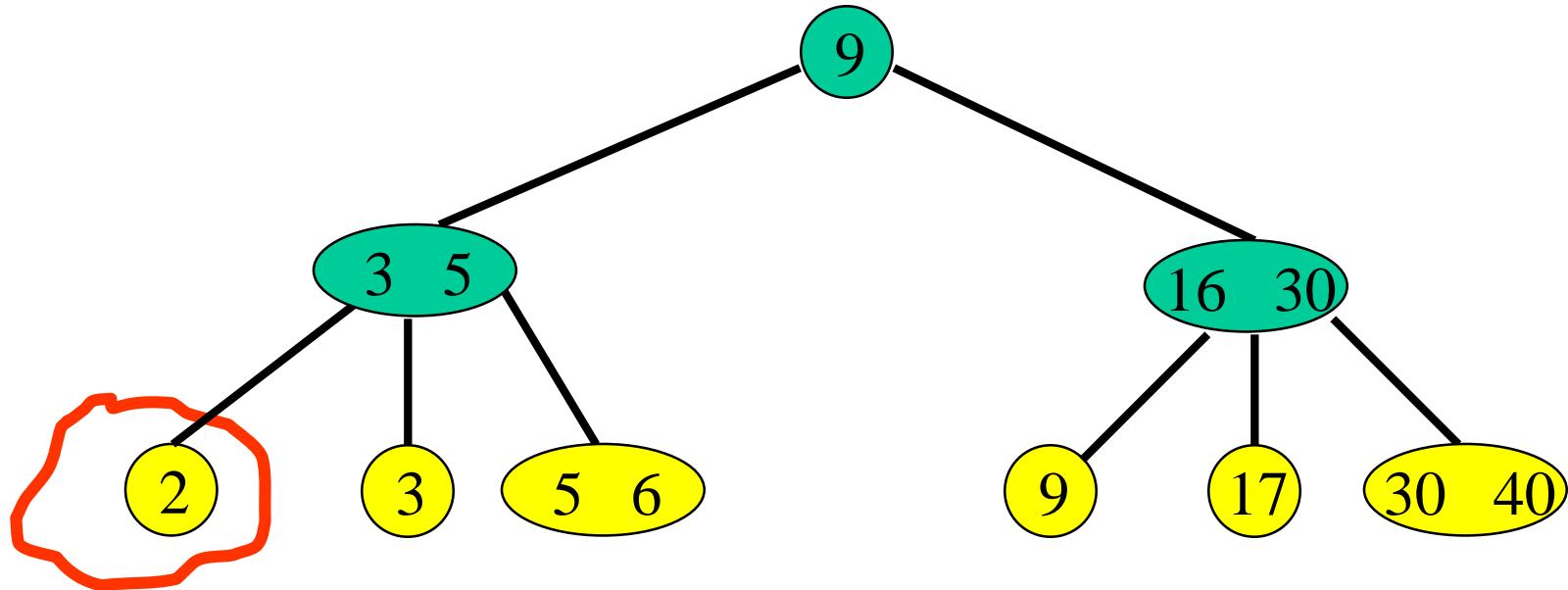
- Delete pair with key = 16.
- Note: delete pair is always in a leaf.

# Delete



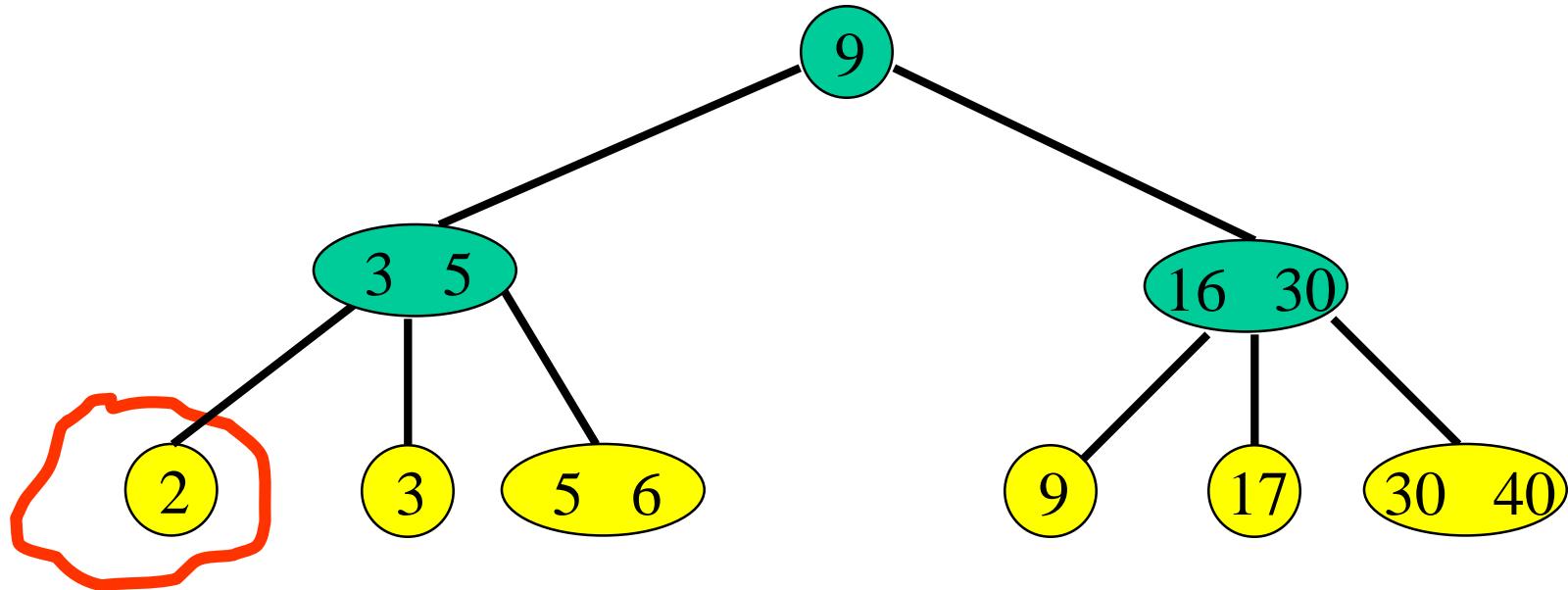
- Delete pair with key = 1.
- Get  $\geq 1$  from adjacent sibling and update parent key.

# Delete



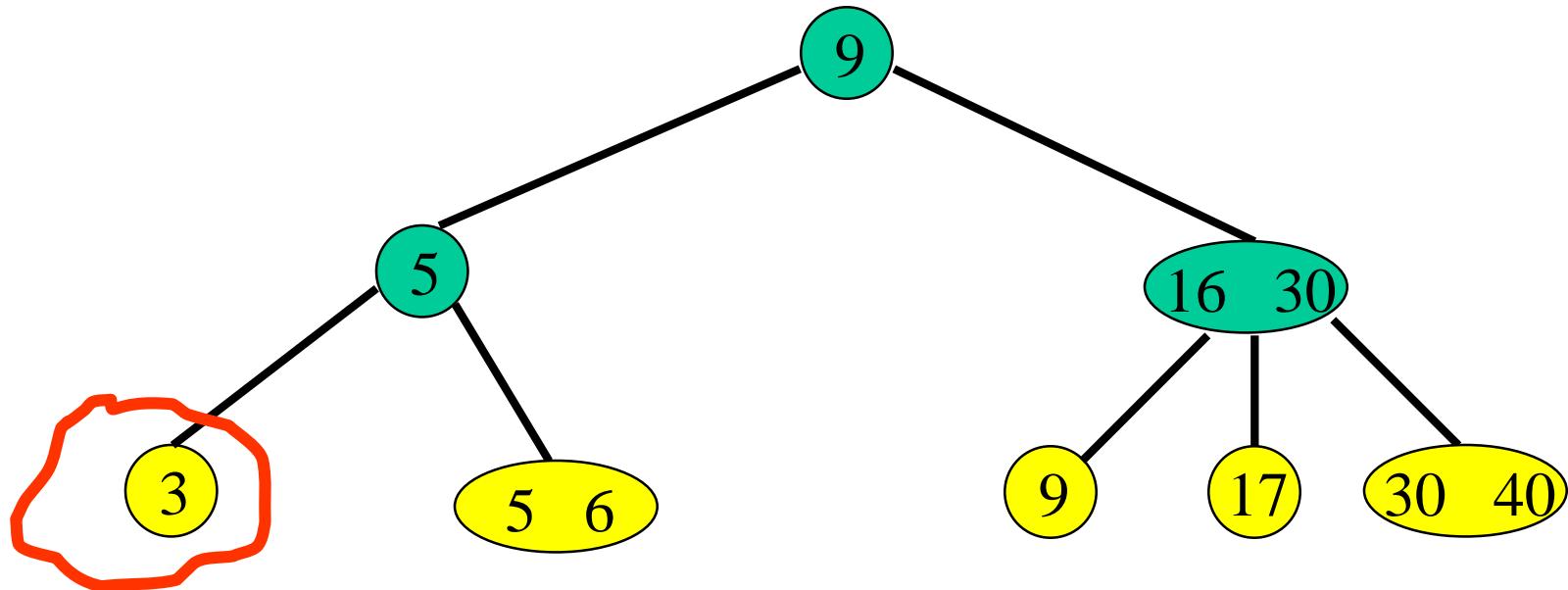
- Delete pair with key = 1.
- Get  $\geq 1$  from sibling and update parent key.

# Delete



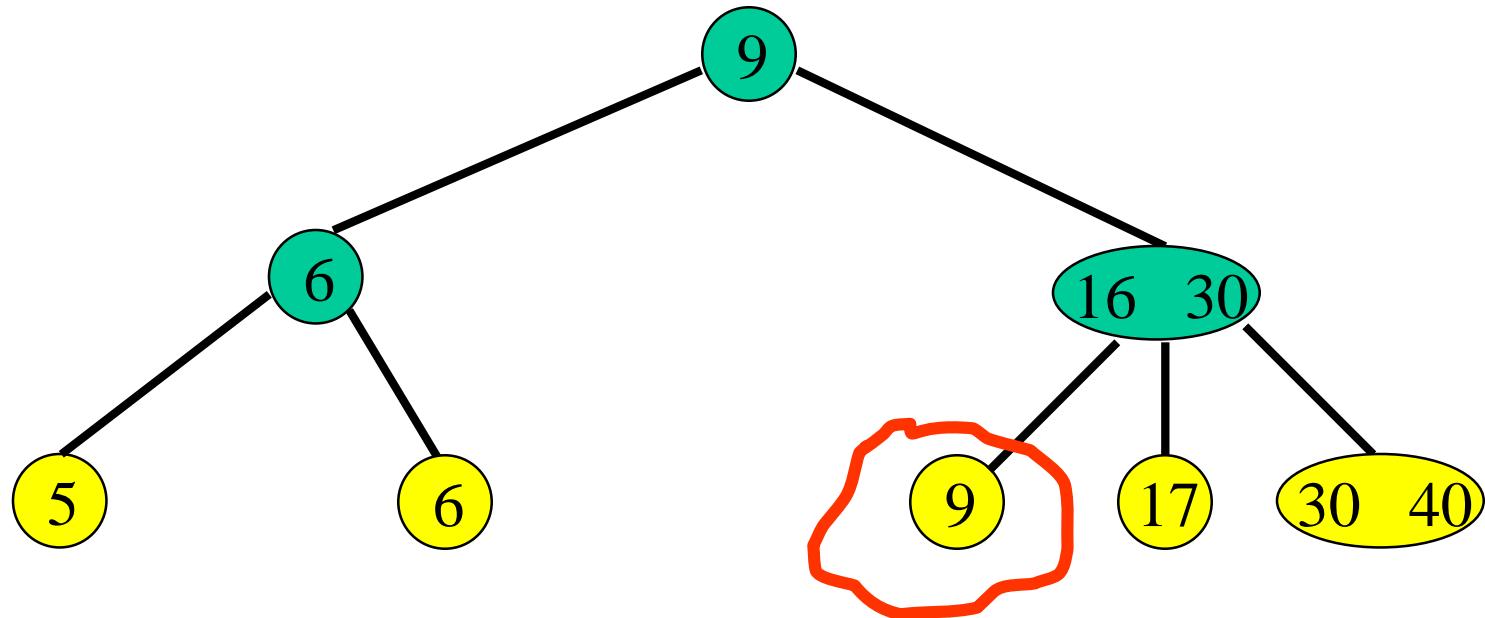
- Delete pair with key = 2.
- Merge with sibling, delete in-between key in parent.

# Delete



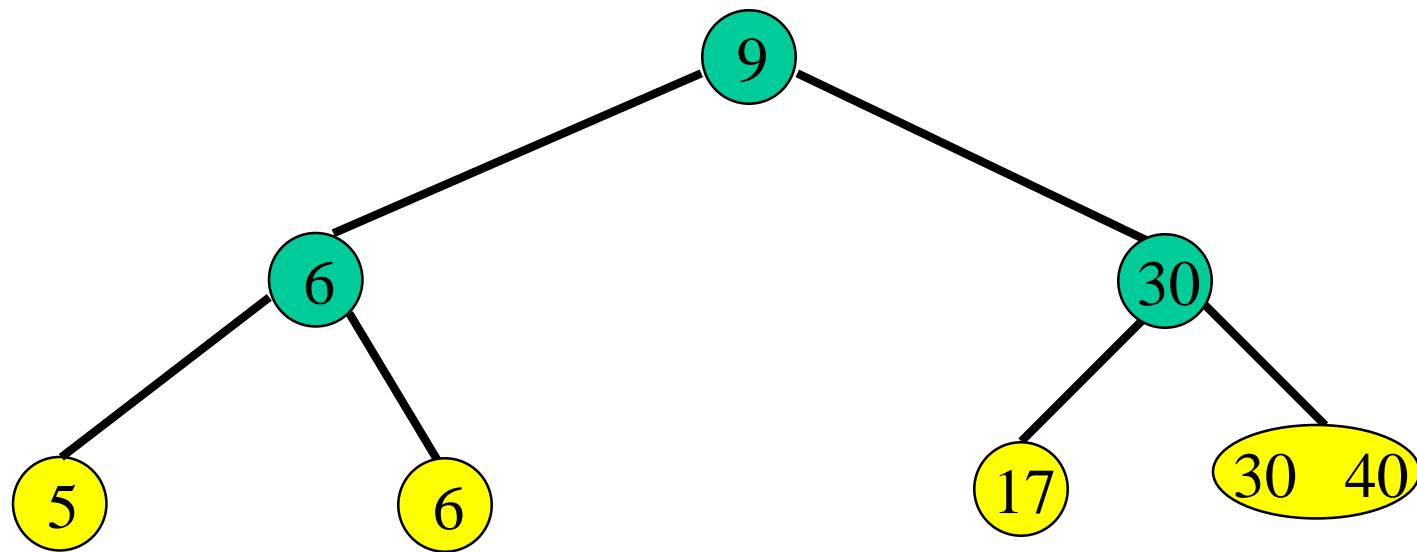
- Delete pair with key = 3.
- Get  $\geq 1$  from sibling and update parent key.

# Delete

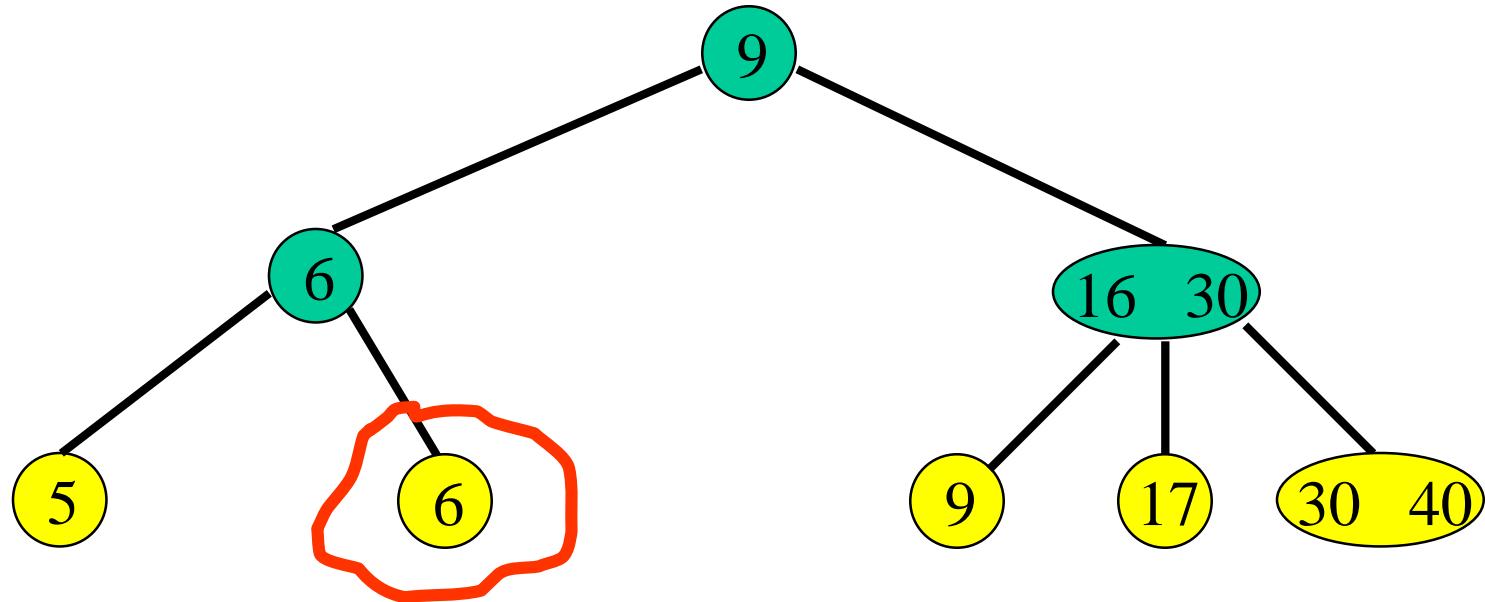


- Delete pair with key = 9.
- Merge with sibling, delete in-between key in parent.

# Delete

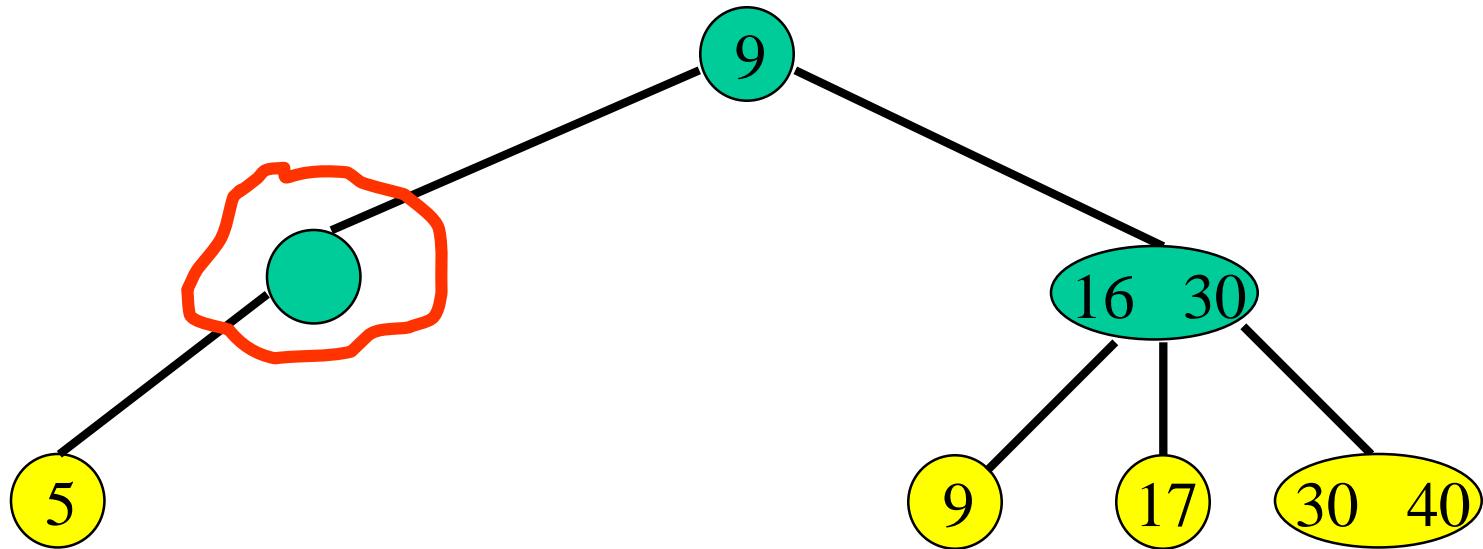


# Delete



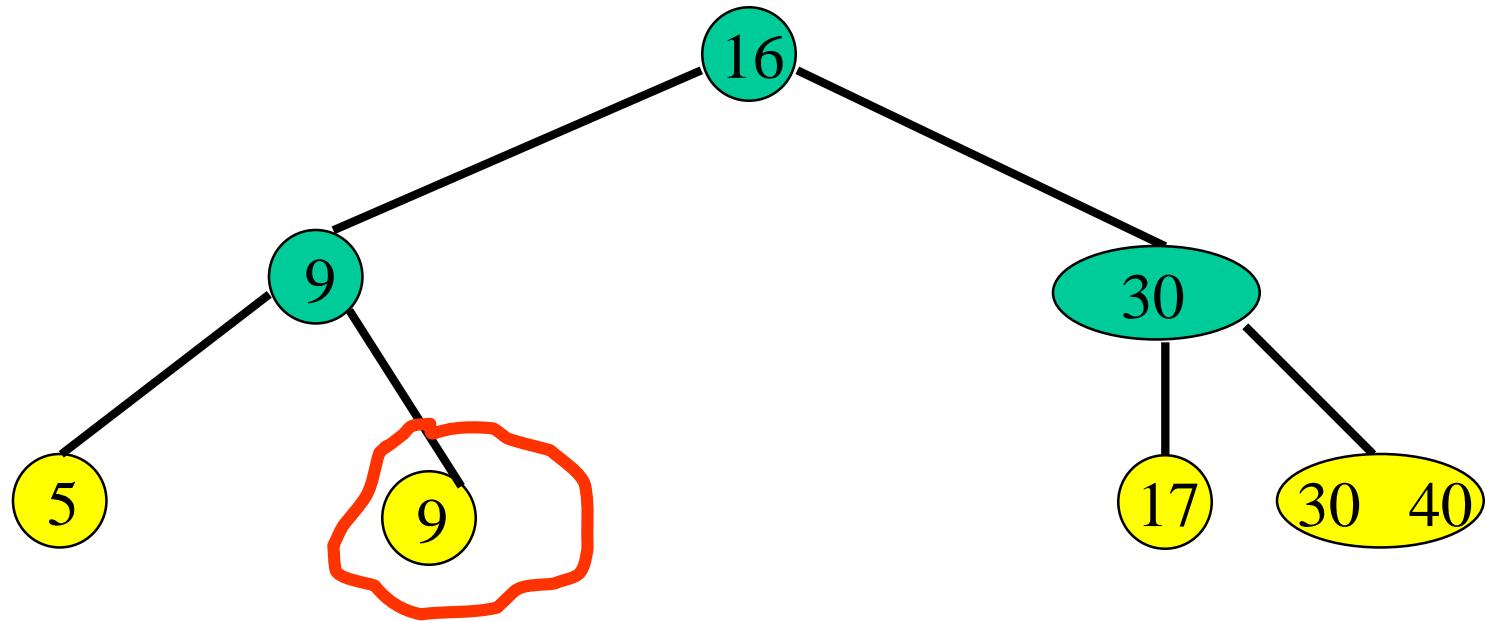
- Delete pair with key = 6.
- Merge with sibling, delete in-between key in parent.

# Delete



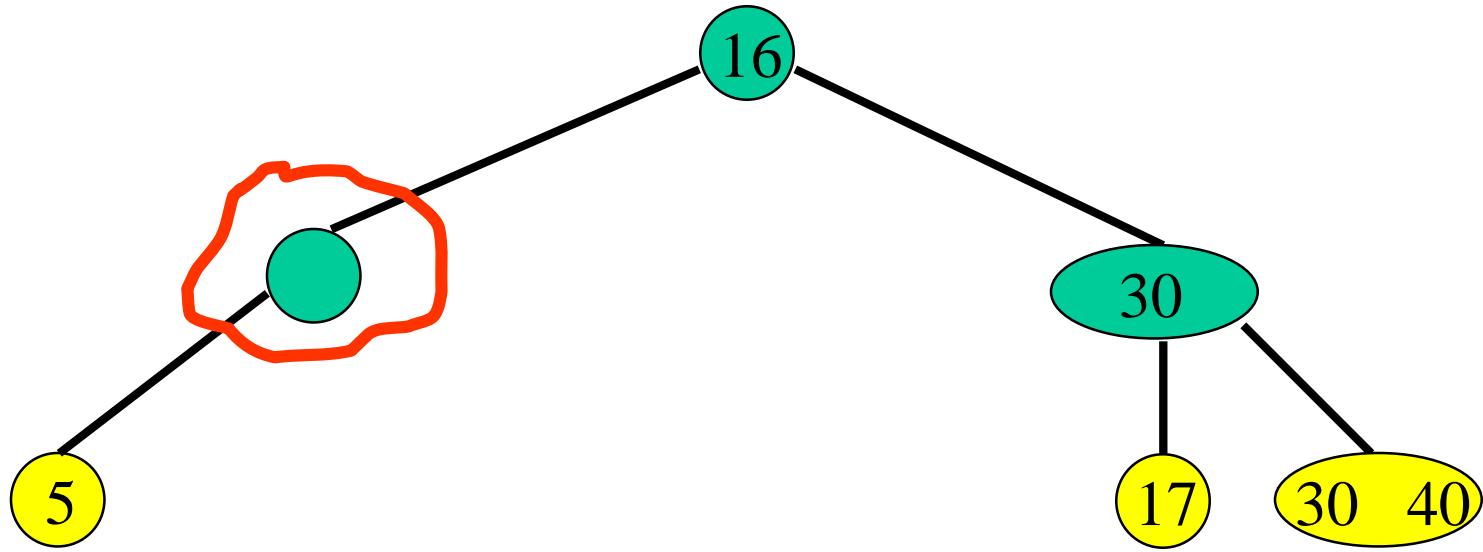
- Index node becomes deficient.
- Get  $\geq 1$  from sibling, move last one to parent, get parent key.

# Delete



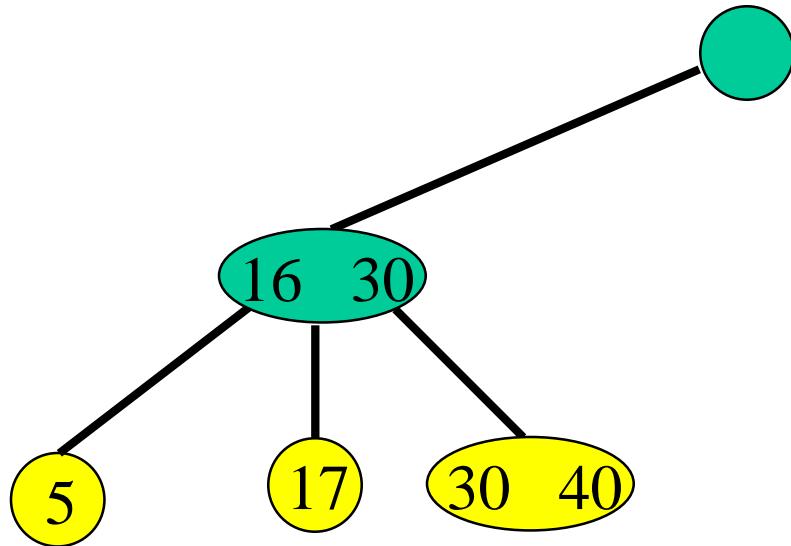
- Delete 9.
- Merge with sibling, delete in-between key in parent.

# Delete



- Index node becomes deficient.
- Merge with sibling and in-between key in parent.

# Delete



- Index node becomes deficient.
- It's the root; discard.

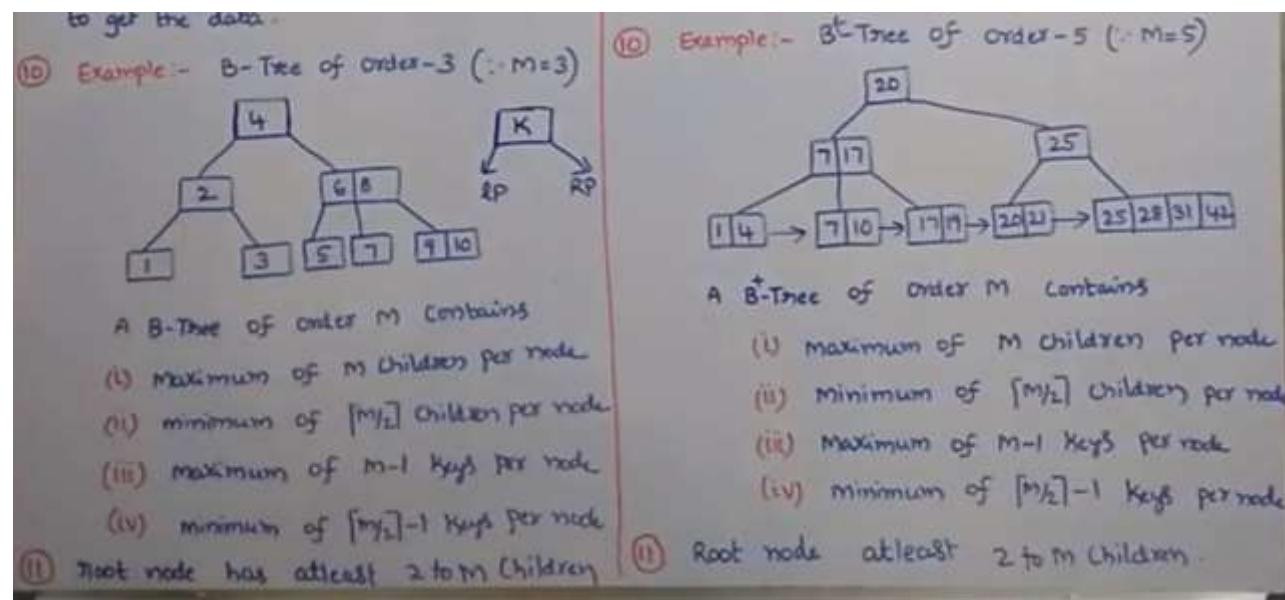
## Comparison between B Tree and B+ Tree:

	<b>B Tree</b>	<b>B+ Tree</b>
Short web descriptions	A B tree is an organizational structure for information storage and retrieval in the form of a tree in which all terminal nodes are at the same distance from the base, and all non-terminal nodes have between n and $2n$ sub-trees or pointers (where n is an integer).	B+ tree is an n-array tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.
Also known as	Balanced tree.	B plus tree.
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log_b n)$
Insert	$O(\log n)$	$O(\log_b n)$
Delete	$O(\log n)$	$O(\log_b n)$

Storage	In a B tree, search keys and data stored in internal or leaf nodes.	In a B+ tree, data stored only in leaf nodes.
Data	The leaf nodes of the three store pointers to records rather than actual records.	The leaf nodes of the tree stores the actual record rather than pointers to records.
Space	These trees waste space	These trees do not waste space.

Function of leaf nodes	In B tree, the leaf node cannot store using linked list.	In B+ tree, leaf node data are ordered in a sequential linked list.
Searching	Here, searching becomes difficult in B- tree as data cannot be found in the leaf node.	Here, searching of any data in a B+ tree is very easy because all data is found in leaf nodes.
Search accessibility	Here in B tree the search is not that easy as compared to a B+ tree.	Here in B+ tree the searching becomes easy.
Redundant key	They do not store redundant search key.	They store redundant search key.
Applications	They are an older version and are not that advantageous as compared to the B+ trees.	Many database system implementers prefer the structural simplicity of a B+ tree.

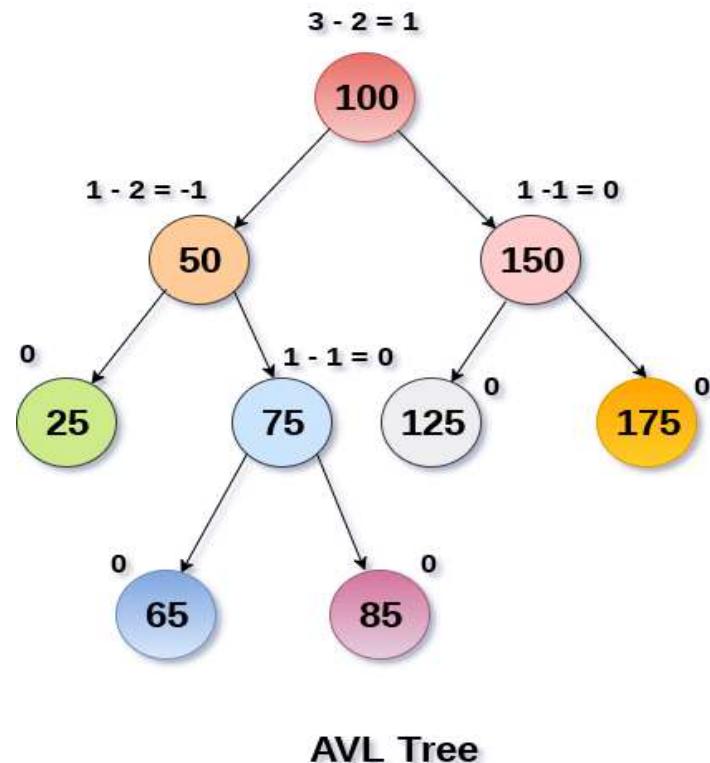
1	In B-Tree keys and records can be stored in internal nodes as well as leaf nodes	In B+ -Tree records are stored in leaf nodes and the keys are stored in internal nodes
2	In B-Tree keys cannot be repeatedly stored	In B+ -Tree redundant keys may be present
3	In B-Tree leaf nodes are not linked together	In B+ -Tree , leaf nodes are linked to each other
4	In B-Tree , searching is slow because records are stored on leaf and internal nodes	In B+ -Tree, leaf nodes are linked together to make the search more accurate and faster
5	In B-Tree , deletion of internal nodes is a complicated and time consuming process	In B+ -Tree, deletion is not difficult because keys are removed from the leaf nodes
6	Operations on B-Tree are performed slower	Operation on B+ -Tree are performed faster than B-Tree
7	In B-Tree , sequential access of data or record not possible	In B+ -Tree, sequential access of data or records can be done fastly
8	B-Tree has more height compared to width	B+ -Tree has more width compared to height
9	In B-Tree ,each node will have atleast two branches and each node will have some record, hence there is no need to traverse till leaf node to get the data	In B+ -Tree,internal nodes contains only pointers to the leaf nodes. all leaf nodes will have records and all are at same distance from the root.all leaf node are at the same level
10	In B-Tree	In B+ -Tree



# AVL Tree

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- Balance Factor (k) = height (left(k)) - height (right(k))
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.
- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



## Complexity

Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

- Operations on AVL tree
- Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree.
- Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

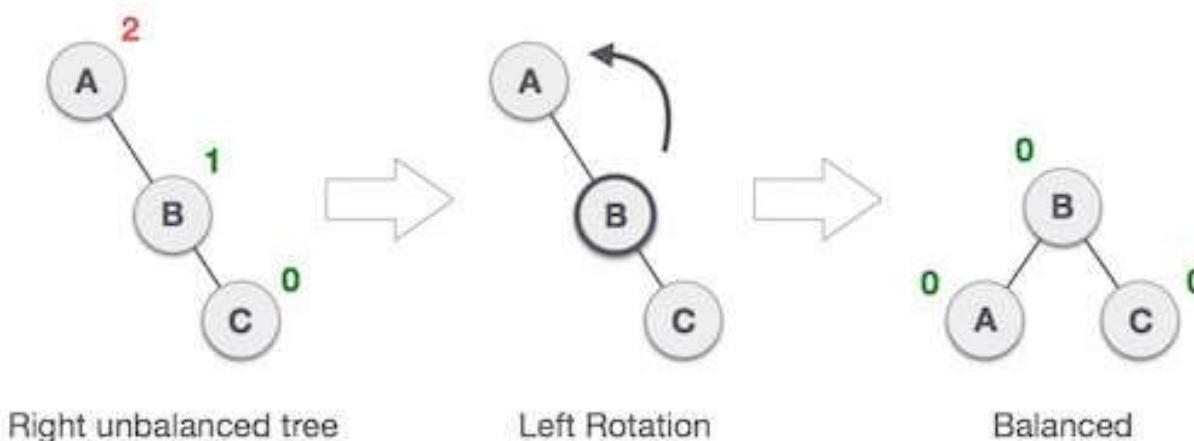
SN	Operation	Description
1	<a href="#"><u>Insertion</u></a>	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	<a href="#"><u>Deletion</u></a>	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

# AVL Rotations

- We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:
  - L L rotation: Inserted node is in the left subtree of left subtree of A
  - R R rotation : Inserted node is in the right subtree of right subtree of A
  - L R rotation : Inserted node is in the right subtree of left subtree of A
  - R L rotation : Inserted node is in the left subtree of right subtree of A
  - Where node A is the node whose balance Factor is other than -1, 0, 1.
- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

# RR Rotation

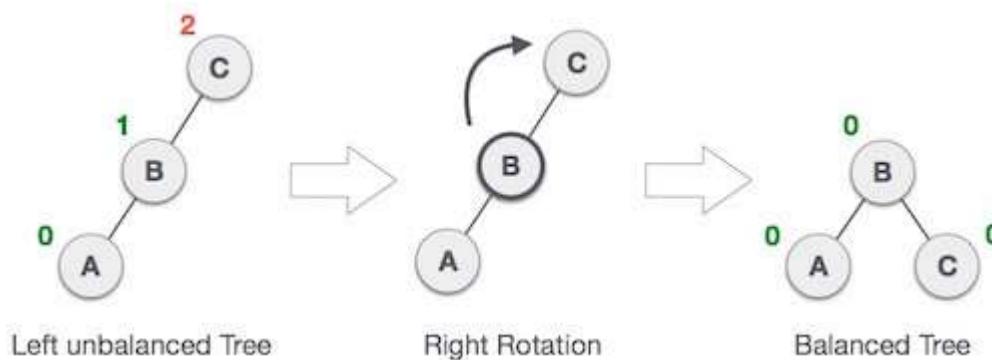
- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

# LL Rotation

- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

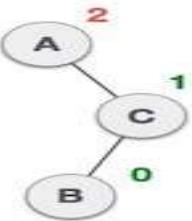
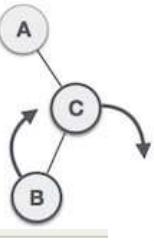
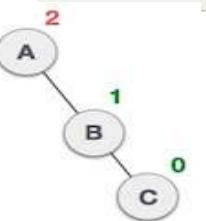
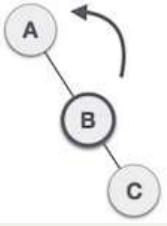
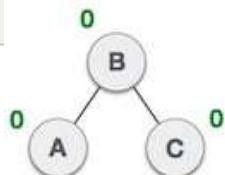
# LR Rotation

- Double rotations are bit tougher than single rotation which has already explained above.  
LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	<p>A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C</p>
	<p>As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node <b>A</b>, has become the left subtree of <b>B</b>.</p>
	<p>After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of <b>C</b></p>
	<p>Now we perform LL clockwise rotation on full tree, i.e. on node C. node <b>C</b> has now become the right subtree of node B, A is left subtree of B</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.</p>

# RL Rotation

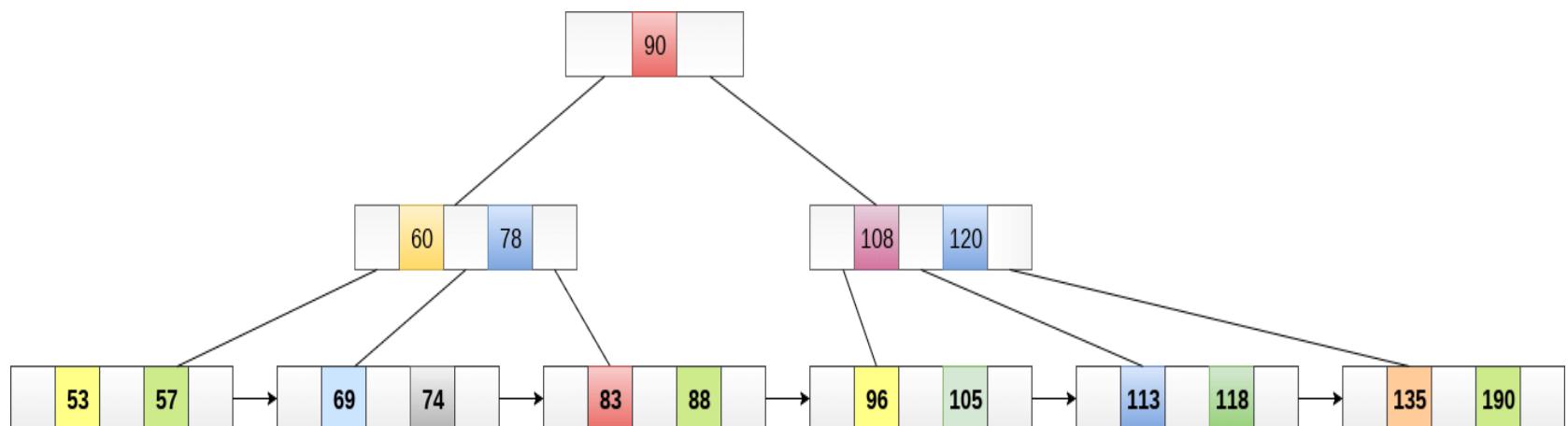
- As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	 <p>A node <b>B</b> has been inserted into the left subtree of <b>C</b>, the right subtree of <b>A</b>, because of which <b>A</b> has become an unbalanced node having balance factor -2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of <b>A</b></p>
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at <b>C</b> is performed first. By doing RR rotation, node <b>C</b> has become the right subtree of <b>B</b> .
	After performing LL rotation, node <b>A</b> is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node <b>A</b> .
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node <b>A</b> . node <b>C</b> has now become the right subtree of node <b>B</b> , and node <b>A</b> has become the left subtree of <b>B</b> .
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

# B+ Tree

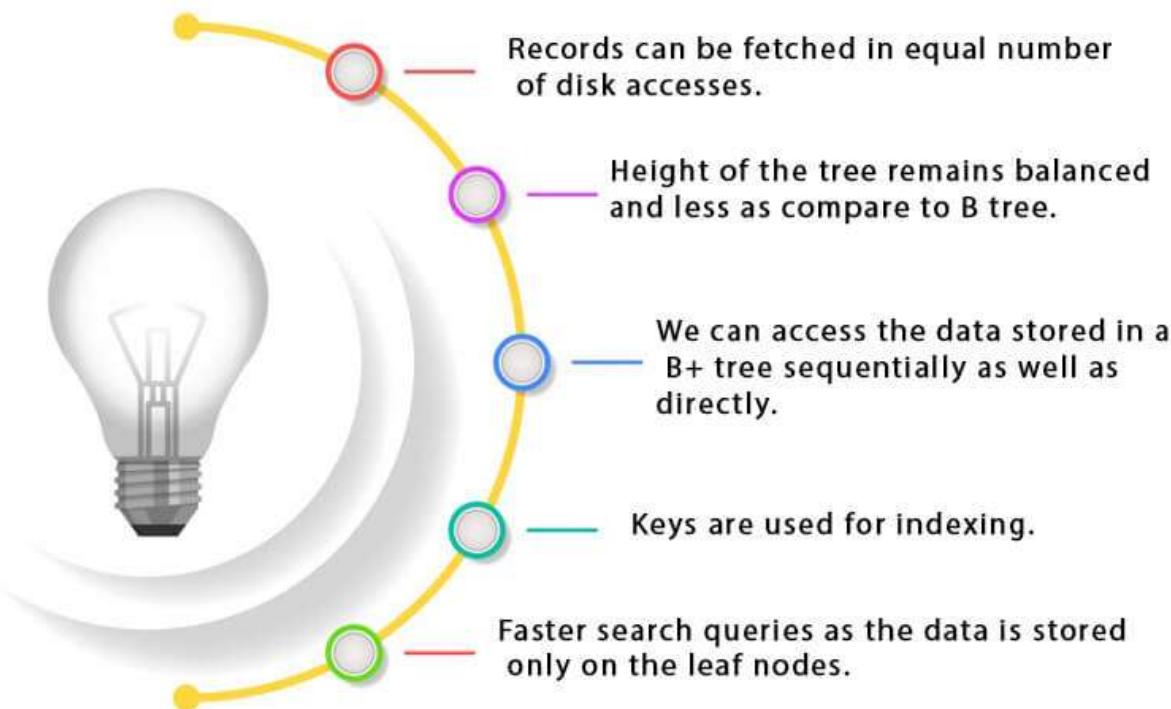
- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.
- B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

- The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



- Advantages of B+ Tree
- Records can be fetched in equal number of disk accesses.
- Height of the tree remains balanced and less as compare to B tree.
- We can access the data stored in a B+ tree sequentially as well as directly.
- Keys are used for indexing.
- Faster search queries as the data is stored only on the leaf nodes.

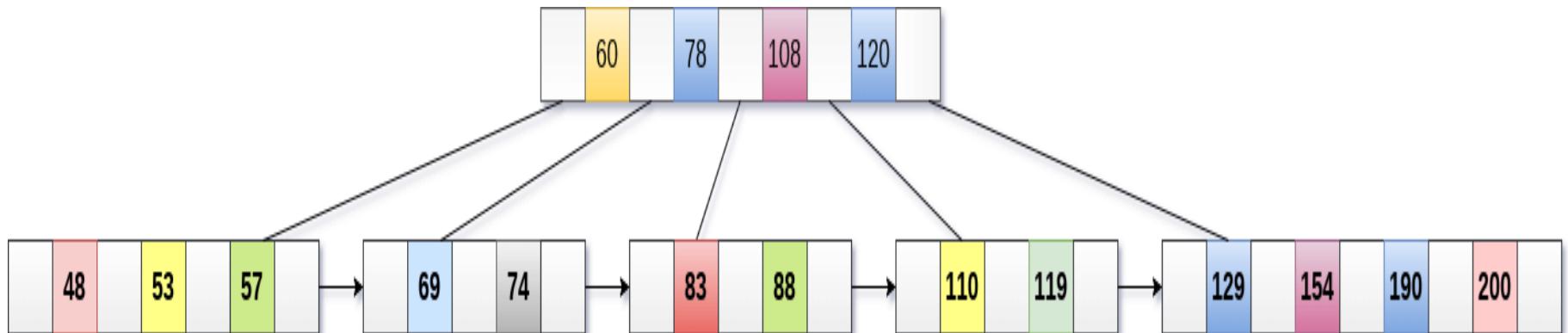
## Advantages of B+ Tree



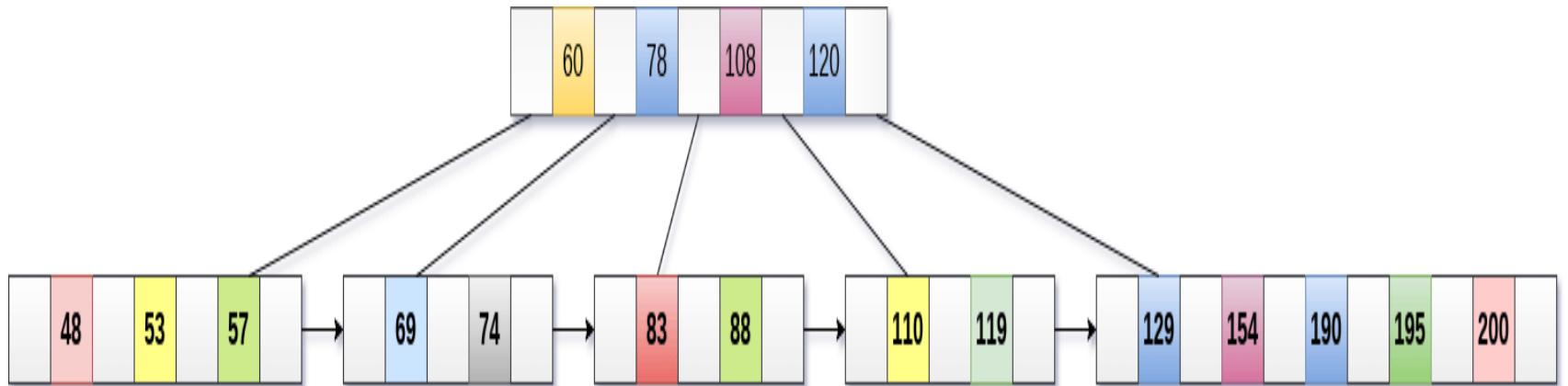
SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

- **Insertion in B+ Tree**
- Step 1: Insert the new node as a leaf node
- Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.
- Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

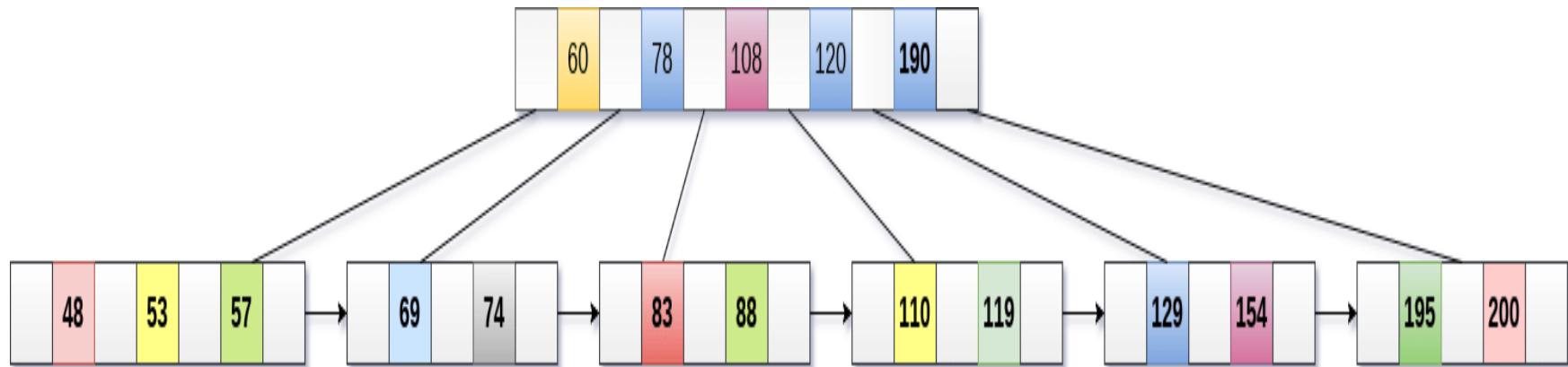
- Example :
- Insert the value 195 into the B+ tree of order 5 shown in the following figure.



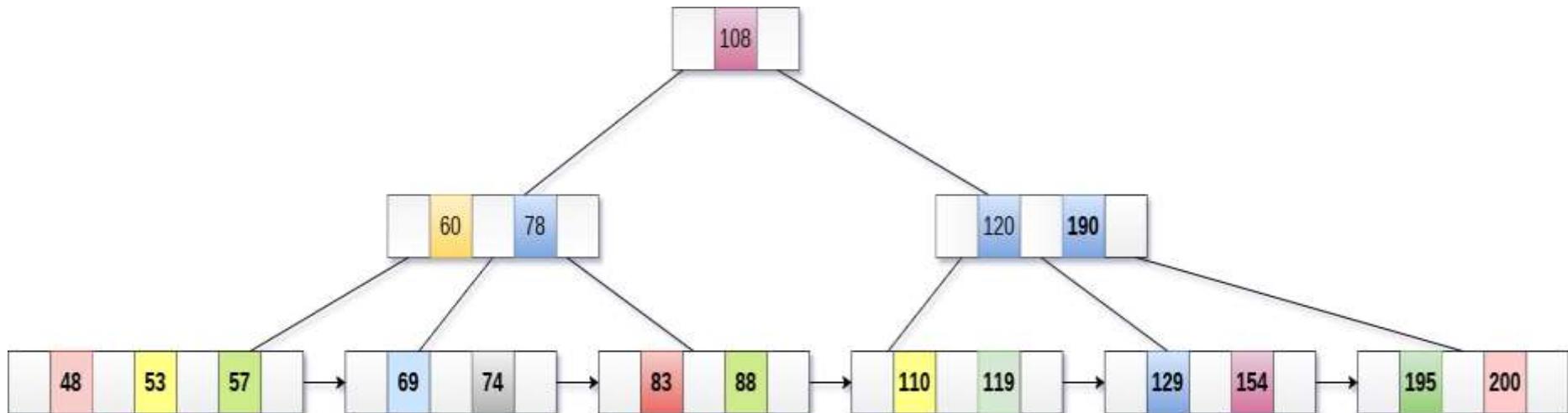
- 195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



- The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.

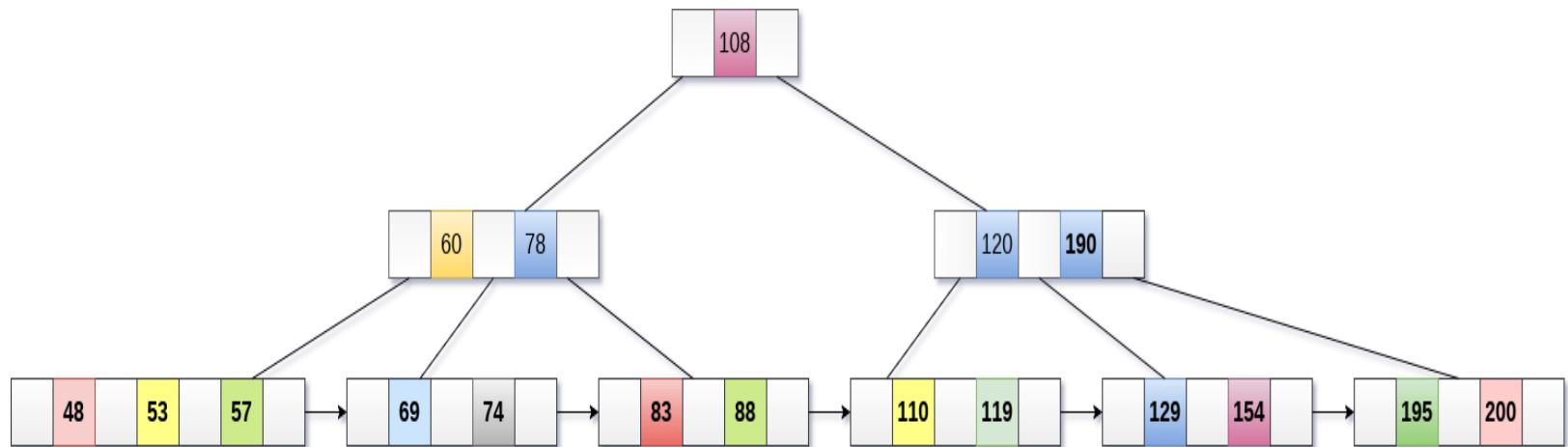


- Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.

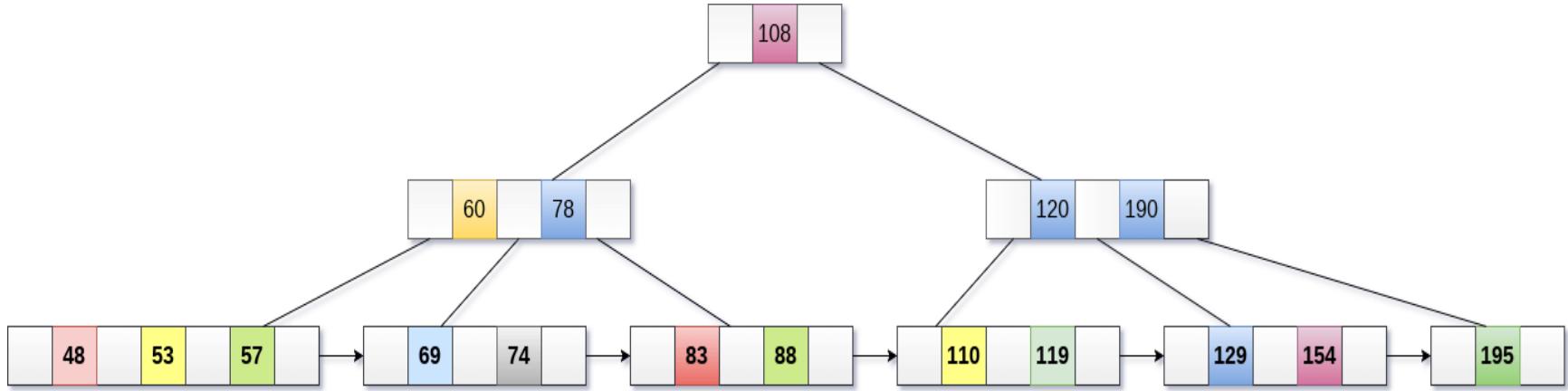


- **Deletion in B+ Tree**
- Step 1: Delete the key and data from the leaves.
- Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.
- Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

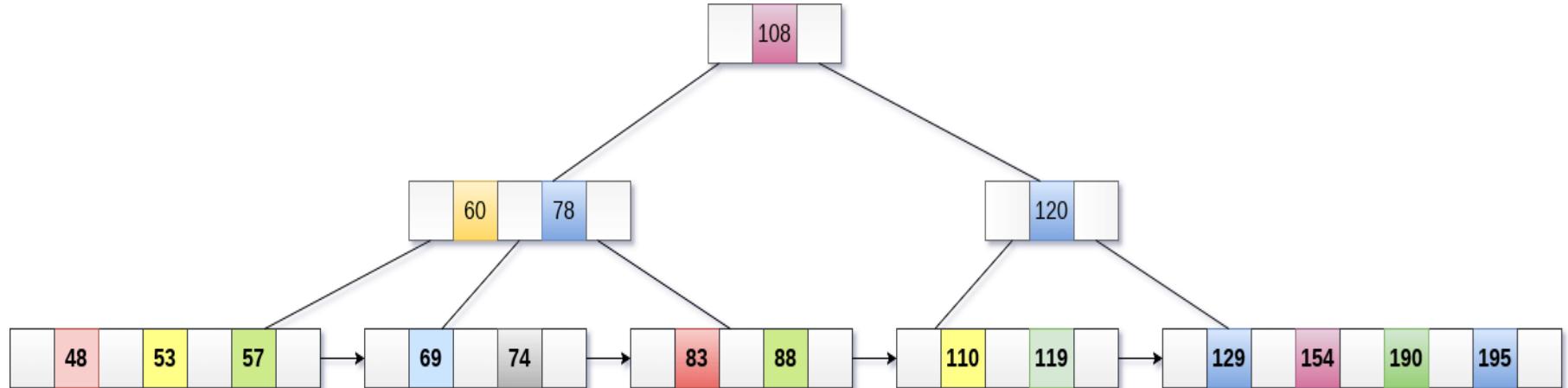
- Example
- Delete the key 200 from the B+ Tree shown in the following figure.



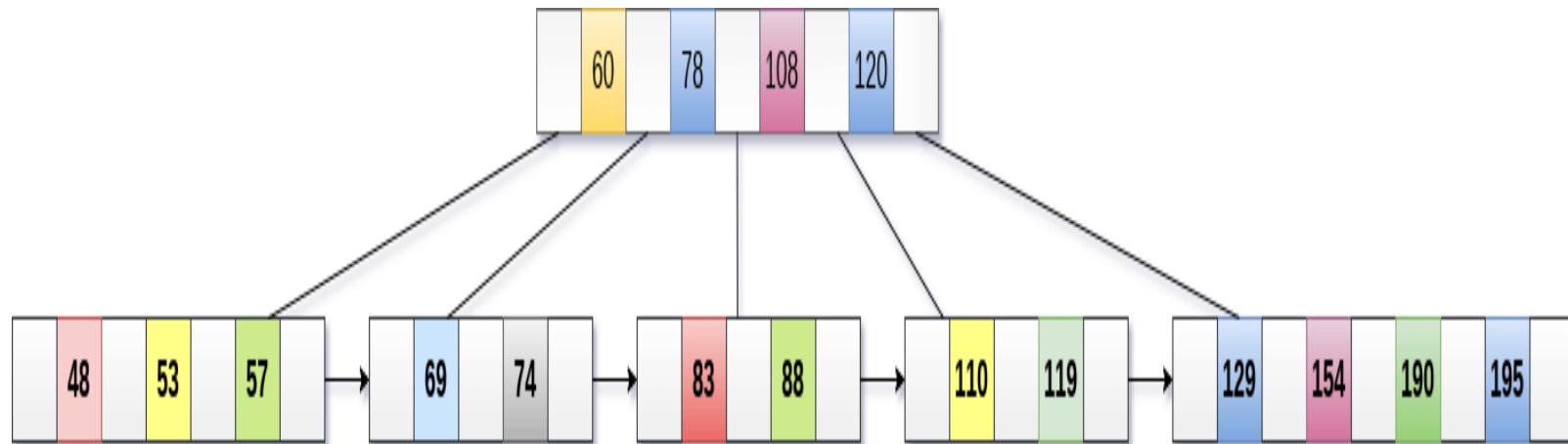
200 is present in the right sub-tree of 190, after 195. delete it.



- Merge the two nodes by using 195, 190, 154 and 129.

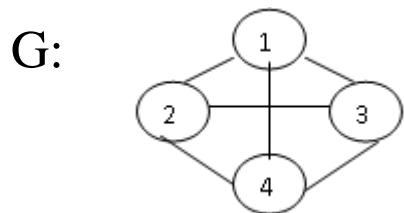


- Now, element 120 is the single element present in the node which is violating the B+ Tree properties.
- Therefore, we need to merge it by using 60, 78, 108 and 120.
- Now, the height of B+ tree will be decreased by 1.



# Graphs – Non- Linear Data Structure

- A graph  $G = (V, E)$  is composed of:
  - $V$ : set of vertices
  - $E$ : set of edges connecting the vertices in  $V$
- Vertices are referred to as nodes and the arc between the nodes are referred to as Edges.
- Each edge is a pair  $(v, w)$  where  $v, w \in V$ . (i.e.)  $v = V1, w = V2..$



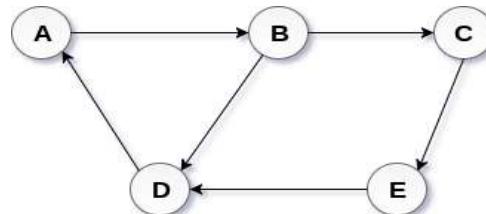
Here  $V(G) = (1, 2, 3, 4)$  and  $E(G) = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4)\}$ .

# BASIC TERMINOLOGIES

## Directed Graph (or) Digraph

Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph.

If  $(v, w)$  is a directed edge then  $(v, w) \neq (w, v)$

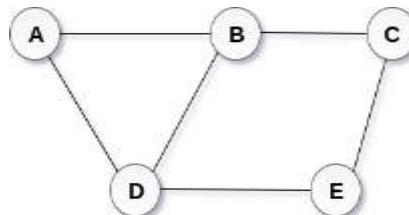


**Directed Graph**

## Undirected Graph

An undirected graph is a graph, which consists of undirected edges.

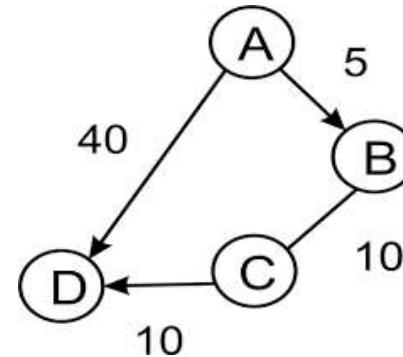
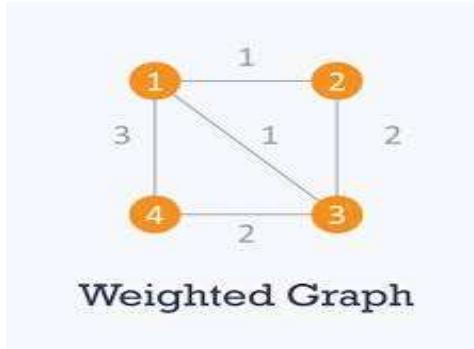
If  $(v, w)$  is an undirected edge then  $(v, w) = (w, v)$



**Undirected Graph**

## **Weighted Graph**

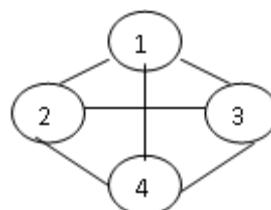
A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.



## **Complete Graph**

A complete graph is a graph in which there is an edge between every pair of vertices.

A complete graph with  $n$  vertices will have  $n(n - 1)/2$  edges.



Number of vertices is 4

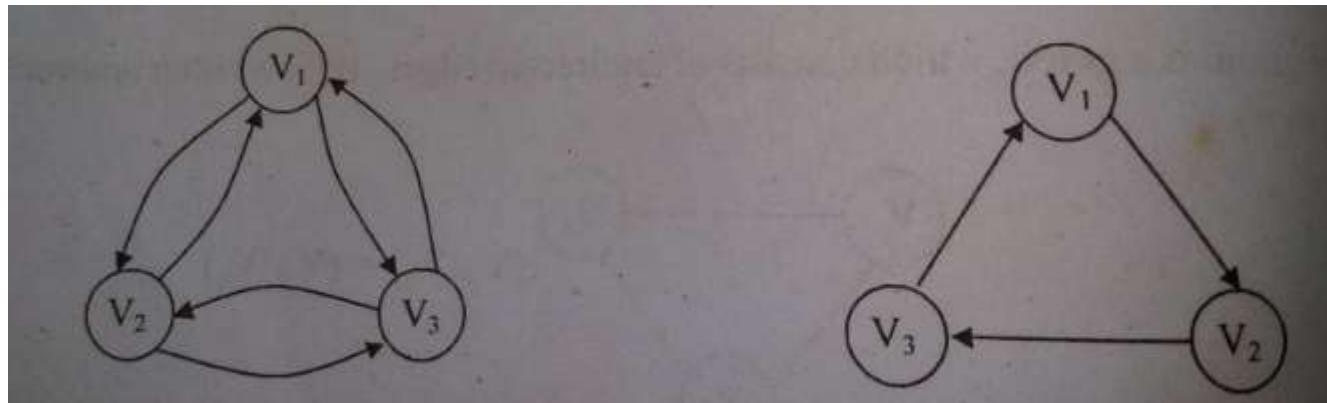
Number of edge is 6

There is a path from every vertex to every other vertex

A complete digraph is a strongly connected graph

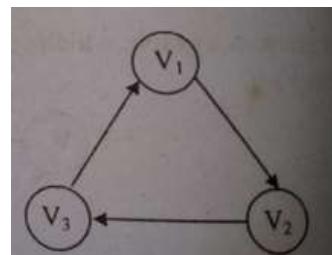
## Strongly Connected Graph

If there is a path from every vertex to every other vertex in a directed graph then it is said to be strongly connected graph. Otherwise, it is said to be weakly connected graph.



## Path

A path in a graph is a sequence of vertices w<sub>1</sub>, w<sub>2</sub>, ..., w<sub>n</sub> such that w<sub>i</sub>, w<sub>i+1</sub> ∈ E for 1 ≤ i ≤ N. The path from v<sub>1</sub> to v<sub>3</sub> is v<sub>1</sub>, v<sub>2</sub>, v<sub>3</sub>.



## **Length**

- The length of the path is the number of edges on the path, which is equal to  $N-1$ , where  $N$  represents the number of vertices.
- The length of the above path  $V_1$  to  $V_3$  is 2. (i.e)  $(V_1, V_2), (V_2, V_3)$ .
- If there is a path from a vertex to itself, with no edges, then the path length is 0.

## **Loop**

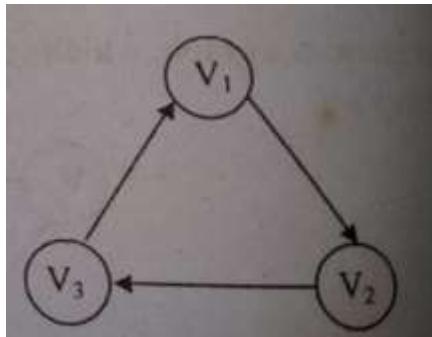
- If the graph contains an edge  $(v, v)$  from a vertex to itself, then the path is referred to as a loop.

## **Simple Path**

- A simple path is a path such that all vertices on the path, except possibly the first and the last are distinct.
- A simple cycle is the simple path of length atleast one that begins and ends at the same vertex.

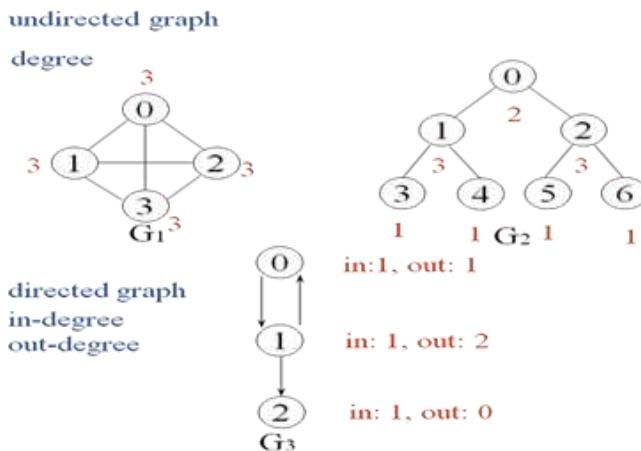
## Cycle

- A cycle in a graph is a path in which first and last vertex are the same.



## Degree

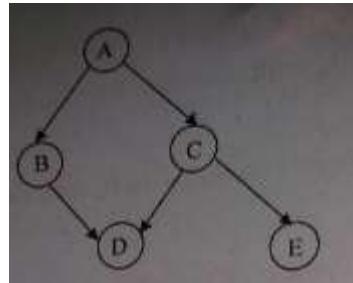
- The number of edges incident on a vertex determines its degree. The degree of the vertex V is written as  $\text{degree}(V)$ .
- The indegree of the vertex V, is the number of edges entering into the vertex V.
- Similarly the out degree of the vertex V is the number of edges exiting from that vertex V.



## **ACyclic Graph**

A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG.

DAG - Directed Acyclic Graph.



# Graph Representations

- Adjacency Matrix
- Adjacency Lists

## Adjacency Matrix

- One simple way to represents a graph is Adjacency Matrix.
- The adjacency Matrix A for a graph  $G = (V, E)$  with n vertices is an  $n \times n$  matrix, such that
- $A_{ij} = 1$ , if there is an edge  $V_i$  to  $V_j$
- $A_{ij} = 0$ , if there is no edge.

Adjacency Matrix For Directed Graph

The diagram shows a directed graph with four vertices labeled  $V_1$ ,  $V_2$ ,  $V_3$ , and  $V_4$ . The edges are directed from  $V_1$  to  $V_2$ , from  $V_1$  to  $V_3$ , from  $V_2$  to  $V_3$ , and from  $V_4$  to  $V_3$ .

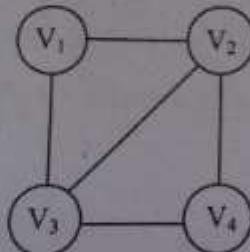
	$V_1$	$V_2$	$V_3$	$V_4$
$V_1$	0	1	1	0
$V_2$	0	0	0	1
$V_3$	0	1	0	0
$V_4$	0	0	1	0

$V_{1,2} = 1$  Since there is an edge  $V_1$  to  $V_2$

$V_{1,3} = 1$ , there is an edge  $V_1$  to  $V_3$

$V_{1,4}$  &  $V_{1,4} = 0$ , there is no edge.

### Adjacency Matrix For Undirected Graph



	V <sub>1</sub>	V <sub>2</sub>	V <sub>3</sub>	V <sub>4</sub>
V <sub>1</sub>	0	1	1	0
V <sub>2</sub>	1	0	1	1
V <sub>3</sub>	1	1	0	1
V <sub>4</sub>	0	1	1	0

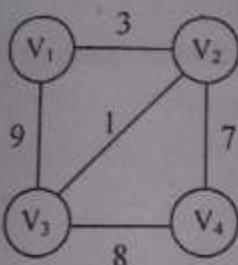
### Adjacency Matrix For Weighted Graph

To solve some graph problems, Adjacency matrix can be constructed as

$$A_{ij} = C_{ij}, \text{ if there exists an edge from } V_i \text{ to } V_j$$

$$A_{ij} = 0, \text{ if there is no edge & } i=j$$

If there is no arc from  $i$  to  $j$ , Assume  $C[i, j] = \infty$  where  $i \neq j$ .



	V <sub>1</sub>	V <sub>2</sub>	V <sub>3</sub>	V <sub>4</sub>
V <sub>1</sub>	0	3	9	$\infty$
V <sub>2</sub>	$\infty$	0	$\infty$	7
V <sub>3</sub>	$\infty$	1	0	$\infty$
V <sub>4</sub>	$\infty$	1	8	0

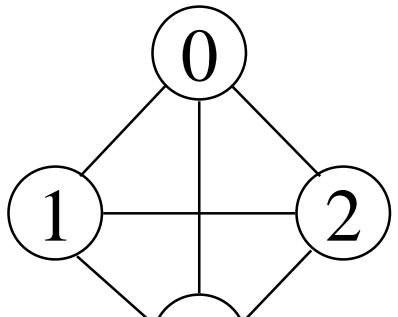
## Advantage

- \* Simple to implement.

## Disadvantage

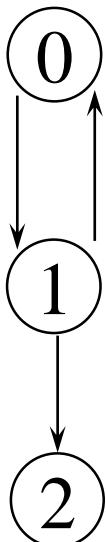
- \* Takes  $O(n^2)$  space to represents the graph
- \* It takes  $O(n^2)$  time to solve the most of the problems.

# Examples for Adjacency Matrix



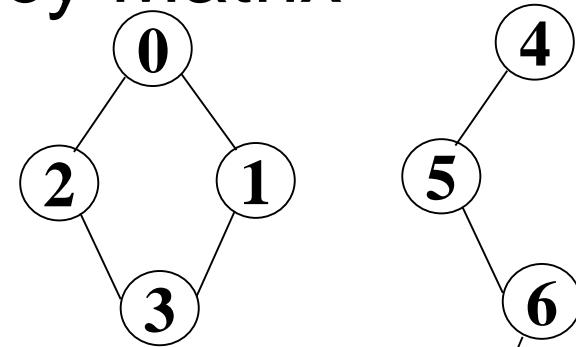
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$G_1$



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$G_2$



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$G_3$

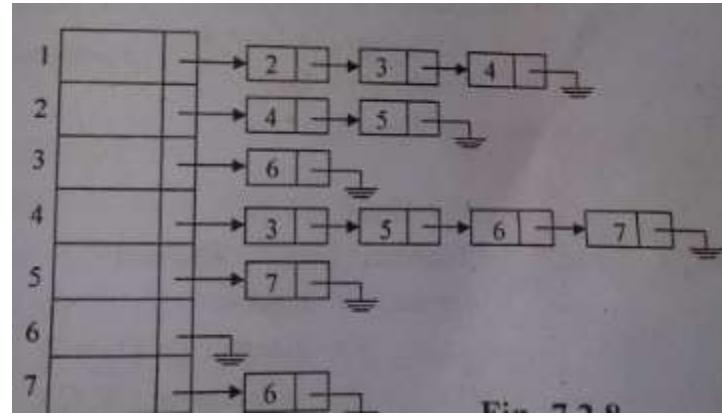
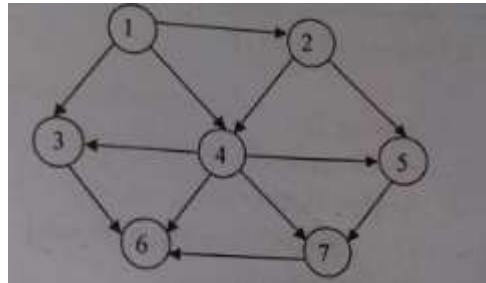
symmetric

undirected:  $n^2/2$

directed:  $n^2$

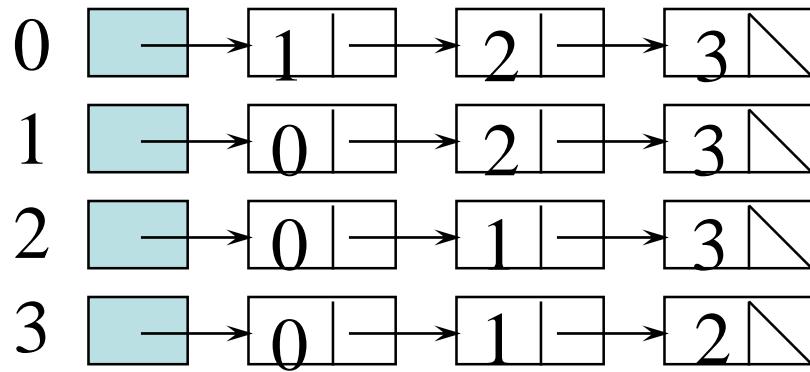
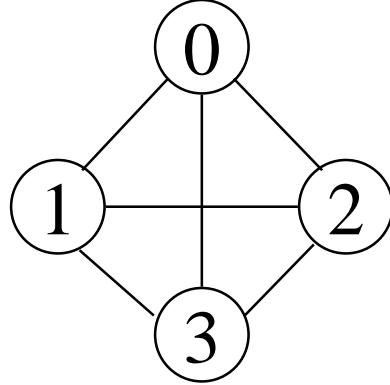
## Adjacency List Representation

- In this representation, we store a graph as a linked structure. We store all vertices in a list and then for each vertex, we have a linked list of its adjacency vertices

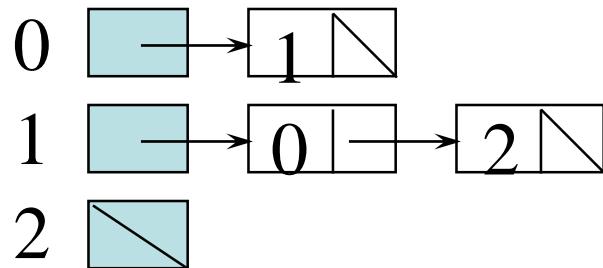
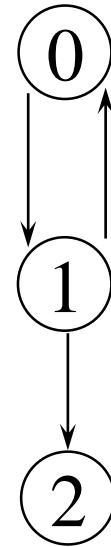


## Disadvantage

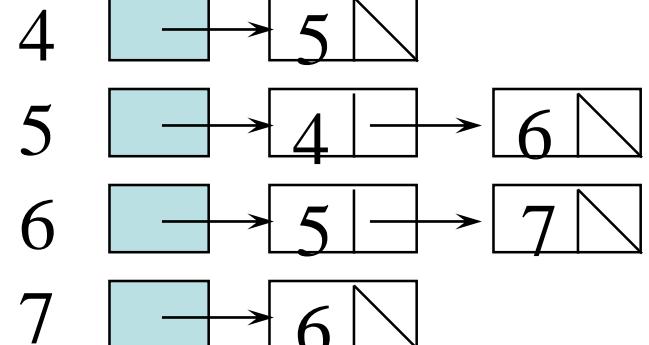
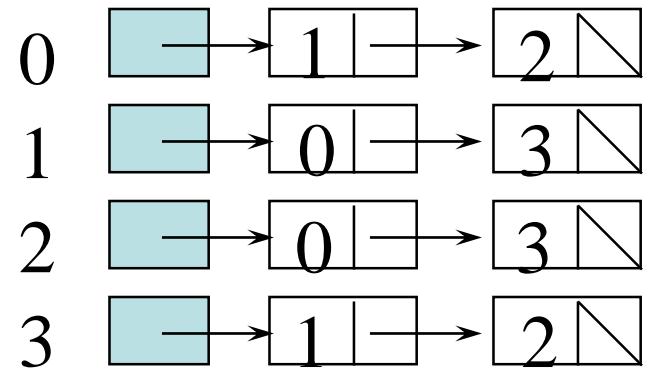
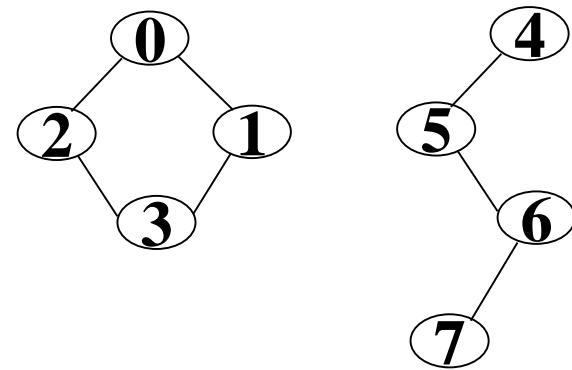
\* It takes  $O(n)$  time to determine whether there is an arc from vertex  $i$  to vertex  $j$ . Since there can be  $O(n)$  vertices on the adjacency list for vertex  $i$ .



$G_1$



$G_3$



$G_4$

An undirected graph with  $n$  vertices and  $e$  edges  $\Rightarrow n$  head nodes and  $2e$  list nodes

# Topological Sort

- A **topological sort** is a linear ordering of vertices in a directed acyclic graph such that if there is a path from  $V_i$  to  $V_j$ , then  $V_j$  appears after  $V_i$  in the linear ordering.
- Topological ordering is not possible. If the graph has a cycle, since for two vertices  $v$  and  $w$  on the cycle,  $v$  precedes  $w$  and  $w$  precedes  $v$ .

To implement the topological sort, perform the following steps.

- **Step 1** : - Find the indegree for every vertex.
- **Step 2** : - Place the vertices whose indegree is `0' on the empty queue.
- **Step 3** : - Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.
- **Step 4** : - Enqueue the vertex on the queue, if its indegree falls to zero.
- **Step 5** : - Repeat from step 3 until the queue becomes empty.
- **Step 6** : - The topological ordering is the order in which the vertices dequeued.

Example 1 :

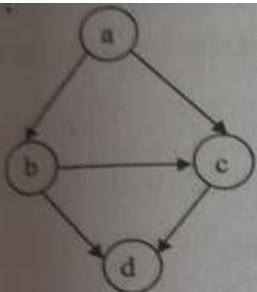


Fig. 7.3.1

	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

Adjacency Matrix

Step 1 : Number of 1's present in each column of adjacency matrix represents the indegree of the corresponding vertex.

In fig 7.3.1 Indegree [a] = 0

Indegree [b] = 1

Indegree [c] = 2

Indegree [d] = 2

Step 2 : Enqueue the vertex, whose indegree is '0'

In fig 7.3.1 vertex 'a' is 0, so place it on the queue.

Step 3 : Dequeue the vertex 'a' from the queue and decrement the indegree's of its adjacent vertex 'b' & 'c'

Hence, Indegree [b] = 0 and Indegree [c] = 1

Step 4 : Enqueue the vertex 'b' as its indegree becomes zero.

Step 5 : Dequeue the vertex 'b' from Q and decrement the indegree's of its adjacent vertex 'c' & 'd'.

Hence, Indegree [c] = 0 and Indegree [d] = 1

Step 6 : Enqueue the vertex 'c' as its indegree falls to zero.

Step 7 : Dequeue the vertex 'c' from Q and decrement the indegree's of its adjacent vertex 'd'.

Hence, Indegree [d] = 0

Step 8 : Enqueue the vertex 'd' as its indegree falls to zero.

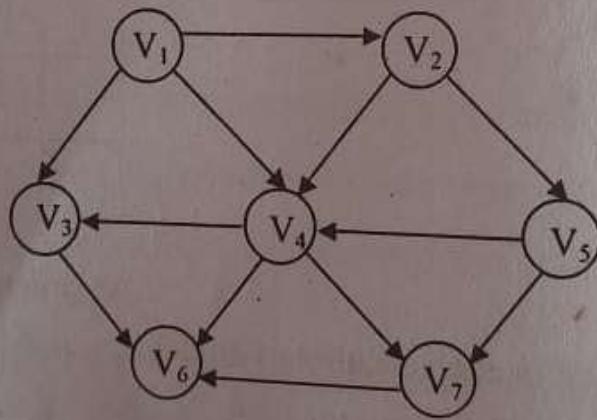
Step 9 : Dequeue the vertex 'd'

Step 7 : As the queue becomes empty, topological ordering is performed, which is nothing but the order in which the vertices are dequeued.

VERTEX	1	2	3	4
a	0	0	0	0
b	1	0	0	0
c	2	1	0	0
d	2	2	1	0
ENQUEUE	a	b	c	d
DEQUEUE	a	b	c	d

RESULT OF APPLYING TOPOLOGICAL SORT TO THE GRAPH IN FIG. 7.31

Example 2 :



## Routine to perform Topological Sort

```
/* Assume that the graph is read into an adjacency matrix and that the indegrees are
computed for every vertices and placed in an array (i.e. Indegree [ ] ) */
void Topsort (Graph G)
{
    Queue Q ;
    int counter = 0;
    Vertex V, W ;
    Q = CreateQueue (NumVertex);
    Makeempty (Q);
    for each vertex V
        if (indegree [V] == 0)
            Enqueue (V, Q);
    while (! IsEmpty (Q))
    {
        V = Dequeue (Q);
        TopNum [V] = + + counter;
        for each W adjacent to V
            if (--Indegree [W] == 0)
                Enqueue (W, Q);
    }
    if (counter != NumVertex)
        Error (" Graph has a cycle");
    DisposeQueue (Q); /* Free the Memory */
}
```

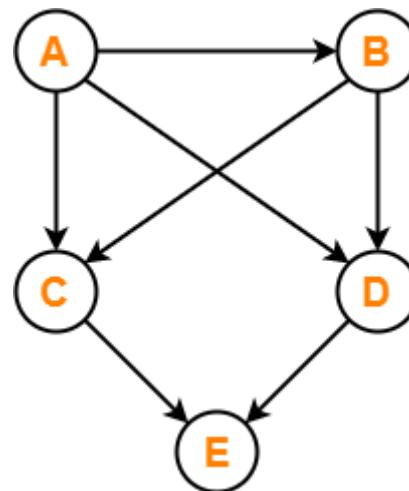
**Note :**

Enqueue ( $V, Q$ ) implies to insert a vertex  $V$  into the queue  $Q$ .

Dequeue ( $Q$ ) implies to delete a vertex from the queue  $Q$ .

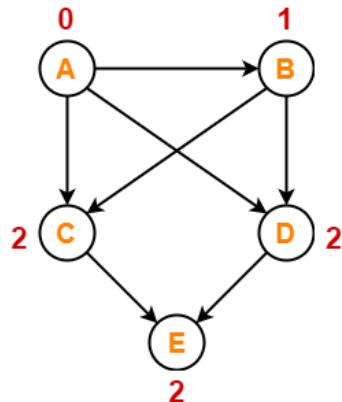
TopNum [ $V$ ] indicates an array to place the topological numbering.

Example



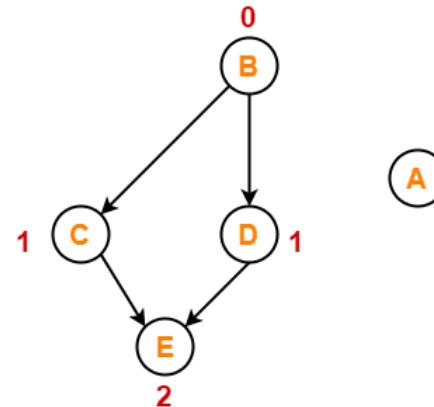
## Step-01:

Write in-degree of each vertex-



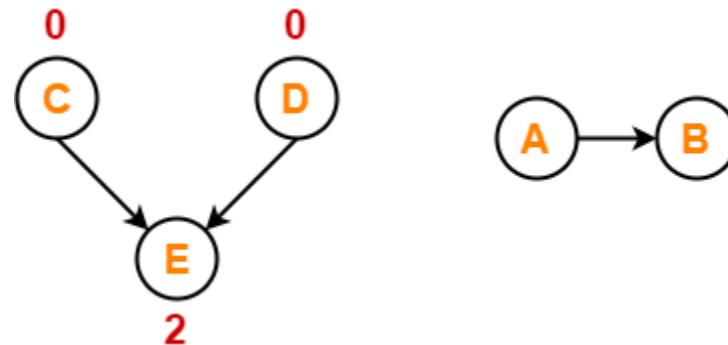
## Step-02:

- Vertex-A has the least in-degree.
- So, remove vertex-A and its associated edges.
- Now, update the in-degree of other vertices.



### Step-03:

- Vertex-B has the least in-degree.
- So, remove vertex-B and its associated edges.
- Now, update the in-degree of other vertices.



### Step-04:

There are two vertices with the least in-degree. So, following 2 cases are possible-

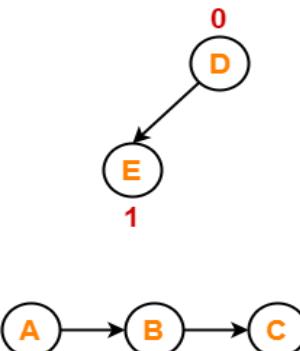
In case-01,

- Remove vertex-C and its associated edges.
- Then, update the in-degree of other vertices.

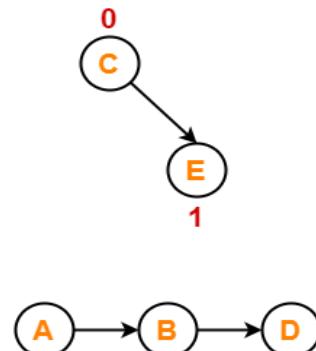
In case-02,

- Remove vertex-D and its associated edges.
- Then, update the in-degree of other vertices.

Case-01



Case-02



## Step-05:

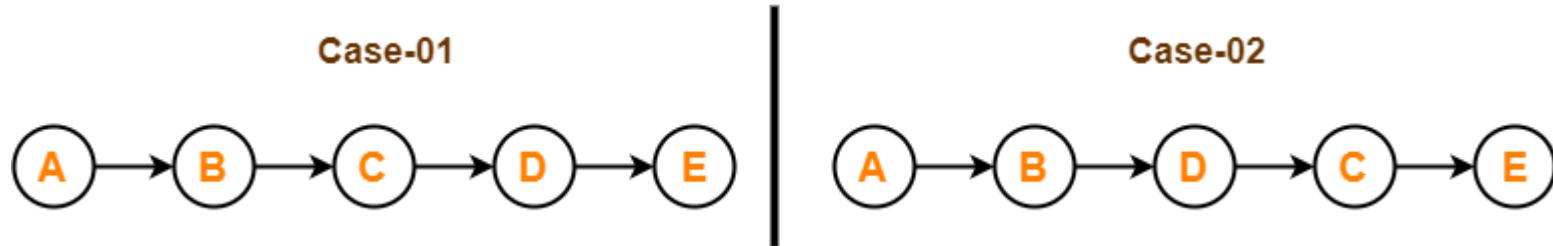
Now, the above two cases are continued separately in the similar manner.

In case-01,

- Remove vertex-D since it has the least in-degree.
- Then, remove the remaining vertex-E.

In case-02,

- Remove vertex-C since it has the least in-degree.
- Then, remove the remaining vertex-E.



## Conclusion-

For the given graph, following **2** different topological orderings are possible-

- A B C D E
- A B D C E

## Graph Traversal

- A graph traversal is a systematic way of visiting the nodes in a specific order.
  - Depth First Traversal
  - Breadth first Traversal
- 
- The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph.
  - In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one.
  - After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

- Step1 : choose any node in the graph, designate it as the search node & mark it as visited
- Step 2: Using the adjacency matrix of the graph, find all the unvisited adjacent node and enqueue them into queue Q
- Step 3: then the node is dequeued from the queue. Mark that node as visited and designate it as the new search node
- Step 4: Repeat the step 2 and 3 using the new search node
- Step 5: this process continues until the queue Q which keeps track of the adjacent nodes is empty.

Example:

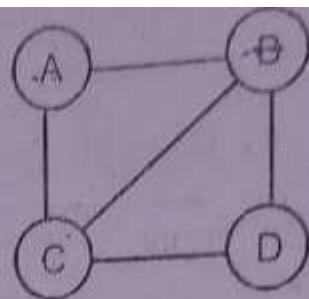


Figure 7.4.1

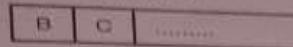
Adjacency matrix

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

Figure 7.4.2

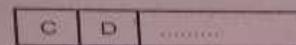
~~Implementation~~

1. Let 'A' be the source vertex. Mark it to as visited.
2. Find the adjacent unvisited vertices of 'A' and enqueue them into the queue.  
Here B and C are adjacent nodes of A



and B and C are enqueued.

3. Then vertex 'B' is dequeued and its adjacent vertices C and D are taken from the adjacency matrix for enqueueing. Since vertex C is already in the queue, vertex D alone is enqueued.



Here B is dequeued, D is enqueued.

4. Then vertex 'C' is dequeued and its adjacent vertices A, B and D are found out. Since vertices A and B are already visited and vertex D is also in the queue, no enqueue operation takes place.



Here C is dequeued

5. Then vertex 'D' is dequeued. This process terminates as all the vertices are visited and the queue is also empty.

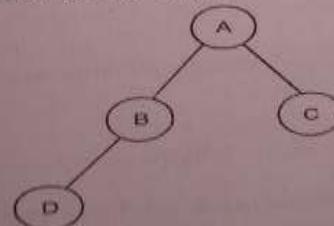


Figure 7.4.3: Breadth first spanning tree

~~Applications of breadth first search~~

1. To check whether the graph is connected or not.

Algorithm BFS gives the details.

**Procedure** BFS( $v$ )

//A breadth first search of  $G$  is carried out beginning at vertex  $v$ . All vertices visited are marked as VISITED( $i$ ) = 1. The graph  $G$  and array VISITED are global and VISITED is initialised to 0.//

VISITED( $v$ )  $\leftarrow 1$

**Initialise Q to be empty //Q is a queue//**

loop

for all vertices  $w$  adjacent to  $v$  do

if VISITED( $w$ ) = 0 //add  $w$  to queue//

then [call ADDQ( $w$ , Q); VISITED( $w$ )  $\leftarrow 1$ ] //mark  $w$  as VISITED//

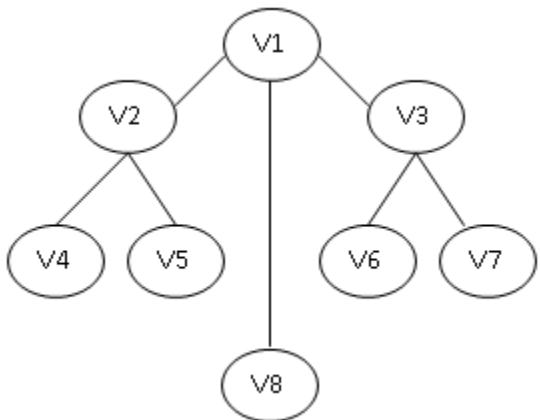
end

if Q is empty then return

call DELETEQ( $v$ , Q)

forever

end BFS



Let us consider the same example, given in figure. We start say, with  $V_1$ . Its adjacent vertices are  $V_2, V_8, V_3$ . we visit all one by one. We pick on one of these, say  $V_2$ . The unvisited adjacent vertices to  $V_2$  are  $V_4, V_5$ . we visit both . we back to the remaining visited vertices of  $V_1$  and pick on one of this, say  $V_3$ . The unvisited adjacent vertices to  $V_3$  are  $V_6, V_7$ . There are no more unvisited adjacent vertices of  $V_8, V_4, V_5, V_6$  and  $V_7$ . Thus the sequence so generated is  $V_1, V_2, V_8, V_3, V_4, V_5, V_6, V_7$ .

## **COMPUTING TIME:**

Each vertex visited gets into the queue exactly once, so the **loop forever** is iterated at most  $n$  times. If an adjacency matrix is used, then the for loop takes  $O(n)$  time for each vertex visited. The total time is, therefore,  $O(n^2)$ . In case adjacency lists are used the for loop as a total cost of  $d_1 + \dots + d_n = O(e)$  where  $d_i = \text{degree}(v_i)$ . Again, all vertices visited. Together with all edges incident to from a connected component of  $G$ .

## **DEPTH FIRST SEARCH :**

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

This procedure is best described recursively as in

Procedure DFS(v)

// Given an undirected graph  $G = (V, E)$  with  $n$  vertices and an array visited ( $n$ ) initially set to zero . This algorithm visits all vertices reachable from  $v$  . $G$  and VISITED are global > //VISITED ( $v$ )  $\leftarrow 1$

for each vertex  $w$  adjacent to  $v$  do

if VISITED ( $w$ ) =0 then call DFS ( $w$ )

end

**end** DFS

The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth first search must be restarted at any one of them.

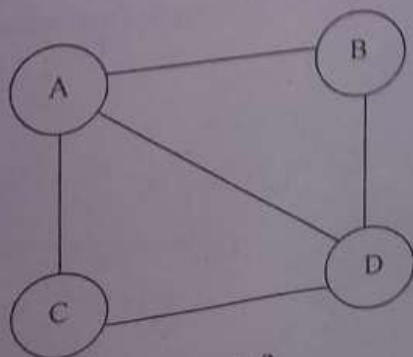
To implement the Depthfirst Search perform the following Steps :

- Step : 1 Choose any node in the graph. Designate it as the search node and mark it as visited.
- Step : 2 Using the adjacency matrix of the graph, find a node adjacent to the search node that has not been visited yet. Designate this as the new search node and mark it as visited.
- Step : 3 Repeat step 2 using the new search node. If no nodes satisfying (2) can be found, return to the previous search node and continue from there.
- Step : 4 When a return to the previous search node in (3) is impossible, the search from the originally chosen search node is complete.
- Step : 5 If the graph still contains unvisited nodes, choose any node that has not been visited and repeat step (1) through (4).

## ROUTINE FOR DEPTH FIRST SEARCH

```
Void DFS (Vertex V)
{
    visited [V] = True;
    for each W adjacent to V
        if (! visited [W])
            DFS (W);
}
```

Example:-

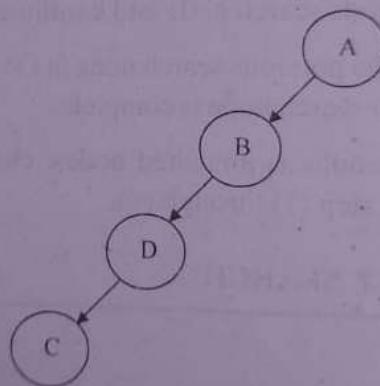


## Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

### Implementation

1. Let 'A' be the source vertex. Mark it to be visited.
  2. Find the immediate adjacent unvisited vertex 'B' of 'A' Mark it to be visited.
  3. From 'B' the next adjacent vertex is 'd' Mark it has visited.
  4. From 'D' the next unvisited vertex is 'C' Mark it to be visited.



## Depth First Spanning Tree

## Applications of Depth First Search

1. To check whether the undirected graph is connected or not.
  2. To check whether the connected undirected graph is Bioconnected or not.
  3. To check the Acyclicity of the directed graph.

# Example

Let us start with  $V_1$ .

**Its adjacent vertices are  $V_2$ ,  $V_8$ , and  $V_3$ . Let us pick on  $v_2$ .**

Its adjacent vertices are  $V_1$ ,  $V_4$ ,  $V_5$ ,  $V_1$  is already visited . let us pick on  $V_4$ .

Its adjacent vertices are  $V_2$ ,  $V_8$ .

$V_2$  is already visited .let us visit  $V_8$ .

Its adjacent vertices are  $V_4$ ,  $V_5$ ,  $V_1$ ,  $V_6$ ,  $V_7$ .

$V_4$  and  $V_1$  are visited. Let us traverse  $V_5$ .

Its adjacent vertices are  $V_2$ ,  $V_8$ . Both are already visited therefore, we back track.

We had  $V_6$  and  $V_7$  unvisited in the list of  $V_8$ . We may visit any. We may visit any. We visit  $V_6$ .

Its adjacent are  $V_8$  and  $V_3$ . Obviously the choice is  $V_3$ .

Its adjacent vertices are  $V_1$ ,  $V_7$  . We visit  $V_7$ .

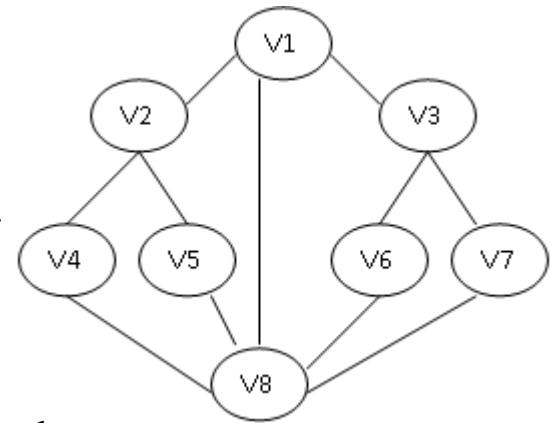
All the adjacent vertices of  $V_7$  are already visited, we back track and find that we have visited all the vertices.

**Therefore the sequence of traversal is**

$V_1, V_2, V_4, V_5, V_6, V_3, V_7$ .

This is not a unique or the only sequence possible using this traversal method.

We may implement the Depth First search by using a stack,pushing all unvisited vertices to the one just visited and poping the stack to find the next vertex to visit.



## **COMPUTING TIME:**

In case G is represented by adjacency lists then the vertices w adjacent to v can be determined by following a chain of links. Since the algorithm DFS would examine each node in the adjacency lists at most once and there are  $2e$  list nodes. The time to complete the search is  $O(e)$ . if G is represented by its adjacency matrix. Then the time to determine all vertices adjacent to v is  $O(n)$ . since at most n vertices are visited. The total time is  $O(n^2)$ .

# Application of Graphs

- Graphs are used to define the **flow of computation**.
- Graphs are used to represent **networks of communication**.
- Graphs are used to represent **data organization**.
- Graph transformation systems work on rule-based in-memory manipulation of graphs. Graph databases ensure **transaction-safe, persistent storing and querying of graph structured data**.
- Graph theory is used to find **shortest path in road or a network**.

- In Computer science graphs are used to represent the flow of computation.
- Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.

- In World Wide Web, web pages are considered to be the vertices. There is an edge from a page  $u$  to other page  $v$  if there is a link of page  $v$  on page  $u$ . This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.
- In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

- **Neural networks:** Vertices represent neurons and edges are the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about  $10^{11}$  neurons and close to  $10^{15}$  synapses.

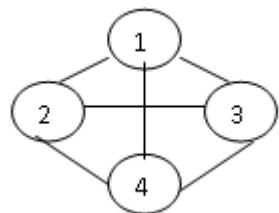
# Application of Trees

- Store hierarchical data, like folder structure, organization structure, XML/HTML data.
- Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
- Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
- B-Tree and B+ Tree : They are used to implement indexing in databases.
- Syntax Tree: Compilers use a syntax tree to validate the syntax of every program you write.

# Graphs – Non- Linear Data Structure

- A graph  $G = (V, E)$  is composed of:
  - $V$ : set of vertices
  - $E$ : set of edges connecting the vertices in  $V$
- Vertices are referred to as nodes and the arc between the nodes are referred to as Edges.

$G:$



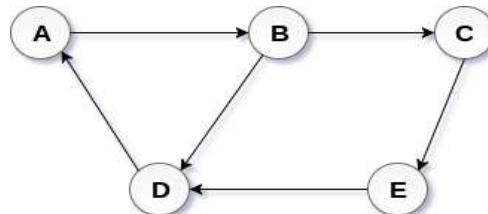
Here  $V(G) = \{1, 2, 3, 4\}$  and  $E(G) = \{(1,2), (1,3), (1,4), (2,3), (2,4)\}$ .

# BASIC TERMINOLOGIES

## Directed Graph (or) Digraph

Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph.

If  $(v, w)$  is a directed edge then  $(v, w) \neq (w, v)$

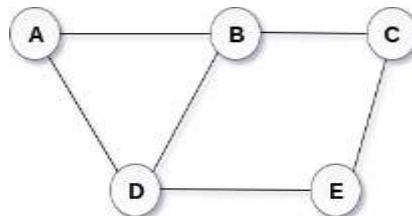


Directed Graph

## Undirected Graph

An undirected graph is a graph, which consists of undirected edges.

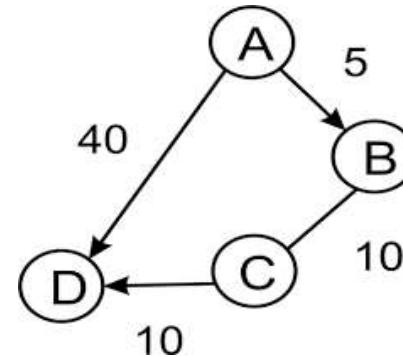
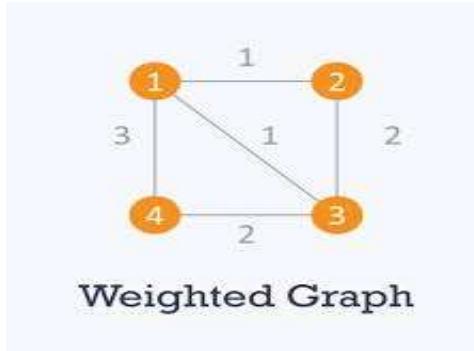
If  $(v, w)$  is an undirected edge then  $(v, w) = (w, v)$



Undirected Graph

## **Weighted Graph**

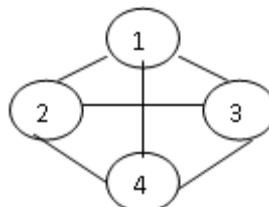
A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.



## **Complete Graph**

A complete graph is a graph in which there is an edge between every pair of vertices.

A complete graph with  $n$  vertices will have  $n(n - 1)/2$  edges.



Number of vertices is 4

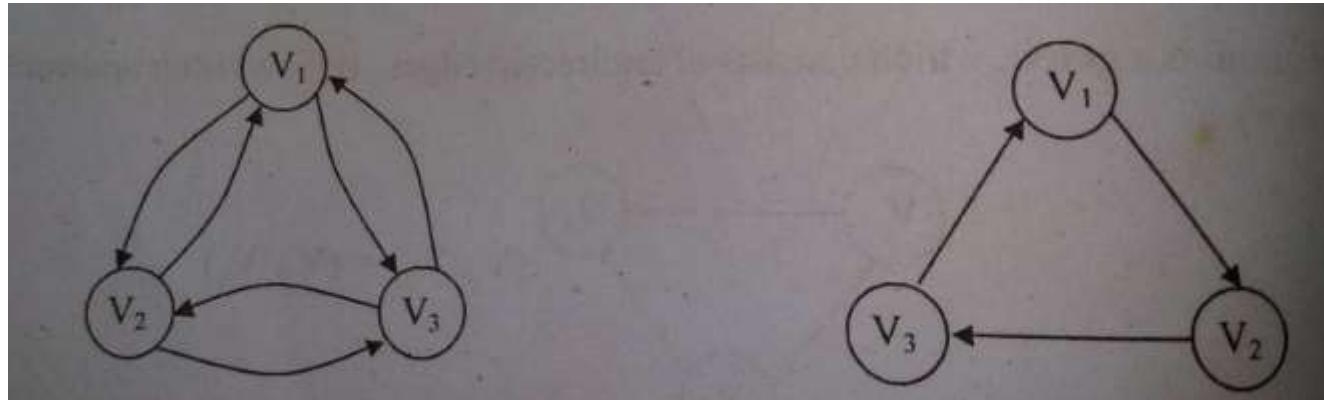
Number of edge is 6

There is a path from every vertex to every other vertex

A complete digraph is a strongly connected graph

## **Strongly Connected Graph**

If there is a path from every vertex to every other vertex in a directed graph then it is said to be strongly connected graph. Otherwise, it is said to be weakly connected graph.

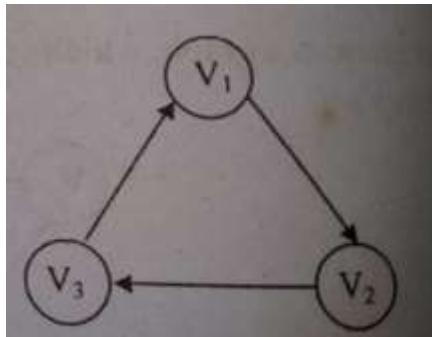


## **Loop**

- If the graph contains an edge  $(v, v)$  from a vertex to itself, then the path is referred to as a loop.

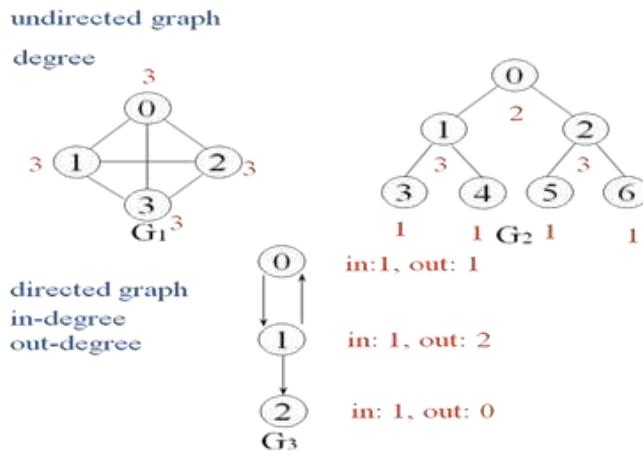
## Cycle

- A cycle in a graph is a path in which first and last vertex are the same.



## Degree

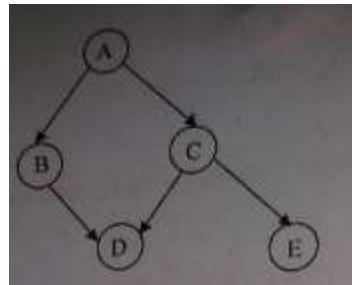
- The number of edges incident on a vertex determines its degree. The degree of the vertex V is written as  $\text{degree}(V)$ .
- The indegree of the vertex V, is the number of edges entering into the vertex V.
- Similarly the out degree of the vertex V is the number of edges exiting from that vertex V.



## **ACyclic Graph**

A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG.

DAG - Directed Acyclic Graph.



# Graph Representations

- Adjacency Matrix
- Adjacency Lists

## Adjacency Matrix

- One simple way to represents a graph is Adjacency Matrix.
- The adjacency Matrix A for a graph  $G = (V, E)$  with n vertices is an  $n \times n$  matrix, such that
- $A_{ij} = 1$ , if there is an edge  $V_i$  to  $V_j$
- $A_{ij} = 0$ , if there is no edge.

**Adjacency Matrix For Directed Graph**

The diagram shows a directed graph with four vertices labeled  $V_1$ ,  $V_2$ ,  $V_3$ , and  $V_4$ . The edges are directed from  $V_1$  to  $V_2$ , from  $V_1$  to  $V_3$ , from  $V_2$  to  $V_3$ , and from  $V_4$  to  $V_3$ .

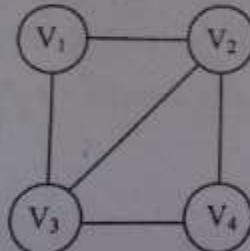
	$V_1$	$V_2$	$V_3$	$V_4$
$V_1$	0	1	1	0
$V_2$	0	0	0	1
$V_3$	0	1	0	0
$V_4$	0	0	1	0

$V_{1,2} = 1$  Since there is an edge  $V_1$  to  $V_2$

$V_{1,3} = 1$ , there is an edge  $V_1$  to  $V_3$

$V_{1,4}$  &  $V_{1,4} = 0$ , there is no edge.

### Adjacency Matrix For Undirected Graph



	V <sub>1</sub>	V <sub>2</sub>	V <sub>3</sub>	V <sub>4</sub>
V <sub>1</sub>	0	1	1	0
V <sub>2</sub>	1	0	1	1
V <sub>3</sub>	1	1	0	1
V <sub>4</sub>	0	1	1	0

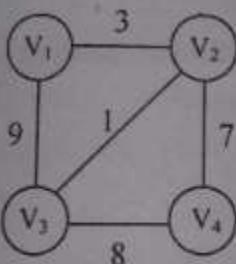
### Adjacency Matrix For Weighted Graph

To solve some graph problems, Adjacency matrix can be constructed as

$$A_{ij} = C_{ij}, \text{ if there exists an edge from } V_i \text{ to } V_j$$

$$A_{ij} = 0, \text{ if there is no edge & } i=j$$

If there is no arc from i to j, Assume C[i, j] =  $\infty$  where  $i \neq j$ .



	V <sub>1</sub>	V <sub>2</sub>	V <sub>3</sub>	V <sub>4</sub>
V <sub>1</sub>	0	3	9	$\infty$
V <sub>2</sub>	$\infty$	0	$\infty$	7
V <sub>3</sub>	$\infty$	1	0	$\infty$
V <sub>4</sub>	$\infty$	1	8	0

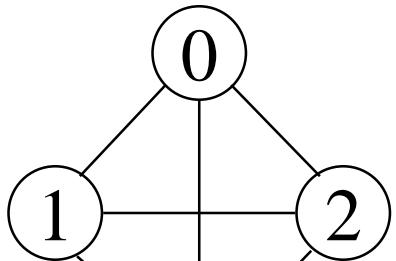
## Advantage

- \* Simple to implement.

## Disadvantage

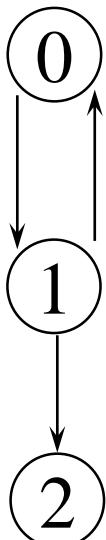
- \* Takes O(n<sup>2</sup>) space to represents the graph
- \* It takes O(n<sup>2</sup>) time to solve the most of the problems.

# Examples for Adjacency Matrix



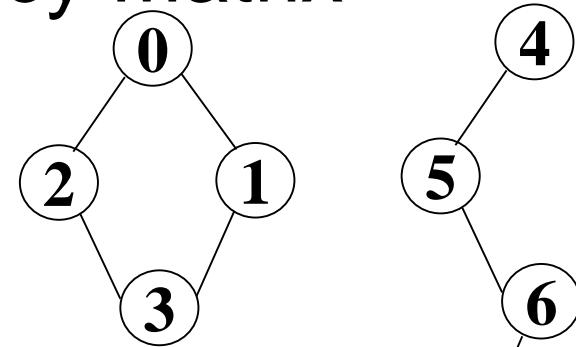
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$G_1$



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$G_2$



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$G_4$

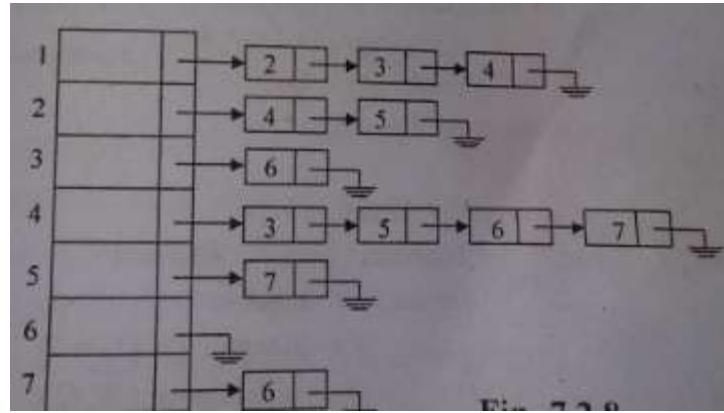
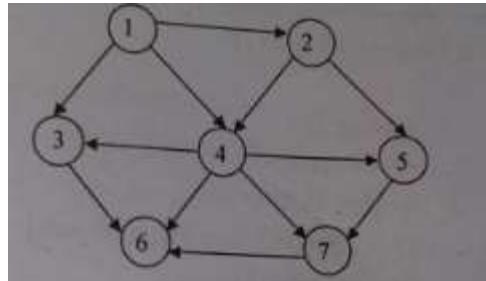
symmetric

undirected:  $n^2/2$

directed:  $n^2$

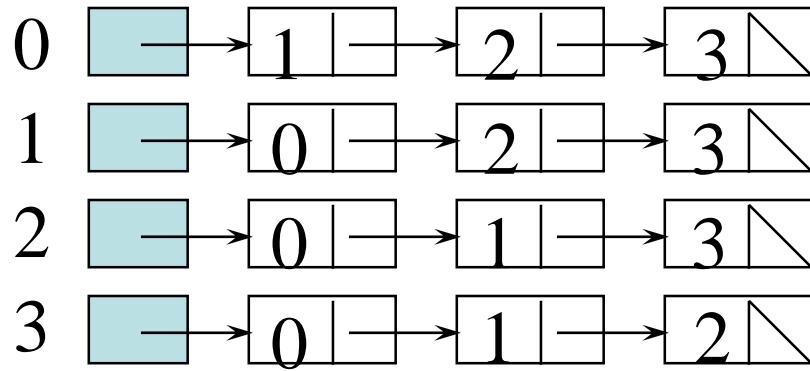
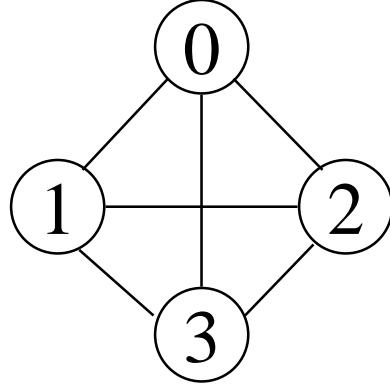
## Adjacency List Representation

- In this representation, we store a graph as a linked structure. We store all vertices in a list and then for each vertex, we have a linked list of its adjacency vertices

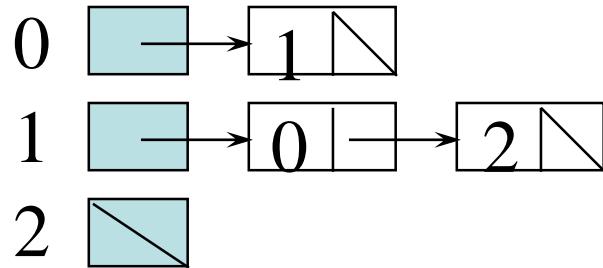


## Disadvantage

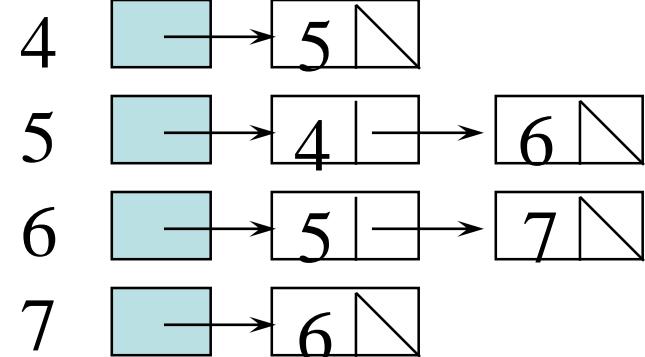
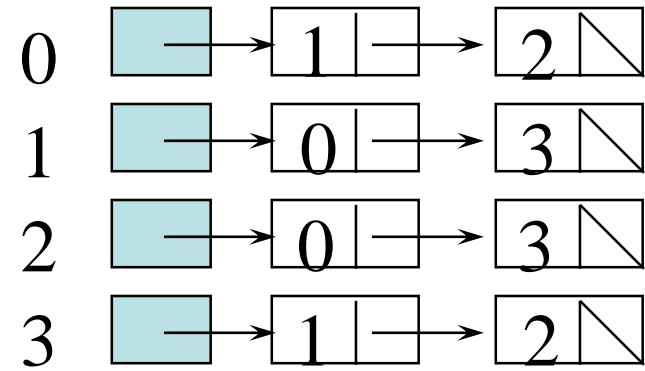
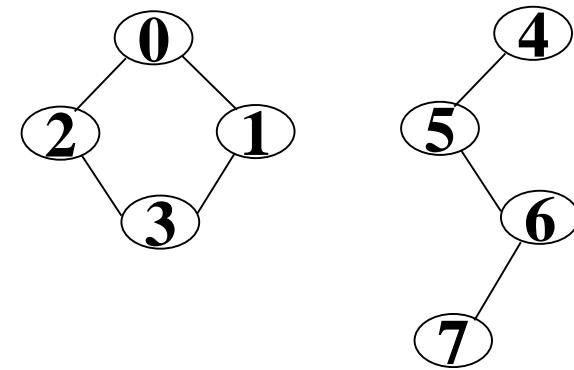
\* It takes  $O(n)$  time to determine whether there is an arc from vertex  $i$  to vertex  $j$ . Since there can be  $O(n)$  vertices on the adjacency list for vertex  $i$ .



$G_1$



$G_3$



An undirected graph with  $n$  vertices and  $e$  edges  $\Rightarrow n$  head nodes and  $2e$  list nodes

# Topological Sort

- A **topological sort** is a linear ordering of vertices in a directed acyclic graph such that if there is a path from  $V_i$  to  $V_j$ , then  $V_j$  appears after  $V_i$  in the linear ordering.
- Topological ordering is not possible. If the graph has a cycle, since for two vertices  $v$  and  $w$  on the cycle,  $v$  precedes  $w$  and  $w$  precedes  $v$ .

To implement the topological sort, perform the following steps.

- **Step 1** : - Find the indegree for every vertex.
- **Step 2** : - Place the vertices whose indegree is `0' on the empty queue.
- **Step 3** : - Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.
- **Step 4** : - Enqueue the vertex on the queue, if its indegree falls to zero.
- **Step 5** : - Repeat from step 3 until the queue becomes empty.
- **Step 6** : - The topological ordering is the order in which the vertices dequeued.

Example 1 :

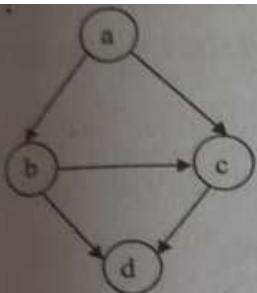


Fig. 7.3.1

	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

Adjacency Matrix

Step 1 : Number of 1's present in each column of adjacency matrix represents the indegree of the corresponding vertex.

In fig 7.3.1 Indegree [a] = 0

Indegree [b] = 1

Indegree [c] = 2

Indegree [d] = 2

Step 2 : Enqueue the vertex, whose indegree is '0'

In fig 7.3.1 vertex 'a' is 0, so place it on the queue.

Step 3 : Dequeue the vertex 'a' from the queue and decrement the indegree's of its adjacent vertex 'b' & 'c'

Hence, Indegree [b] = 0 and Indegree [c] = 1

Step 4 : Enqueue the vertex 'b' as its indegree becomes zero.

Step 5 : Dequeue the vertex 'b' from Q and decrement the indegree's of its adjacent vertex 'c' & 'd'.

Hence, Indegree [c] = 0 and Indegree [d] = 1

Step 6 : Enqueue the vertex 'c' as its indegree falls to zero.

Step 7 : Dequeue the vertex 'c' from Q and decrement the indegree's of its adjacent vertex 'd'.

Hence, Indegree [d] = 0

Step 8 : Enqueue the vertex 'd' as its indegree falls to zero.

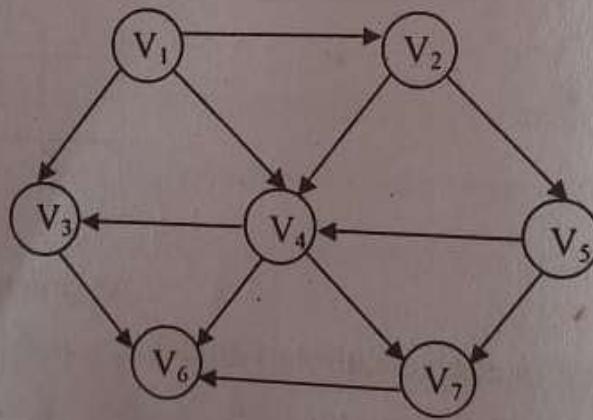
Step 9 : Dequeue the vertex 'd'

Step 7 : As the queue becomes empty, topological ordering is performed, which is nothing but the order in which the vertices are dequeued.

VERTEX	1	2	3	4
a	0	0	0	0
b	1	0	0	0
c	2	1	0	0
d	2	2	1	0
ENQUEUE	a	b	c	d
DEQUEUE	a	b	c	d

RESULT OF APPLYING TOPOLOGICAL SORT TO THE GRAPH IN FIG. 7.31

Example 2 :



## Routine to perform Topological Sort

```
/* Assume that the graph is read into an adjacency matrix and that the indegrees are
computed for every vertices and placed in an array (i.e. Indegree [ ] ) */
void Topsort (Graph G)
{
    Queue Q ;
    int counter = 0;
    Vertex V, W ;
    Q = CreateQueue (NumVertex);
    Makeempty (Q);
    for each vertex V
        if (indegree [V] == 0)
            Enqueue (V, Q);
    while (! IsEmpty (Q))
    {
        V = Dequeue (Q);
        TopNum [V] = ++ counter;
        for each W adjacent to V
            if (--Indegree [W] == 0)
                Enqueue (W, Q);
    }
    if (counter != NumVertex)
        Error (" Graph has a cycle");
    DisposeQueue (Q); /* Free the Memory */
}
```

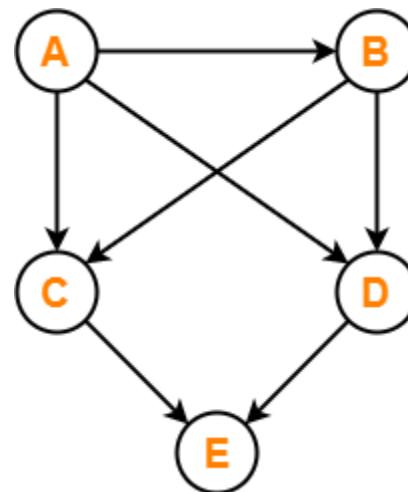
## Note :

Enqueue ( $V, Q$ ) implies to insert a vertex  $V$  into the queue  $Q$ .

Dequeue ( $Q$ ) implies to delete a vertex from the queue  $Q$ .

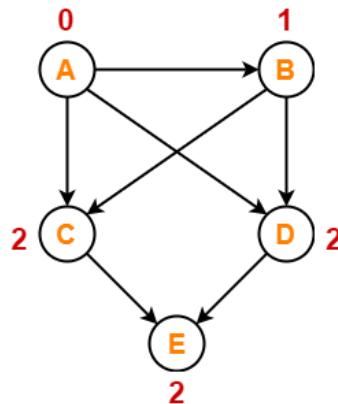
TopNum [ $V$ ] indicates an array to place the topological numbering.

Example



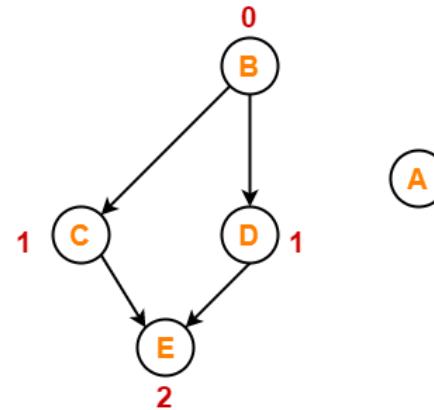
## Step-01:

Write in-degree of each vertex-



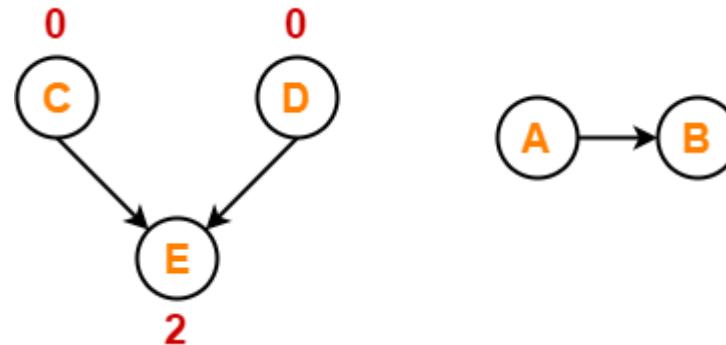
## Step-02:

- Vertex-A has the least in-degree.
- So, remove vertex-A and its associated edges.
- Now, update the in-degree of other vertices.



### Step-03:

- Vertex-B has the least in-degree.
- So, remove vertex-B and its associated edges.
- Now, update the in-degree of other vertices.



### Step-04:

There are two vertices with the least in-degree. So, following 2 cases are possible-

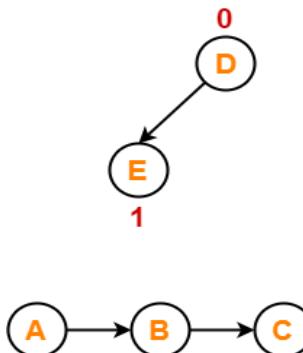
In case-01,

- Remove vertex-C and its associated edges.
- Then, update the in-degree of other vertices.

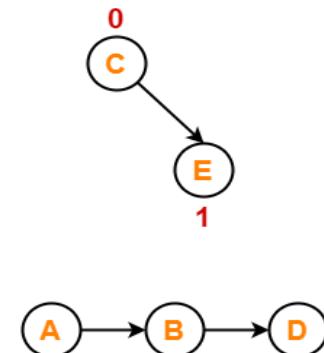
In case-02,

- Remove vertex-D and its associated edges.
- Then, update the in-degree of other vertices.

Case-01



Case-02



## Step-05:

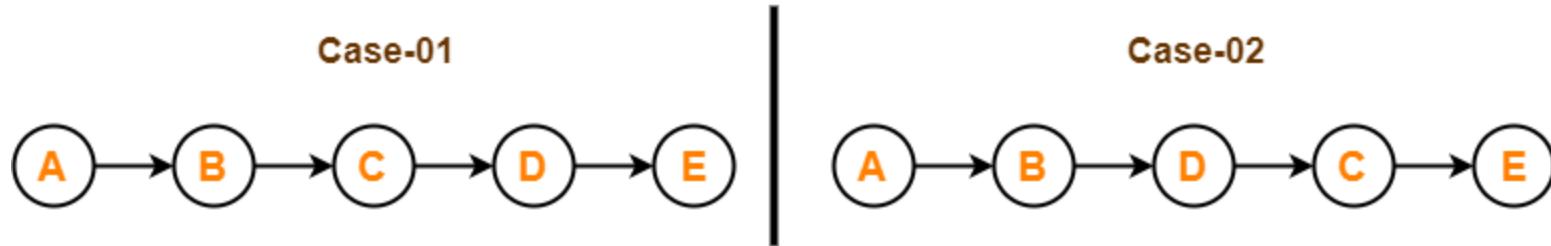
Now, the above two cases are continued separately in the similar manner.

In case-01,

- Remove vertex-D since it has the least in-degree.
- Then, remove the remaining vertex-E.

In case-02,

- Remove vertex-C since it has the least in-degree.
- Then, remove the remaining vertex-E.



## Conclusion-

For the given graph, following **2** different topological orderings are possible-

- A B C D E
- A B D C E

## Graph Traversal

- A graph traversal is a systematic way of visiting the nodes in a specific order.
  - Depth First Traversal
  - Breadth first Traversal
- 
- The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph.
  - In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one.
  - After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

- Step1 : choose any node in the graph, designate it as the search node & mark it as visited
- Step 2: Using the adjacency matrix of the graph, find all the unvisited adjacent node and enqueue them into queue Q
- Step 3: then the node is dequeued from the queue. Mark that node as visited and designate it as the new search node
- Step 4: Repeat the step 2 and 3 using the new search node
- Step 5: this process continues until the queue Q which keeps track of the adjacent nodes is empty.

Example:

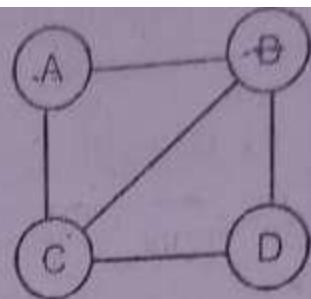


Figure 7.4.1

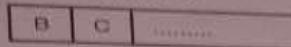
Adjacency matrix

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

Figure 7.4.2

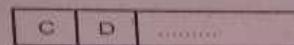
~~Implementation~~

1. Let 'A' be the source vertex. Mark it to as visited.
2. Find the adjacent unvisited vertices of 'A' and enqueue them into the queue.  
Here B and C are adjacent nodes of A



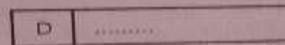
and B and C are enqueued.

3. Then vertex 'B' is dequeued and its adjacent vertices C and D are taken from the adjacency matrix for enqueueing. Since vertex C is already in the queue, vertex D alone is enqueued.



Here B is dequeued, D is enqueued.

4. Then vertex 'C' is dequeued and its adjacent vertices A, B and D are found out. Since vertices A and B are already visited and vertex D is also in the queue, no enqueue operation takes place.



Here C is dequeued

5. Then vertex 'D' is dequeued. This process terminates as all the vertices are visited and the queue is also empty.

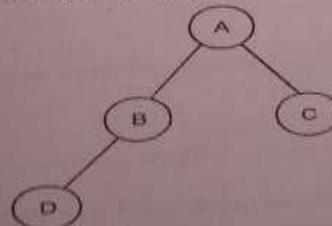


Figure 7.4.3: Breadth first spanning tree

~~Applications of breadth first search~~

1. To check whether the graph is connected or not.

Algorithm BFS gives the details.

**Procedure** BFS( $v$ )

//A breadth first search of  $G$  is carried out beginning at vertex  $v$ . All vertices visited are marked as VISITED( $i$ ) = 1. The graph  $G$  and array VISITED are global and VISITED is initialised to 0.//

VISITED( $v$ )  $\leftarrow 1$

**Initialise Q to be empty //Q is a queue//**

loop

for all vertices  $w$  adjacent to  $v$  do

if VISITED( $w$ ) = 0 //add  $w$  to queue//

then [call ADDQ( $w$ , Q); VISITED( $w$ )  $\leftarrow 1$ ] //mark  $w$  as VISITED//

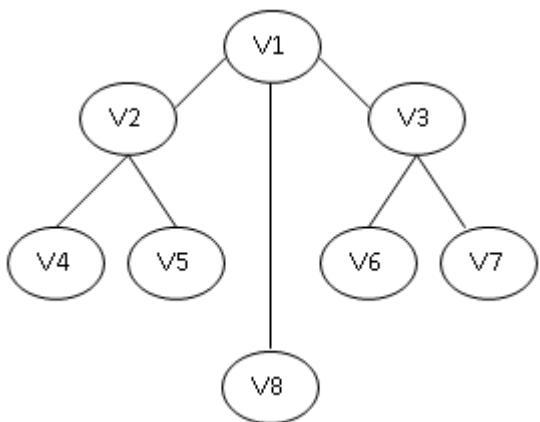
end

if Q is empty then return

call DELETEQ( $v$ , Q)

forever

end BFS



Let us consider the same example, given in figure. We start say, with  $V_1$ . Its adjacent vertices are  $V_2, V_8, V_3$ . we visit all one by one. We pick on one of these, say  $V_2$ . The unvisited adjacent vertices to  $V_2$  are  $V_4, V_5$ . we visit both . we back to the remaining visited vertices of  $V_1$  and pick on one of this, say  $V_3$ . The unvisited adjacent vertices to  $V_3$  are  $V_6, V_7$ . There are no more unvisited adjacent vertices of  $V_8, V_4, V_5, V_6$  and  $V_7$ . Thus the sequence so generated is  $V_1, V_2, V_8, V_3, V_4, V_5, V_6, V_7$ .

## **COMPUTING TIME:**

Each vertex visited gets into the queue exactly once, so the **loop forever** is iterated at most  $n$  times. If an adjacency matrix is used, then the for loop takes  $O(n)$  time for each vertex visited. The total time is, therefore,  $O(n^2)$ . In case adjacency lists are used the for loop as a total cost of  $d_1 + \dots + d_n = O(e)$  where  $d_i = \text{degree}(v_i)$ . Again, all vertices visited. Together with all edges incident to from a connected component of  $G$ .

## **DEPTH FIRST SEARCH :**

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

This procedure is best described recursively as in

Procedure DFS(v)

// Given an undirected graph  $G = (V, E)$  with  $n$  vertices and an array visited ( $n$ ) initially set to zero . This algorithm visits all vertices reachable from  $v$  . $G$  and VISITED are global > //VISITED ( $v$ )  $\leftarrow 1$

for each vertex  $w$  adjacent to  $v$  do

if VISITED ( $w$ ) =0 then call DFS ( $w$ )

end

**end** DFS

The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth first search must be restarted at any one of them.

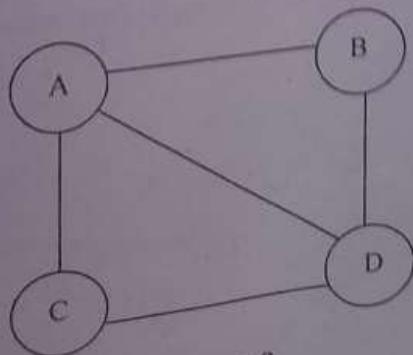
To implement the Depthfirst Search perform the following Steps :

- Step : 1 Choose any node in the graph. Designate it as the search node and mark it as visited.
- Step : 2 Using the adjacency matrix of the graph, find a node adjacent to the search node that has not been visited yet. Designate this as the new search node and mark it as visited.
- Step : 3 Repeat step 2 using the new search node. If no nodes satisfying (2) can be found, return to the previous search node and continue from there.
- Step : 4 When a return to the previous search node in (3) is impossible, the search from the originally chosen search node is complete.
- Step : 5 If the graph still contains unvisited nodes, choose any node that has not been visited and repeat step (1) through (4).

## ROUTINE FOR DEPTH FIRST SEARCH

```
Void DFS (Vertex V)
{
    visited [V] = True;
    for each W adjacent to V
        if (! visited [W])
            DFS (W);
}
```

Example:-

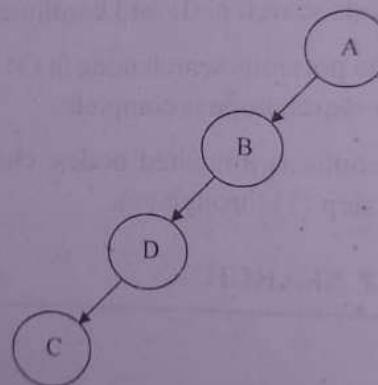


## Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

### Implementation

1. Let 'A' be the source vertex. Mark it to be visited.
  2. Find the immediate adjacent unvisited vertex 'B' of 'A' Mark it to be visited.
  3. From 'B' the next adjacent vertex is 'd' Mark it has visited.
  4. From 'D' the next unvisited vertex is 'C' Mark it to be visited.



## Depth First Spanning Tree

## Applications of Depth First Search

1. To check whether the undirected graph is connected or not.
  2. To check whether the connected undirected graph is Bioconnected or not.
  3. To check the Acyclicity of the directed graph.

# Example

Let us start with  $V_1$ .

**Its adjacent vertices are  $V_2$ ,  $V_8$ , and  $V_3$ . Let us pick on  $v_2$ .**

Its adjacent vertices are  $V_1$ ,  $V_4$ ,  $V_5$ ,  $V_1$  is already visited . let us pick on  $V_4$ .

Its adjacent vertices are  $V_2$ ,  $V_8$ .

$V_2$  is already visited .let us visit  $V_8$ .

Its adjacent vertices are  $V_4$ ,  $V_5$ ,  $V_1$ ,  $V_6$ ,  $V_7$ .

$V_4$  and  $V_1$  are visited. Let us traverse  $V_5$ .

Its adjacent vertices are  $V_2$ ,  $V_8$ . Both are already visited therefore, we back track.

We had  $V_6$  and  $V_7$  unvisited in the list of  $V_8$ . We may visit any. We may visit any. We visit  $V_6$ .

Its adjacent are  $V_8$  and  $V_3$ . Obviously the choice is  $V_3$ .

Its adjacent vertices are  $V_1$ ,  $V_7$  . We visit  $V_7$ .

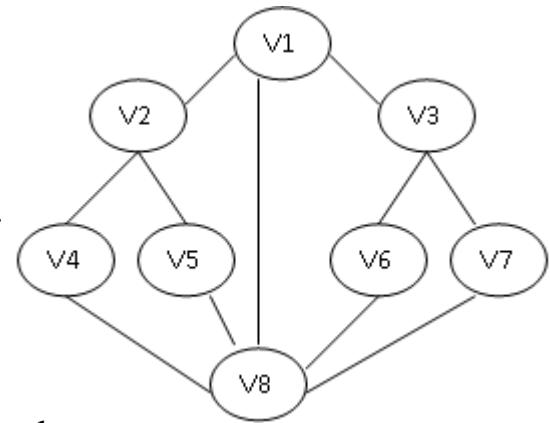
All the adjacent vertices of  $V_7$  are already visited, we back track and find that we have visited all the vertices.

**Therefore the sequence of traversal is**

$V_1, V_2, V_4, V_5, V_6, V_3, V_7$ .

This is not a unique or the only sequence possible using this traversal method.

We may implement the Depth First search by using a stack,pushing all unvisited vertices to the one just visited and poping the stack to find the next vertex to visit.



## **COMPUTING TIME:**

In case G is represented by adjacency lists then the vertices w adjacent to v can be determined by following a chain of links. Since the algorithm DFS would examine each node in the adjacency lists at most once and there are  $2e$  list nodes. The time to complete the search is  $O(e)$ . if G is represented by its adjacency matrix. Then the time to determine all vertices adjacent to v is  $O(n)$ . since at most n vertices are visited. The total time is  $O(n^2)$ .

# Application of Graphs

- Graphs are used to define the **flow of computation**.
- Graphs are used to represent **networks of communication**.
- Graphs are used to represent **data organization**.
- Graph transformation systems work on rule-based in-memory manipulation of graphs. Graph databases ensure **transaction-safe, persistent storing and querying of graph structured data**.
- Graph theory is used to find **shortest path in road or a network**.

- In Computer science graphs are used to represent the flow of computation.
- Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.

- In World Wide Web, web pages are considered to be the vertices. There is an edge from a page  $u$  to other page  $v$  if there is a link of page  $v$  on page  $u$ . This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.
- In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

- **Neural networks:** Vertices represent neurons and edges are the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about  $10^{11}$  neurons and close to  $10^{15}$  synapses.

# Application of Trees

- Store hierarchical data, like folder structure, organization structure, XML/HTML data.
- Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
- Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
- B-Tree and B+ Tree : They are used to implement indexing in databases.
- Syntax Tree: Compilers use a syntax tree to validate the syntax of every program you write.