BINARY TREE DELETION

Kanak Kalyani

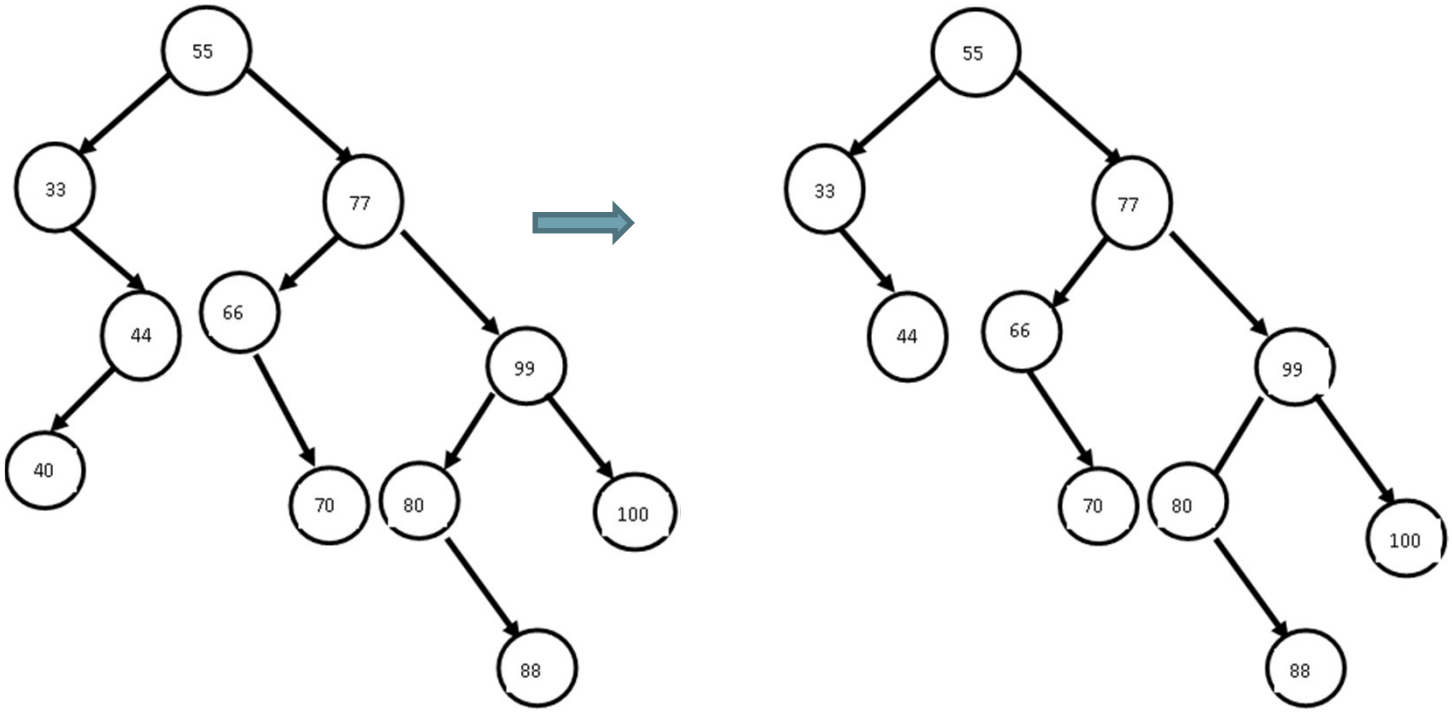
BINARY SEARCH TREE DELETION

When we delete a node, three possibilities arise.

- ***1. Node to be deleted is leaf:***
  - Simply remove the Node from the tree
- ***2. Node to be deleted has only one child:***
  - Replace the Node by its only Child Node.
- ***3. Node to be deleted has two children:***
  - Find inorder successor of the Node.
  - Replace the Node by its inorder successor.

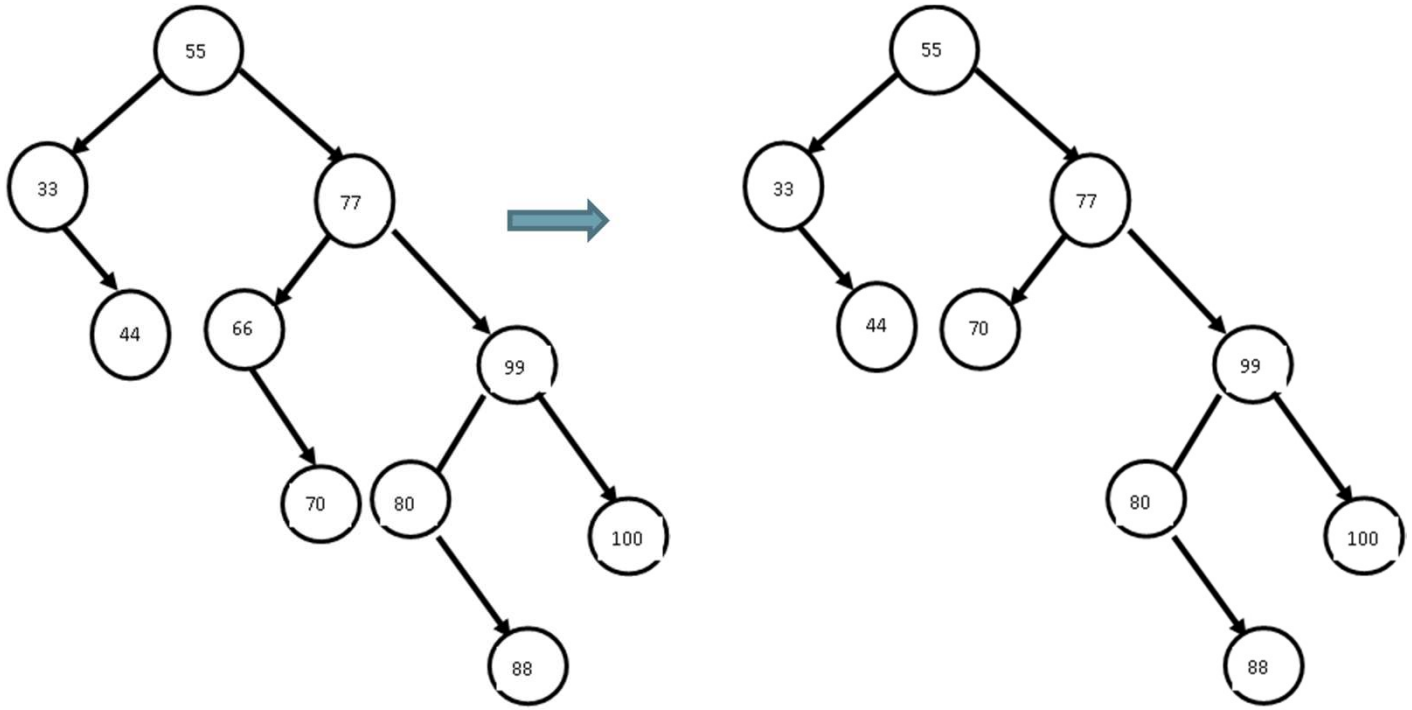
BINARY SEARCH TREE DELETION

- Delete 40



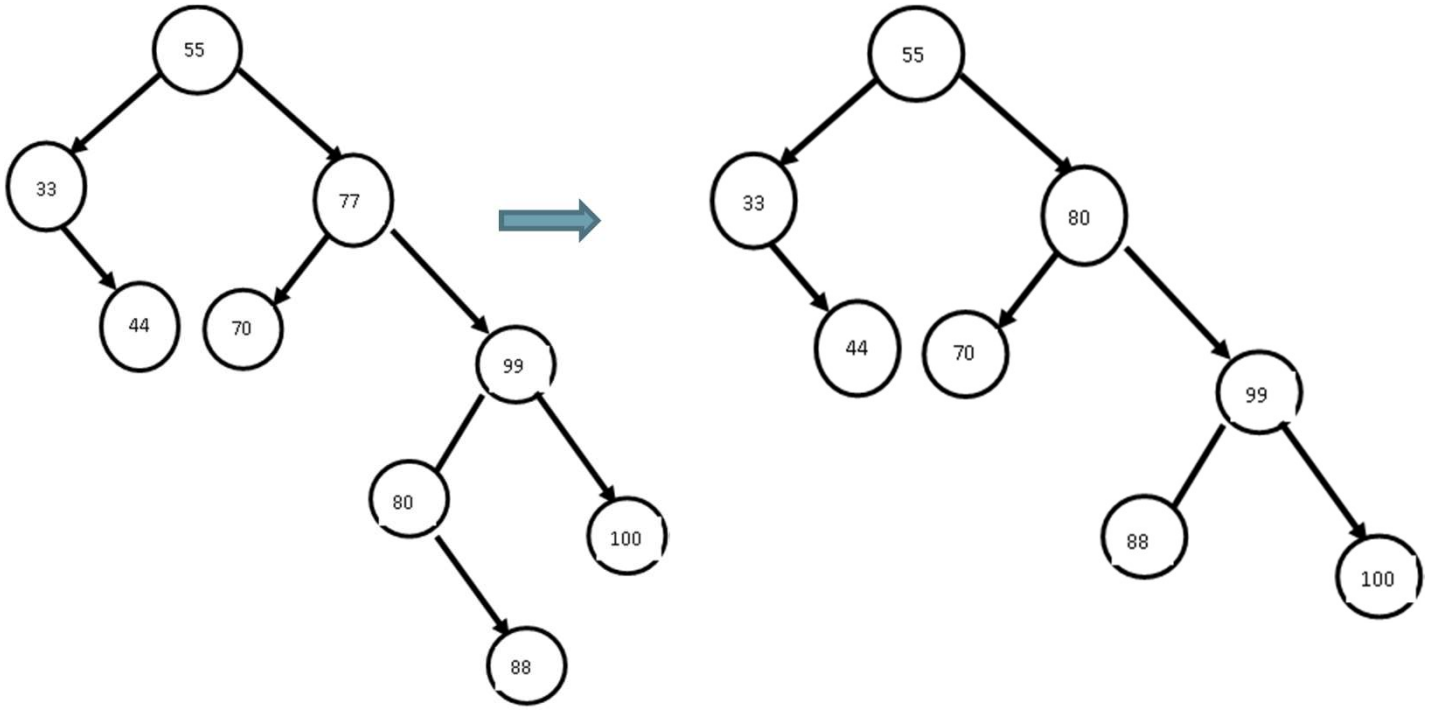
BINARY SEARCH TREE DELETION

- Delete 66



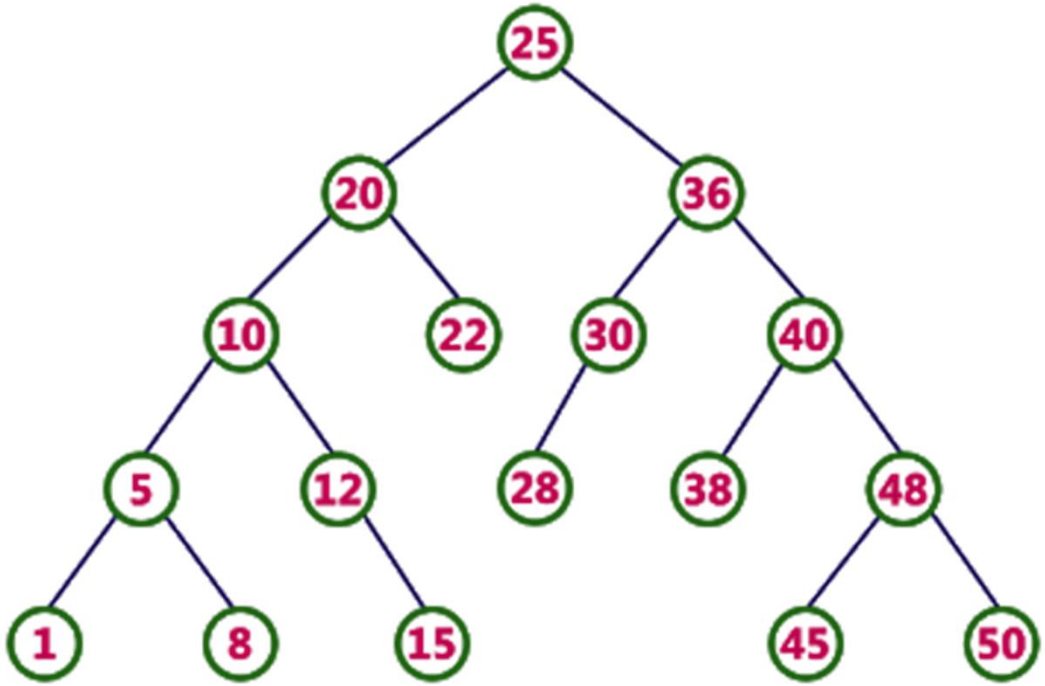
BINARY SEARCH TREE DELETION

- Delete 77



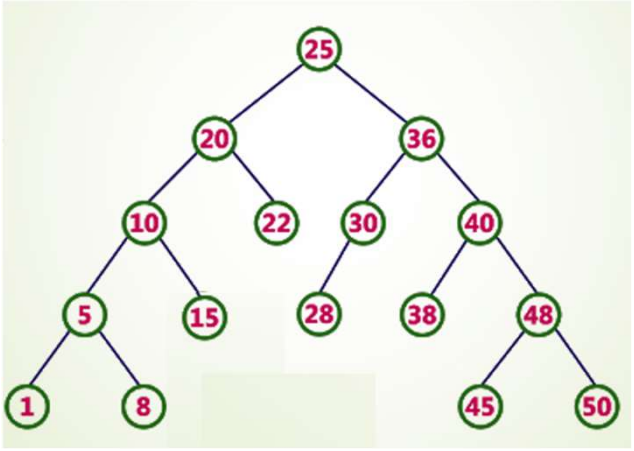
Example…

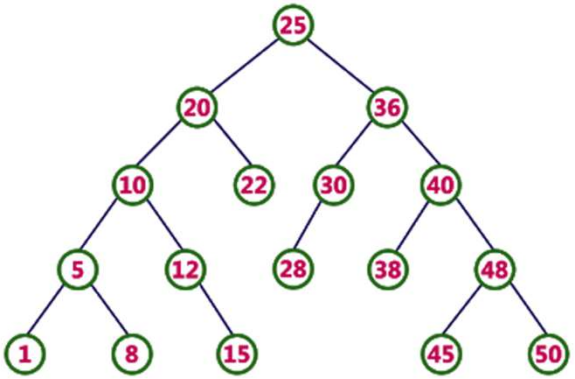
- Delete nodes in given sequence: 12, 22, 36, 25



Example...

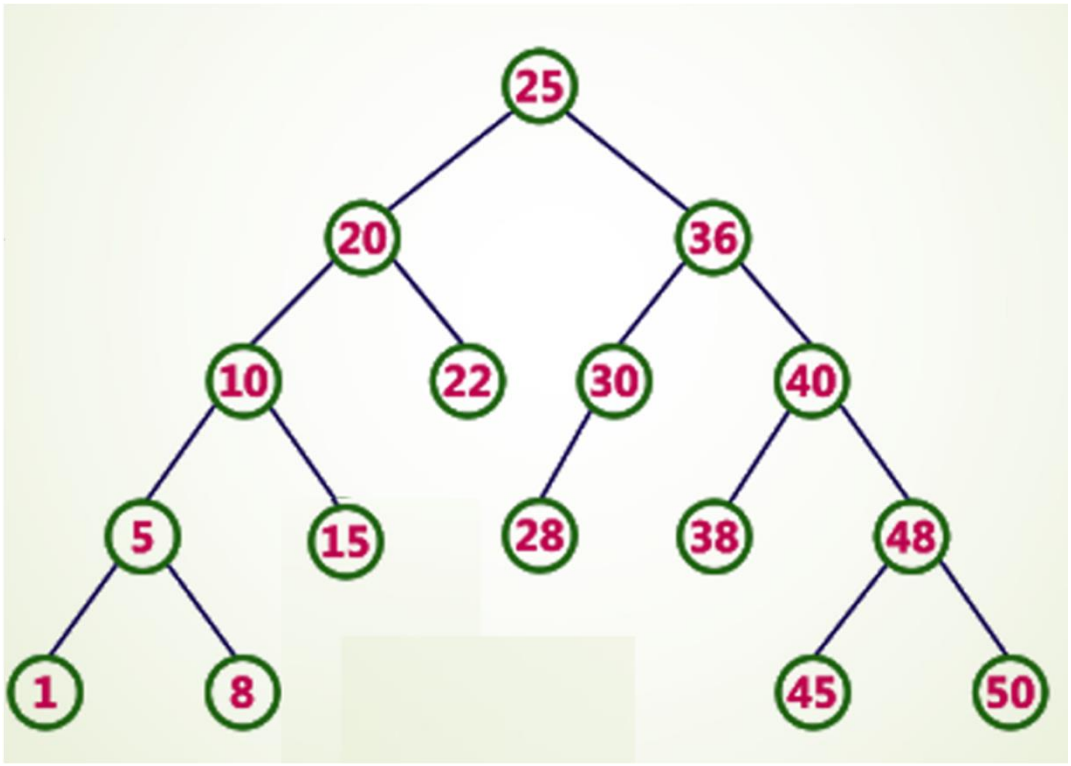
- Delete node 12





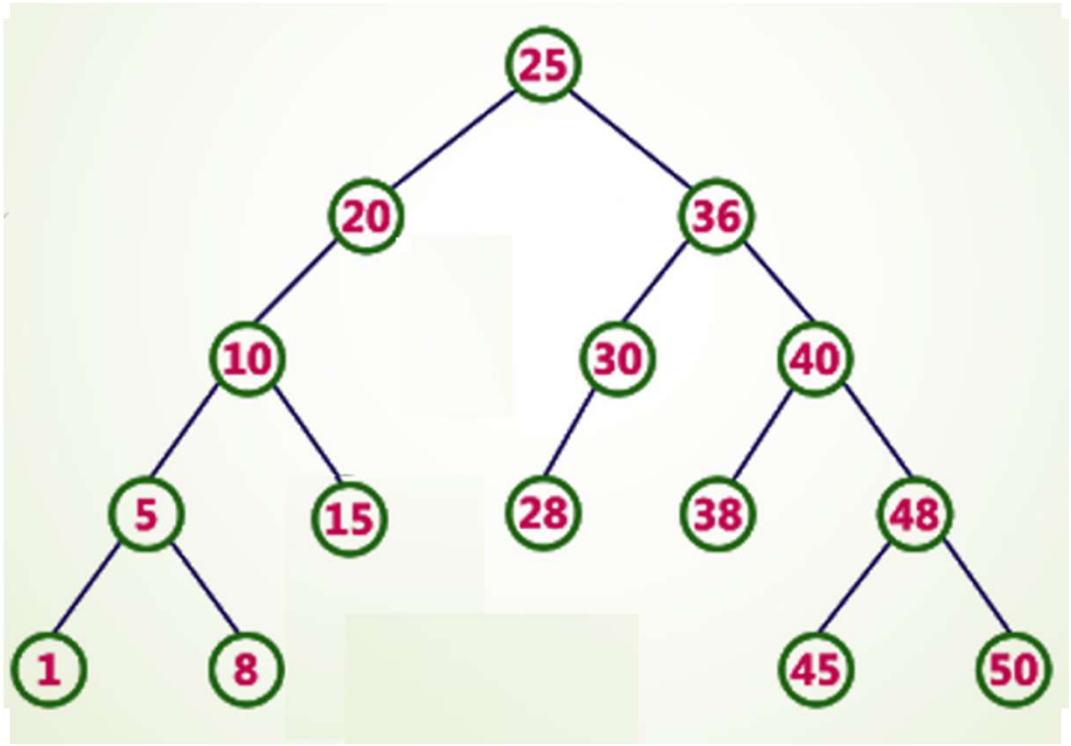
Example…

- Delete node 22



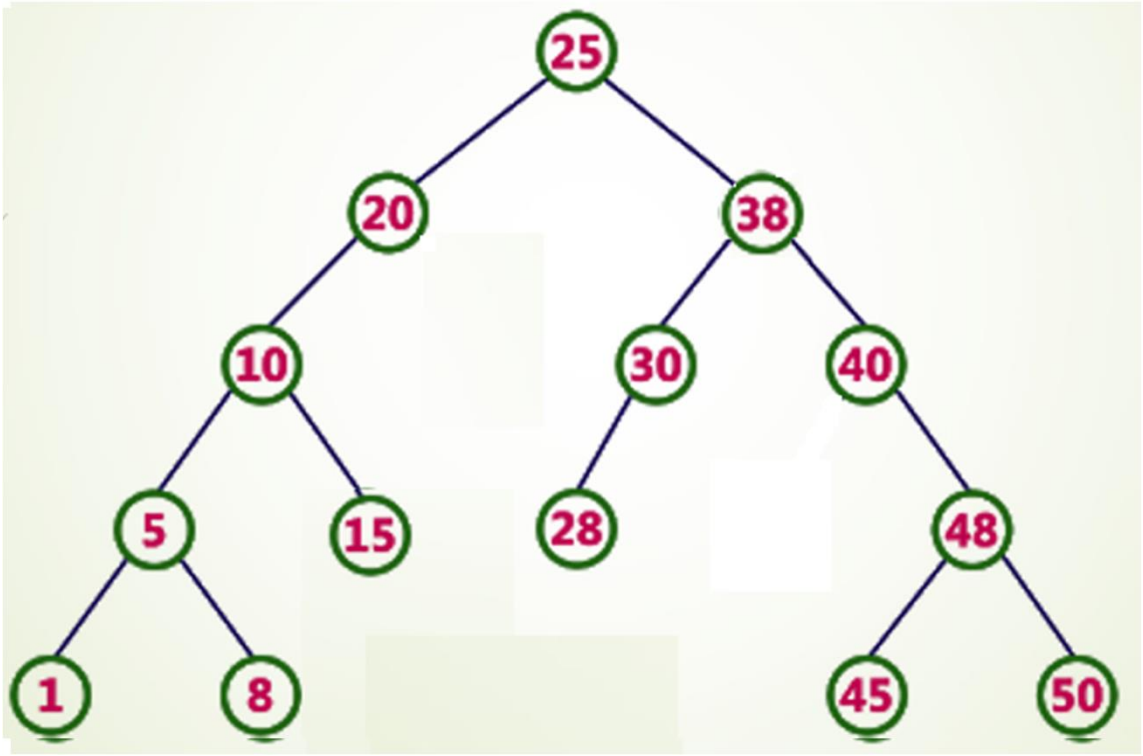
Example...

- Delete node 36

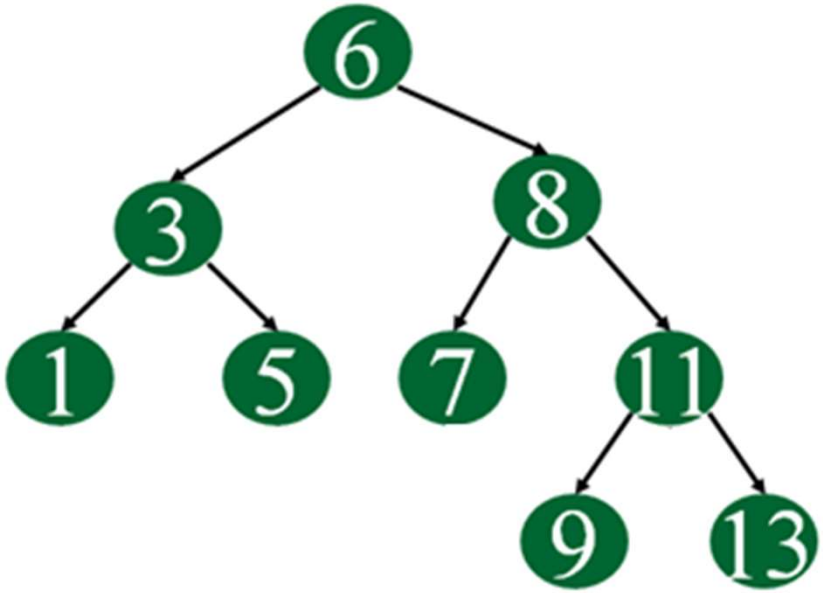


Example…

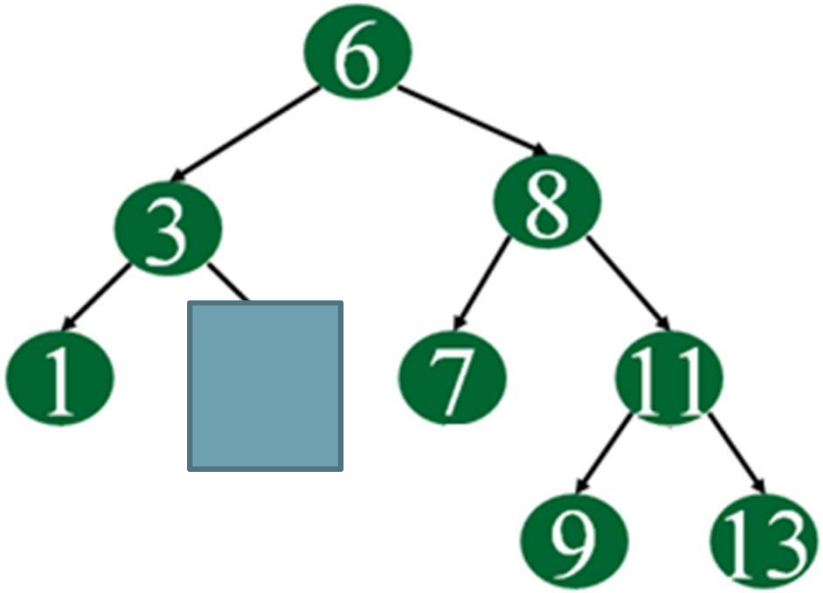
- Delete node 25



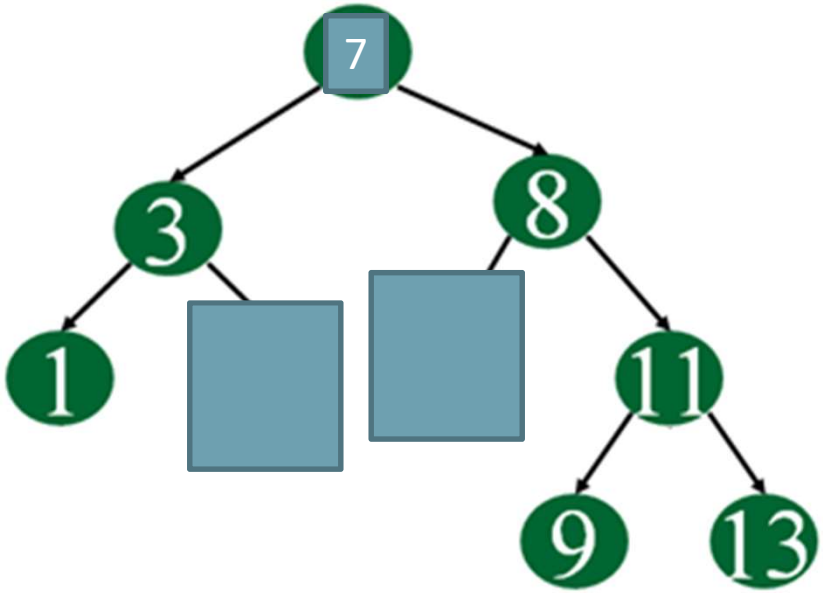
Delete 5,6,8,11



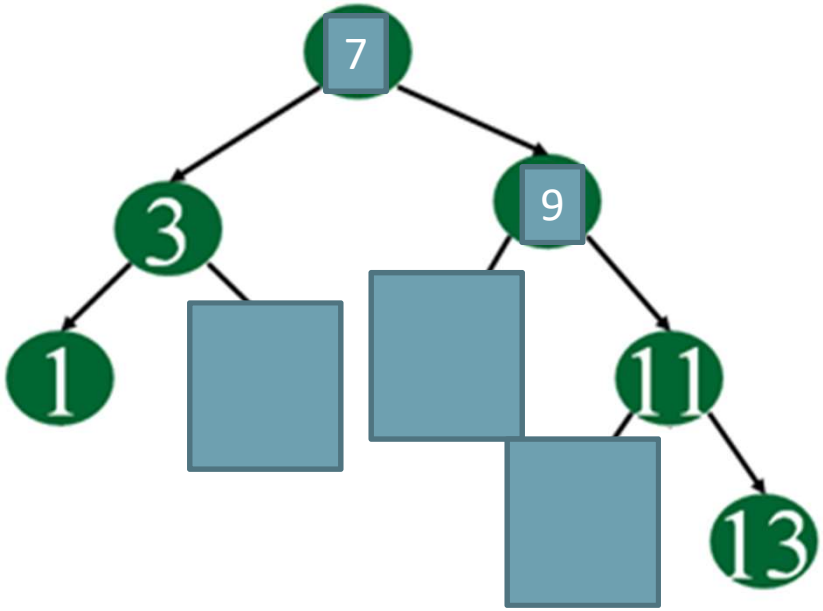
Delete 5



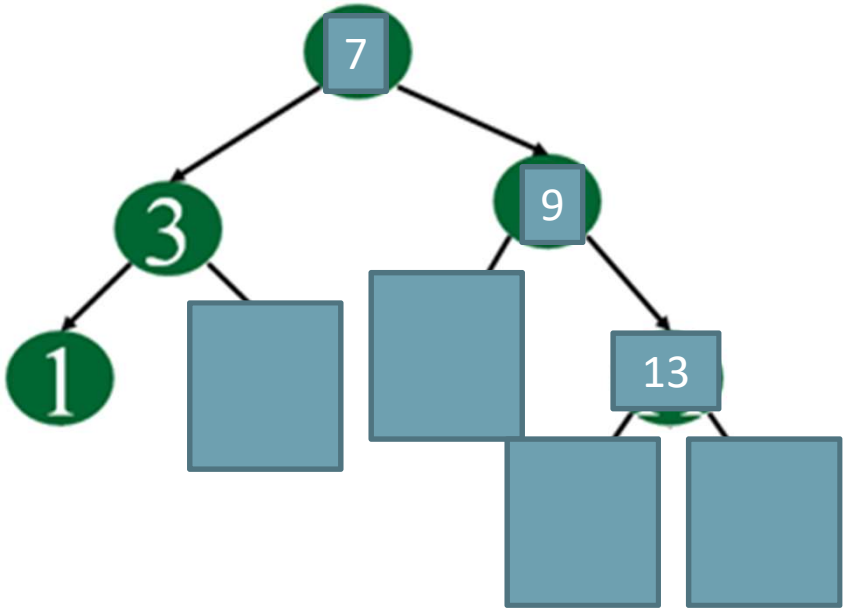
Delete 6



Delete 8



Delete 11

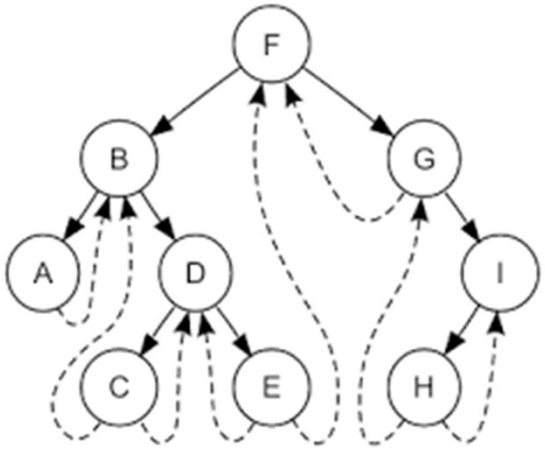


THREADED BINARY TREE

Kanak Kalyani

THREADED BINARY TREES

- Binary Trees have a lot of wasted space: The leaf nodes each have 2 NULL pointers.
- We can use these pointers to help us in inorder traversal.
- Thread: a pointer to other node in the tree for replacing NULL links.
- But we need to know if pointer is an actual link to child node or a thread



INORDER THREADED BINARY TREES

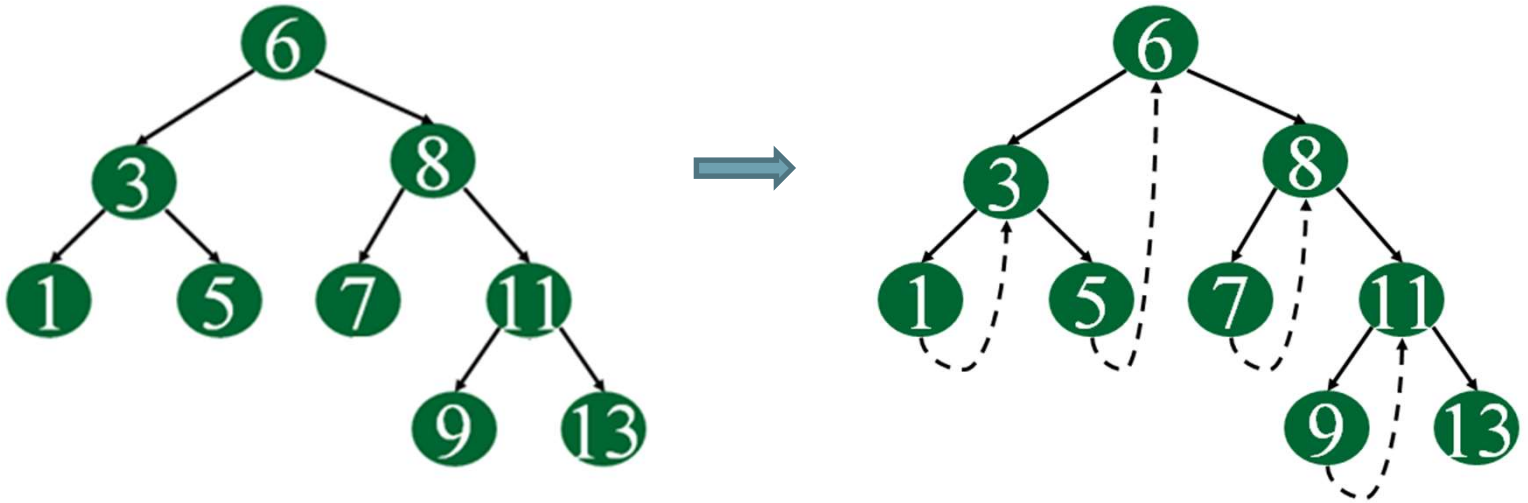
- An Inorder Threaded Binary Tree is a binary tree in which every node that does not have a rightchild has a THREAD to its INORDER successor.

- By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

INORDER THREADED BINARY TREES

- A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

- The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion.



THREADED BINARY TREES

There are two types of threaded binary trees.

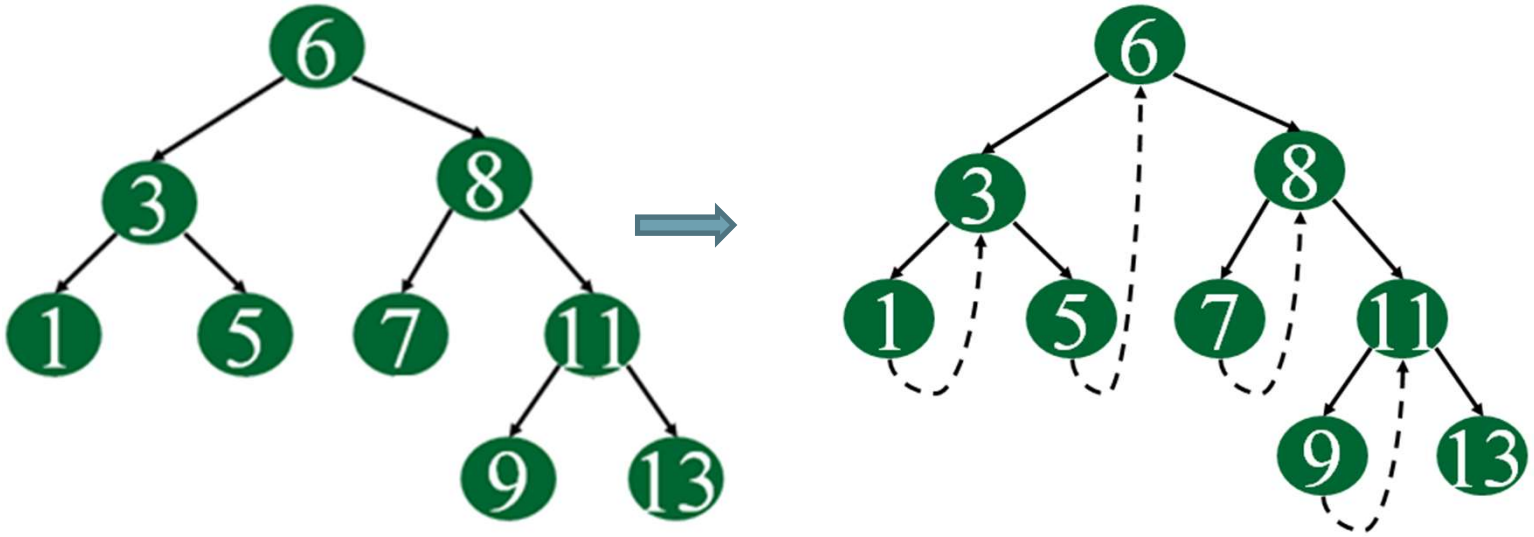
- **Single Threaded:** NULL right pointers is made to point to the inorder successor.
- **Double Threaded:** Both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

THREADED BINARY TREES

```
struct Node
{
        Node *left;
        int data;
        Node * right;
        bool rightThread;
}
```

left	data	right	rightThread

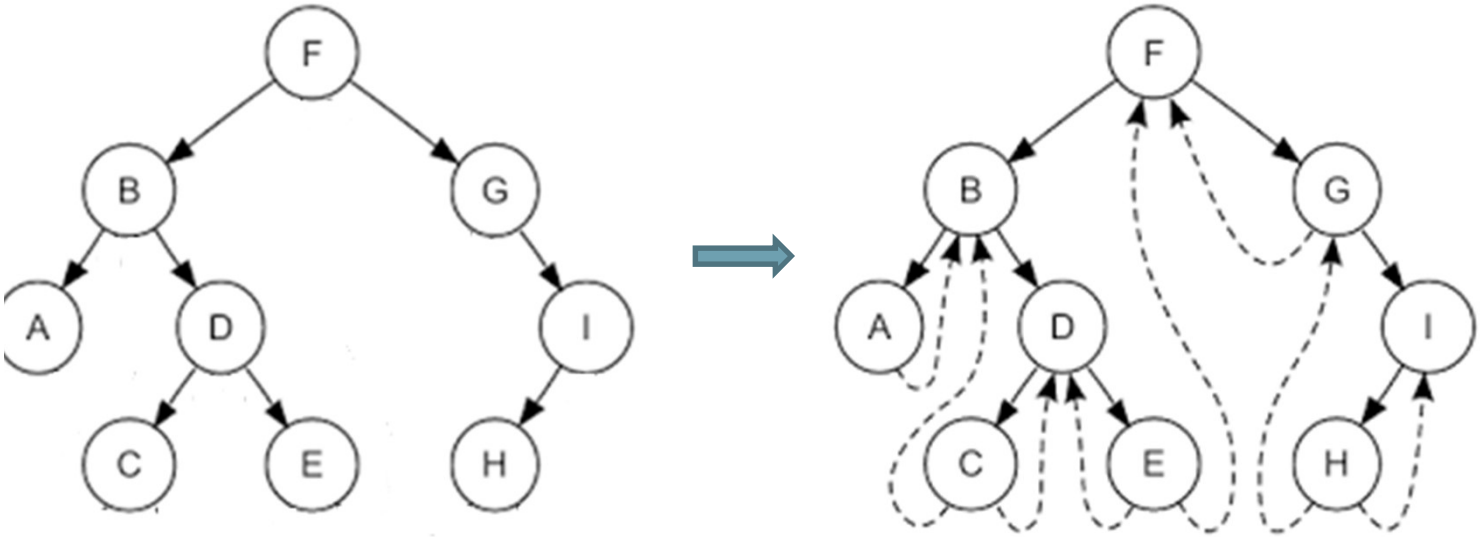
left	data	right	0/1



DOUBLE THREADED BINARY TREES

```
struct Node
{
        bool leftThread;
        Node *left;
        int data;
        Node *right;
        bool rightThread;
}
```

Left Thread	left	data	right	Right Thread



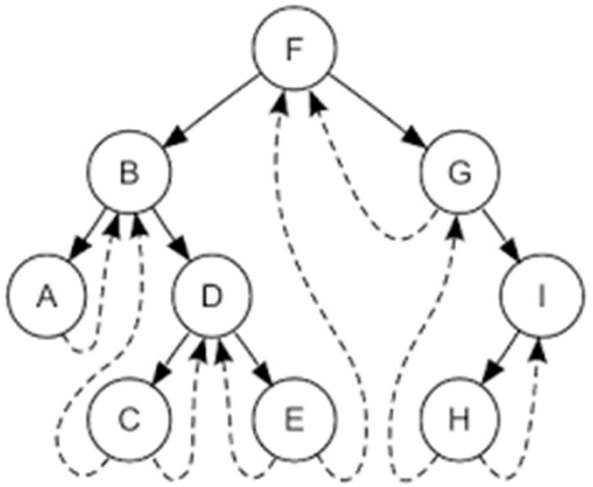
Program for inorder traversal of threaded binary tree

```
struct Node* leftMost(struct Node *node)
{

        if (node == NULL)
        return NULL;
        while (node ->left != NULL)
                node = node ->left;
        return node;
}
```

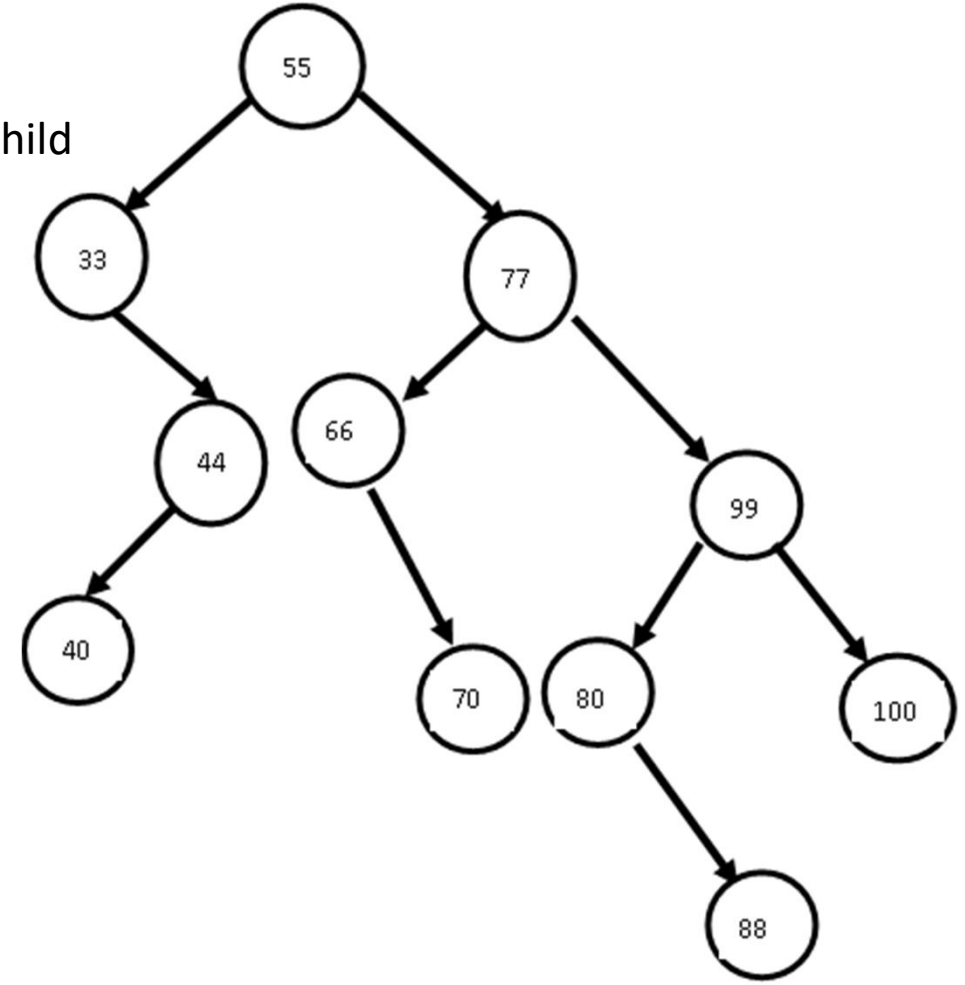
Program for inorder traversal of threaded binary tree

```
void inOrder(struct Node *root)
{
        struct Node *cur = leftmost(root);
        while (cur != NULL)
        {
                printf("%d ", cur->data);
                // Check if this node is a thread node
                if (cur->rightThread)
                        cur = cur->right;
                else
                        cur = leftmost(cur->right);
        }
}
```



Trees

1) Count number of nodes in the tree
2) Count number of leaf nodes
3) Count number of nonleaf nodes
4) Count number of nodes with 1 child



AVL TREES

Kanak Kalyani

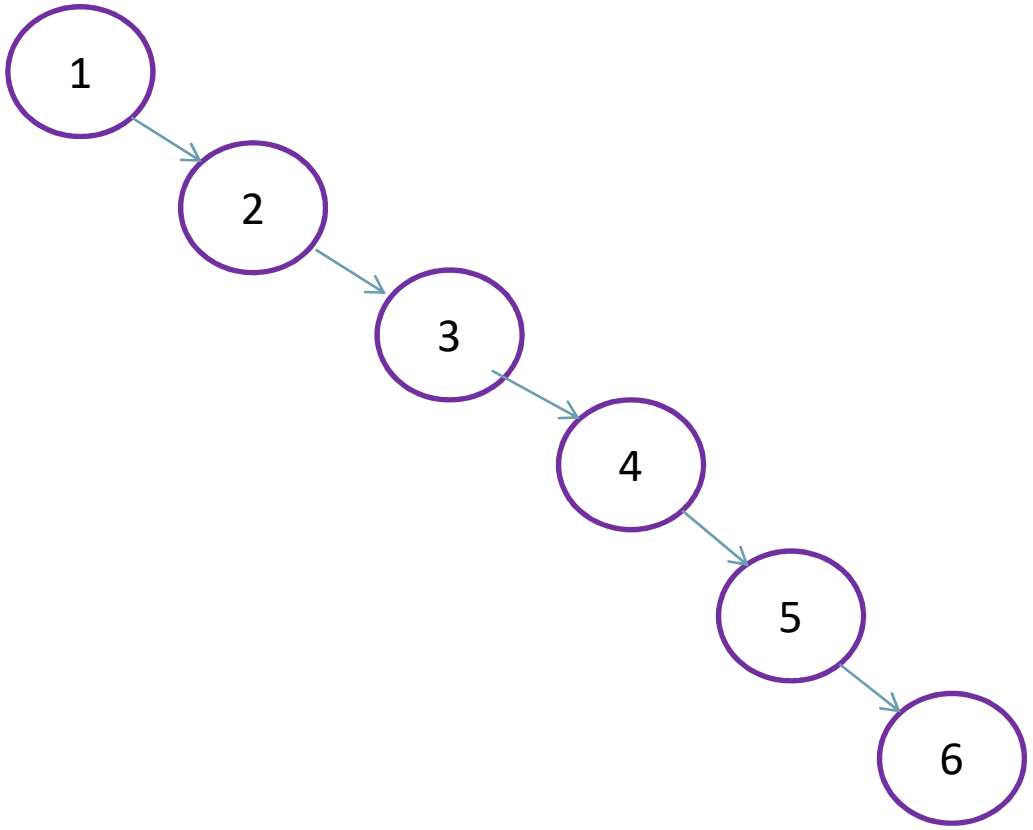
AVL TREES

- AVL tree is a self-balancing binary search tree invented by Adelson, Velsky and Landis in 1962

AVL TREES

- In an AVL tree, the heights of the two sub-trees of a node may differ by at most one.
- Due to this property, the AVL tree is also known as a height-balanced tree.
- The key advantage of using an AVL tree is that it takes **O(log n)** time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to **O(log n).**

- Create a BST with following nodes: 1,2,3,4,5,6



Skewed BST

AVL TREES

- The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the Balance Factor.

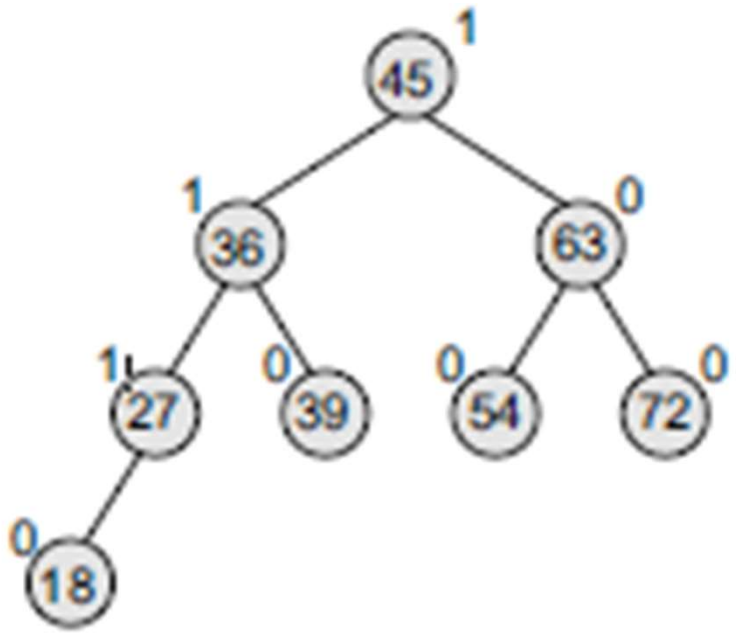
- Every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

Balance factor = Height (left sub-tree) – Height (right sub-tree)

- A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

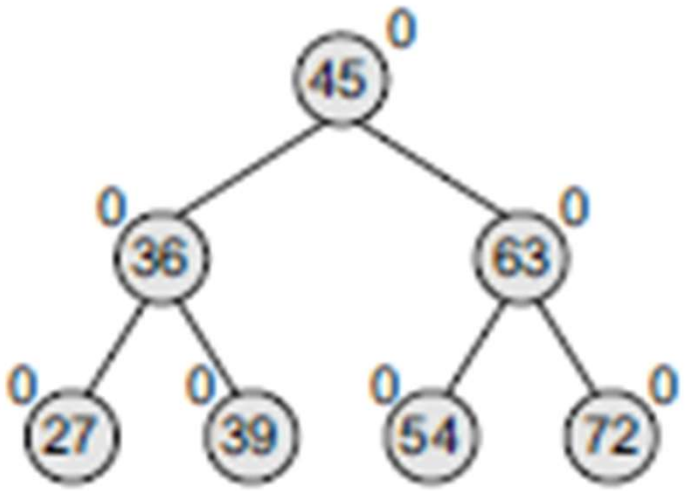
Left-Heavy Tree

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a left-heavy tree.



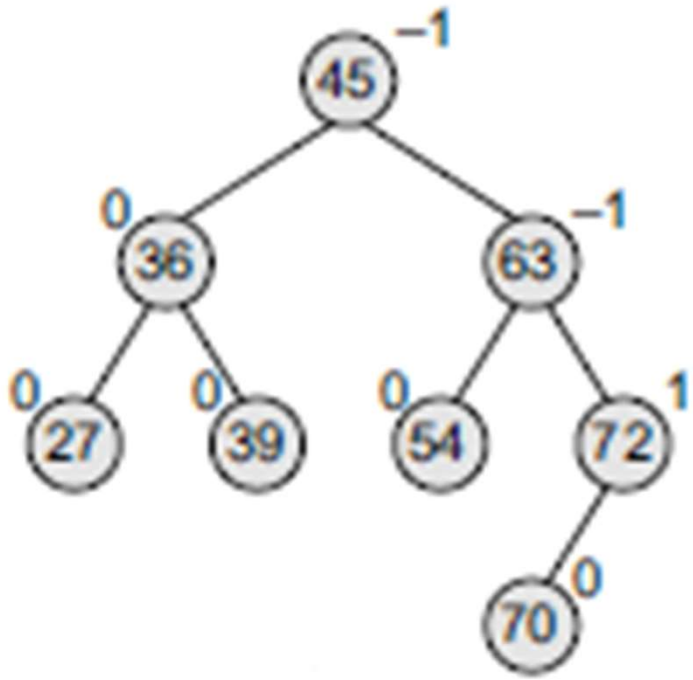
Balanced **AVL TREES**

- If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.



Right-Heavy tree

- If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a right-heavy tree.



Operations on AVL Trees

Searching for a Node in an AVL Tree:

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Due to the height-balancing of the tree, the search operation takes O(log n) time to complete.
- Since the operation does not modify the structure of the tree, no special provisions are required.

Inserting a New Node

Inserting a New Node in an AVL Tree:

- Insertion in an AVL tree is also done in the same way as it is done in a binary search tree.
- But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree.
- However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still –1, 0, or 1, then rotations are not required.
- During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node.

Inserting a New Node

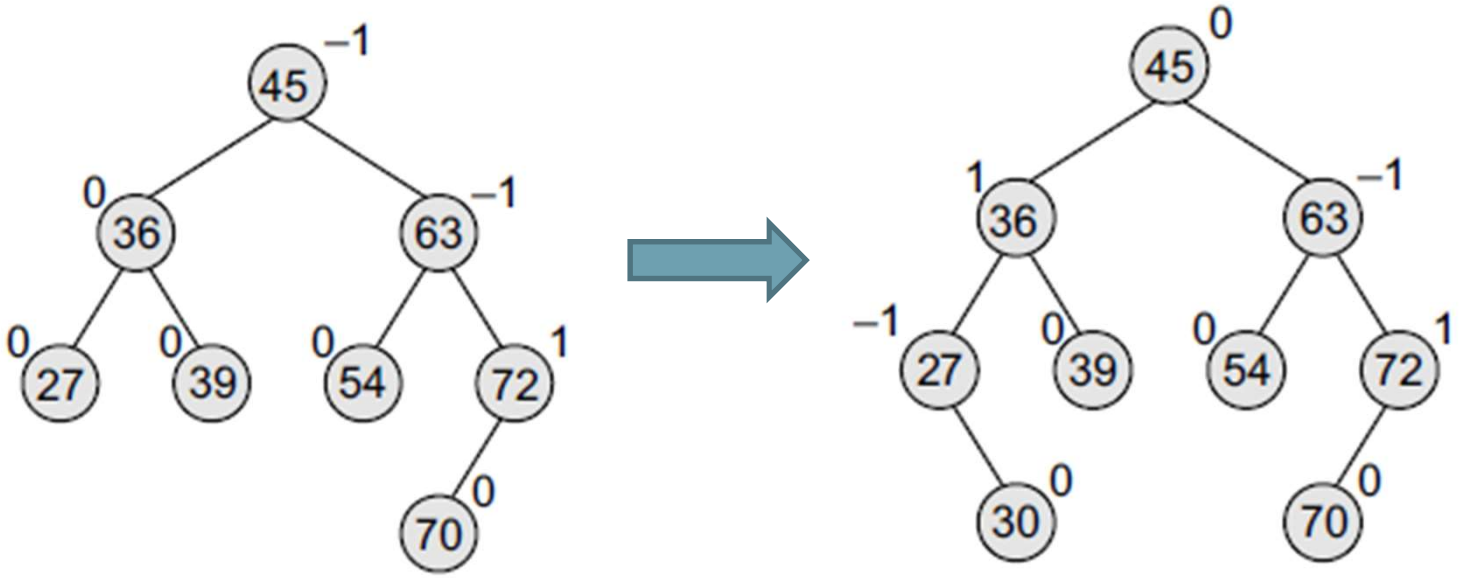
Inserting a New Node in an AVL Tree:

The possible changes which may take place in any node on the path are as follows:

- Initially, the node was either left- or right-heavy and after insertion, it becomes balanced.
- Initially, the node was balanced and after insertion, it becomes either left-heavy or right-heavy.
- Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a ***critical node***.

Inserting a New Node

Insert 30

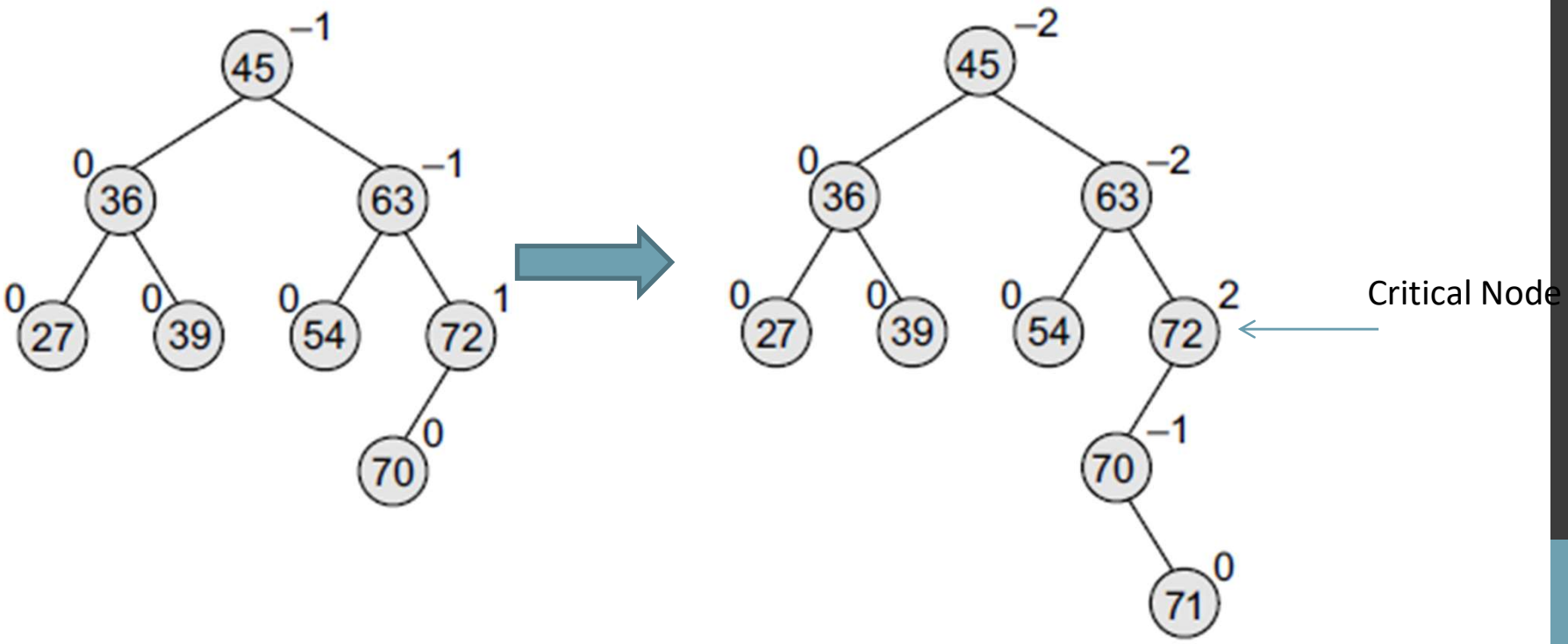


Balanced

Balanced

Inserting a New Node

Insert 71



Balanced

Not Balanced

Rotations in AVL tree

- ***LL rotation:***

  The new node is inserted in the left sub-tree of the left sub-tree of the critical node.

- ***RR rotation:***

  The new node is inserted in the right sub-tree of the right sub-tree of the critical node.

- ***LR rotation:***

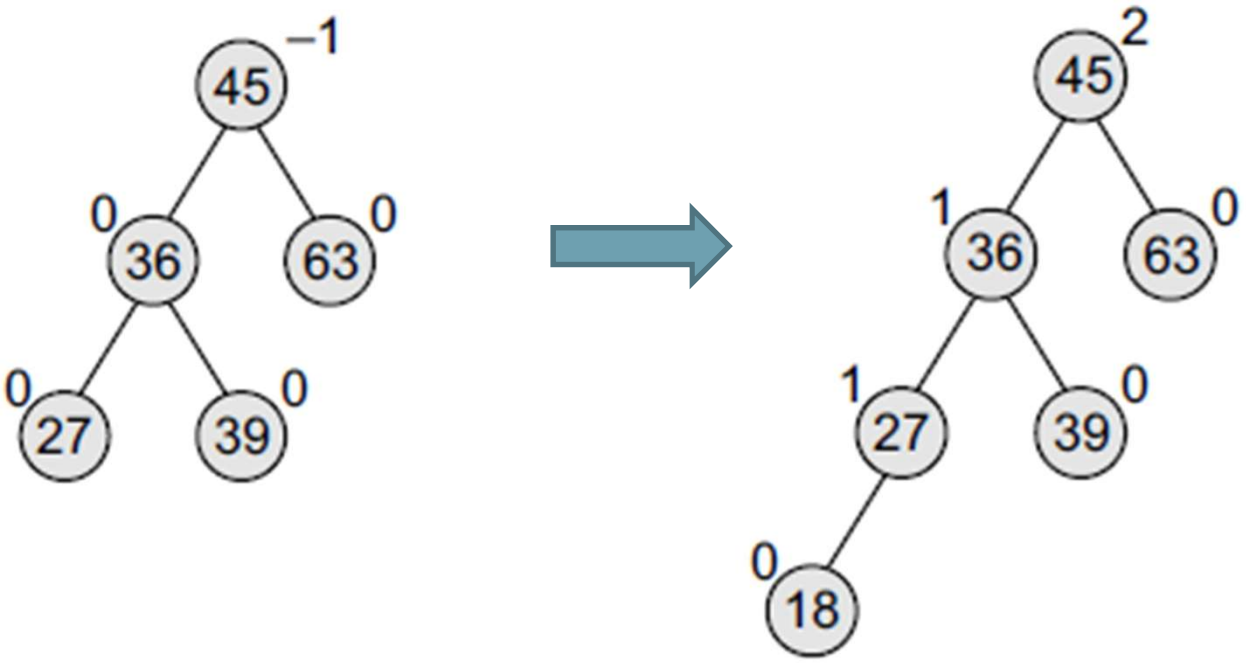
  The new node is inserted in the right sub-tree of the left sub-tree of the critical node.

- ***RL rotation:***

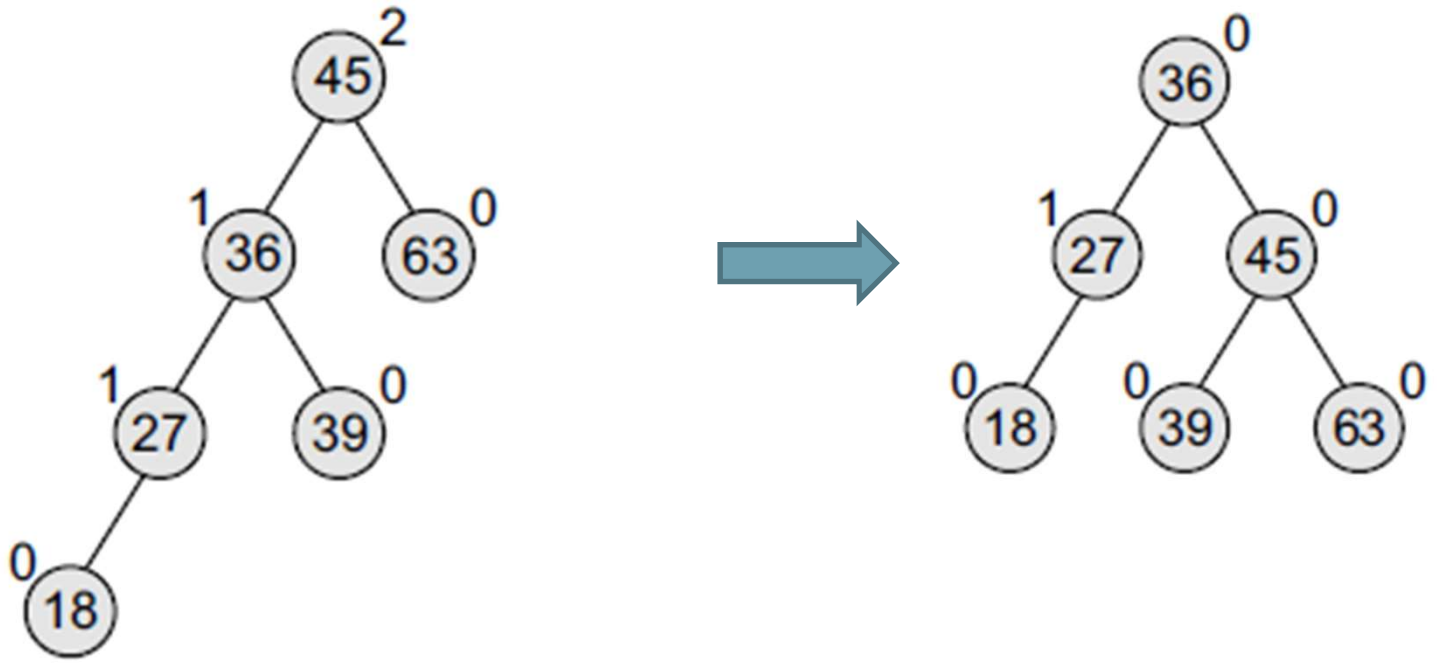
  The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

LL rotation in AVL tree

Insert 18

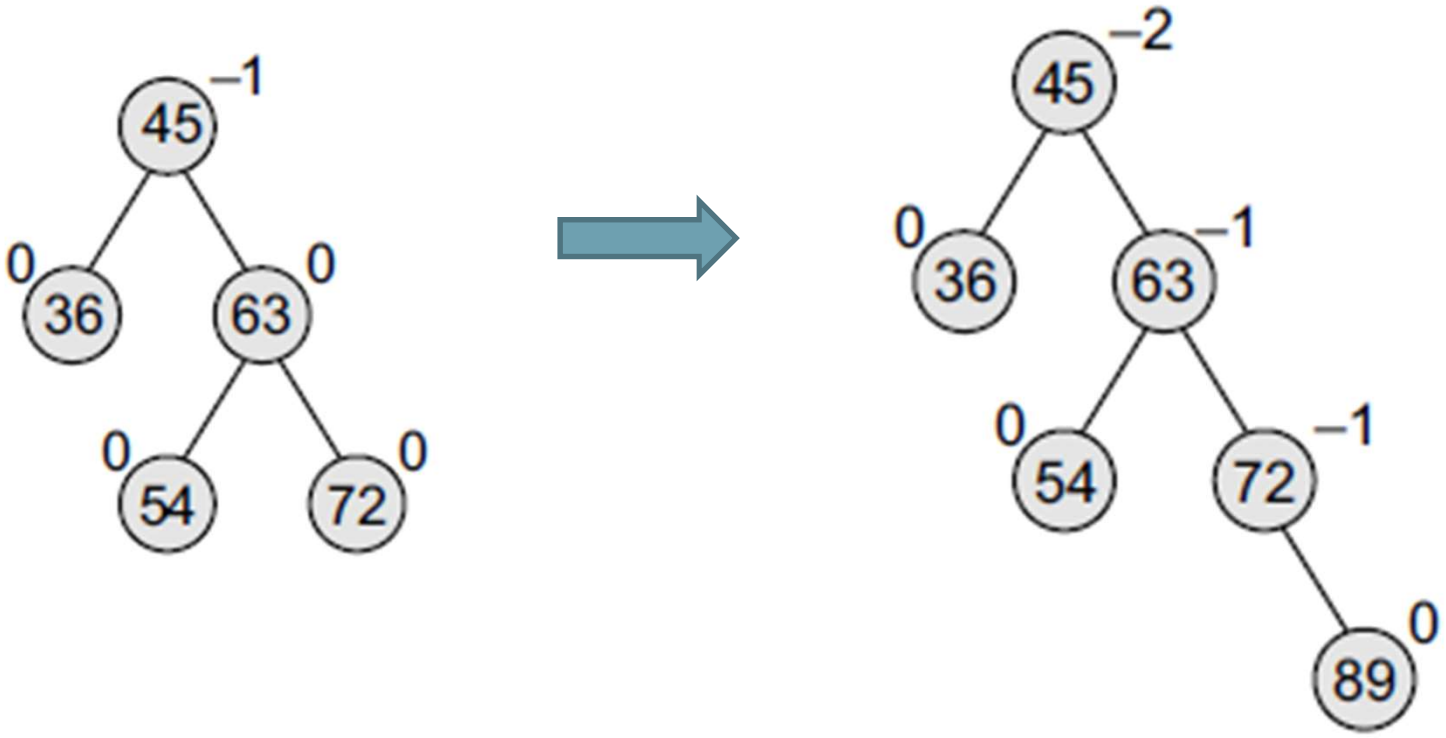


LL Rotations in AVL Tree

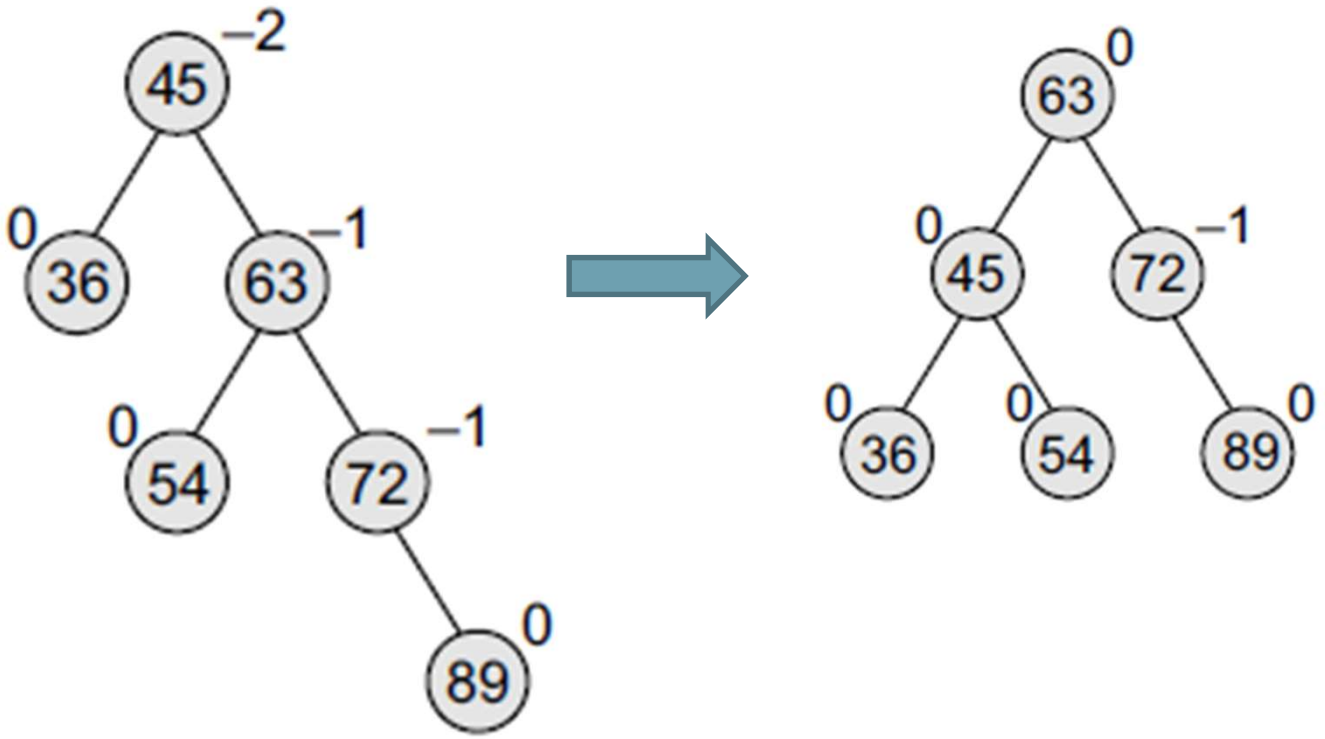


RR rotation in AVL Tree

Insert 89



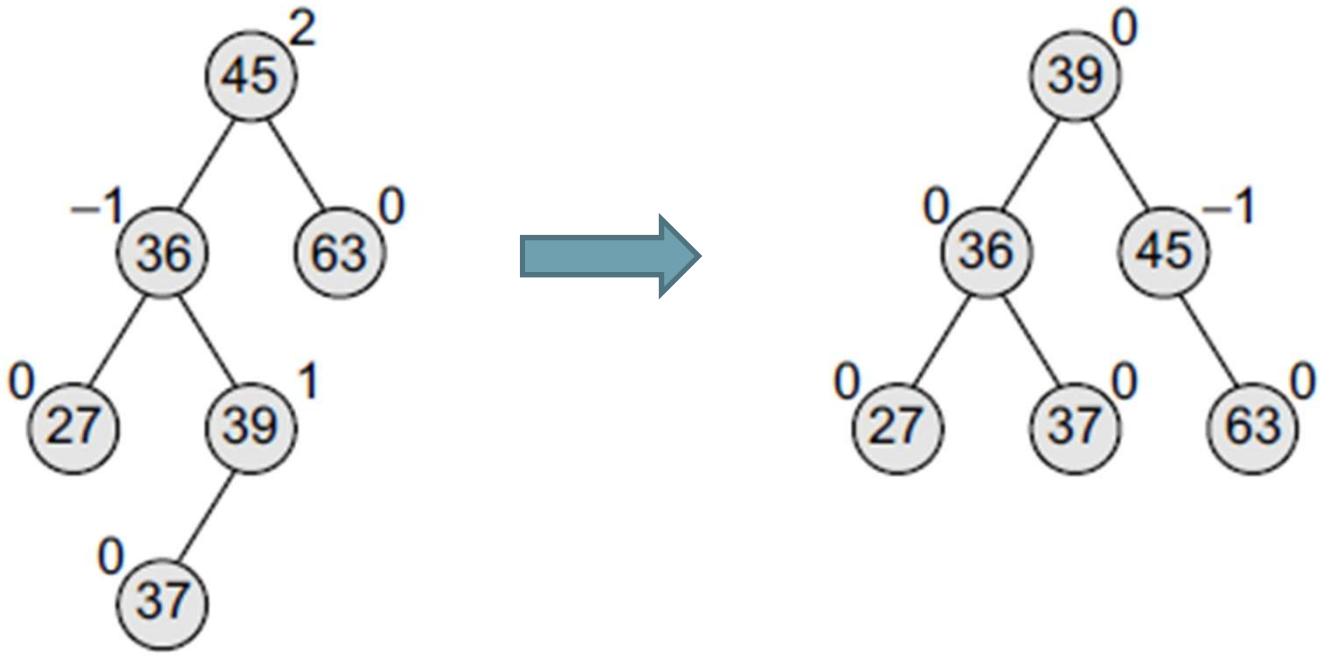
RR Rotations in AVL Tree



LR Rotation in AVL Tree

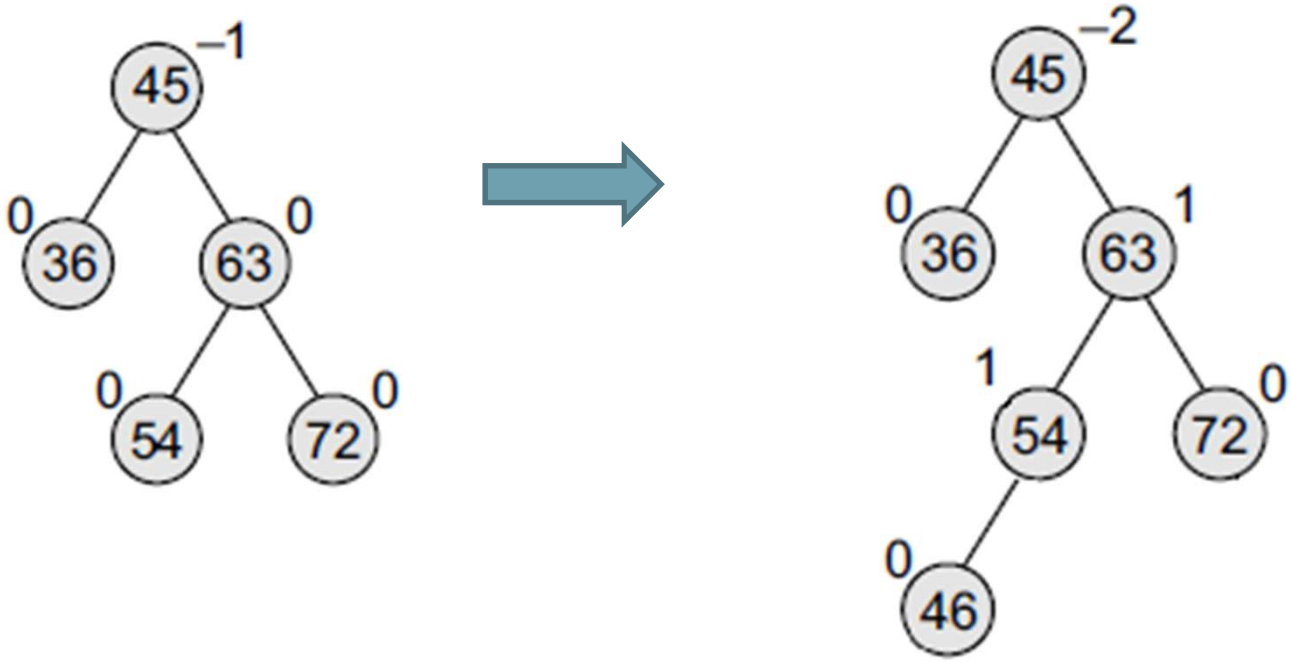
Insert 37

LR Rotation in AVL Tree

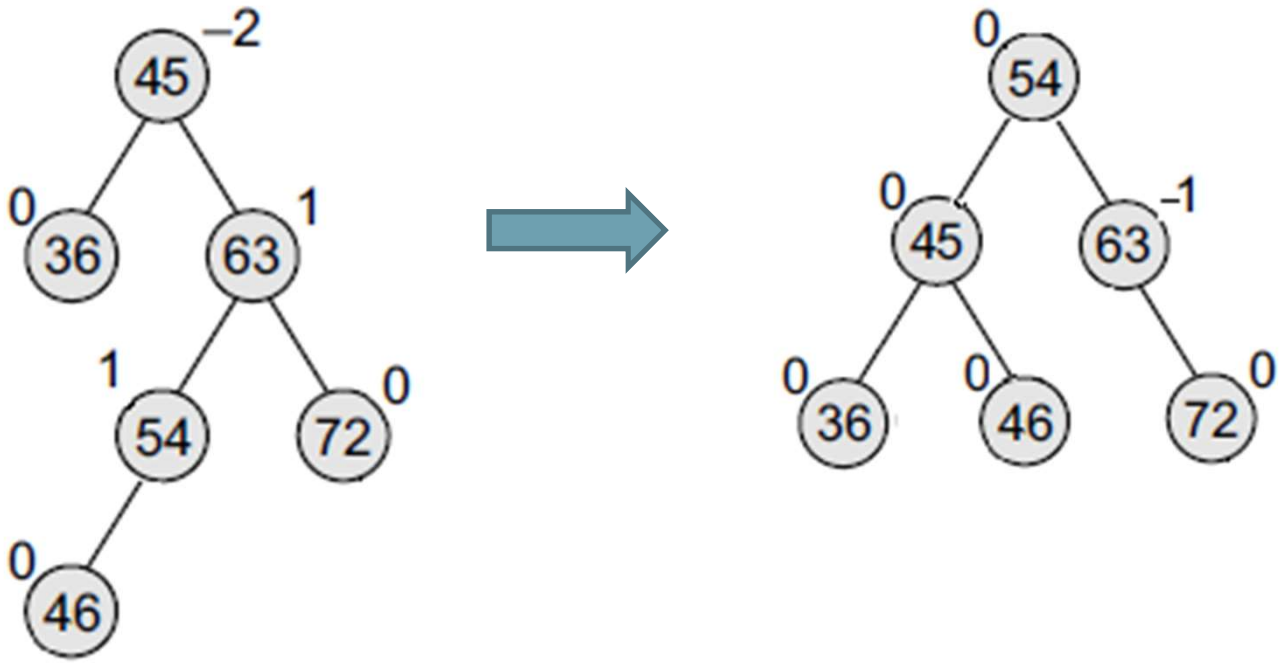


RL Rotation in AVL Tree

Insert 46



RL Rotation in AVL Tree

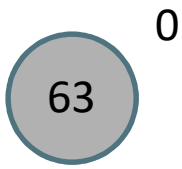


Example to Construct an AVL Tree

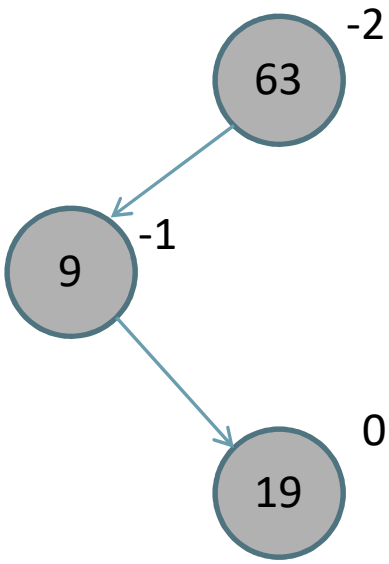
- Construct an AVL tree by inserting the following elements in the given order.

  63, 9, 19, 27, 18, 108, 99, 81.

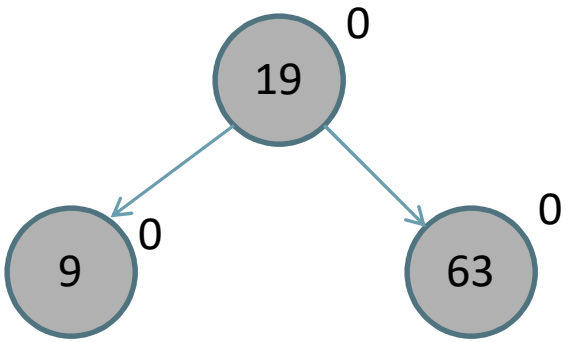
63, 9, 19, 27, 18, 108, 99, 81



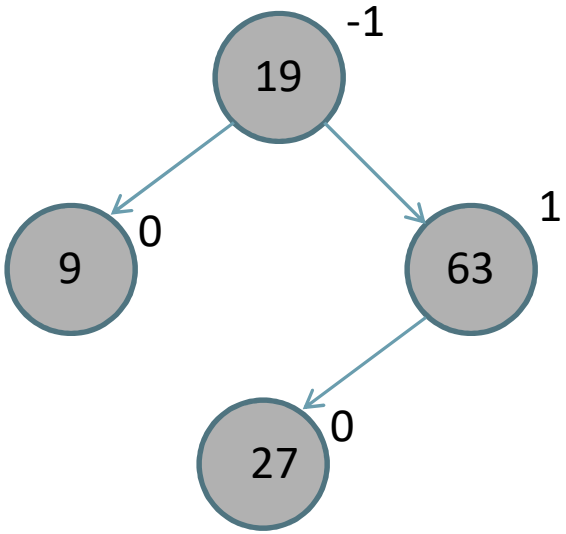
63, 9, 19, 27, 18, 108, 99, 81



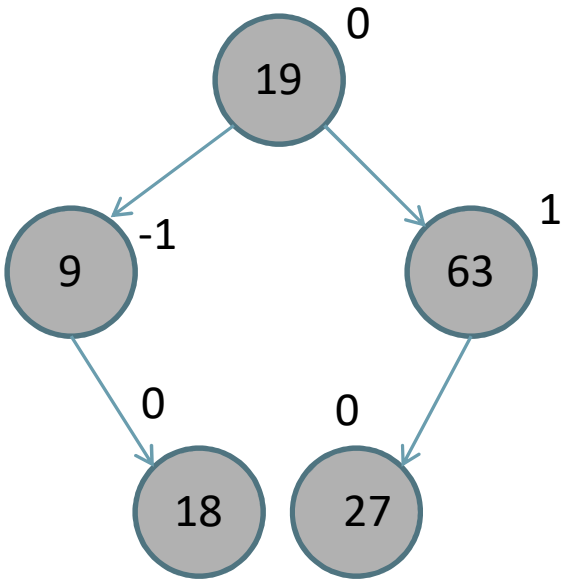
63, 9, 19, 27, 18, 108, 99, 81



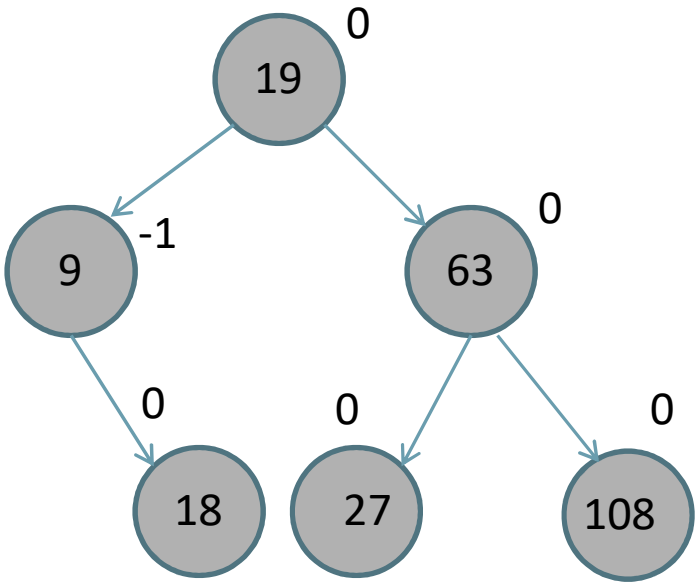
63, 9, 19, 27, 18, 108, 99, 81



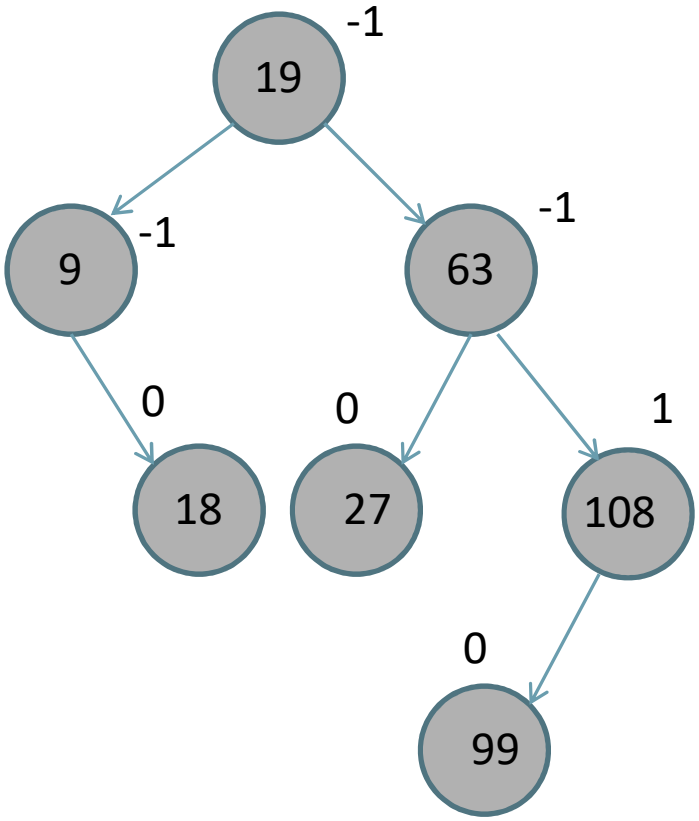
63, 9, 19, 27, 18, 108, 99, 81



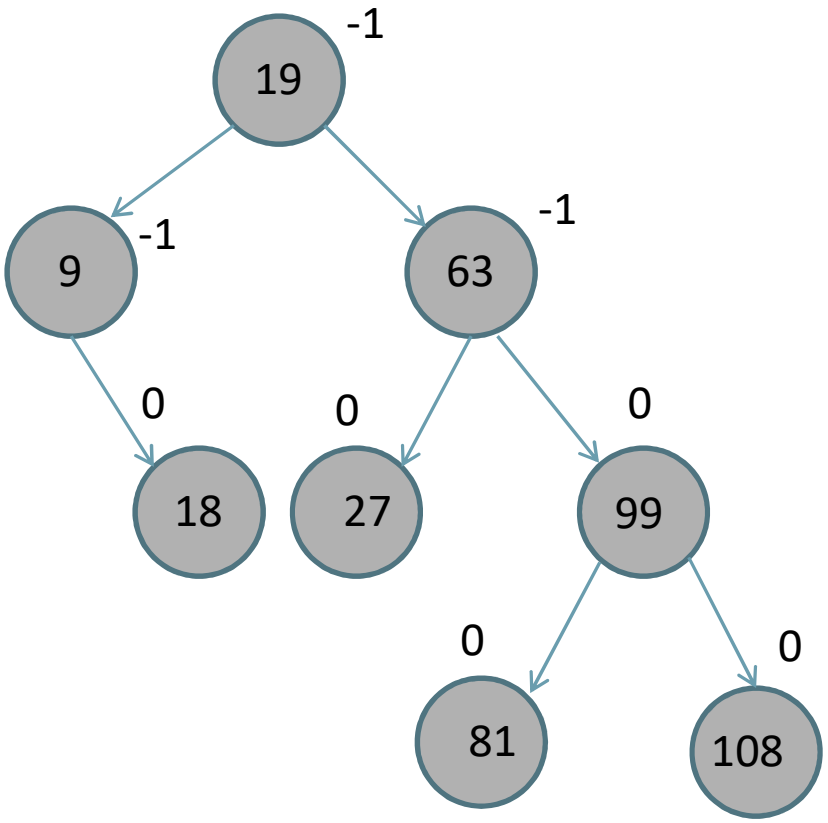
63, 9, 19, 27, 18, 108, 99, 81



63, 9, 19, 27, 18, 108, 99, 81

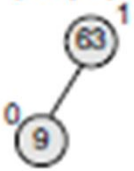


63, 9, 19, 27, 18, 108, 99, 81

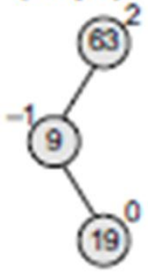


(Step 1)

(Step 2)

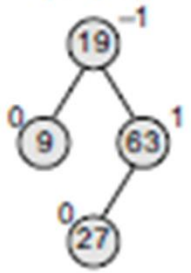


(Step 3)

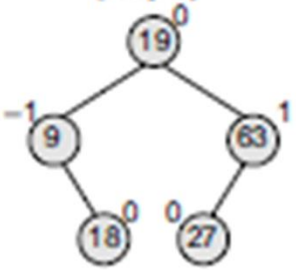


After LR Rotation
(Step 4)

(Step 5)

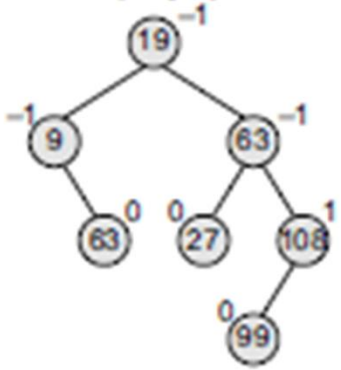


(Step 6)

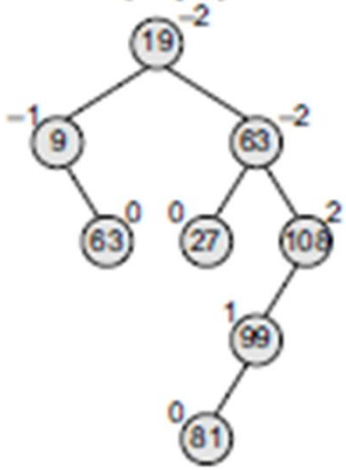


(Step 7)

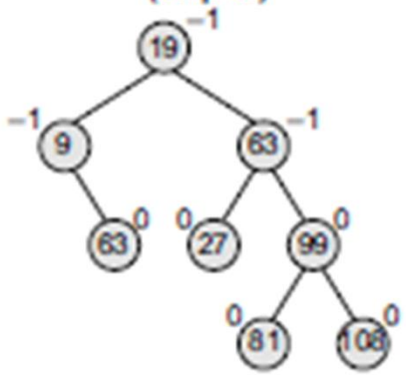
(Step 8)



(Step 9)



After LL Rotation
(Step 10)

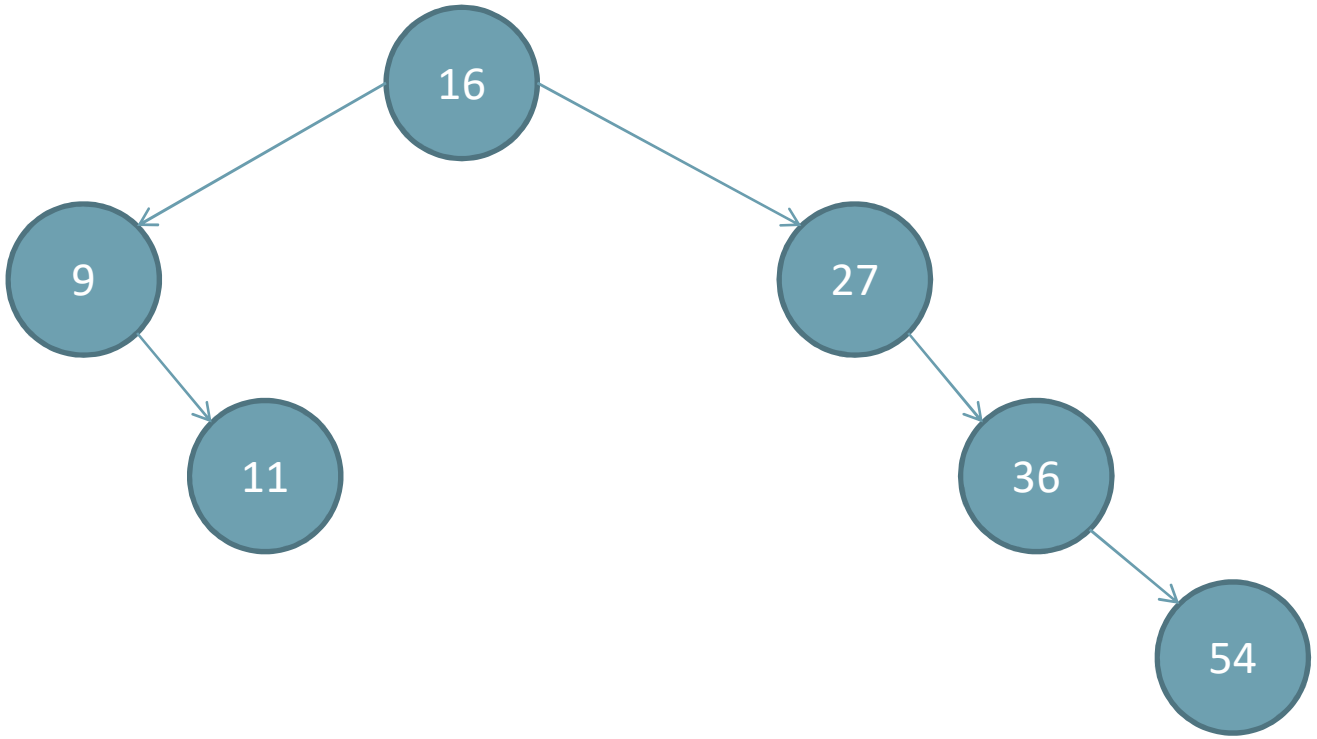


Kanak Kalyani

AVL TREE

- Construct an AVL Tree with following nodes:

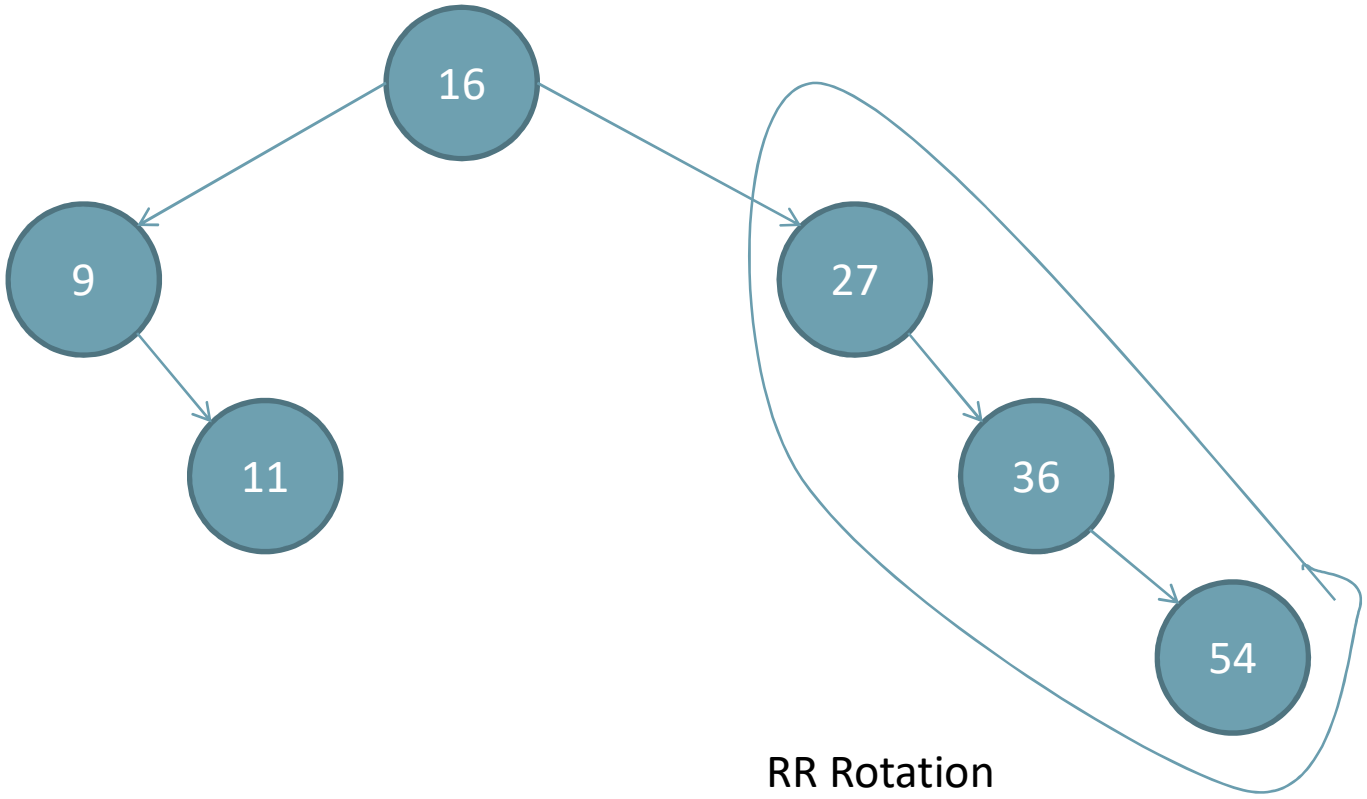
16, 27, 9, 11, 36, 54, 81, 63, 72



AVL TREE

- Construct an AVL Tree with following nodes:

16, 27, 9, 11, 36, 54, 81, 63, 72

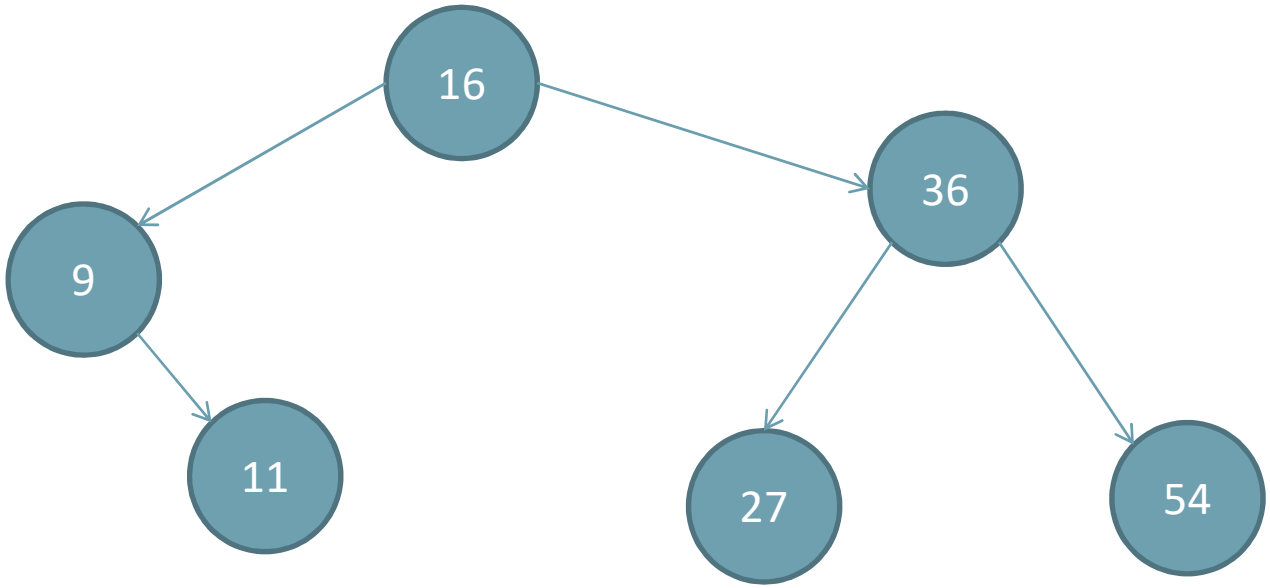


RR Rotation

AVL TREE

- Construct an AVL Tree with following nodes:

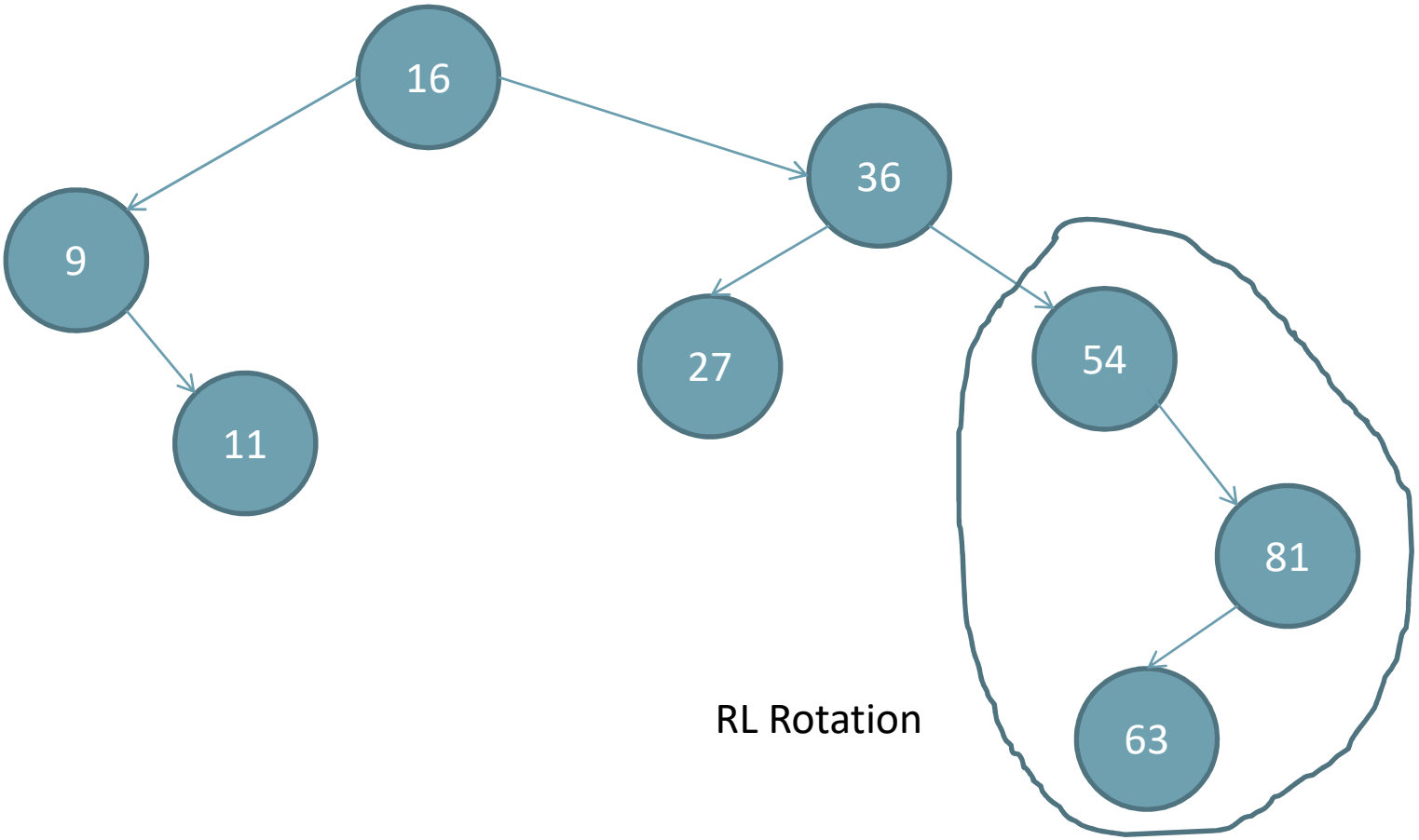
16, 27, 9, 11, 36, 54, 81, 63, 72



AVL TREE

- Construct an AVL Tree with following nodes:

16, 27, 9, 11, 36, 54, 81, 63, 72

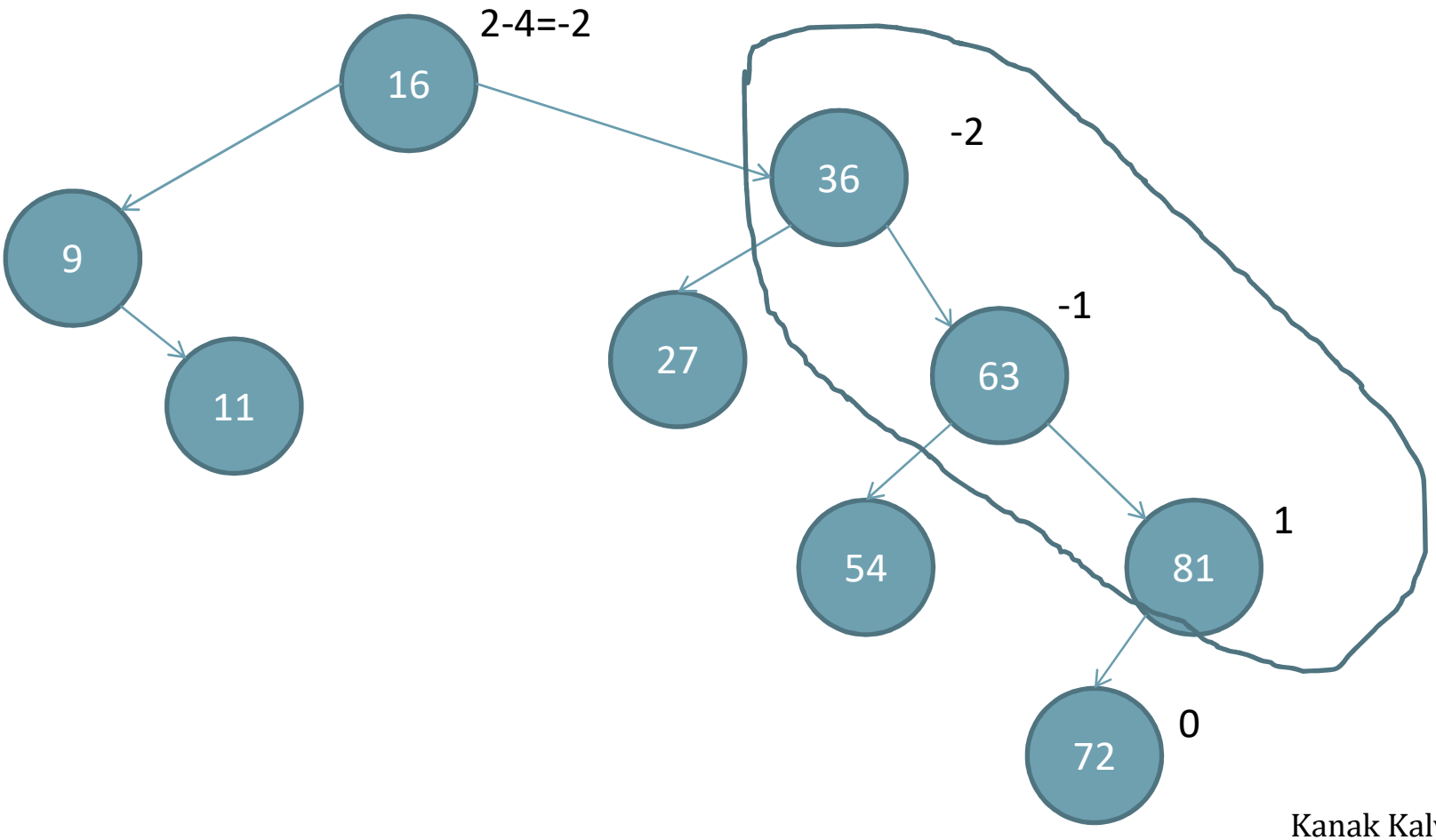


RL Rotation

AVL TREE

- Construct an AVL Tree with following nodes:

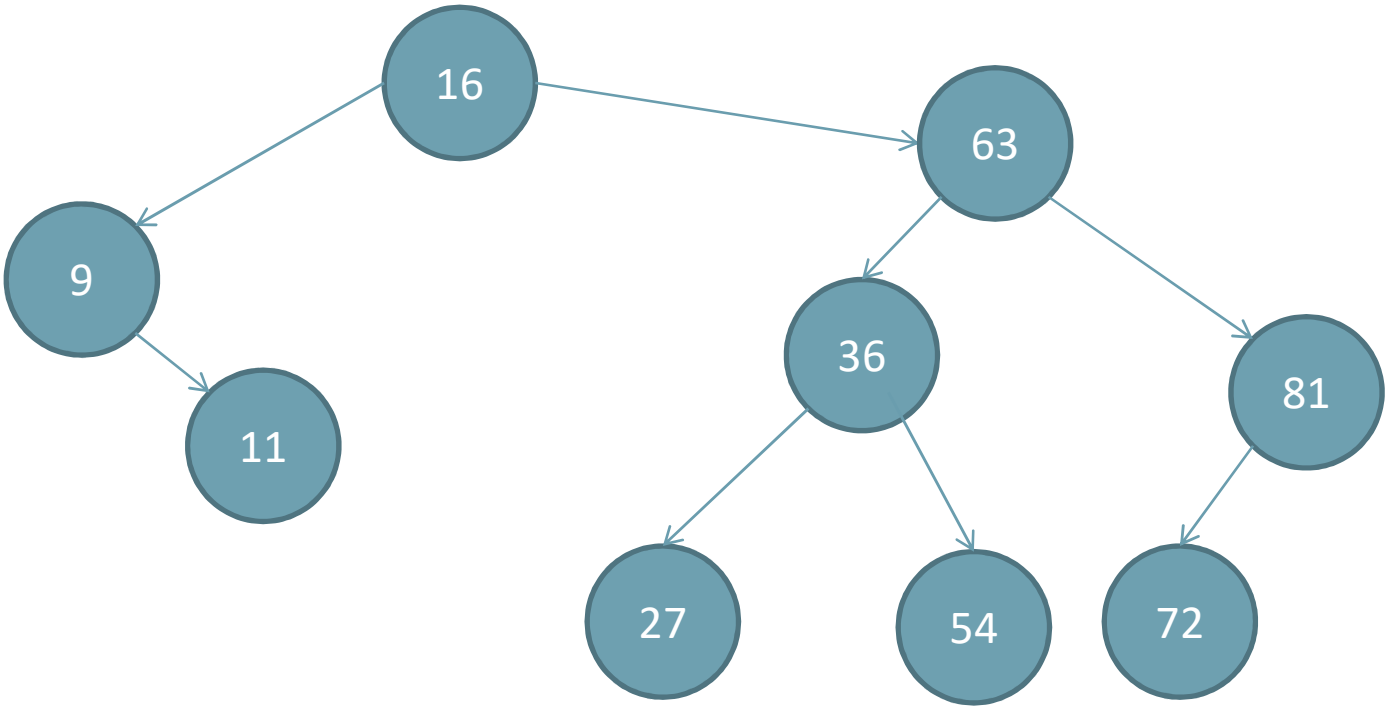
16, 27, 9, 11, 36, 54, 81, 63, 72



AVL TREE

- Construct an AVL Tree with following nodes:

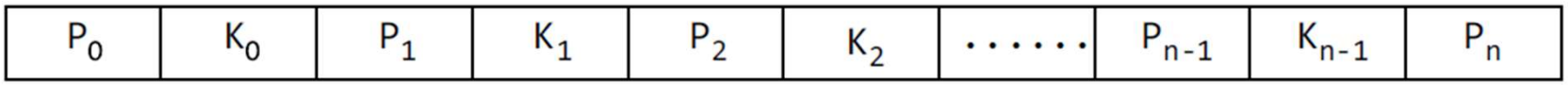
16, 27, 9, 11, 36, 54, 81, 63, 72



Multi-way Search Trees

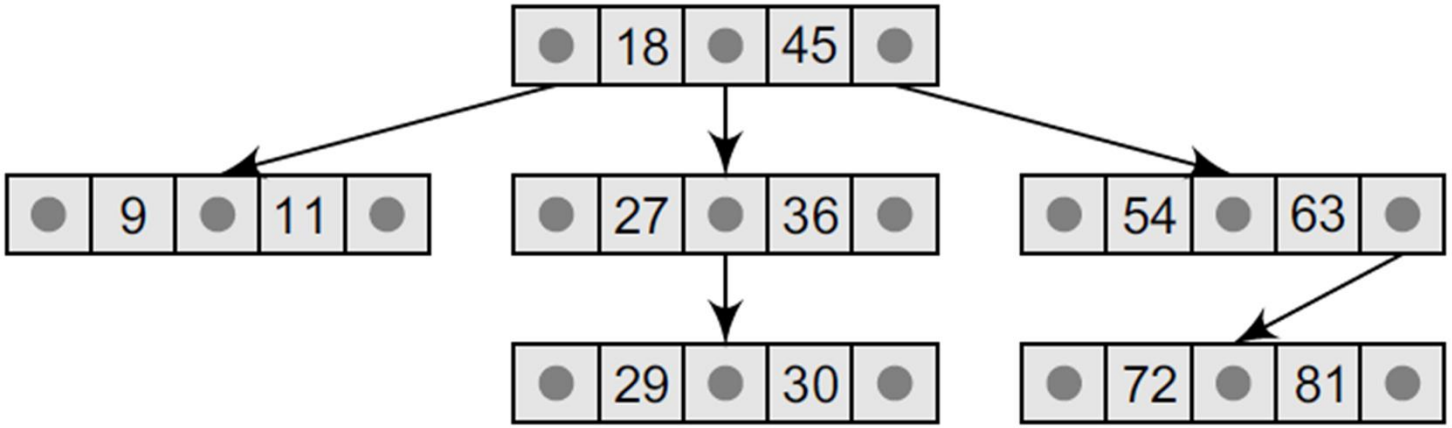
M-way search tree

- *M-way* search tree is similar to binary tree but has M – 1 values per node and M subtrees.
- M is called the degree of the tree
- If M = 2, means it has one value and two sub-trees
- every internal node of an M-way search tree consists of pointers to M sub-trees and contains M – 1 keys, where M > 2.



M-way search tree

- M-way search tree of order 3



M-way search tree

Properties of an M-way search tree

- Key values in the sub-tree pointed by P0 are less than the key value K0. Similarly, all the key values in the sub-tree pointed by P1 are less than K1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by Pi are less than Ki, where 0 <= i <= n–1.
- Key values in the sub-tree pointed by P1 are greater than the key value K0. Similarly, all the key values in the sub-tree pointed by P2 are greater than K1, so on and so forth. Thus,the generalized rule is that all the key values in the sub-tree pointed by Pi are greater than Ki–1, where 0 <= i <= n–1.

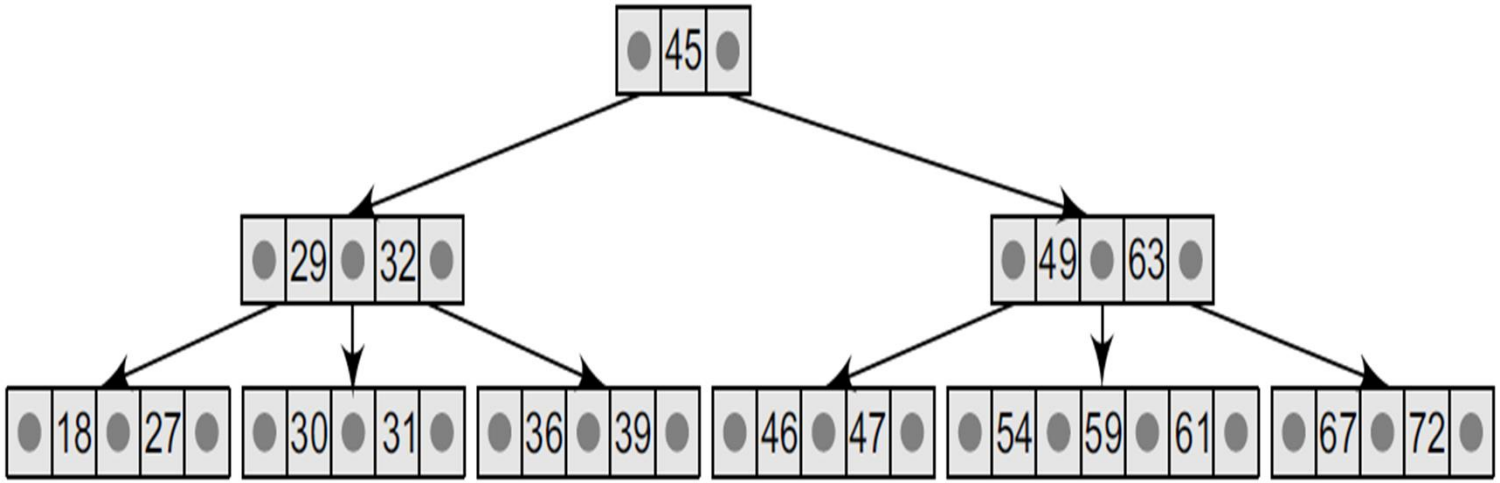
B - Tree

- B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970
- It was widely used for Disk access.
- A B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees.
- A B tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small

B - Tree

- A B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree.
- In addition it has the following properties:
  1. Every node in the B tree has at most m children.
  2. Every node in the B tree except the root node and leaf nodes has at least (minimum) m/2 children.
  3. The root node has at least two children if it is not a terminal (leaf) node.
  4. All leaf nodes are at the same level.

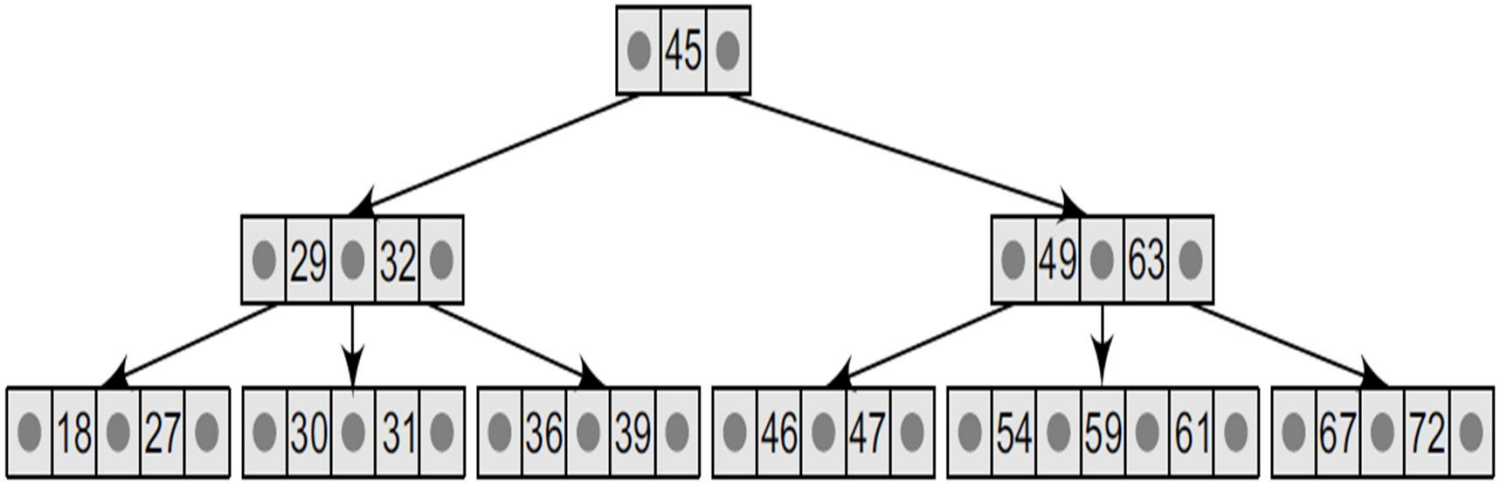
B - Tree of order 4



Kanak Kalyani

Searching for an Element in a B Tree

- Search 31



Inserting a node in B Tree

1. Search the B tree to find the leaf node where the new key value should be inserted.
2. If the leaf node is not full, that is, it contains less than m–1 key values, then insert the new element in the node keeping the node's elements ordered.
3. If the leaf node is full, that is, the leaf node already contains m–1 key values, then

   (a) insert the new value in order into the existing set of keys,

   (b) split the node at its median into two nodes (note that the split nodes are half full), and

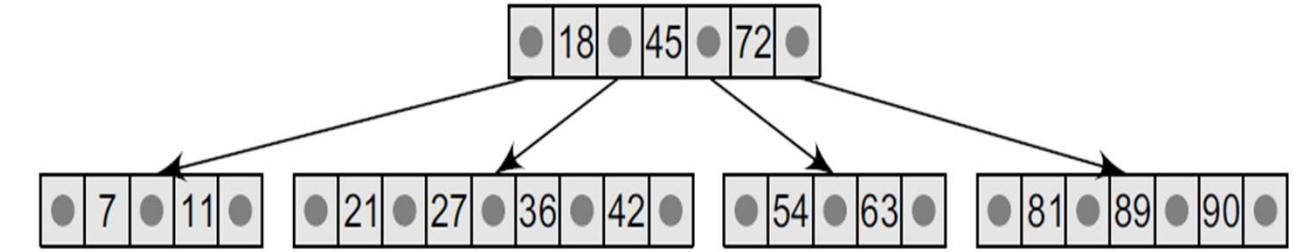
   (c) push the median element up to its parent's node. If the parent's node is already full, then

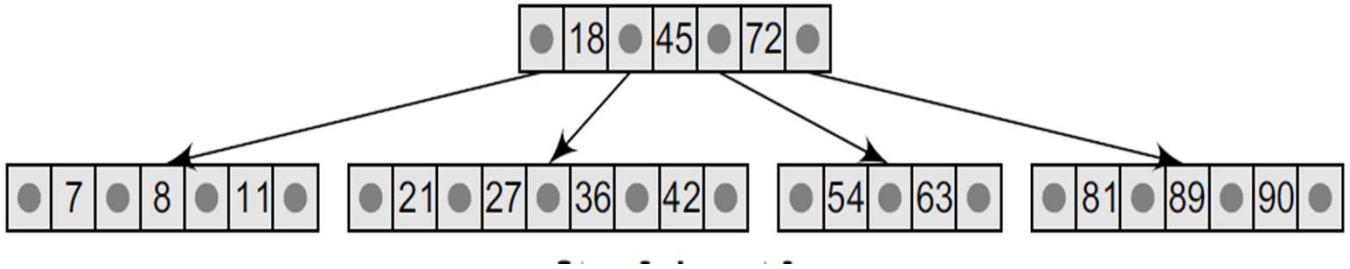
   split the parent node by following the same steps.

Inserting a node in B Tree

B Tree of Order 5

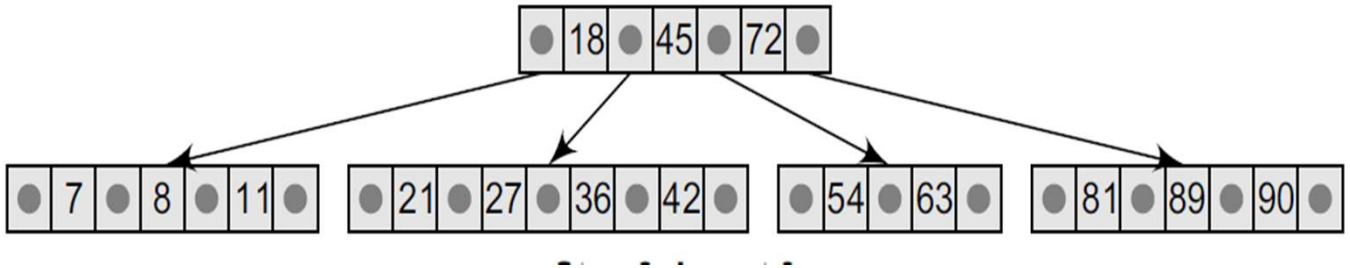
- Insert 8

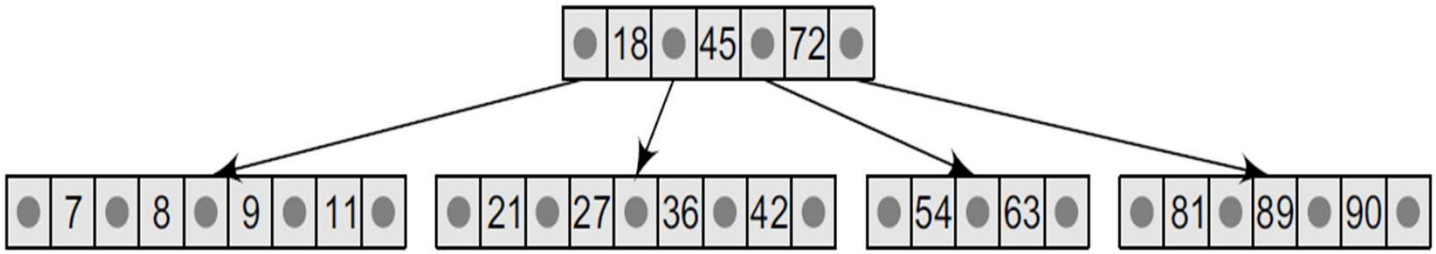




Inserting a node in B Tree

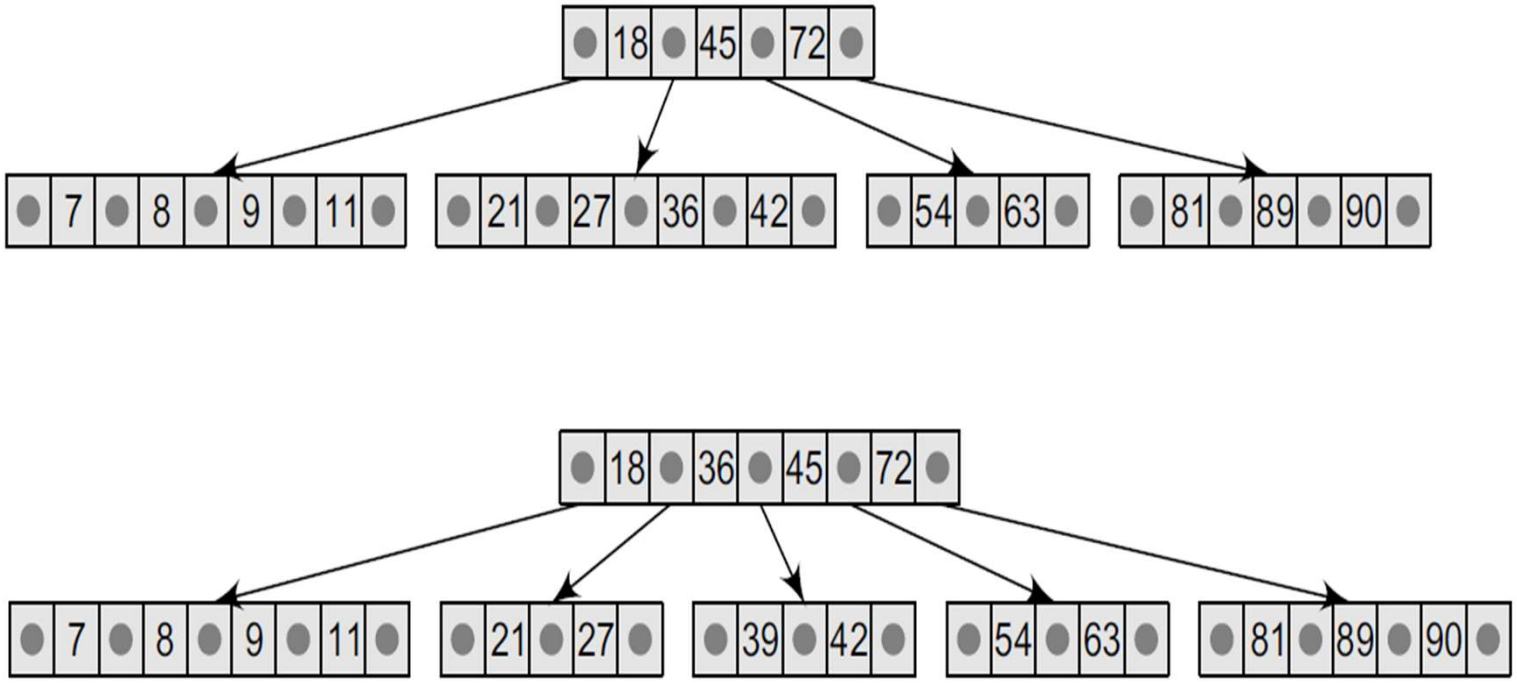
- **B Tree of Order 5**
- Insert 9





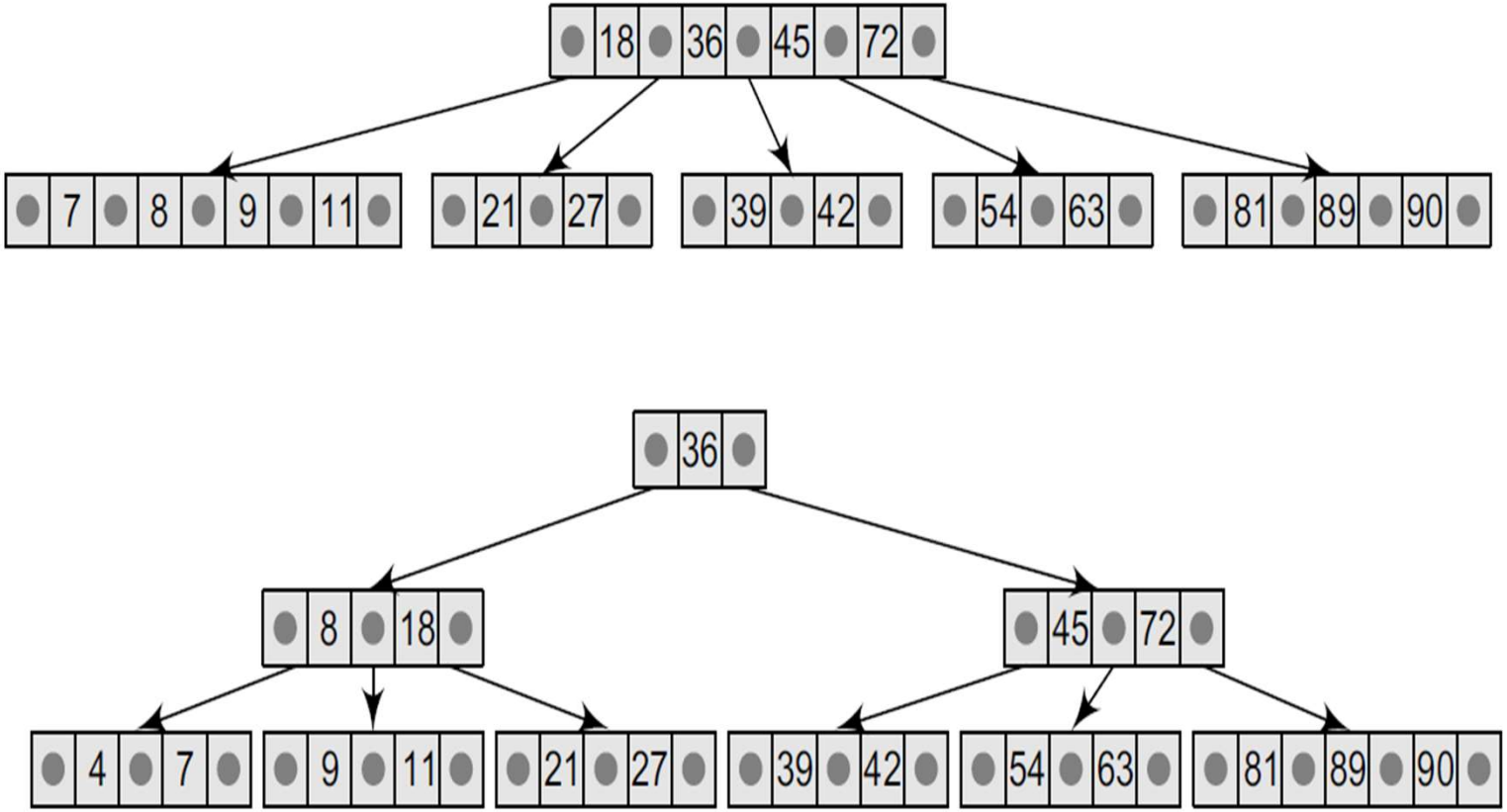
Inserting a node in B Tree

- **B Tree of Order 5**
- Insert 39



Inserting a node in B Tree

- **B Tree of Order 5**
- Insert 4

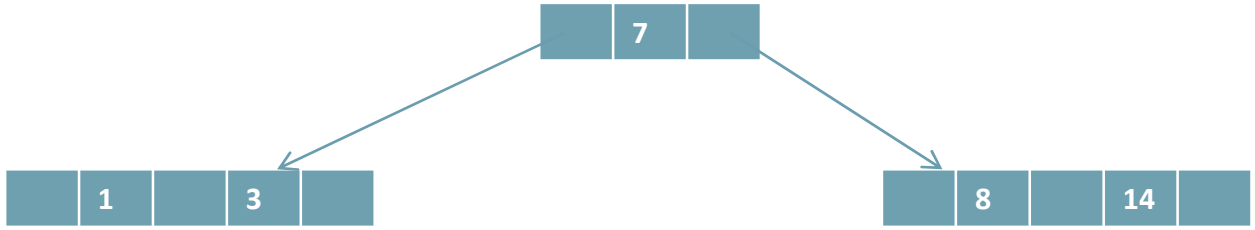


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 8

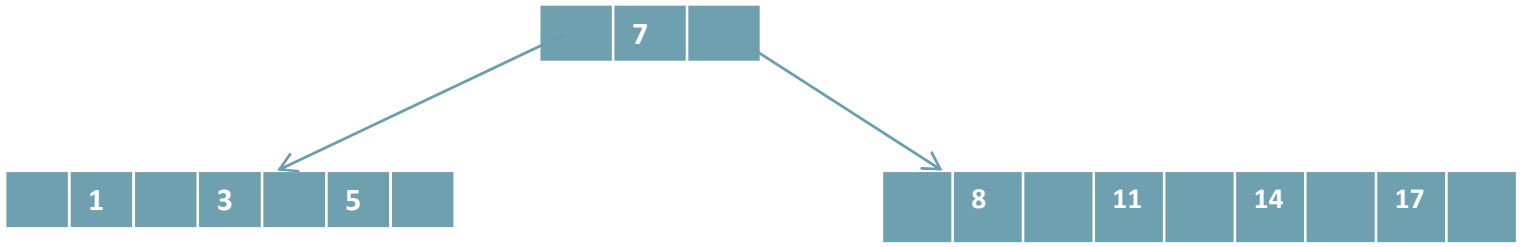


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 5, 11,17

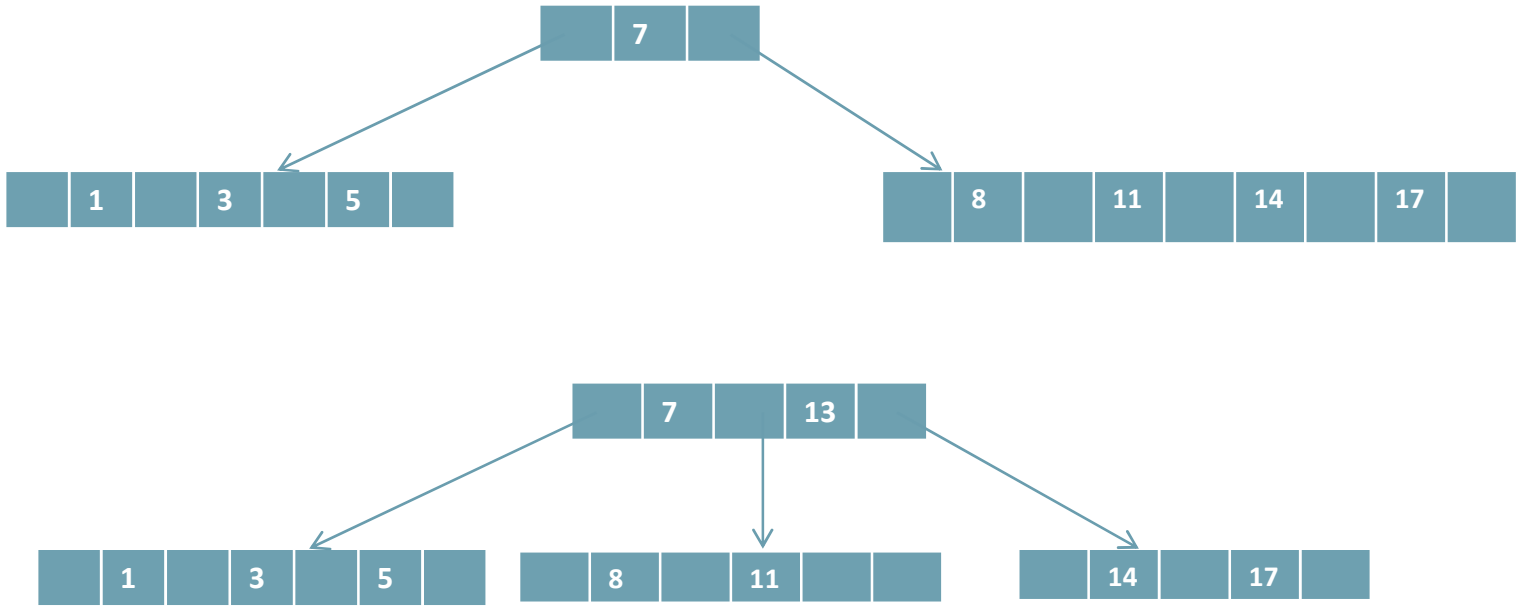


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 13

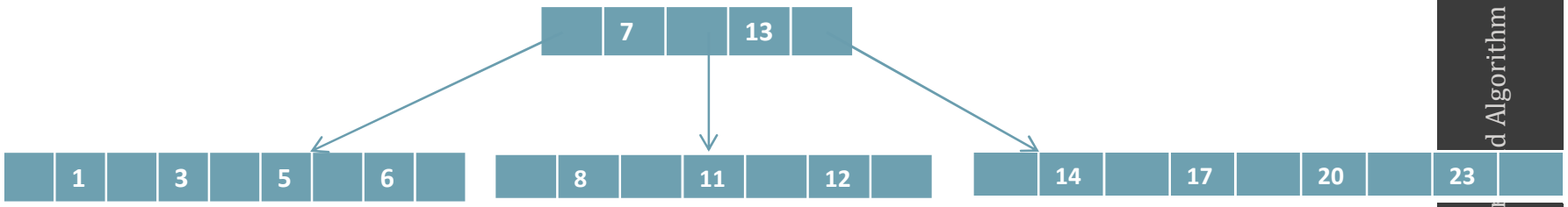


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 6,23,12,20

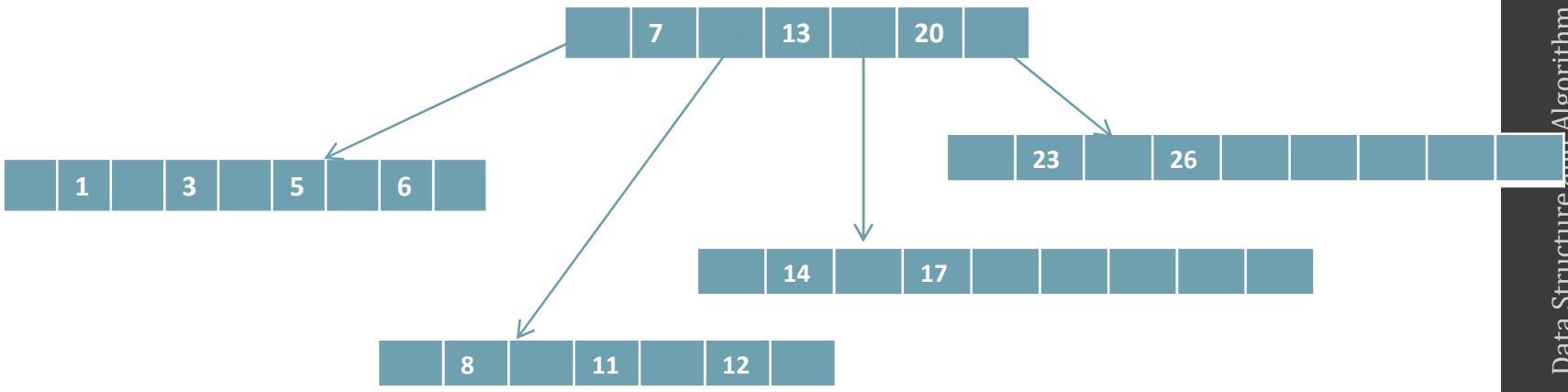


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 26

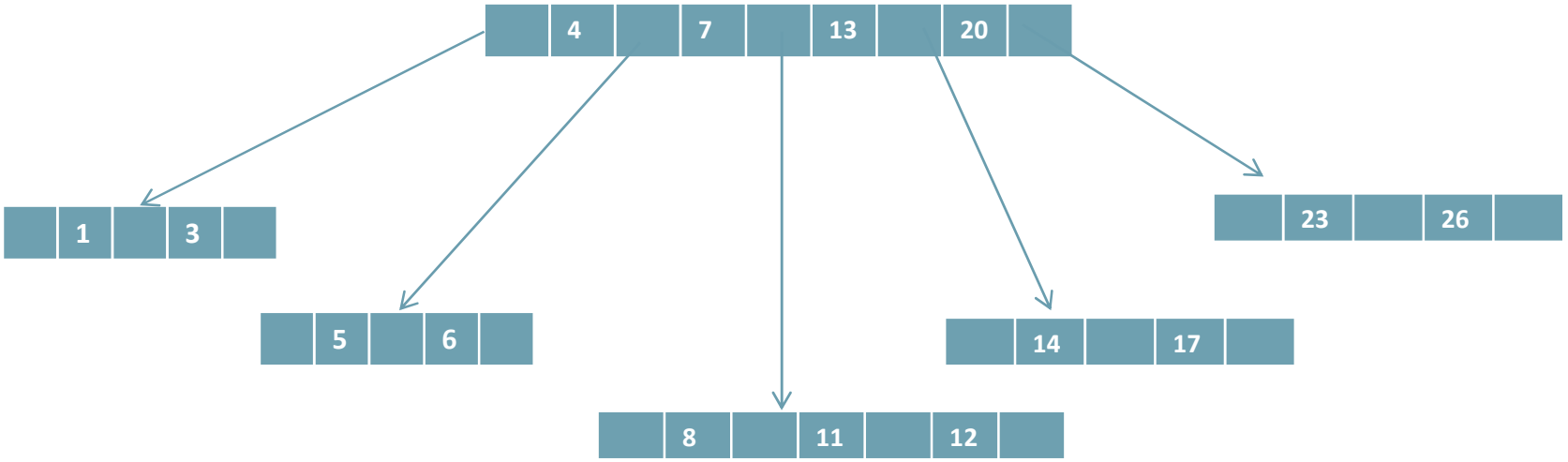


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

Insert 4

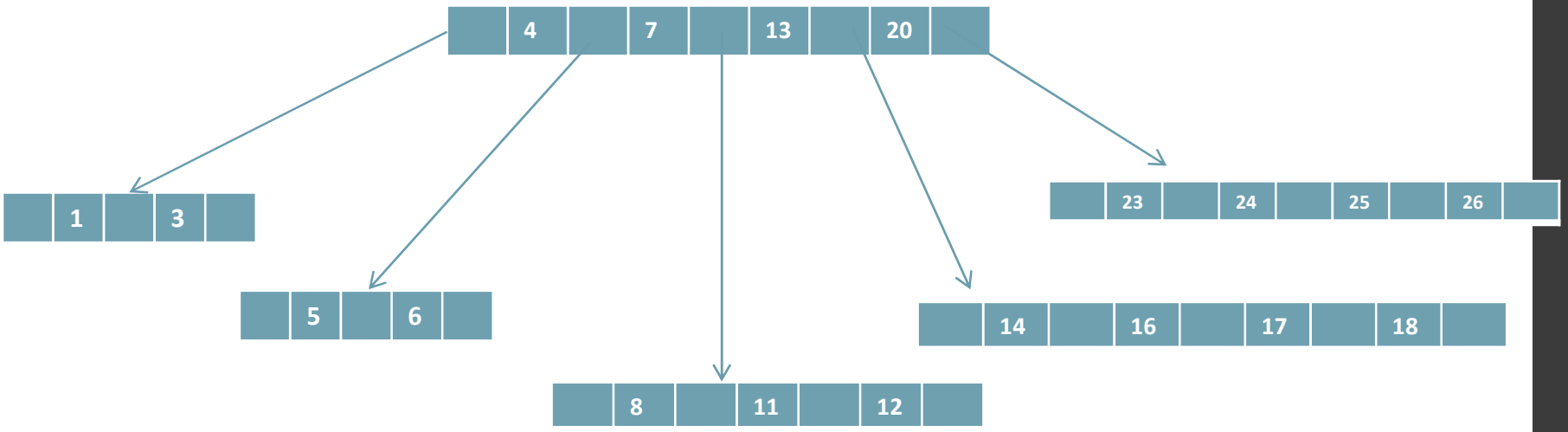


Create a B Tree

- Create a B Tree of order 5

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25

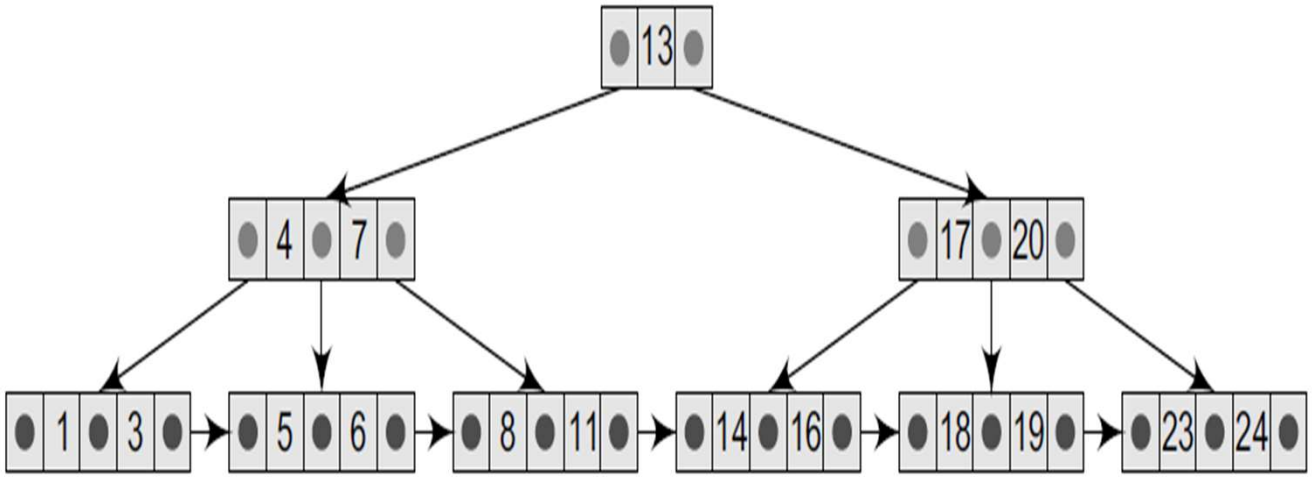
Insert 16,18,24,25



B+ Tree

- A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a *key*
- A B+ tree stores all the records at the leaf level of the tree; only keys are stored in the interior nodes
- The leaf nodes of a B+ tree are often linked to one another in a linked list
- B+ trees are used to store large amounts of data that cannot be stored in the main memory
- With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory

B+ Tree



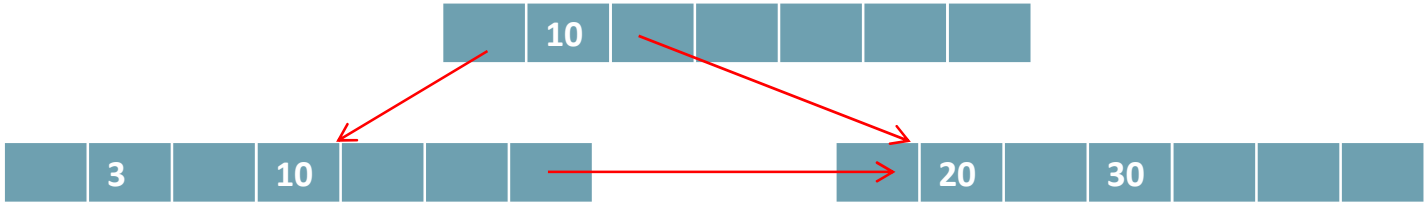
Creating B+ Tree

- Construct a B+ Tree of Order 4

  10,20,30,3,6,9,15,27,32,34,48,

Insert 10, 20, 30

Insert 3



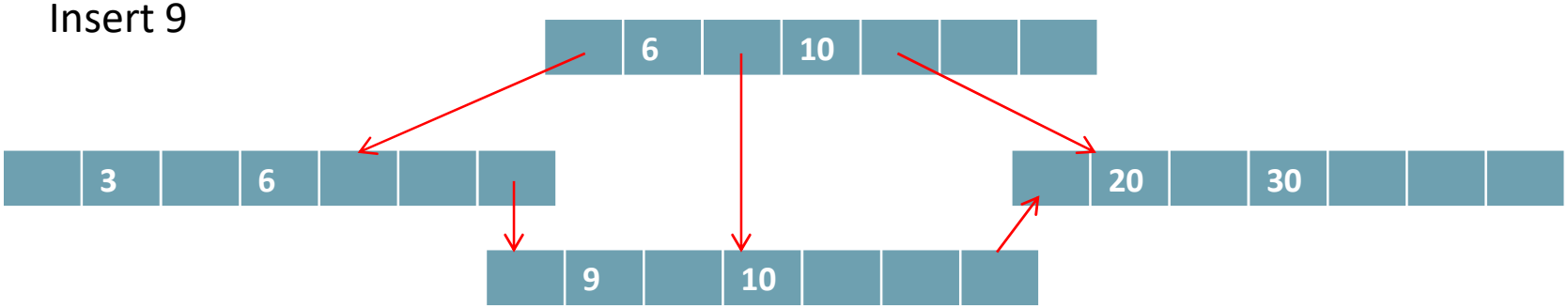
Creating B+ Tree

- Construct a B+ Tree of Order 4

  10,20,30,3,6,9,15,27,32,34,48,

Insert 6

Insert 9

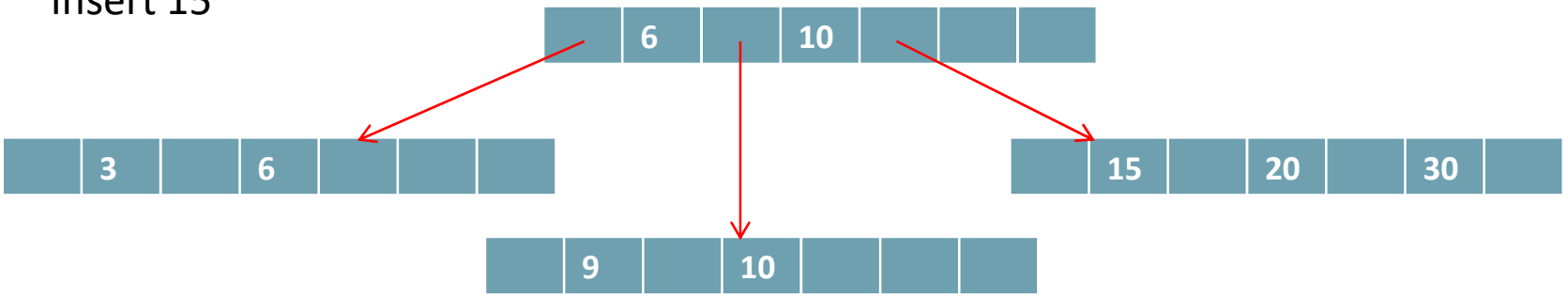


Creating B+ Tree

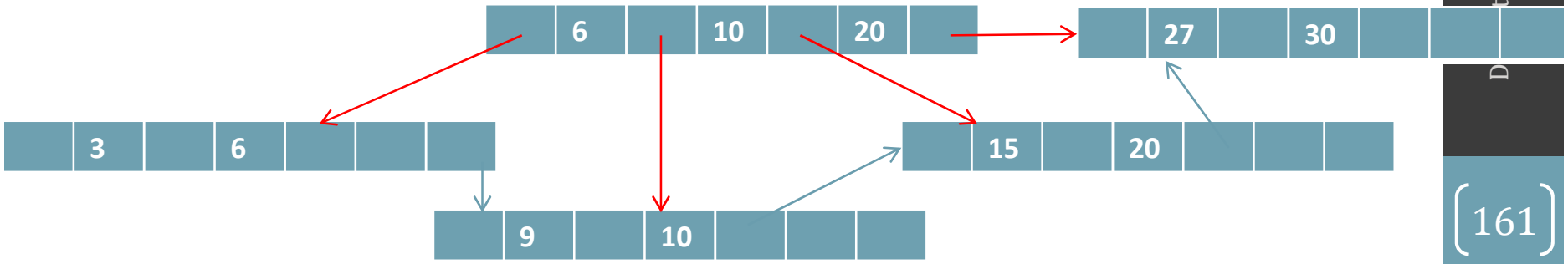
- Construct a B+ Tree of Order 4

  10,20,30,3,6,9,15,27,32,34,48,

Insert 15



Insert 27

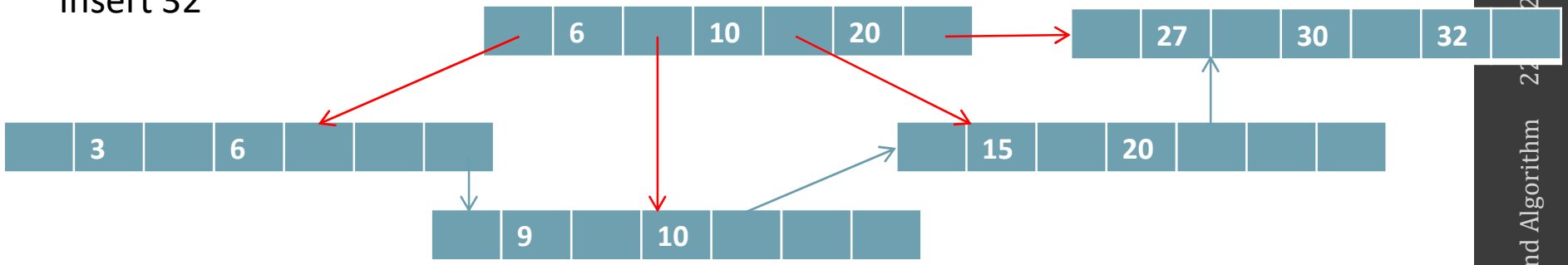


Creating B+ Tree

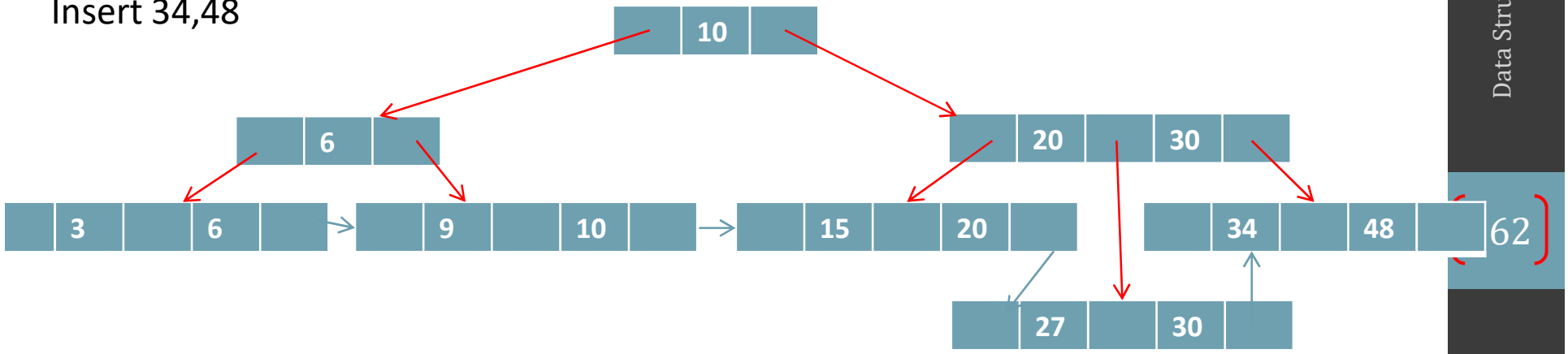
- Construct a B+ Tree of Order 4

  10,20,30,3,6,9,15,27,32,34,48,

Insert 32



Insert 34,48



Advantages of B+ trees

1. Records can be fetched in equal number of disk accesses
2. It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level
3. Height of the tree is less and balanced
4. Supports both random and sequential access to records
5. Keys are used for indexing

Comparison Between B Trees and B+ Trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

- Create a B+ Tree of order 4
- 1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42