Module 2

Advanced Data Structures

Advanced Data Structures (5 Hrs) :

B/B+ tree
Red-Black Trees
Heap operations
Implementation of queue using heap
Topological sort

Self-learning Topics: Implementation of Red-Black Tree and Heaps.

Data structures : Topics marked in green are in Syllabus

1. Array
2. Binary Tree (BST)
3. AVL Tree
4. B Tree
5. B+ Tree
6. 2-3 Tree
7. Read Black Tree
8. Binary Heap
9. Graphs : Topological Sort

Data Structures :

1. Array :

Worst case time complexity to search an element in an array = n searches = O(n)

1	2	3	4	5	6	7 = n
6	3	9	1	5	7	11

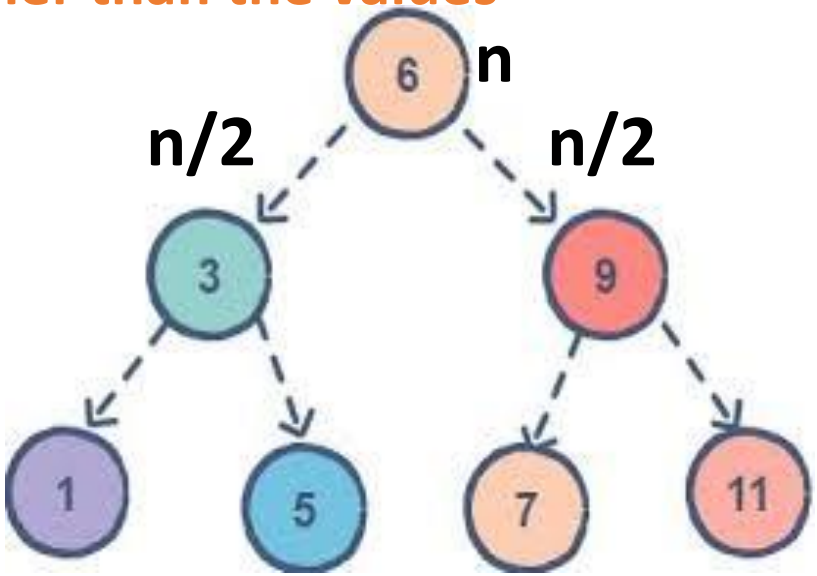
Max No of comparisons needed to search , insert, delete in BST = Height of the BST = O (h) = O ($\log_2$ n)

Max No of comparisons needed to insert ele 14 in above BST =3 =$\log_2$ 7 = 2.81 approx. = 3 = height of BST

2. Binary Search Tree (BST) :

Advantages of BSTs :

1. **At every node, the value in the left sub-tree is always smaller than the values in the right sub tree.**



2. **This helps in fast search operation as search reduces to half of the tree at each level .**

3. **Thus, in Binary Search Tree (BST) , search takes less time.**

We divide the tree into two sub trees of size n/2 each giving a time complexity of (log n) which is very less compared to n.

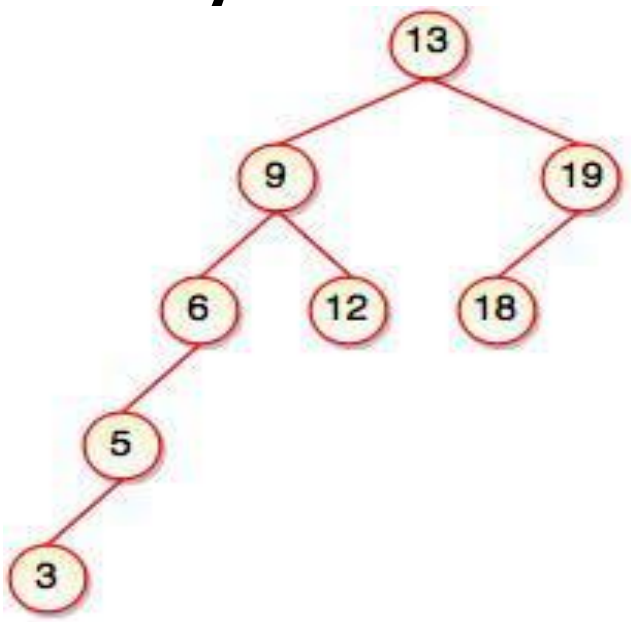
Disadvantages of BSTs :

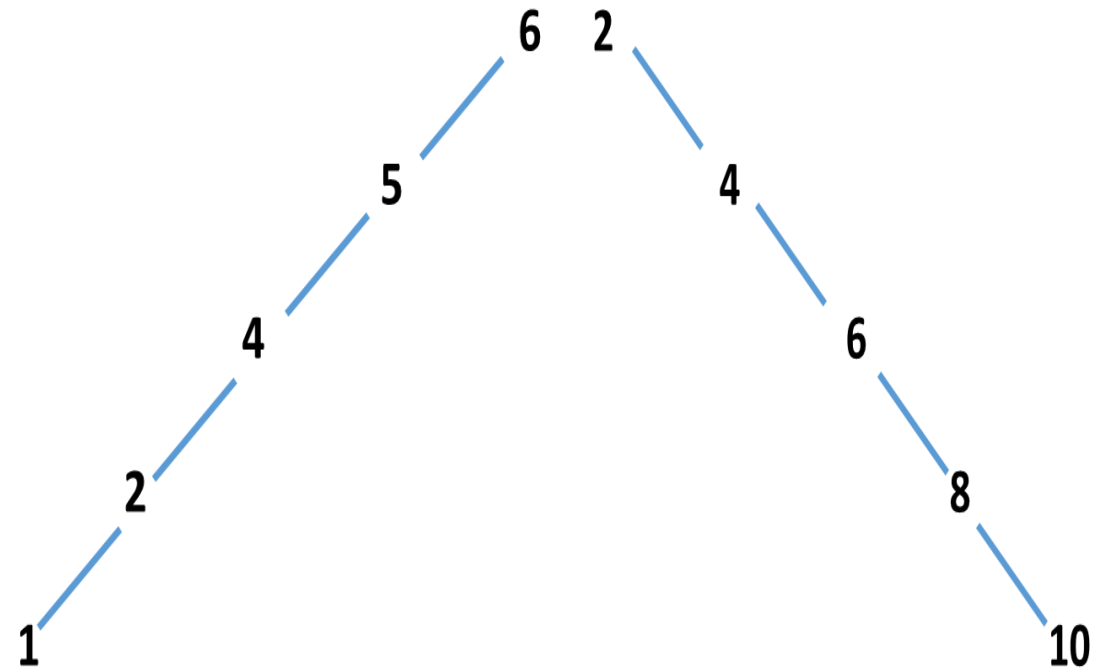
1	2	3	4	5	6	7
6	3	9	1	5	7	11

1. BSTs require additional space for storage of two pointers : left pointer and right pointer along with data value (which is not required in an array).

2. BST may be skewed to the left or to the right giving a time complexity of O(n) which is same as that of array.

3. This problem occurs because the BSTs are not balanced.





Left skewed tree Right skewed tree

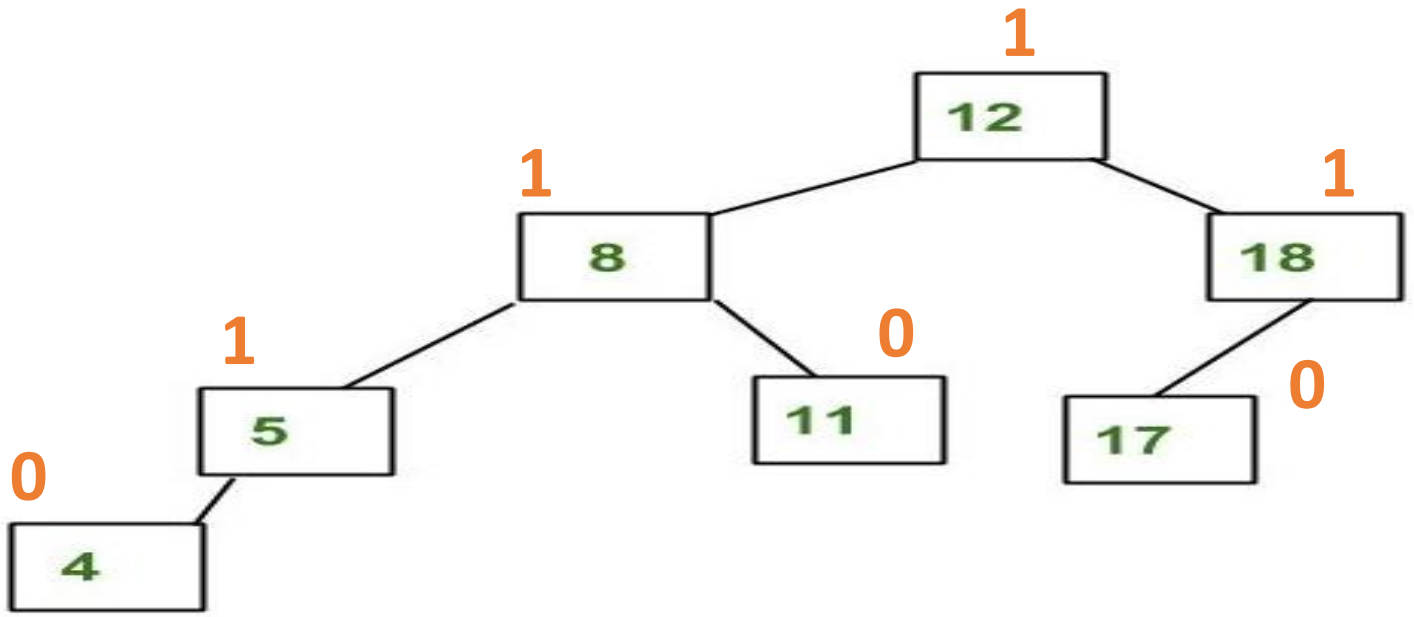
3. AVL tree :

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right sub trees cannot be more than one for all nodes.

AVL trees are balanced on the height of the tree, so also called as height balanced search trees.

Balancing Factor (BF) =
Height of Left Sub tree (LST) -
Height of right Sub tree (RST)

In AVL tree, Only three values
Permissible for BF are : = { -1 , 0 , 1 }



Due to their height balancing, AVL trees result in faster search / retrieval of data values.
In extreme case, AVL tree gives a cost of O(log n) for search, insertion and deletion operation.

Advanced Data Structures and Analysis
ITDO 5014

Current session :

B Tree :

- **What is a B Tree**
- **Properties of B Tree**
- **Searching and Traversal in B Tree**
- **Comparison of B tree with BST and AVL Tree**

B tree : [Cormen]

B-Tree is a self-balancing search tree.

In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory.

B-Trees are used to store the huge amount of data that cannot fit in main memory.

When the number of keys is high, the data is stored in and read from disk in the form of blocks.

Disk access time is very high compared to main memory access time.

The main idea of using B-Trees is to reduce the number of disk accesses.

Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree = O(log n)

B-tree is usually a shallow and fat tree.

Generally, a B-Tree node size is kept equal to the disk block size.

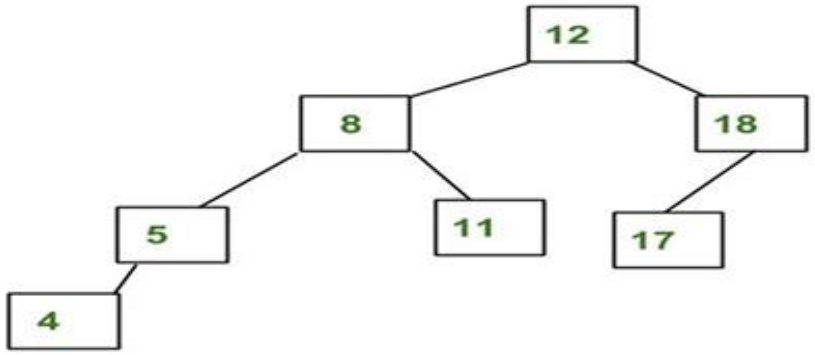
The height – h of B-Trees is kept low by putting maximum possible keys in a B-Tree node.

Since height - h is low for B-Tree :

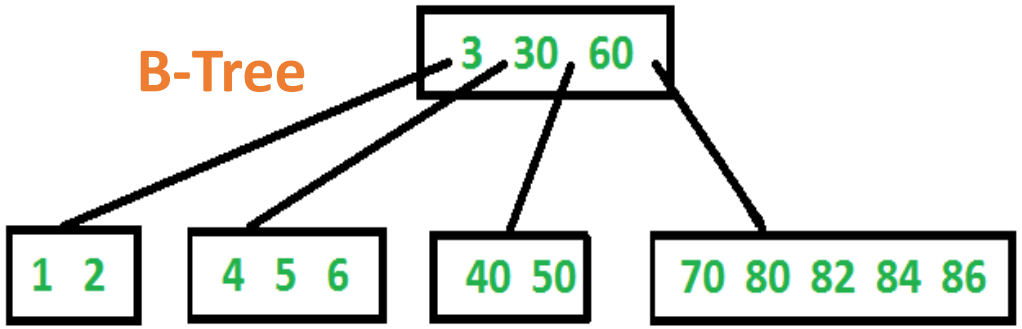
significantly reduces total number of disk accesses for most of the operations compared to balanced BSTs like AVL Tree, Red-Black Tree, ..etc.

Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree = O(log n)

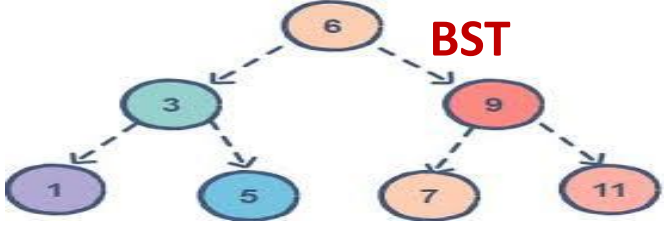
AVL Tree



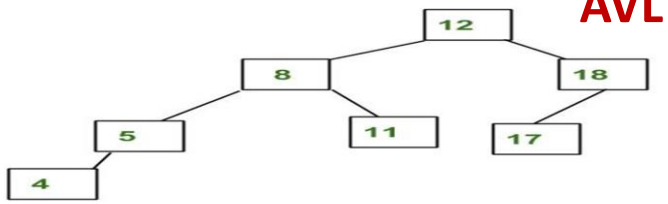
B-Tree



Properties of B-Tree : [Cormen]



BST

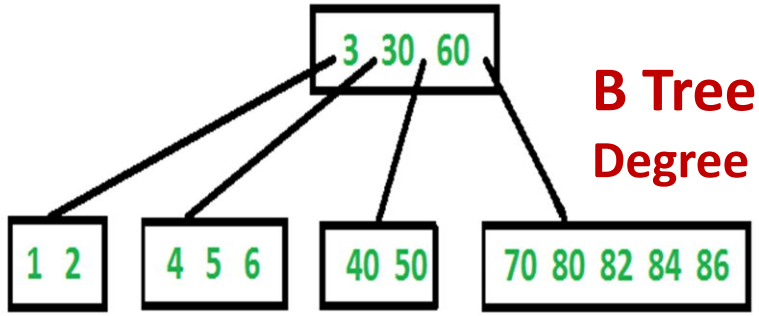


AVL Tree

1) All leaves are at same level.

2) A B-Tree is defined by the term minimum degree ‘t’. Value of t depends upon disk block size.

3) Every node except root must contain at least t-1 keys.
Root may contain minimum 1 key (i.e. two children).



B Tree
Degree t = 3

4) All nodes (including root) may contain at most 2t – 1 keys.

5) Number of children of a node is equal to the number of keys in it plus 1.

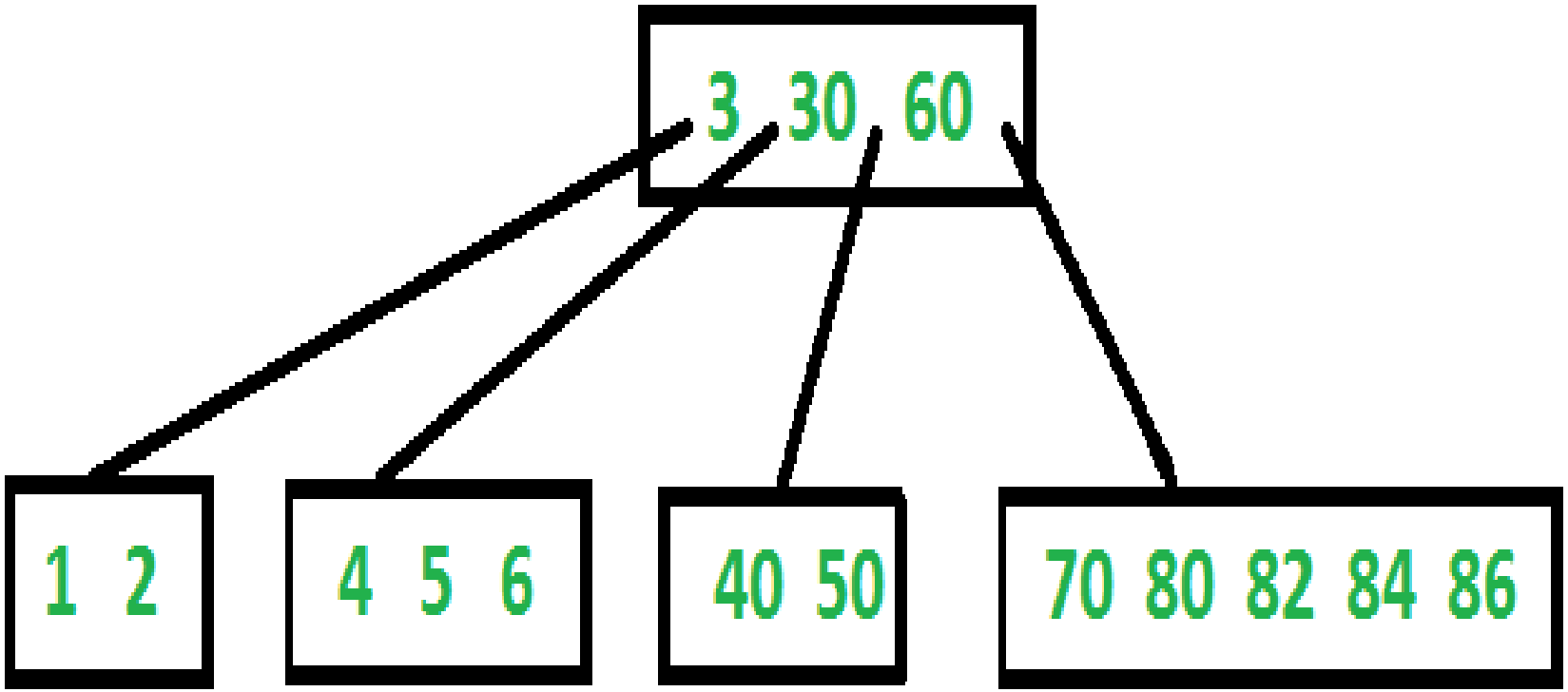
6) All keys of a node are sorted in increasing order.
The child between two keys k1 and k2 contains all keys in the range from k1 and k2.

7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree.
Binary Search Trees grow downward and also shrink from downward.

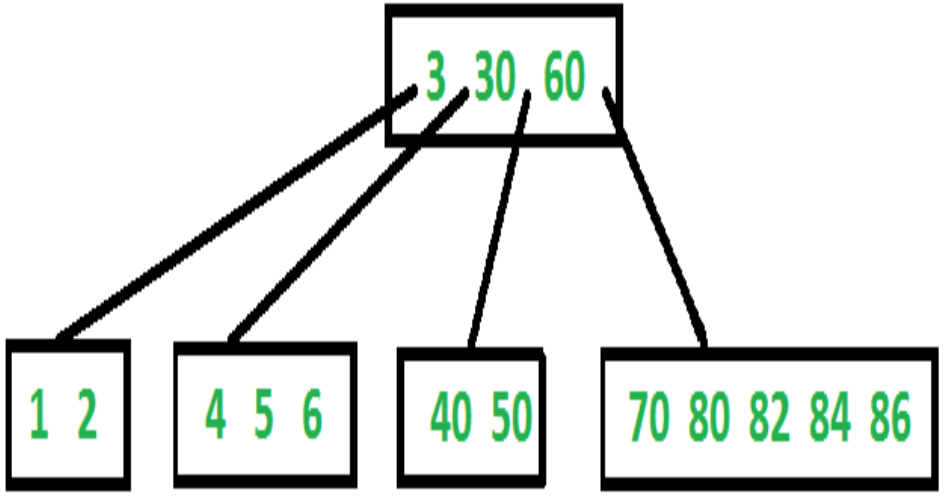
8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Log n) =O(h).

Following is an example B-Tree of minimum degree 3.

Note that in practical B-Trees, the value of minimum degree is much more than 3.



Searching in a B tree :



Search is similar to the search in Binary Search Tree.

Let the key to be searched be k.

We start from the root and recursively traverse down.

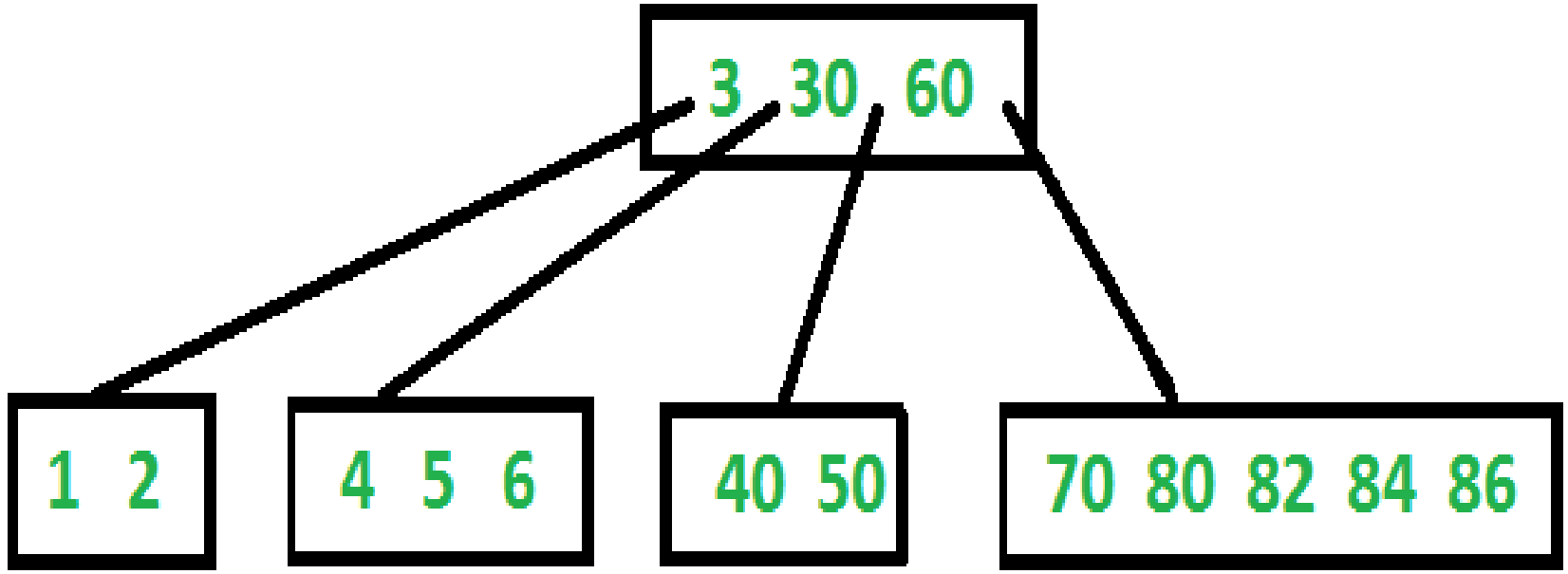
For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.

If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traversal in a B tree :

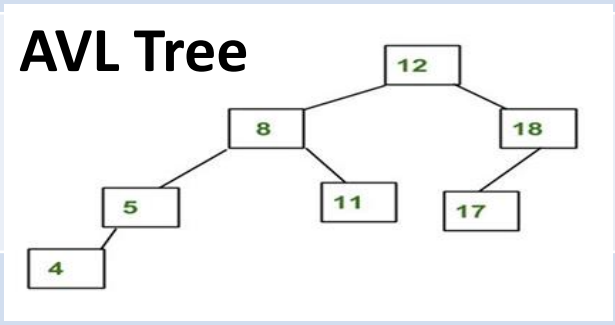
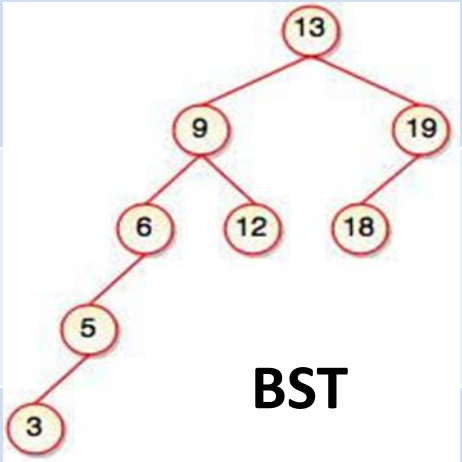
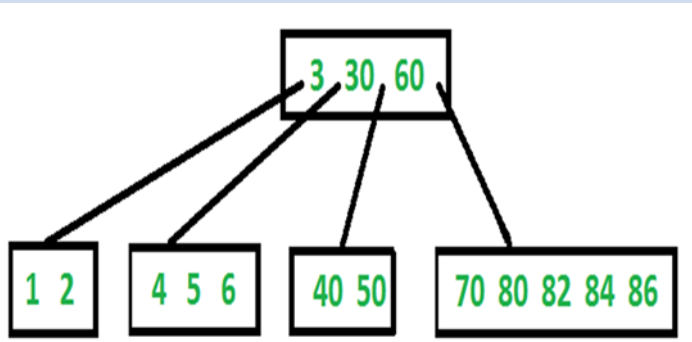
Traversal is also similar to In-order traversal of Binary Tree.

We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.



Summary : Comparison with BST and AVL Tree :

B-Tree is a self-balancing search tree. B-tree is usually a shallow and fat tree.

AVL Tree	B Tree
In a BST and AVL tree, every node stores a single data value .	In a B Tree, every node stores multiple data values .
Each node in a BST and AVL tree can have maximum of two children.	Depending on the degree of a B tree, each node can have more than two children in contrast to a BST or AVL tree.
AVL and Red-Black Trees are used for storing small amount of data that resides in the primary memory.	Unlike AVL and Red-Black Trees, B-Trees are used for storing huge amount of data that resides in the secondary memory for example disks.
AVL Tree   BST	B Tree Degree t = 3 

Insertion in B Tree :

A new key is always inserted at leaf node.

Let the key to be inserted be k. Like BST, we start from root and traverse down till we reach a leaf node.

Once we reach a leaf node, we insert the key in that leaf node.

Unlike BSTs, we have a predefined range on number of keys that a node can contain. So before inserting a key to node, we make sure that the node has extra space.

If a node is FULL (i.e. node has 2t-1 elements) then we use an operation called splitChild() that is used to split a child of a node at the mid element.

In the following diagram, child y of x is being split into two nodes y and z.
Note that the splitChild operation moves a key up and this is the reason B-Trees grow up unlike BSTs which grow down.

Insertion in B Tree :

1) Initialize x as root.

2) While x is not leaf, do following

    a) Find the child of x that is going to be traversed next. Let the child be y.

    b) If y is not full, change x to point to y.

    c) If y is full, split it and change x to point to one of the two parts of y.

        If k is smaller than mid key in y, then set x as first part of y. Else second part of y.
        When we split y, we move a key from y to its parent x.

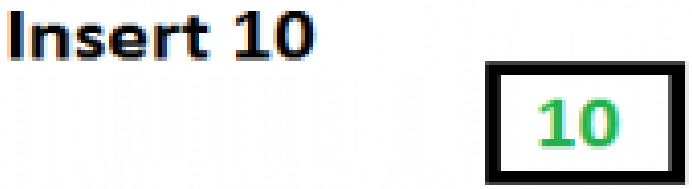
3) The loop in step 2 stops when x is leaf.
    After exiting from loop in step 2, x must have space for 1 extra key as we have been splitting all nodes in advance.
    So simply insert k to x.

Insert a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree of minimum degree 't' as 3.

Solution: MIN = t – 1 = 3-1 = 2 Max = 2t – 1 = 2*3 - 1 = 5

Initially root is NULL. Let us first insert 10.

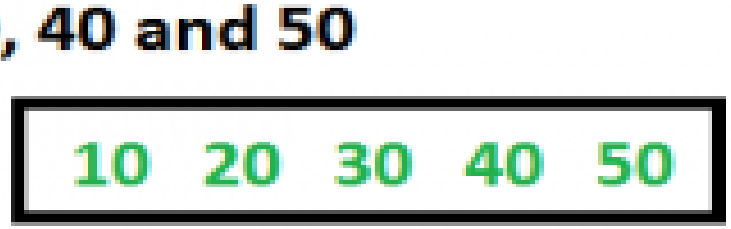
1.



Let us now insert 20, 30, 40 and 50.

They all will be inserted in root because maximum number of keys a node can accommodate is 2*t – 1 which is 5.

2.

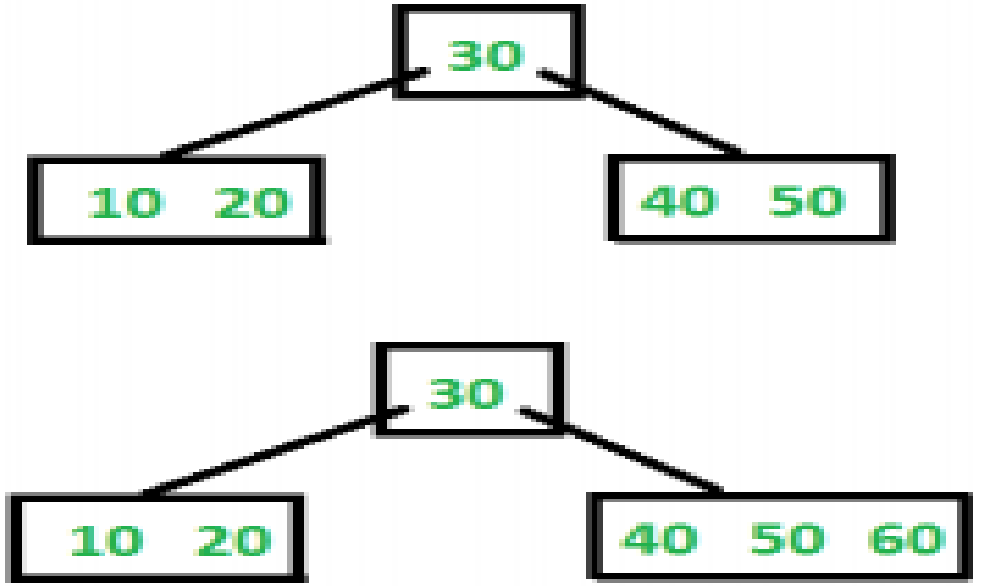


MIN = t – 1 = 2 Max = 2t – 1 = 5

Insert 20, 30, 40 and 50

Insert 60 : Since root node is full, it will first split into two (always at the mid element), then 60 will be inserted into the appropriate child.

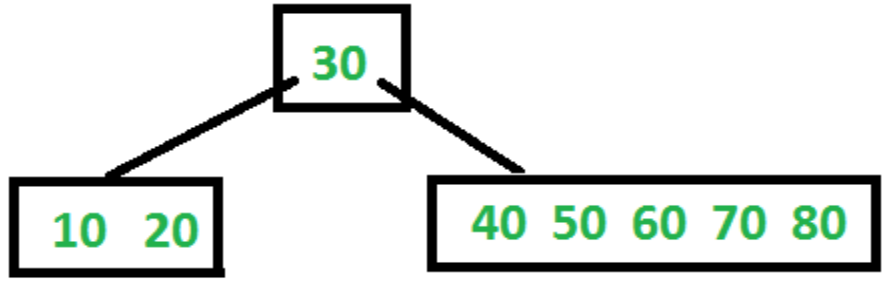
Insert 60



MIN = t – 1 = 2 Max = 2t – 1 = 2*3 - 1 = 5

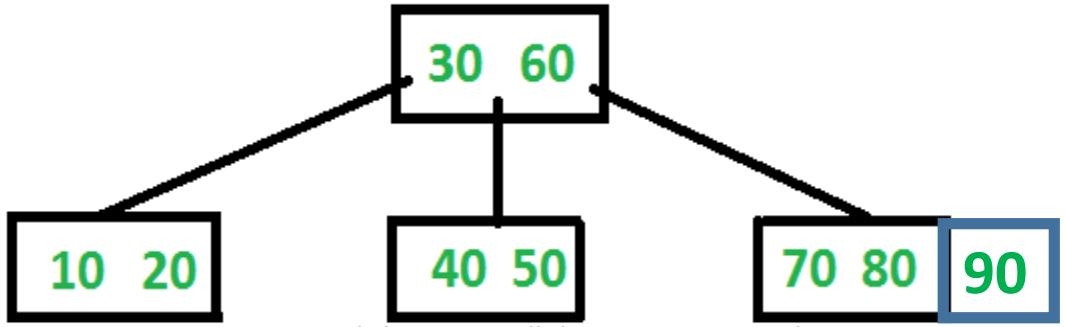
Insert 70 and 80 : These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Insert 90 : This insertion will cause a split. The middle key will go up to the parent.

Insert 90



Example 2 : Insert a sequence of characters R, Y, F, X, A, M, C, D, E, T, H, V, L, W, G in given order in an initially empty B-Tree of minimum degree 't' as 3.

Solution: t = 3 ➔ Min No of keys per Node = t -1 = 3-1 = 2

Max No of keys per Node = 2t -1 = 2*3 - 1 = 5

1. Initially root is NULL. Let us first insert R.

Insert R

Insert Y

Insert F

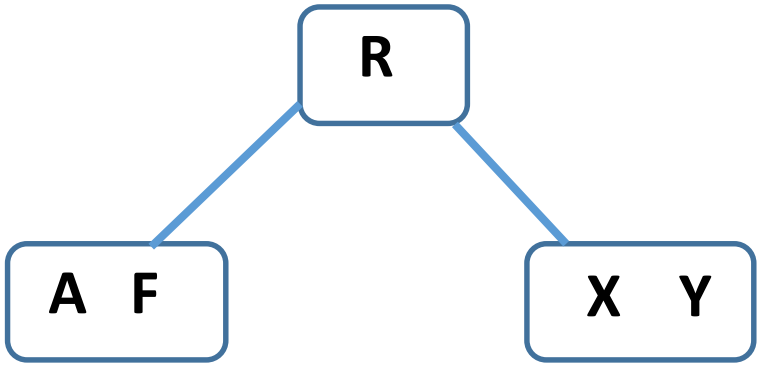
Insert X

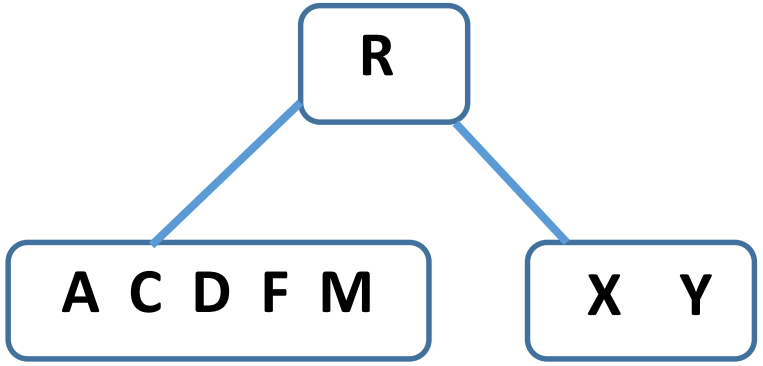
Insert A

Example 2 : Insert a sequence of characters R, Y, F, X, A, M, C, D, E, T, H, V, L, W, G in given order in an initially empty B-Tree of minimum degree 't' as 3.

Solution: t = 3 ➔ Min = t -1 = 2 Max = 2t -1 = 5

Insert M : Reached MAX limit = 5 keys / node. So, split the node at the middle element

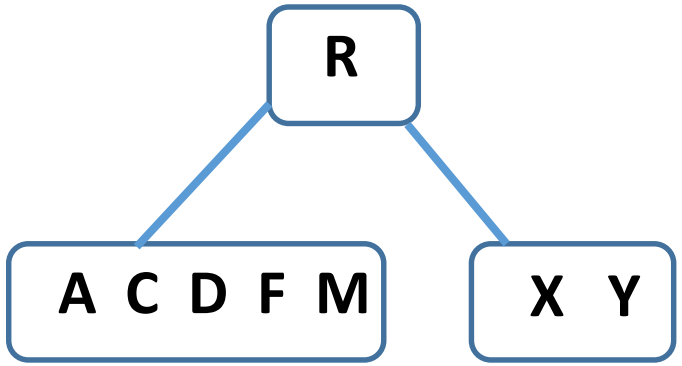


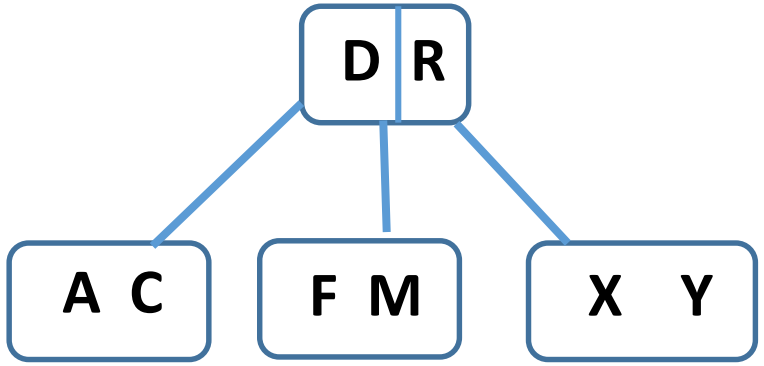


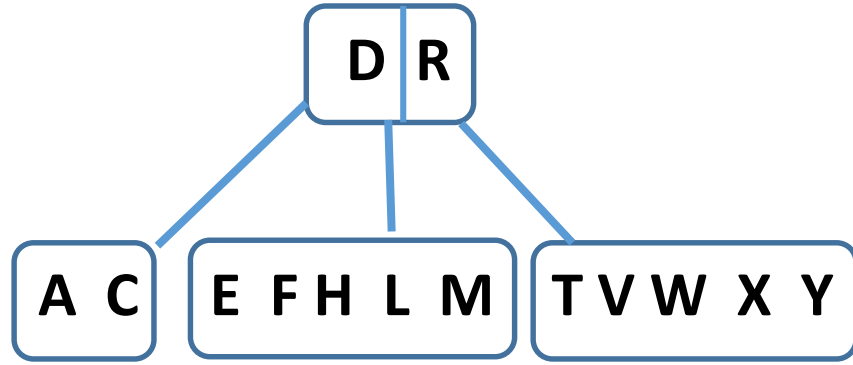
Example 2 : Insert a sequence of characters R, Y, F, X, A, M, C, D, E, T, H, V, L, W, G in given order in an initially empty B-Tree of minimum degree 't' as 3.

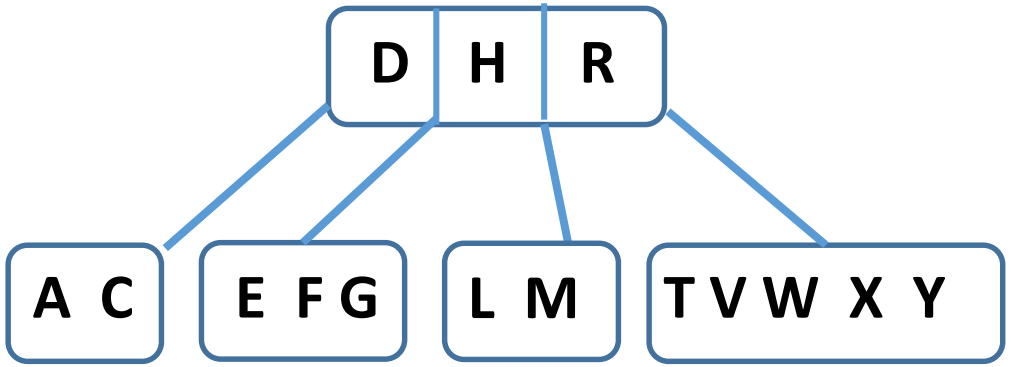
Solution: t = 3 ➔ Min = t -1 = 2 Max = 2t -1 = 5

Insert M : Reached MAX limit = 5 keys / node. So, split the node at the middle element









Deletion in B tree :

Deletion from a B-tree is more complicated than insertion,

because we can delete a key from any node-not just a leaf—and

when we delete a key from an internal node, we will have to rearrange the node's children.

As in insertion, we must make sure the deletion doesn't violate the B-tree properties.

Just as we had to ensure that a node didn't get too big due to insertion,

we must ensure that a node doesn't get too small during deletion

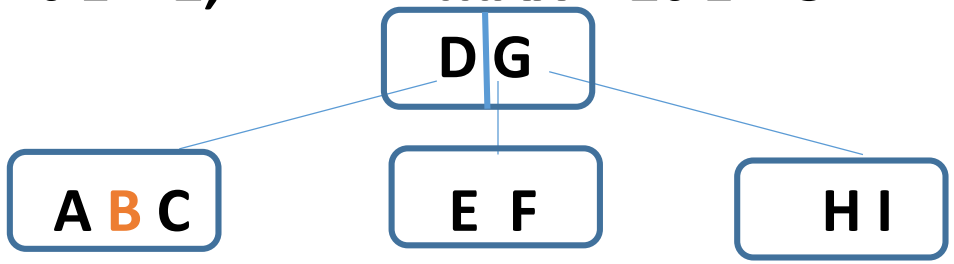
(except that the root is allowed to have fewer than the minimum number t-1 of keys).

Deleting a key from a B Tree :

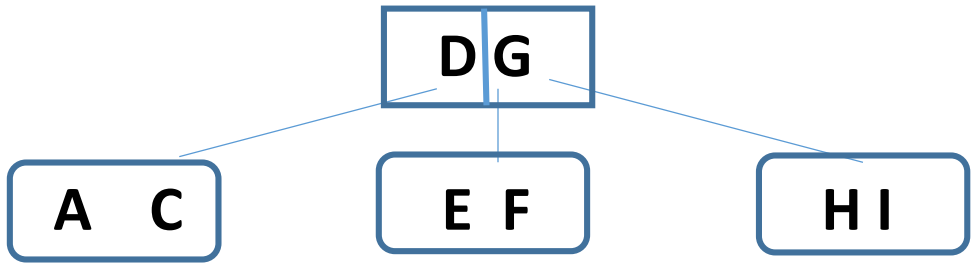
Case 1: Deletion from Leaf node :

If the key k to be deleted is inside a leaf node x and this **leaf node x has > = t keys , (t = degree of B tree)** then just delete the key from node x.

T=3 MIN = t-1 = 2, MAX = 2t-1 = 5



Delete B ➔

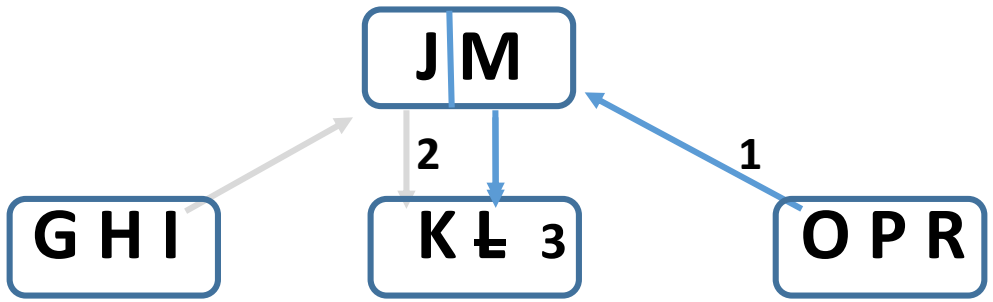


Case 2: Deletion from Leaf node :

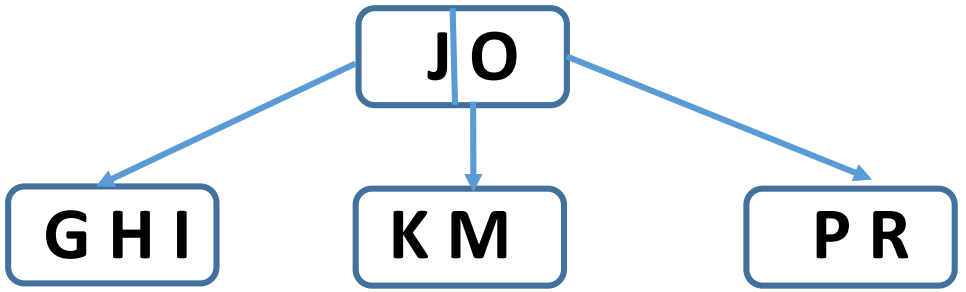
If node x containing the target key to be deleted is a **leaf node** ,
but it **has exactly (t – 1) keys** , i.e. the MINIMUM no. of keys that x should have then

Case 2a : If x has a sibling with at least t keys , then move x's parent key into x
and Move the appropriate extreme key from x's sibling into the open slot in the parent node and then delete the target key.

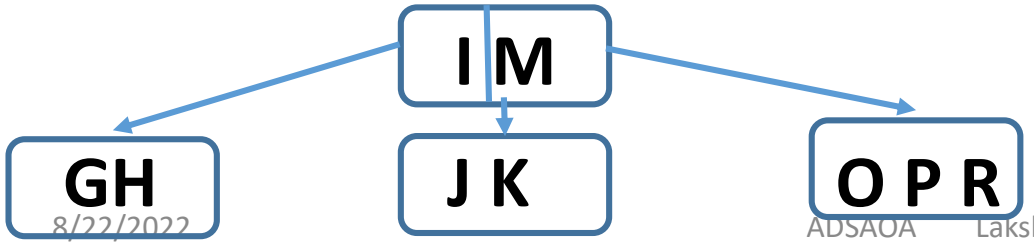
t=3 MIN = t-1 = 2, MAX = 2t-1 = 5



Del L ➔



OR

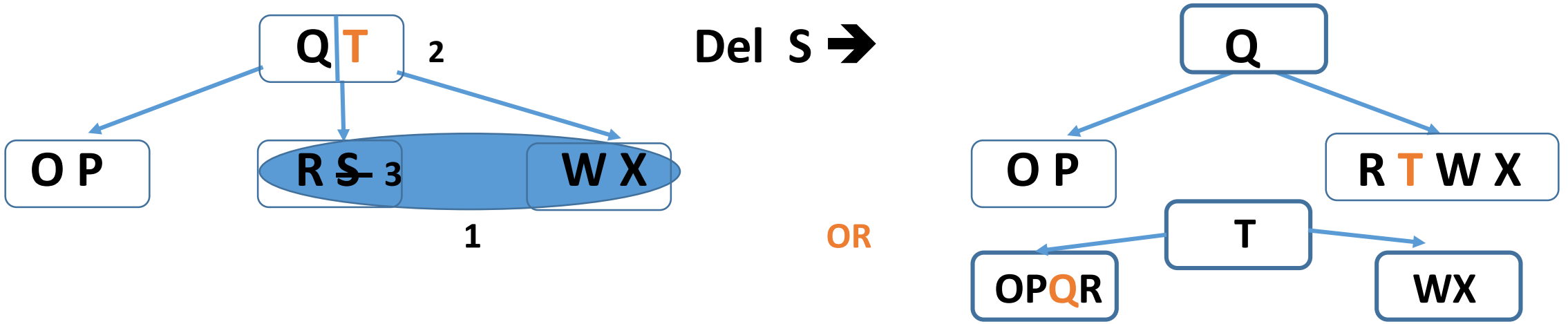


Case 2: Deletion from Leaf node :

If node x containing the target key to be deleted is a **leaf node** ,
but it **has exactly (t – 1) keys** , i.e. the MINIMUM no. of keys that x should have then

Case 2b : If x's sibling also has (t – 1) keys , then MERGE x with one of its siblings by bringing down the parent key as the median key. **Then delete the target key.**

T=3 MIN = t-1 = 2, MAX = 2t-1 = 5



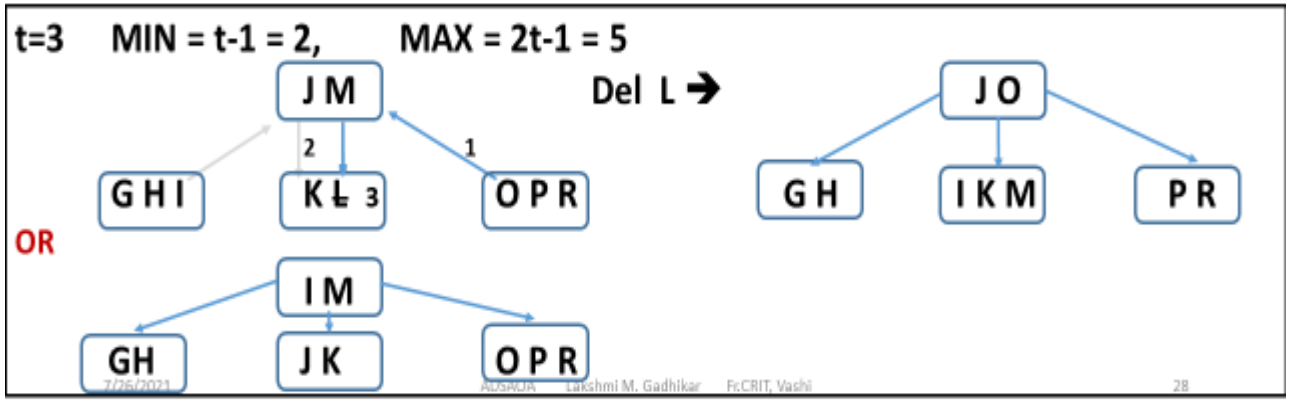
Case 3 : Deletion from internal node :

If the node x containing the target key to be deleted is an internal node, then

a. If the target key's left child has at least t keys , then its largest value can be moved to the parent to replace the target key

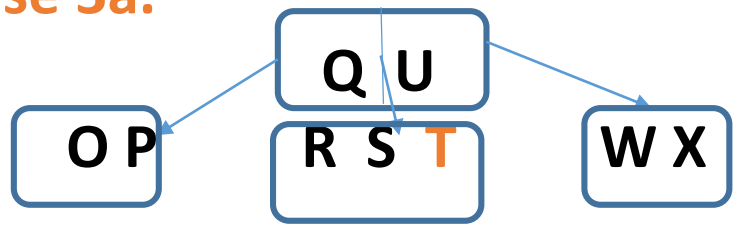
b. If the target key's right child has at least t keys, then its smallest value can be moved to the parent to replace the target key

c. If none of the target key's children have at least t keys, then the children must be MERGED into one and then delete the target key



t = 3 MIN = t-1 = 2, MAX = 2t-1 = 5

Case 3a.

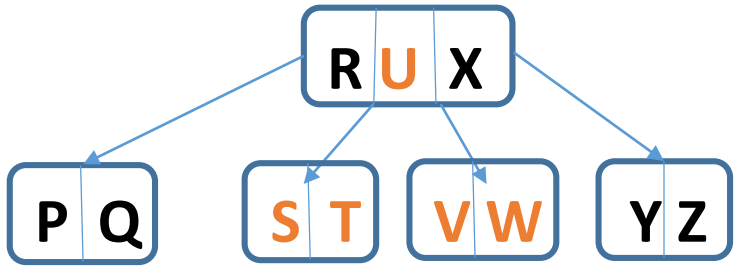


del U ➔

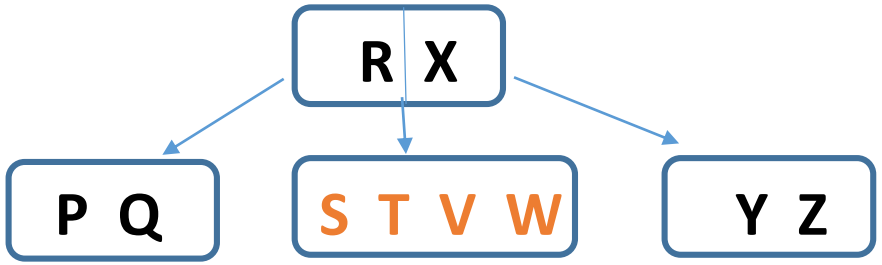
Case 3b.

del I ➔

Case 3c.



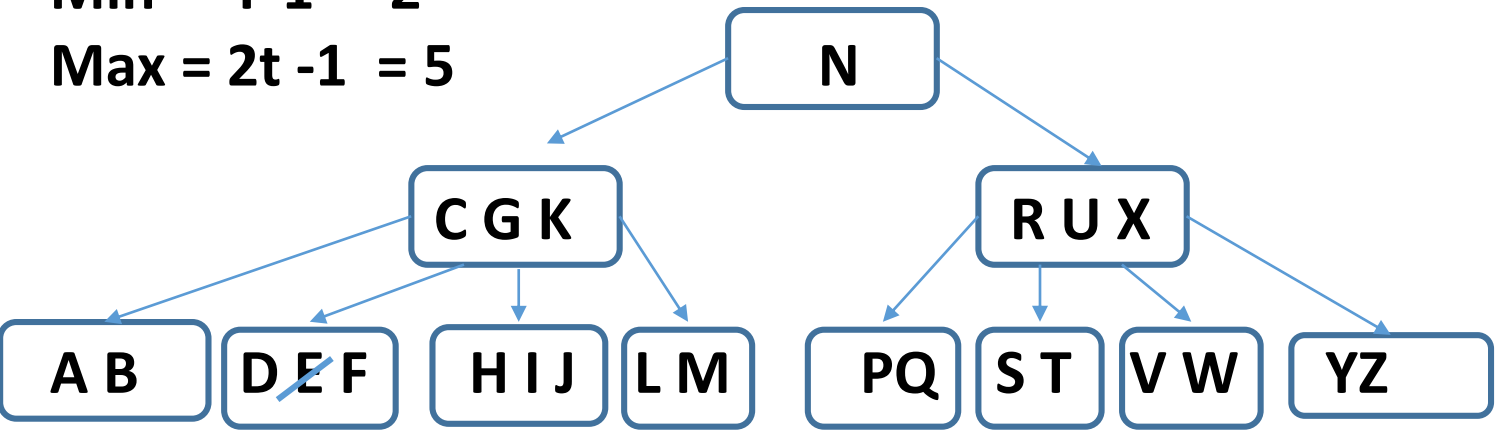
del U ➔



Example1 : Delete the nodes E, F, A , R in order from following B tree of order 3.

Min = T-1 = 2

Max = 2t -1 = 5

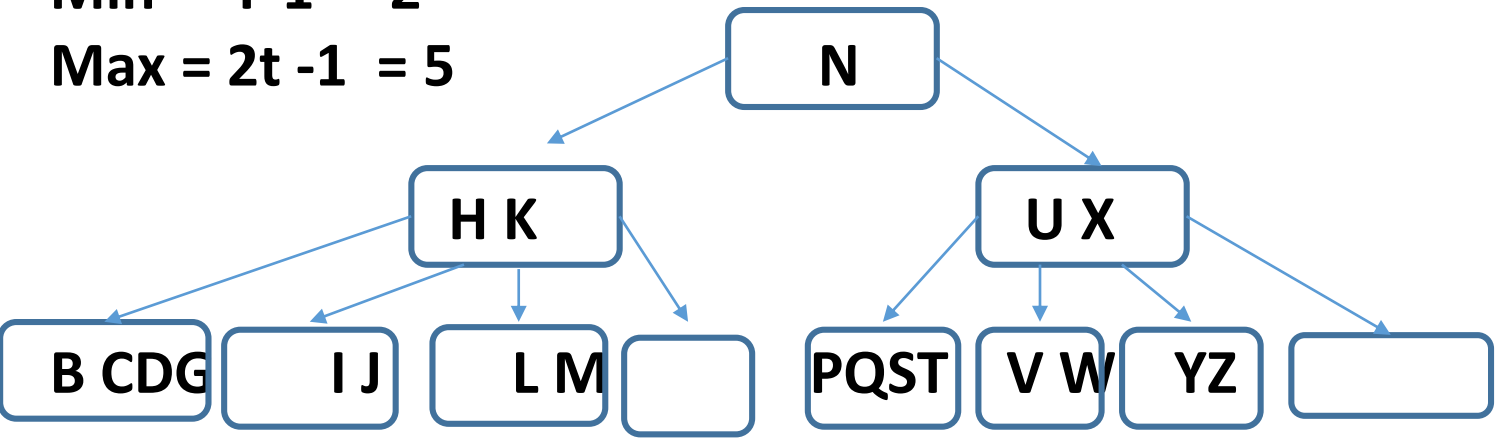


N

Example1 : Delete the nodes E, F, A , R in order from following B tree of order 3.

Min = T-1 = 2

Max = 2t -1 = 5

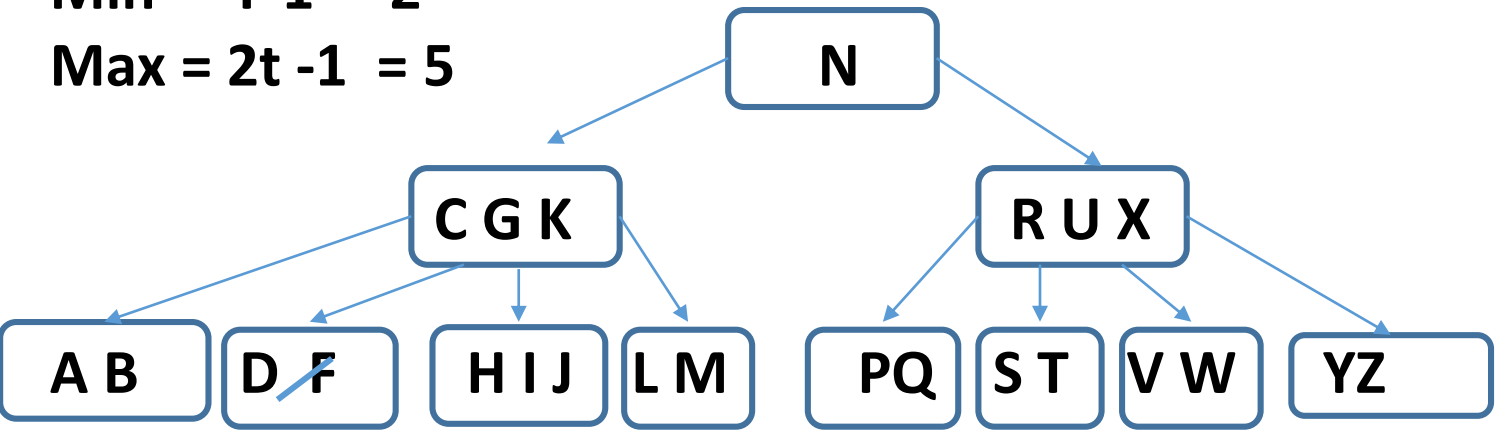


N

Example1 : Delete the nodes E, F, A , R in order from following B tree of order 3.

Min = T-1 = 2

Max = 2t -1 = 5

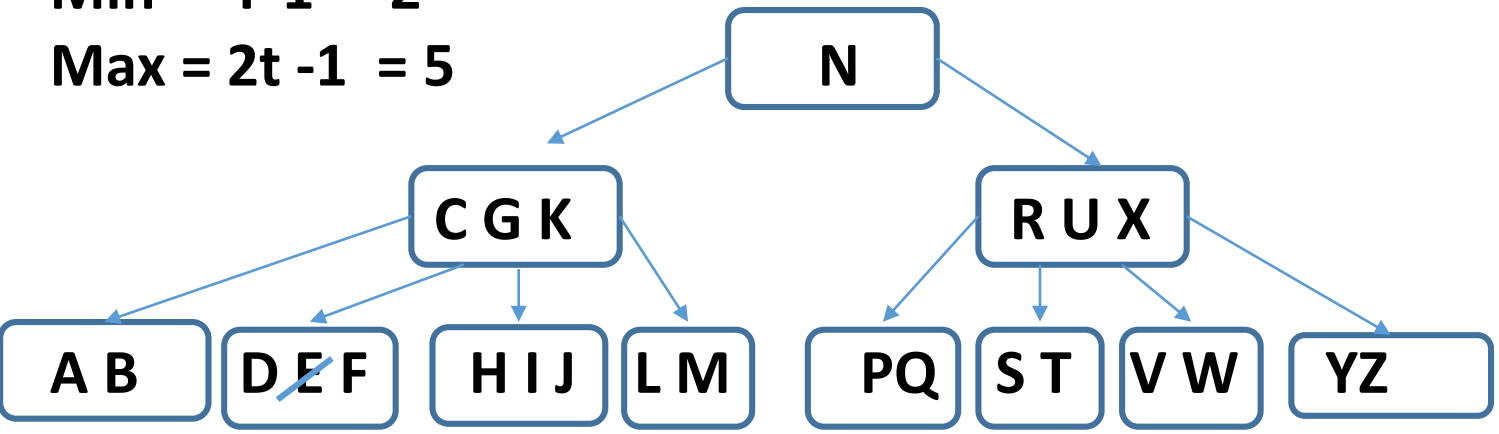


N

Example1 : Delete the nodes E, F, A , R in order from following B tree of order 3.

Min = T-1 = 2

Max = 2t -1 = 5

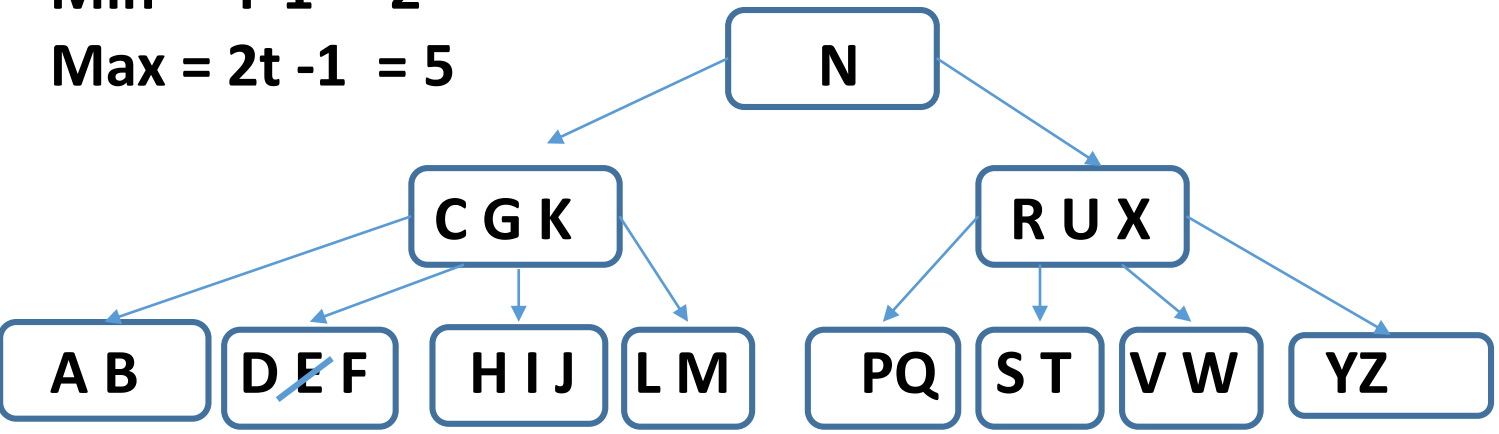


N

Example1 : Delete the nodes E, F, A , R in order from following B tree of order 3.

Min = T-1 = 2

Max = 2t -1 = 5



N

Advanced Data Structures and Analysis of Algorithms
ITCDLO 5011

Last session :

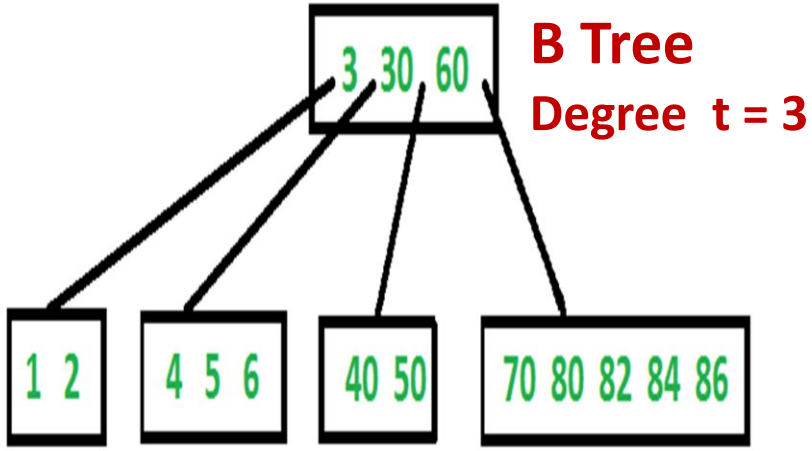
B Tree – Introduction, Properties , Insertion , Deletion

Current session :

B+ Tree :

- **What is a B+ tree**
- **Properties of B+ trees**
- **Examples of B+ trees**
- **Differences between B tree and B+ tree**

B+ Tree :



B Tree
Degree t = 3

B-tree and B+ tree are generally used to implement dynamic multilevel indexing.

Disadvantage B-tree for indexing :

Along with storing the key value in the node of a B-tree, it also stores the data pointer (a pointer to the disk file block containing the key value) :corresponding to a particular key value,.

This significantly reduces the number of keys that can be packed into a node of a B-tree.

This results in an increase in the height (i.e. increase in number of levels) in the B-tree.

Thus, increasing the time to search a key value in B tree.

Above drawback is overcomed in a B+ tree by storing data pointers only at the leaf nodes of the tree.

This results in differences in :

the structure of leaf nodes and

the structure of internal nodes of the B+ tree.

(hence we have two different limits for leaf nodes & internal nodes in a B+ tree.)

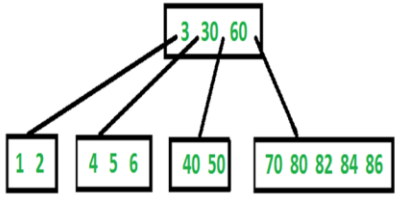


Fig. B tree

The leaf nodes store all the key values along with their corresponding data pointers **to the disk file block, in order to access them.**

All the leaf nodes are linked to provide ordered access to the records.

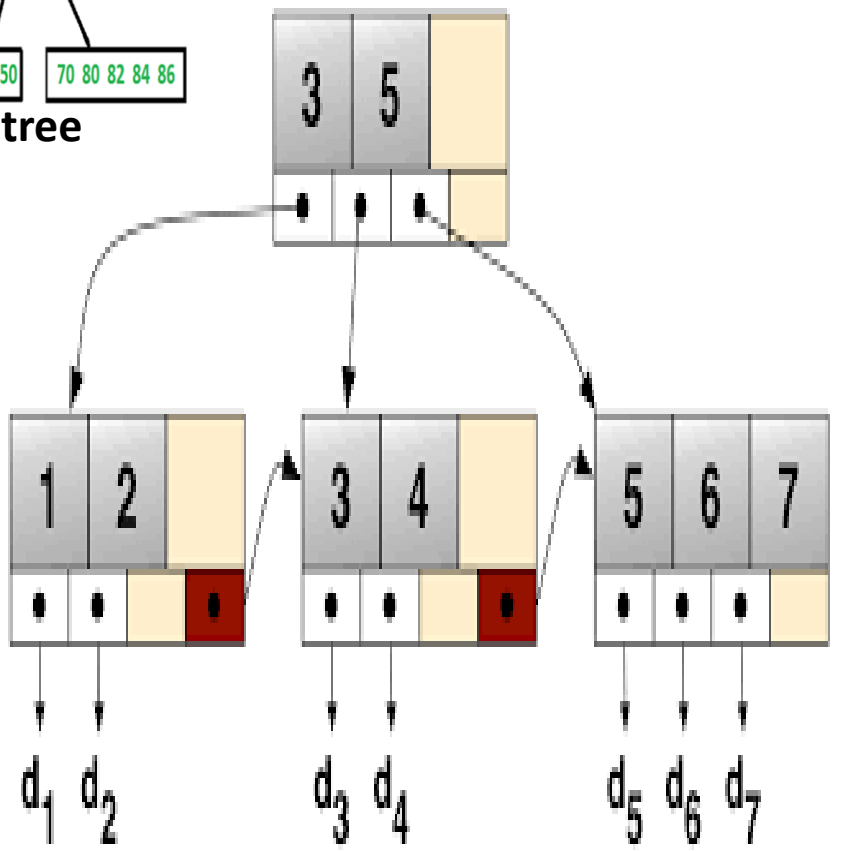


Fig. B+ tree

The leaf nodes, therefore form the first level of index, and the internal nodes form the other levels of a multilevel index.

Some of the key values of the leaf nodes also appear in the internal nodes, to control the searching of a record.

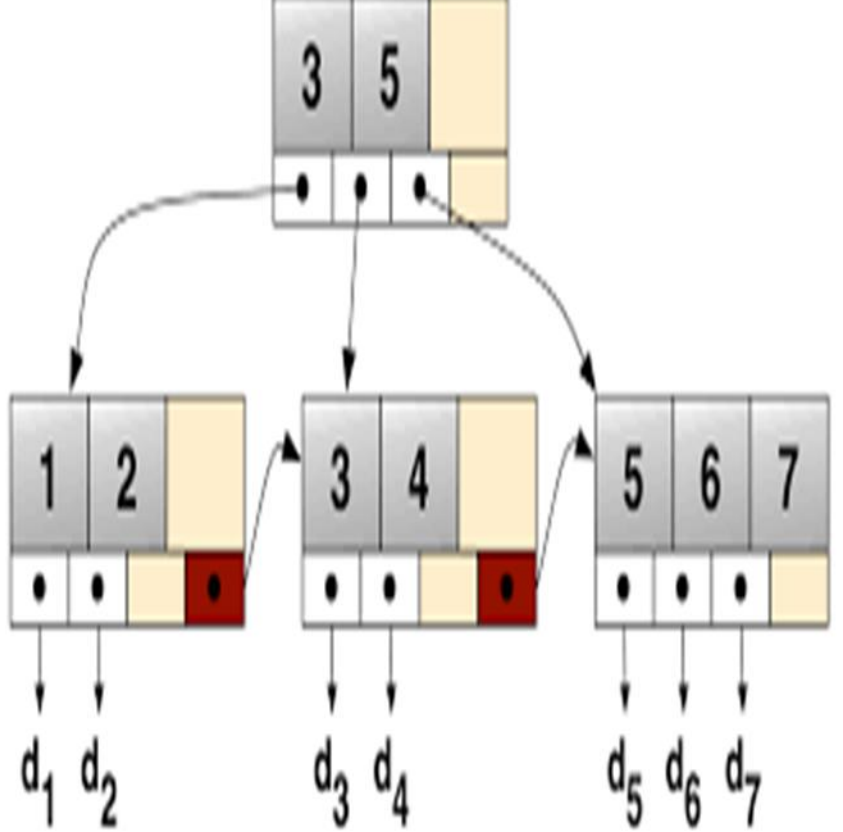
Properties of B+ Trees :

All leaf nodes are at same level.

Data values are present only in the leaves.

Internal nodes (including the root) store only the keys

All leaf nodes are interconnected with each other like a linked list for faster access.



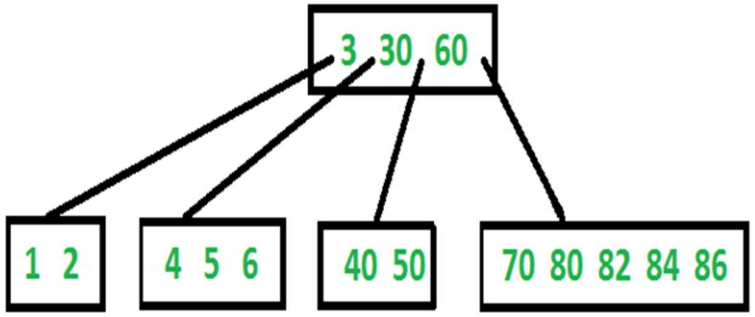
Keys are used for directing the searching of a data item in proper leaf.

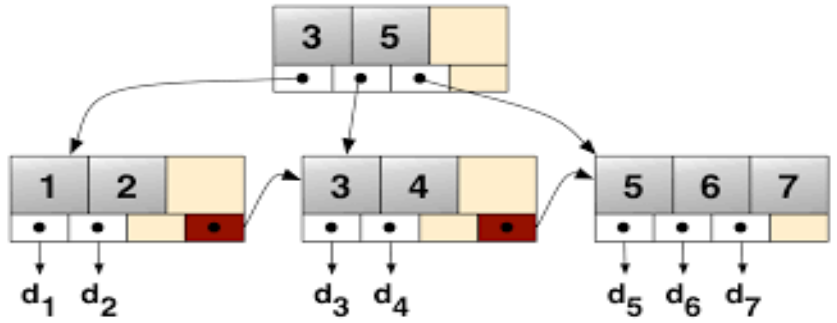
B+ tree combines the features of ISAM (Indexed Sequential Access Method) and B trees.

Summary : Differences between B and B+ tree

The primary application of the B tree and B+ tree is for storage and efficient retrieval of large volumes of data in a block-oriented storage context (Eg. Disks and in particular file-systems).

B-tree	B+ tree
in a B-tree, the keys and data can be stored in both the internal and leaf nodes	In a B+ tree, the data can only be stored in the leaf nodes.
Leaf nodes in the B tree are not linked with each other	Leaf nodes in the B+ tree are linked with each other to provide ordered access to the records.
Each (Internal) node in the B tree contains key–value pairs	Each internal node in the B+ tree contains only keys and not key–value pairs.





Advantage of B+ Trees :

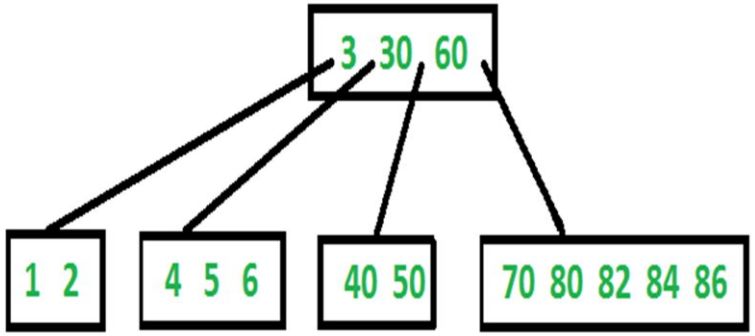
A B+ tree with ‘l’ levels can store more entries in its internal nodes compared to a B-tree having the same ‘l’ levels.

This significantly reduces the search time for any given key**.**

B+ tree are very quick and efficient in accessing records from disks due to lesser levels and presence of P_{next} pointers.

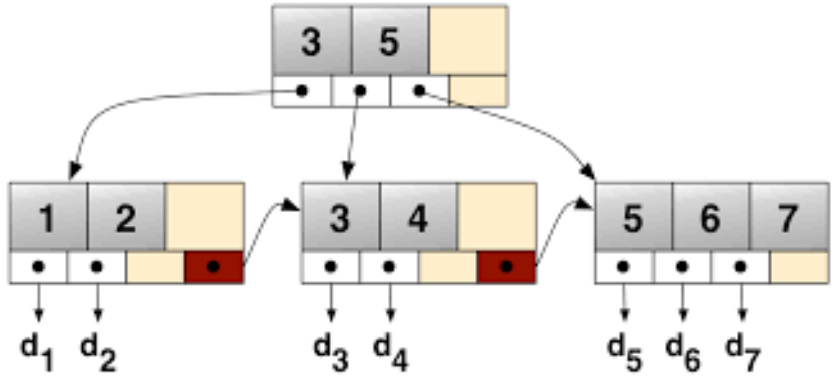
Internal node : contains more pointers :

pointers to disk blocks, 4 child pointers



Internal node : contains 3 pointers :

0 pointers to disk blocks, 3 child pointers



Internal Nodes :

Eg. a = m = 4 = No. of pointers in internal nodes

Root node has at least one key value and minimum two pointers independent of the value of a.

MIN no of keys in internal nodes = [ceil (m/2) -1] = 2 -1 =1

MAX no of keys in internal nodes = m-1 = a-1 = 4 -1 = 3

Leaf Nodes :

Eg. b = L= 5 = No. of data values in leaf nodes

MIN no of keys in leaf nodes = ceil (L/2) = 3

MAX no of keys in leaf nodes = L = b = 5

Insertion in B+ tree :

Insert following elements in order to build a B+ tree where m =3 and L =3

Internal Nodes :

m = 3 ➔ max elements in internal nodes including root = m-1 = 2

min elements in internal nodes = ceil (m/2) -1 = ceil(3/2)-1 = 2-1 =1
Root has min 1 key value and two child pointers.

leaf Nodes :

L = 3 ➔ max elements in leaf = L = 3

min elements in leaf nodes = ceil (L/2) = ceil(3/2) = 2

Example 1 : Insert the following elements in order to build a B+ tree of minimum degree 'm' as 3. 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45.

Solution: m = 3 ➔ Internal Nodes : min = ceil (m/2) -1 = ceil(3/2) -1= 2-1 = 1

max = m-1 = 2

Leaf Nodes : min = ceil (m/2) = ceil(3/2) = 2

max = m = 3

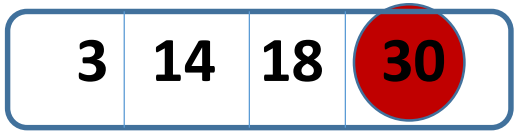
1. Initially root is NULL. Let us first insert 3, 18, 14 .

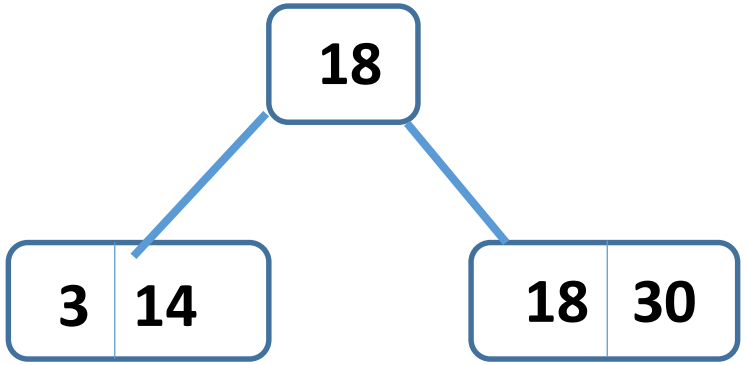
Insert 3

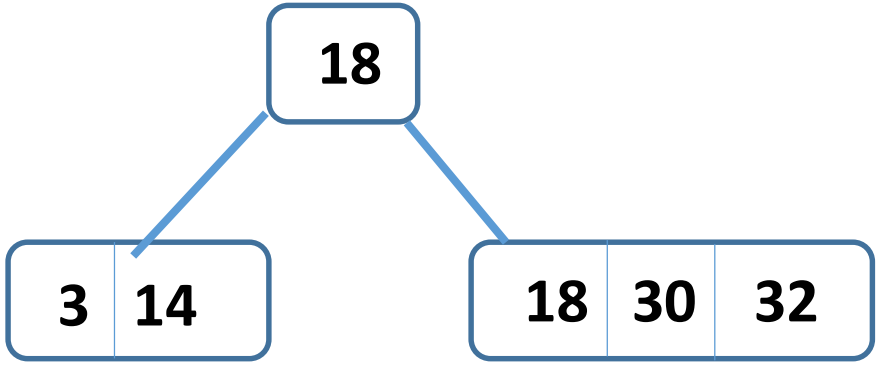
Insert 18

Insert 14

Insert 30







Example 1 : Insert the following elements in order to build a B+ tree of minimum degree 'm' as 3. 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45.

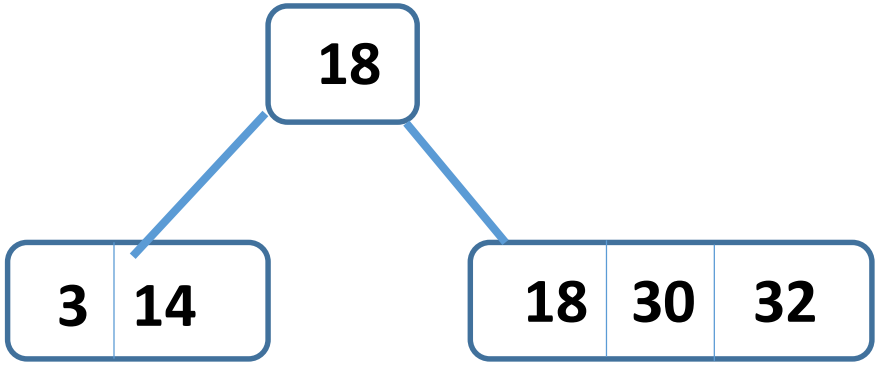
Solution: m = 3 → Internal Nodes : min = ceil (m/2) -1 = ceil(3/2) = 2-1 = 1

max = m-1 = 2

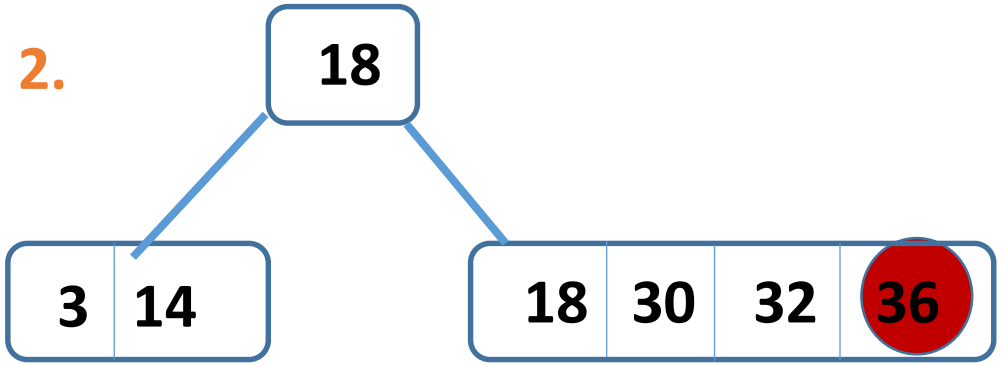
Leaf Nodes : min = ceil (m/2) = ceil(3/2) = 2

max = m = 3

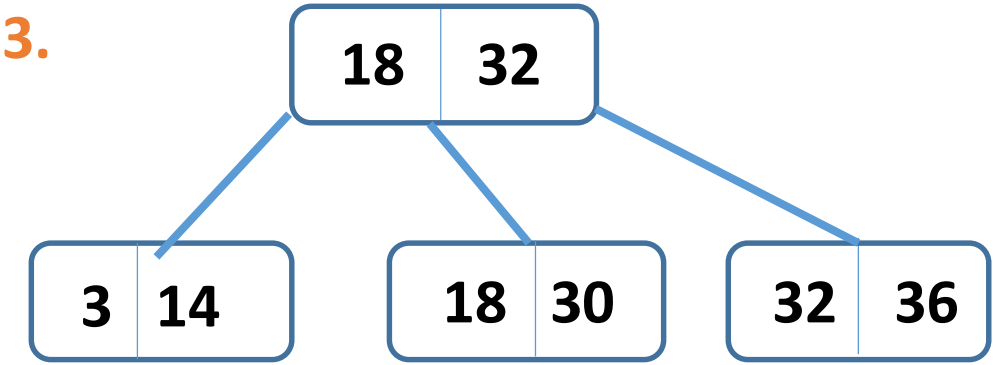
1.



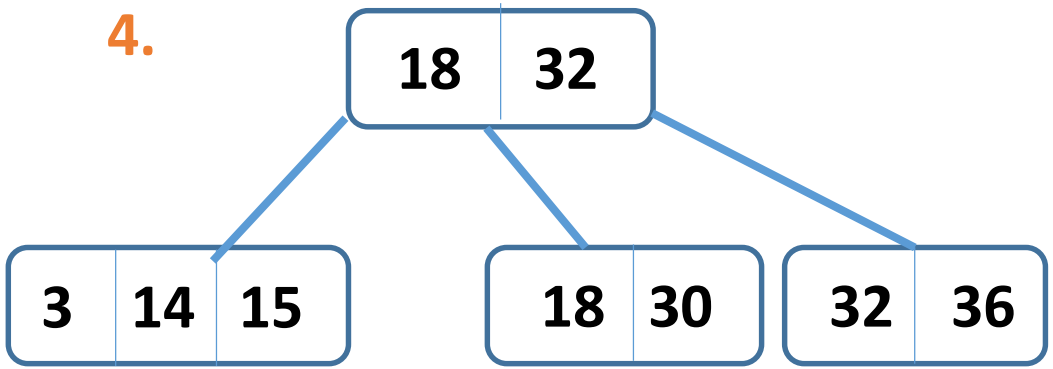
2.



3.



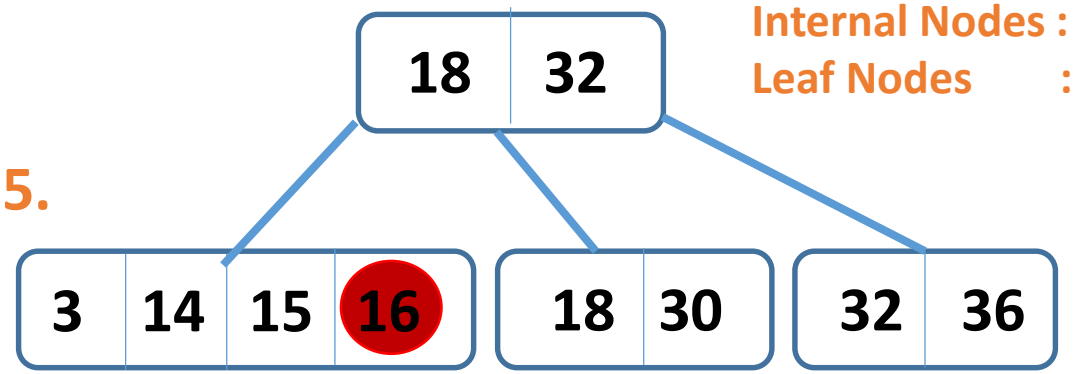
4.



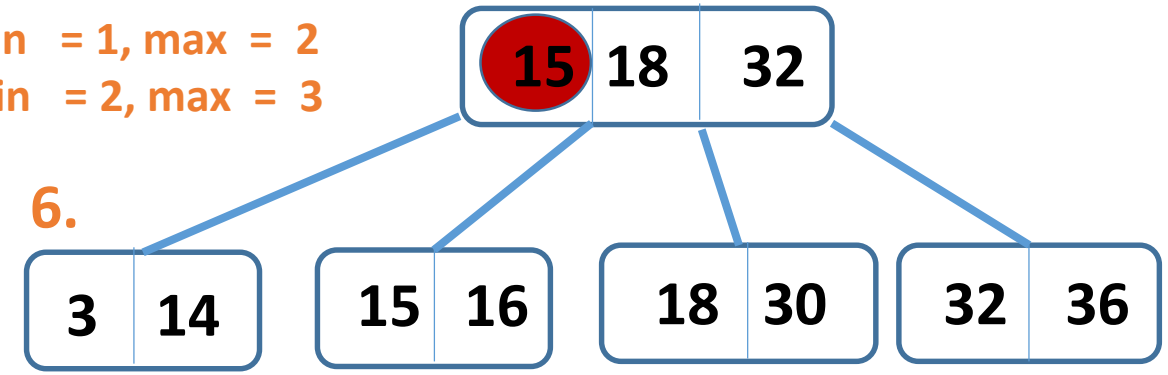
Example 1 : Insert the following elements in order to build a B+ tree of minimum degree 'm' as 3. 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45.

m = 3 ➔
Internal Nodes : min = 1, max = 2
Leaf Nodes : min = 2, max = 3

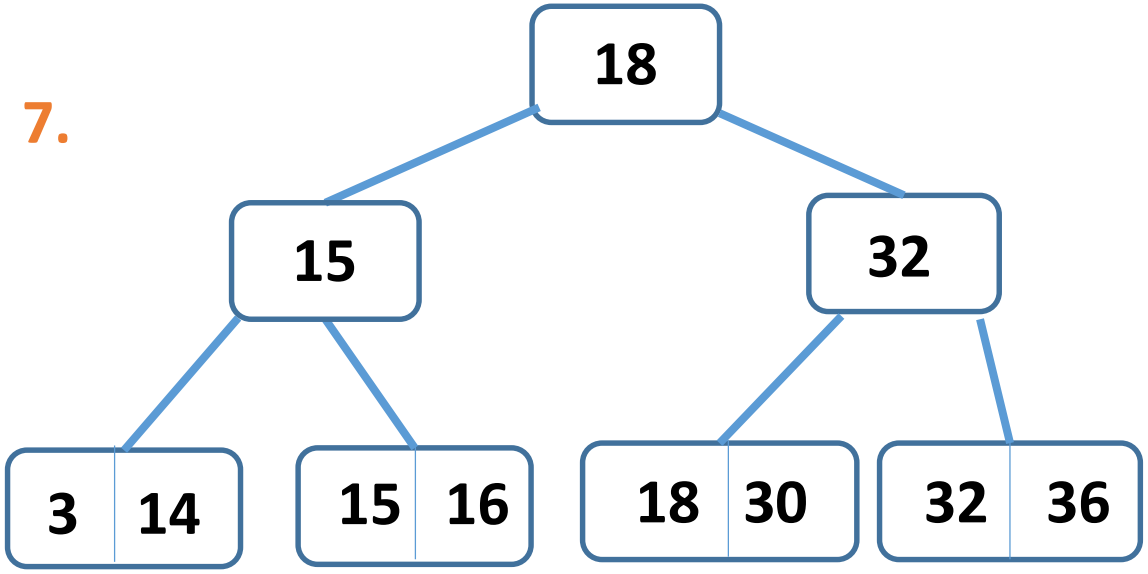
5.



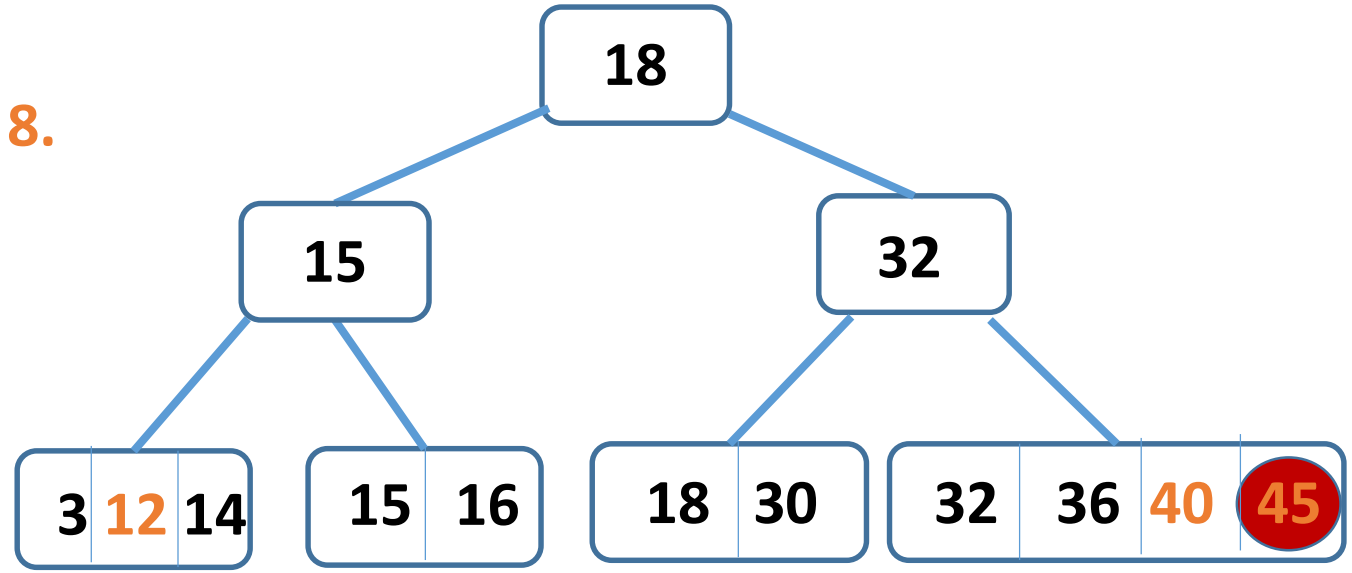
6.



7.



8.



Example 1 : Insert the following elements in order to build a B+ tree of minimum degree 'm' as 3. 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45.

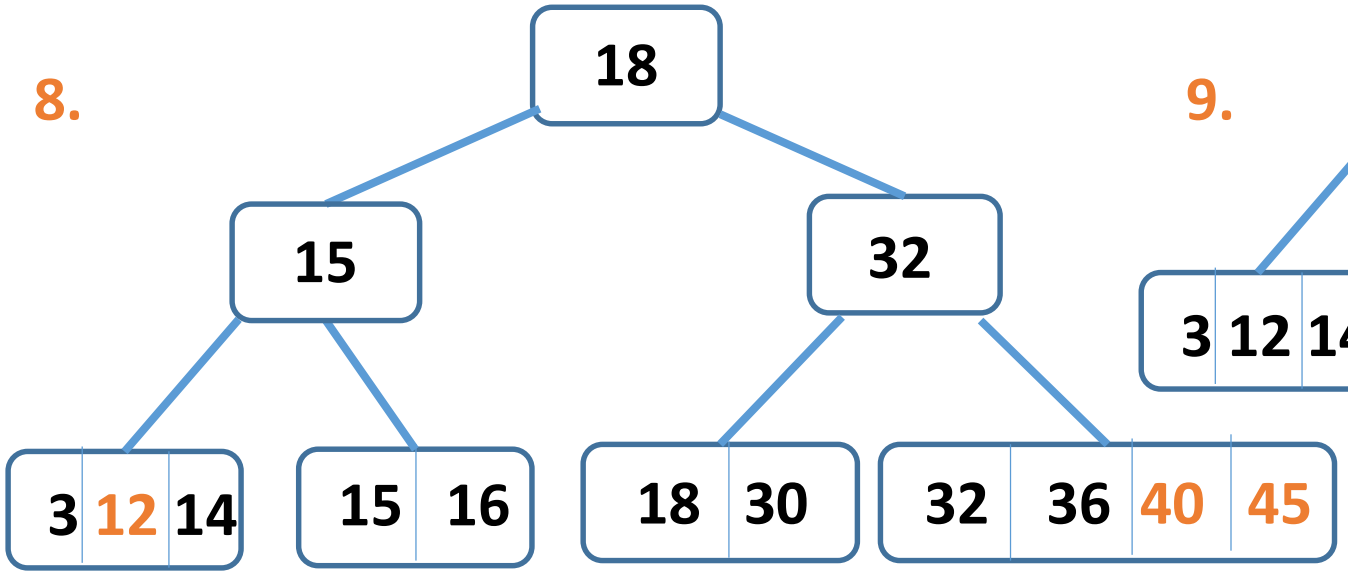
Solution: m = 3 ➔ Internal Nodes : min = ceil (m/2) -1 = ceil(3/2) = 2-1 = 1

max = m-1 = 2

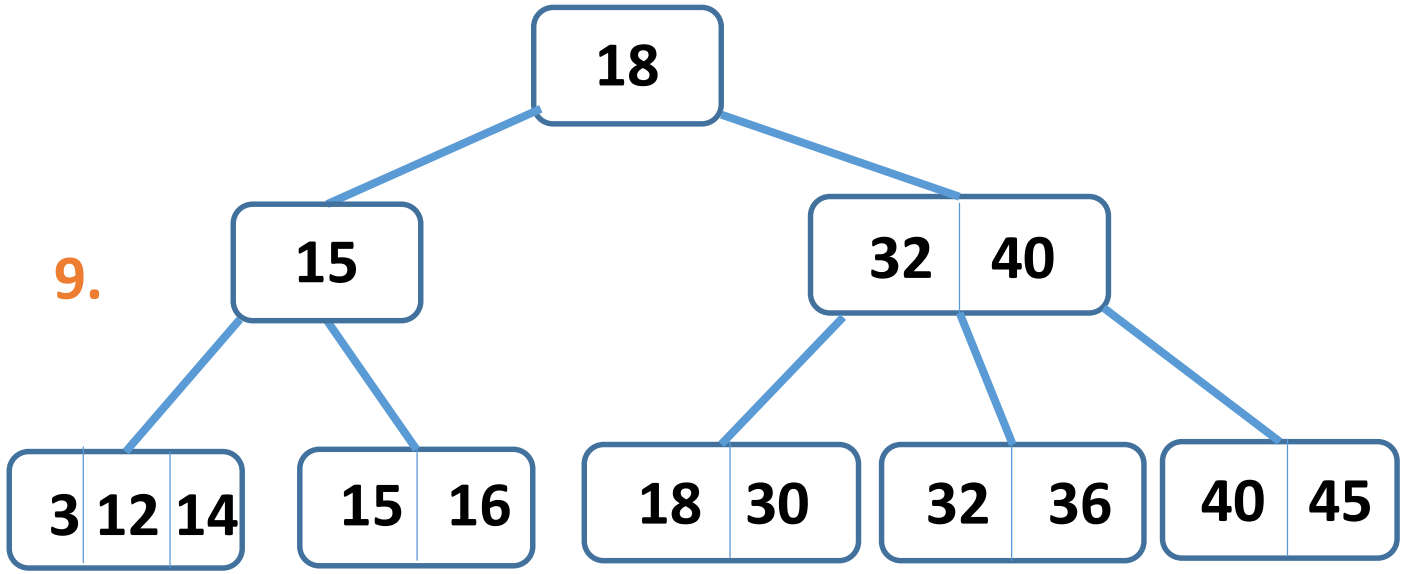
Leaf Nodes : min = ceil (m/2) = ceil(3/2) = 2

max = m = 3

8.



9.



Deletion in B+ tree :

Deletion from Leaf nodes :

1. Remove the key from the leaf
2. If the leaf ends up to have less than ceil (L/2) items, then Underflow :
   - Adopt data from the neighbor (max of left or min of right neighbor)
   - If adopting does not work, then delete the node and merge with the neighbor.
3. If the parent ends up to have less than ceil (m/2) items, then Underflow :

Deletion from internal nodes :

1. Remove the key from the internal node.
2. If the internal node ends up to have less than ceil (m/2) items, then Underflow :
   - Adopt data from the neighbor (max of left or min of right neighbor)
   - If adopting does not work, then delete the node and merge with the neighbor.
3. If the parent ends up to have less than ceil (m/2) items, then Underflow :
4. If the root ends up with only one child , make the child the new root of the tree.
5. Propagate keys up through the tree.

Advantage of B+ Trees :

A B+ tree with ‘l’ levels can store more entries in its internal nodes compared to a B-tree having the same ‘l’ levels.

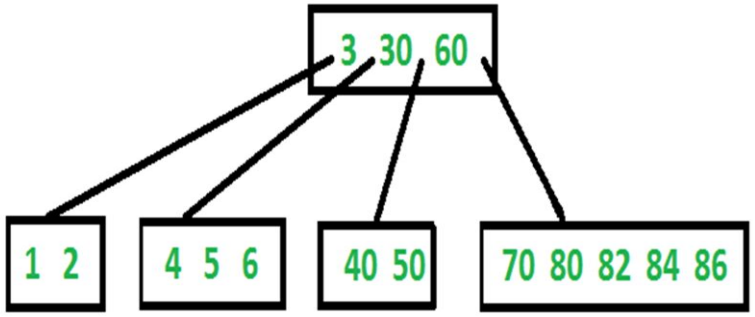
This significantly reduces the search time for any given key.

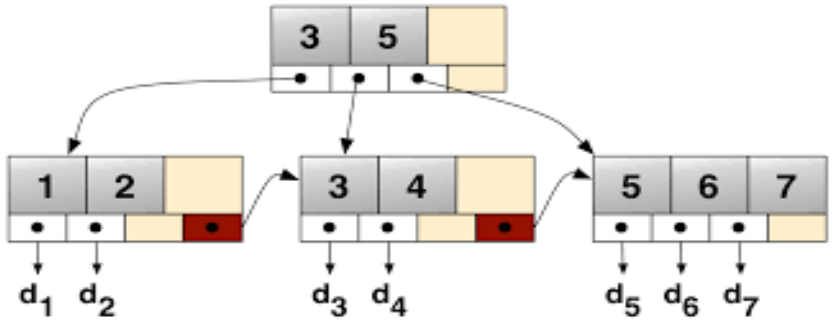
Lesser levels and presence of P_{next} pointers imply that B+ tree are very quick and efficient in accessing records from disks.

Summary : Differences between B and B+ tree

The primary application of the B tree and B+ tree is for storage and efficient retrieval of large volumes of data in a block-oriented storage context (Eg. Disks and in particular file-systems).

B-tree	B+ tree
in a B-tree, the keys and data can be stored in both the internal and leaf nodes	In a B+ tree, the data can only be stored in the leaf nodes.
Leaf nodes in the B tree are not linked with each other	Leaf nodes in the B+ tree are linked with each other to provide ordered access to the records.
Each (Internal) node in the B tree contains key–value pairs	Each internal node in the B+ tree contains only keys and not key–value pairs.





2–3 Tree : (not in syllabus)

a 2–3 tree is a tree data structure, where every internal node (with children) has either

two children (2-node) and one key or

three children (3-nodes) and two keys. (=> 2-3 tree is not a binary tree)

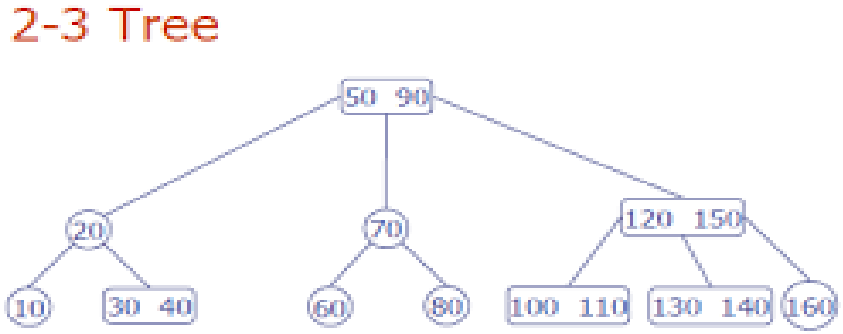
All leaf nodes are at the same level. Each path from the root to a leaf has the same length.

2-3 trees are also a balanced tree like AVL trees, red-black trees, and B trees.

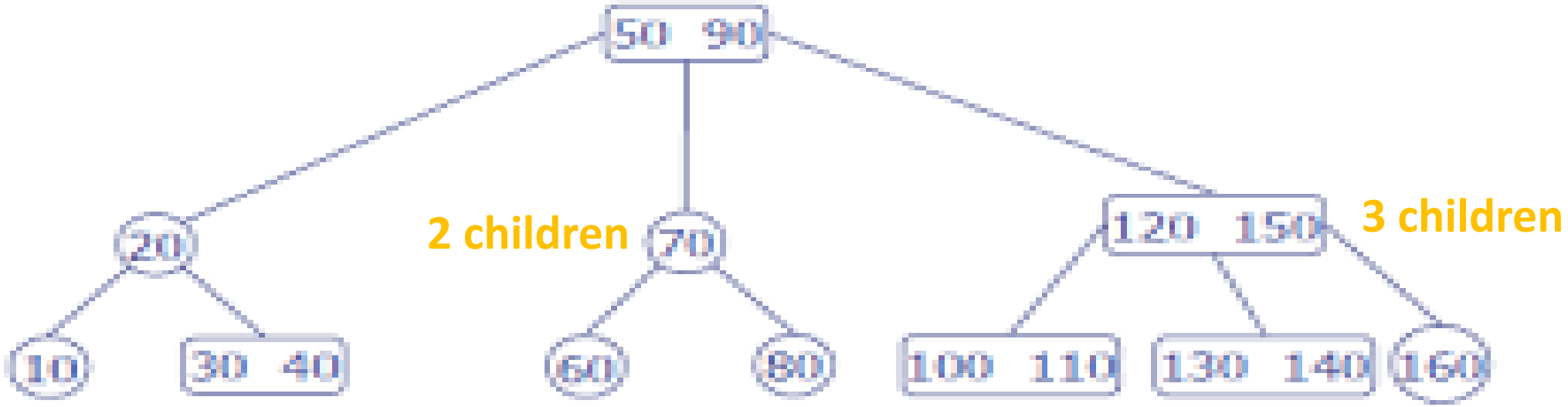
As for binary search trees, the same values can usually be represented by more than one tree.

According to Knuth, "a B-tree of order 3 is a 2-3 tree."

2–3 trees were invented by John Hopcroft in 1970.



2-3 Tree


50 90
20
2 children
70
120 150
3 children
10
30 40
60
80
100 110
130 140
160

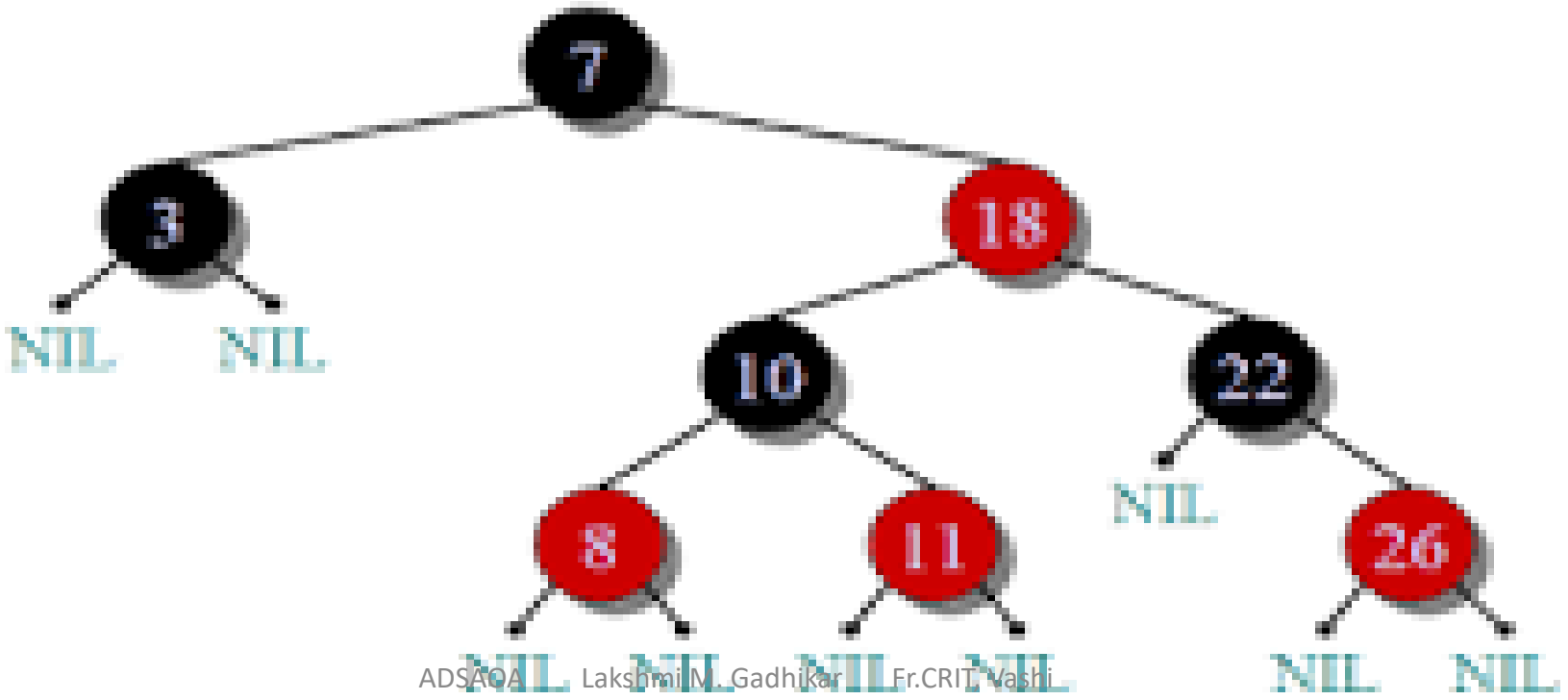
A red–black tree :

Q. Explain red-black trees (10M FH - 19)

is a kind of self-balancing binary search tree.

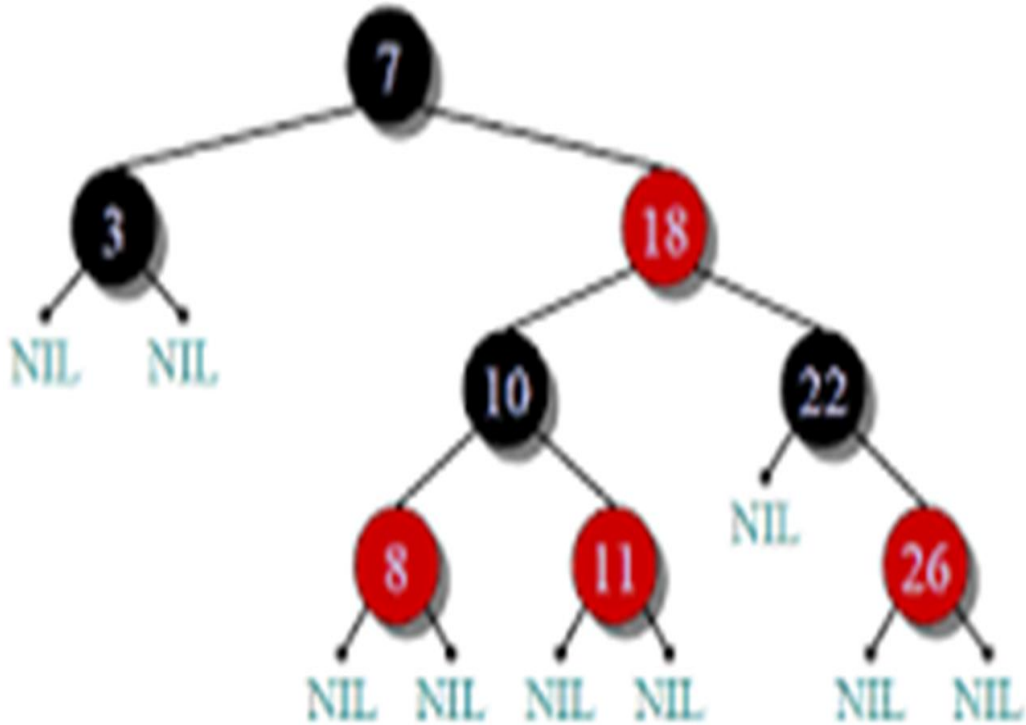
Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node.

These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.



Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.

RedBlackTree



1) Every node has a color either red or black.

2) Root of tree is always black.

3) All leaf nodes are always black.

4) No data is stored in leaf nodes.

5) There are no two adjacent red nodes
(A red node cannot have a red parent or red child).

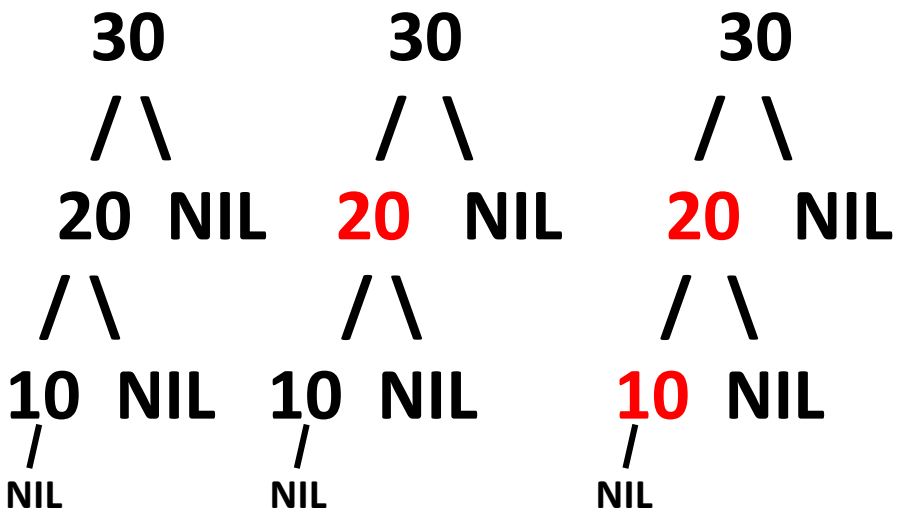
6) Every path from root to a NULL node has same number of black nodes (=3 in tree shown).

How does a Red-Black Tree ensure balance?

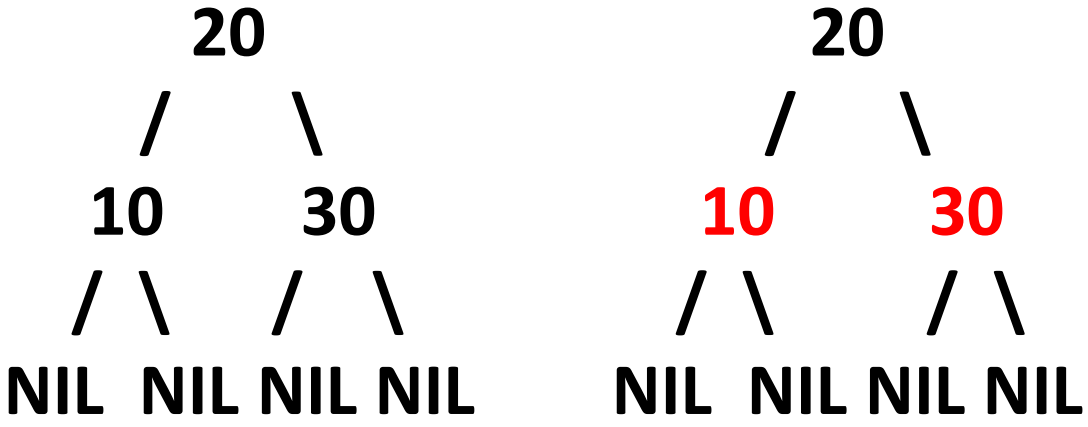
If a red black tree contains only 3 nodes then , a chain of 3 nodes is not possible in the Red-Black tree.

We can try any combination of colors and see all of them violate Red-Black tree property.

Following are NOT Red-Black Trees



Following are different possible Red-Black Trees with above 3 keys



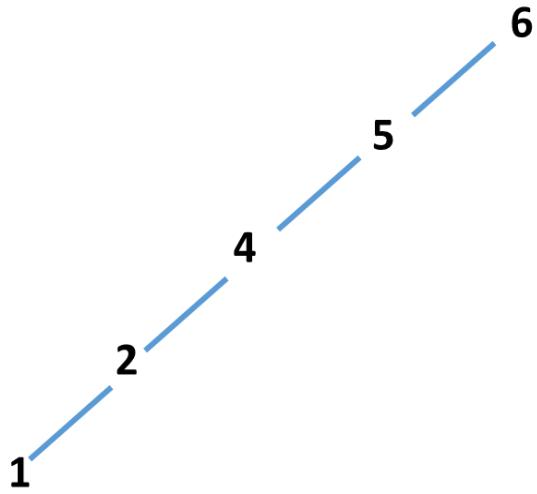
Why Red-Black Trees over binary trees ?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc)

take O(h) = O(Log n) time where,

n is the number of nodes in the tree.

and h is the height of the BST.



Left skewed tree

The cost of these operations may become O(n) for a skewed Binary tree.

If we make sure that height of the tree remains O(Log n) after every insertion and deletion, then we can guarantee an upper bound of O(Log n) for all these operations.

The height of a Red-Black tree is always O(Log n) where n is the number of nodes in the tree.

Q1. When to choose Red-Black tree, AVL tree and B-trees?

Ans : many inserts, many searches and when managing more items respectively

Q2. When it would be optimal to prefer Red-black trees over AVL trees?

Ans : when there are more insertions or deletions

Q3. How is the color information stored in nodes of Red black tree?

Ans : using least significant bit of one of the pointers in the node for color information

Heap Data Structure :

Heap is a binary tree with special characteristics.

In a heap data structure, nodes are arranged based on their value.

A heap data structure is some times called as Binary Heap.

There are two types of heap data structures :

- **Max Heap**
- **Min Heap**

Every heap data structure has the following properties:

1 . **Ordering : Nodes must be arranged in a order according to values based on Max heap or Min heap.**
2. **Structural : All levels in a heap must be full, except last level and nodes must be filled from left to right strictly.**

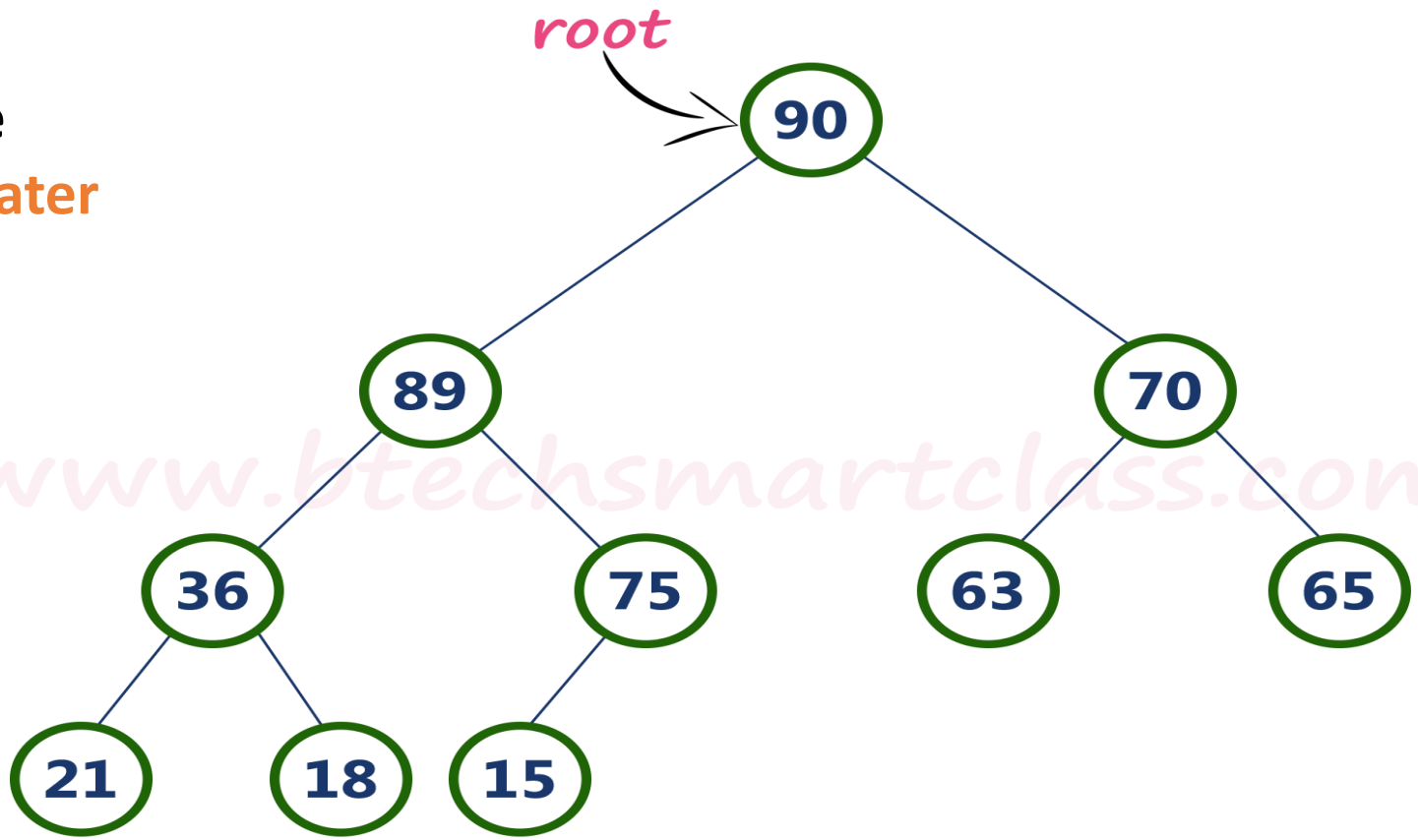
Max heap :

data structure is a specialized complete or a full binary tree data structure except last leaf node can be alone. In a max heap nodes are arranged based on node value.

Max heap is defined as follows :

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes.
Last leaf node can be alone.

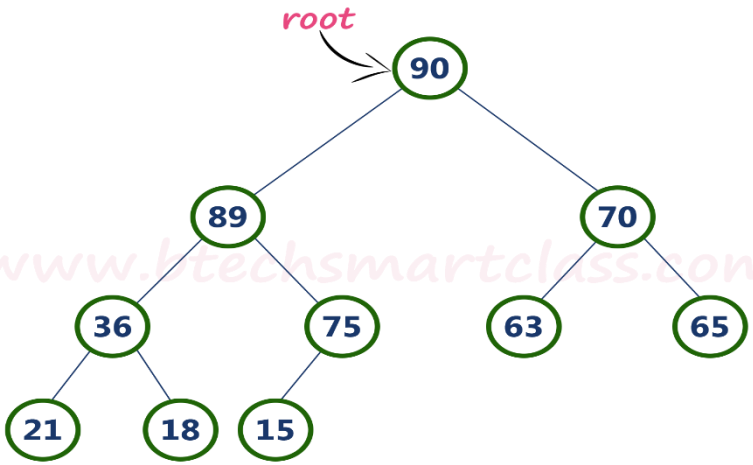
Example



Operations on Max Heap :

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**



Finding Maximum Value Operation in Max Heap :

Finding the node which has maximum value in a max heap is very simple.

In max heap, the root node has the maximum value than all other nodes in the max heap.

So, directly we can display root node value as maximum value in max heap.

Insertion Operation in Max Heap :

Insertion Operation in max heap is performed as follows...

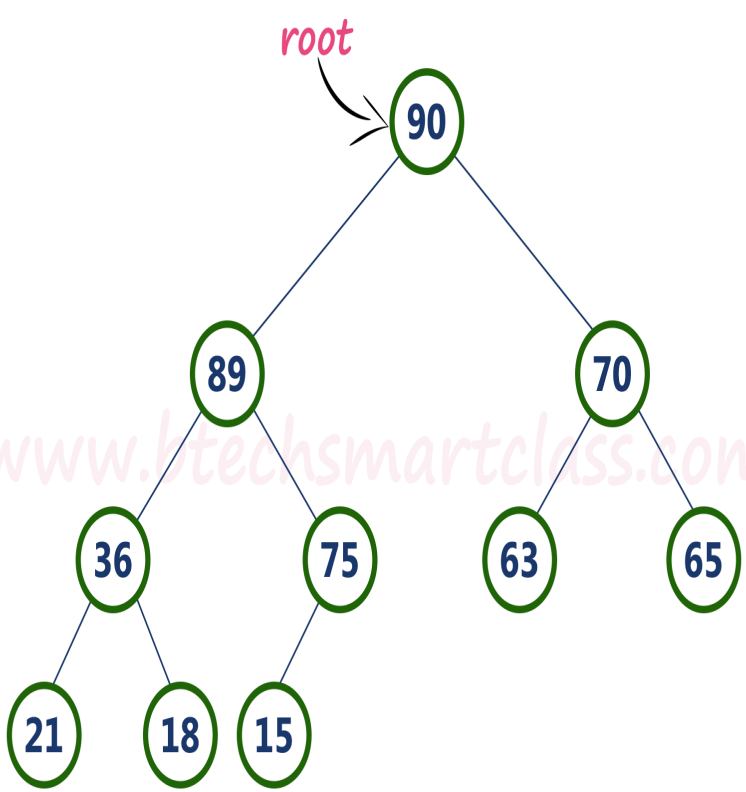
Step 1: Insert the newNode as last leaf from left to right.

Step 2: Compare newNode value with its Parent node.

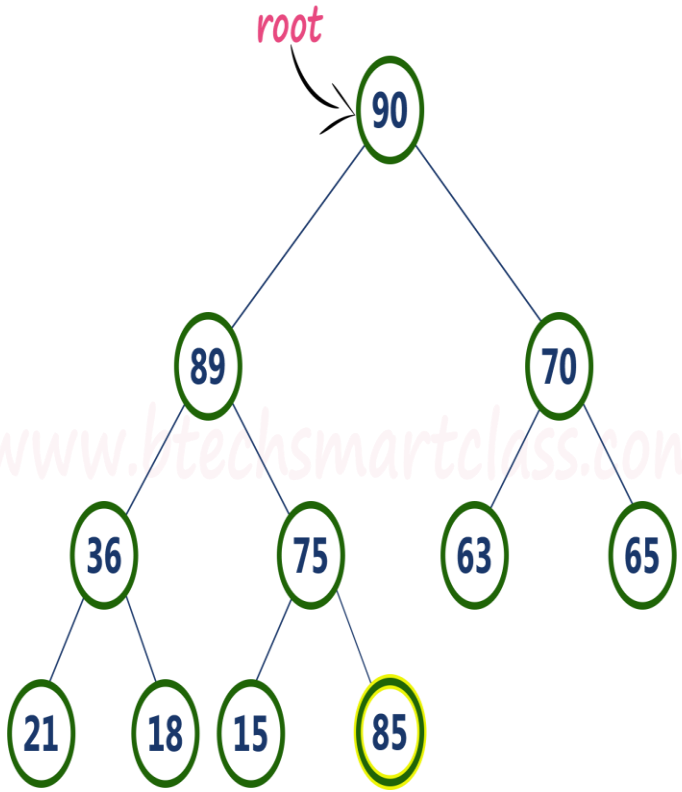
Step 3: If newNode value is greater than its parent, then swap both of them.

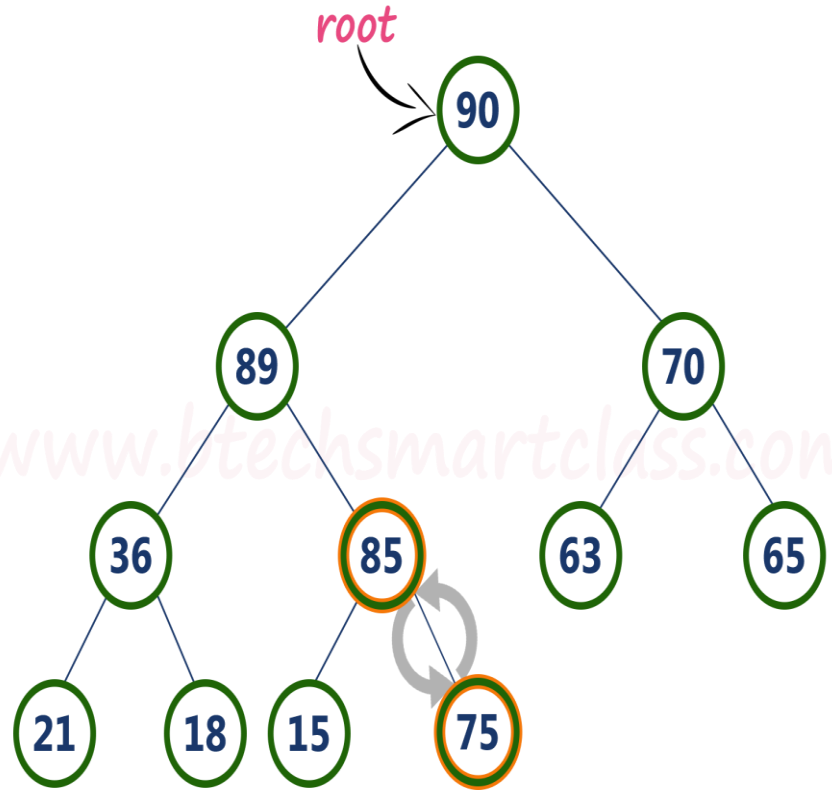
Step 4: Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reached to root.

Example
Consider the above max heap. Insert a new node with value 85.



(a)





(b)

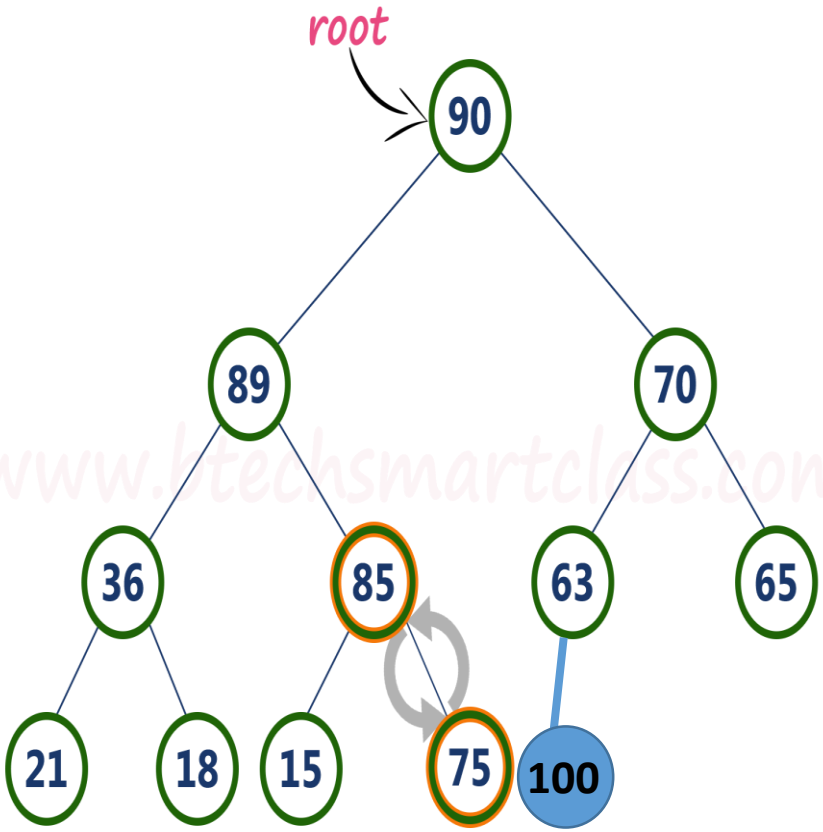
Step 2: Compare newNode value (85) with its Parent node value (75). That means 85 > 75 (a)

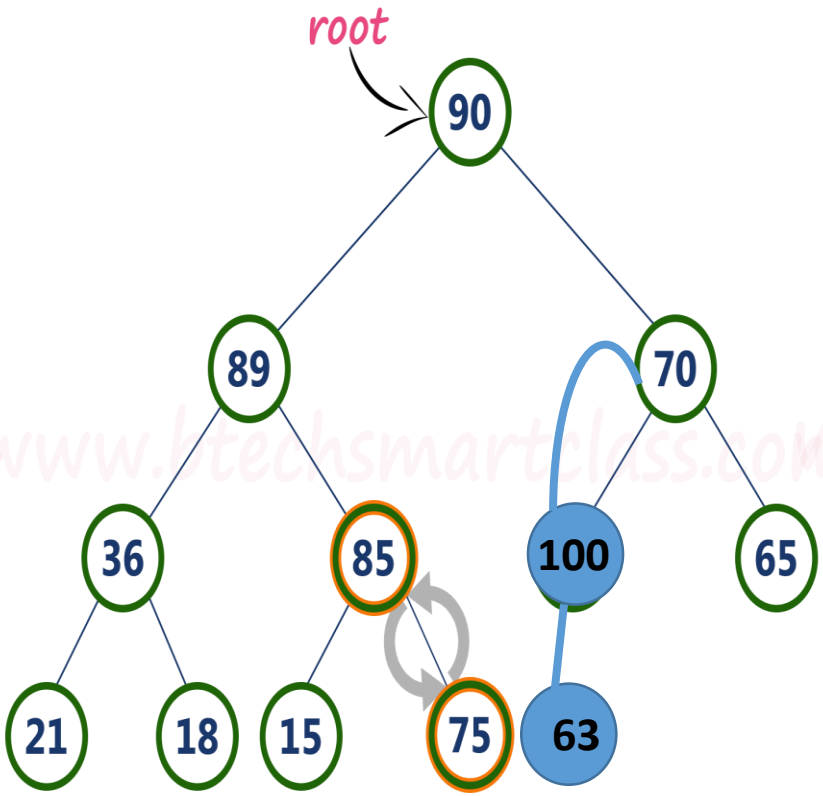
Step 3: Here newNode value (85) is greater than its parent value (75), then swap both of them. After wsapping, max heap is : (b)

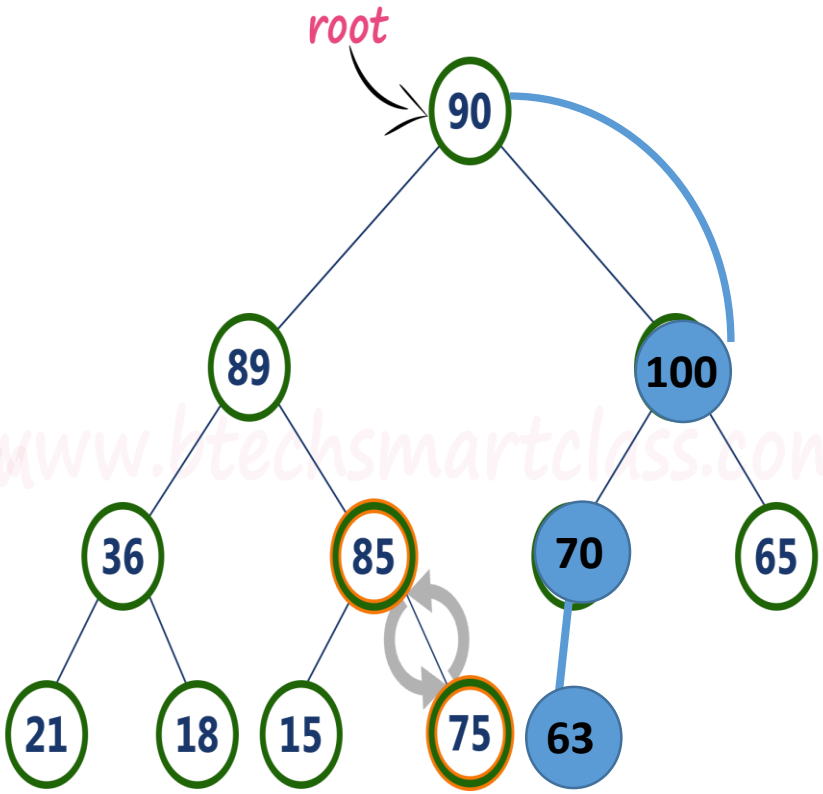
Step 4: Now, again compare newNode value (85) with its parent nede value (89).

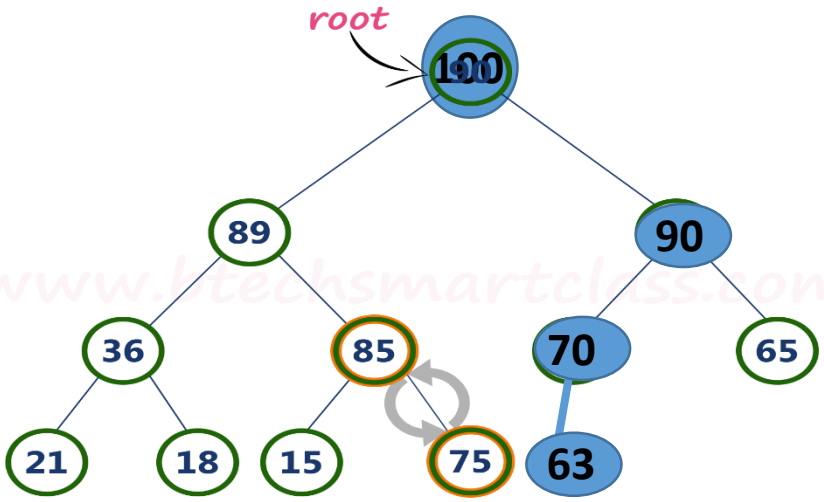
Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process.

Ex2. Insert 100.









Deletion Operation in Max Heap :

In a max heap, deleting last node is very simple as it is not disturbing max heap properties.

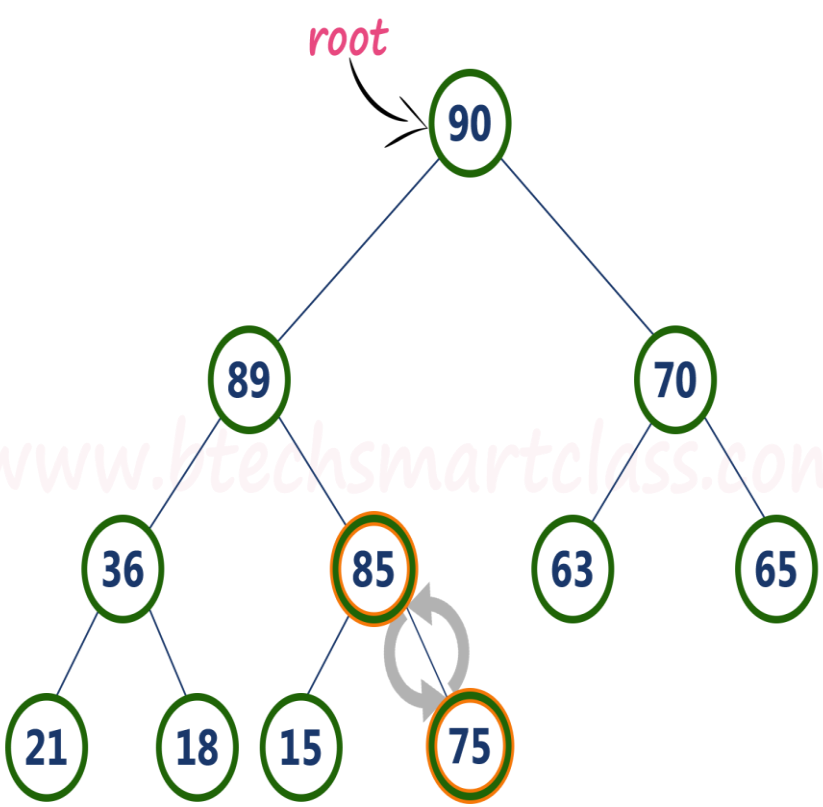
Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete root node from a max heap...

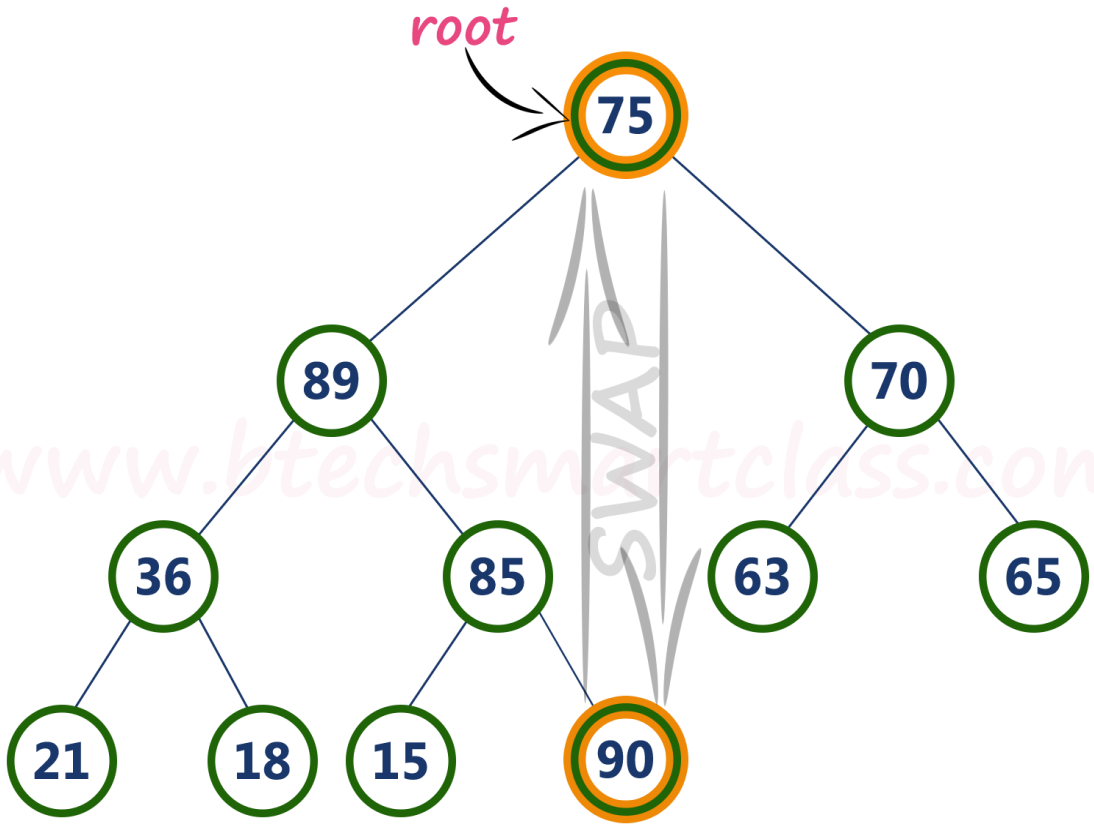
Step 1: Swap the root node with last node in max heap
Step 2: Delete last node.
Step 3: Now, compare root value with its left child value.
Step 4: If root value is smaller than its left child, then compare left child with its right sibling. Else goto Step 6
Step 5: If left child value is larger than its right sibling, then swap root with left child. otherwise swap root with its right child.
Step 6: If root value is larger than its left child, then compare root value with its right child value.
Step 7: If root value is smaller than its right child, then swap root with right child. otherwise stop the process.
Step 8: Repeat the same until root node is fixed at its exact position.

Example

Consider the above max heap. Delete root node (90) from the max heap.

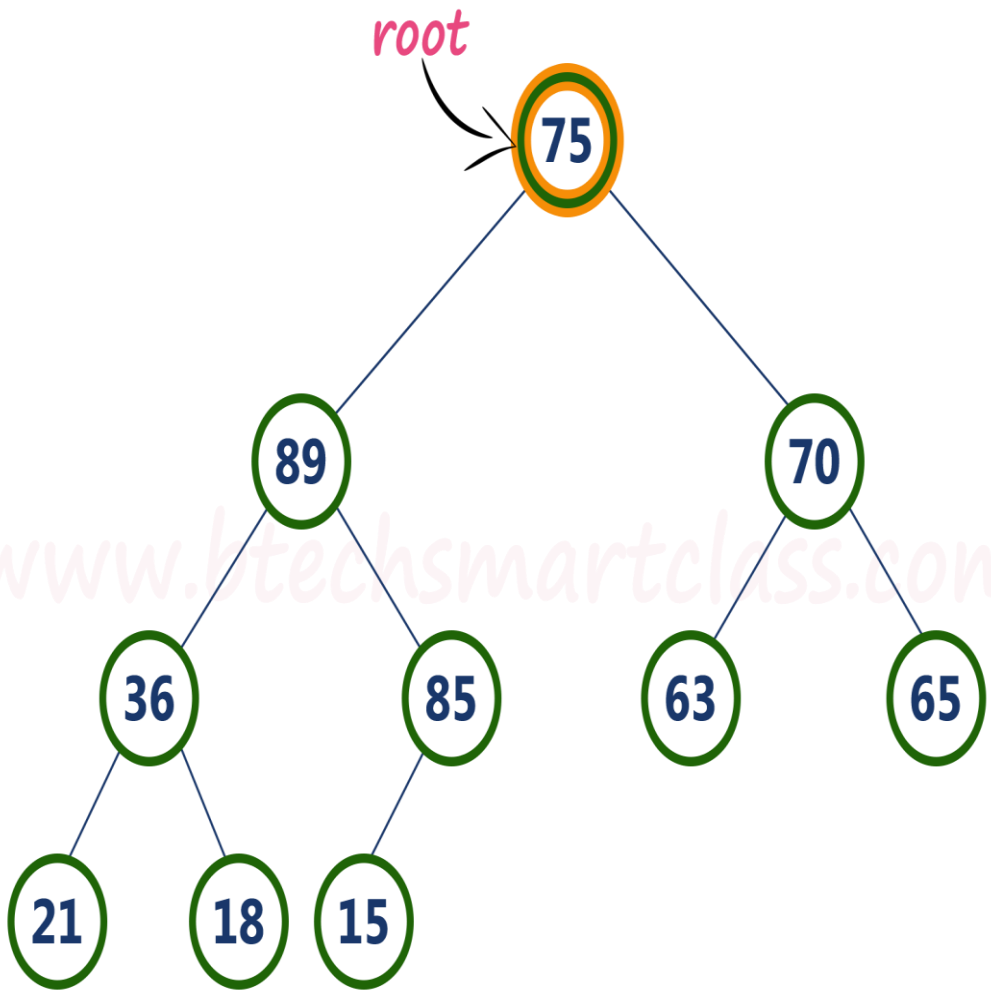
Step 1: Swap the root node (90) with last node 75 in max heap After swapping max heap is as follows...

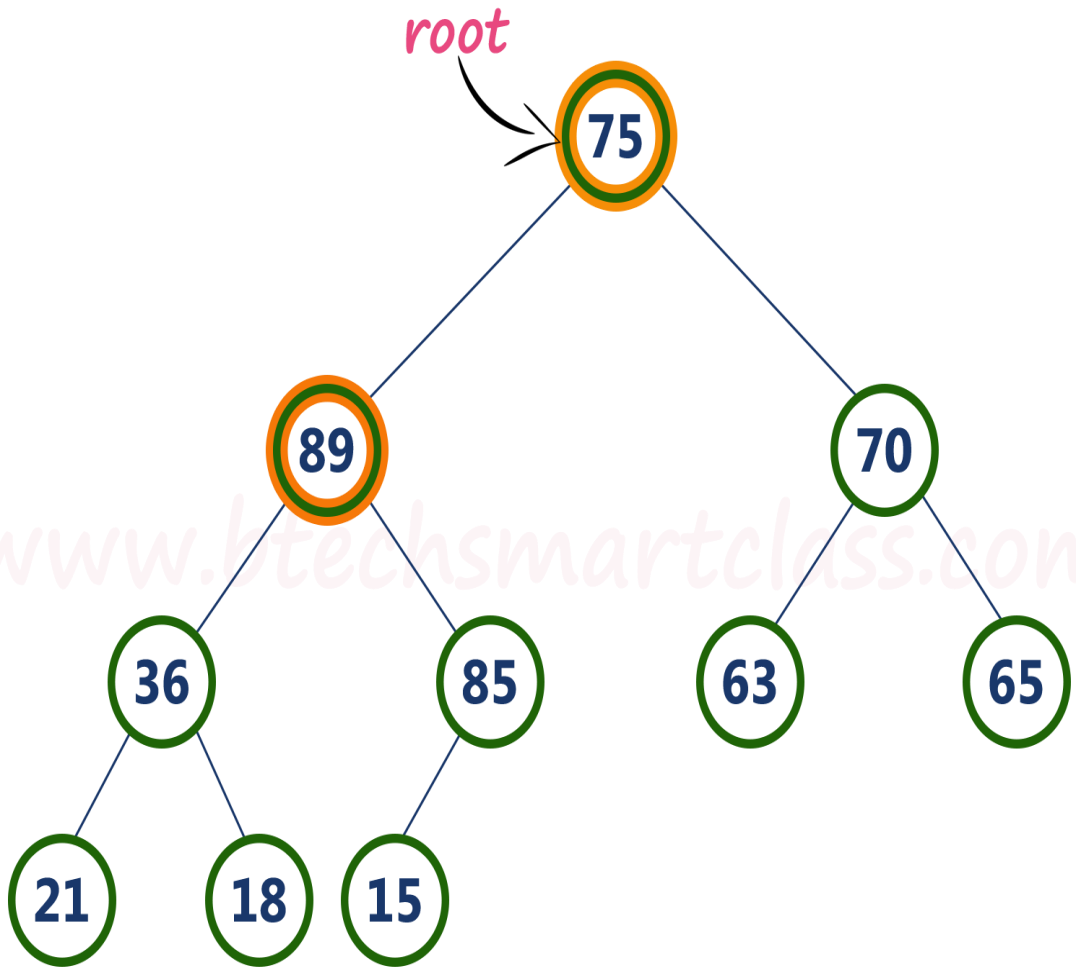




Step 2: Delete last node. Here node with value 90. After deleting node with value 90 from heap, max heap is as follows...

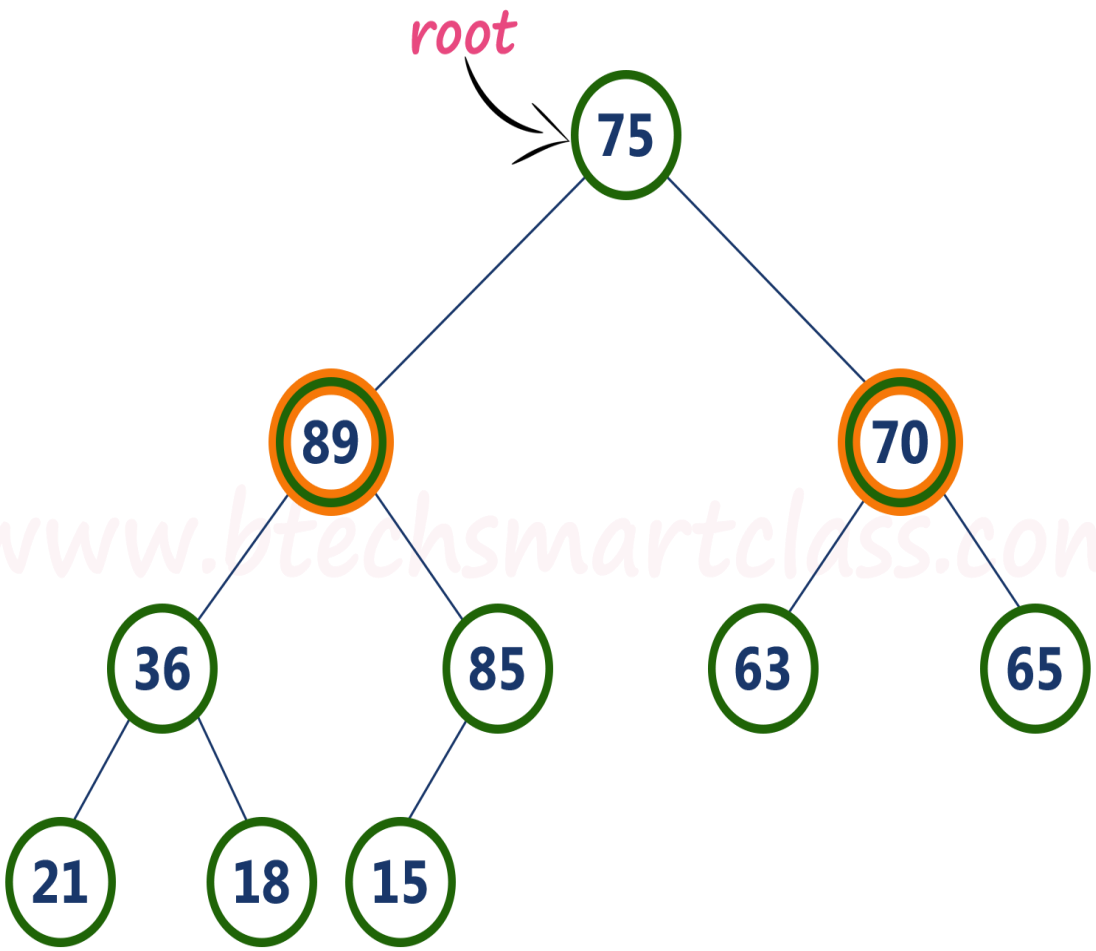
Step 3: Compare **root node (75)** with its **left child (89)**.

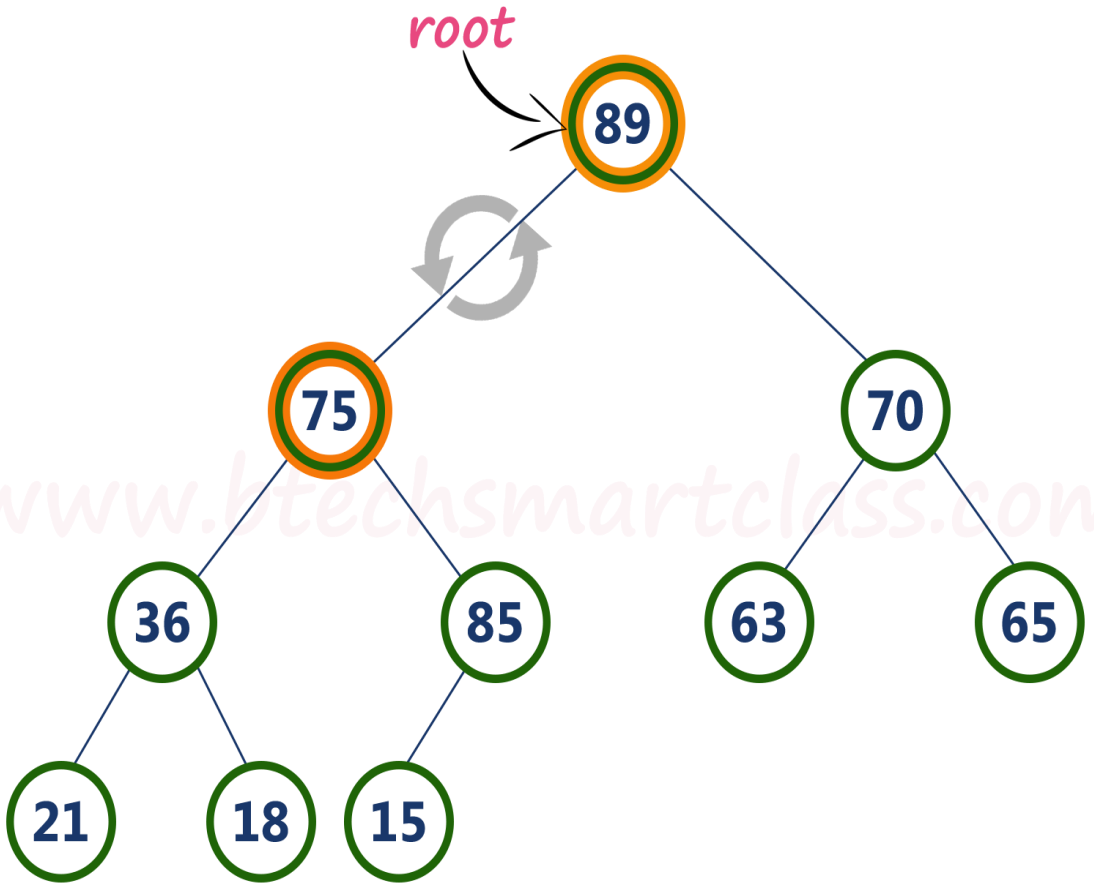




Here, root value (75) is smaller than its left child value (89). So, compare left child (89) with its right sibling (70).

Step 4: Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



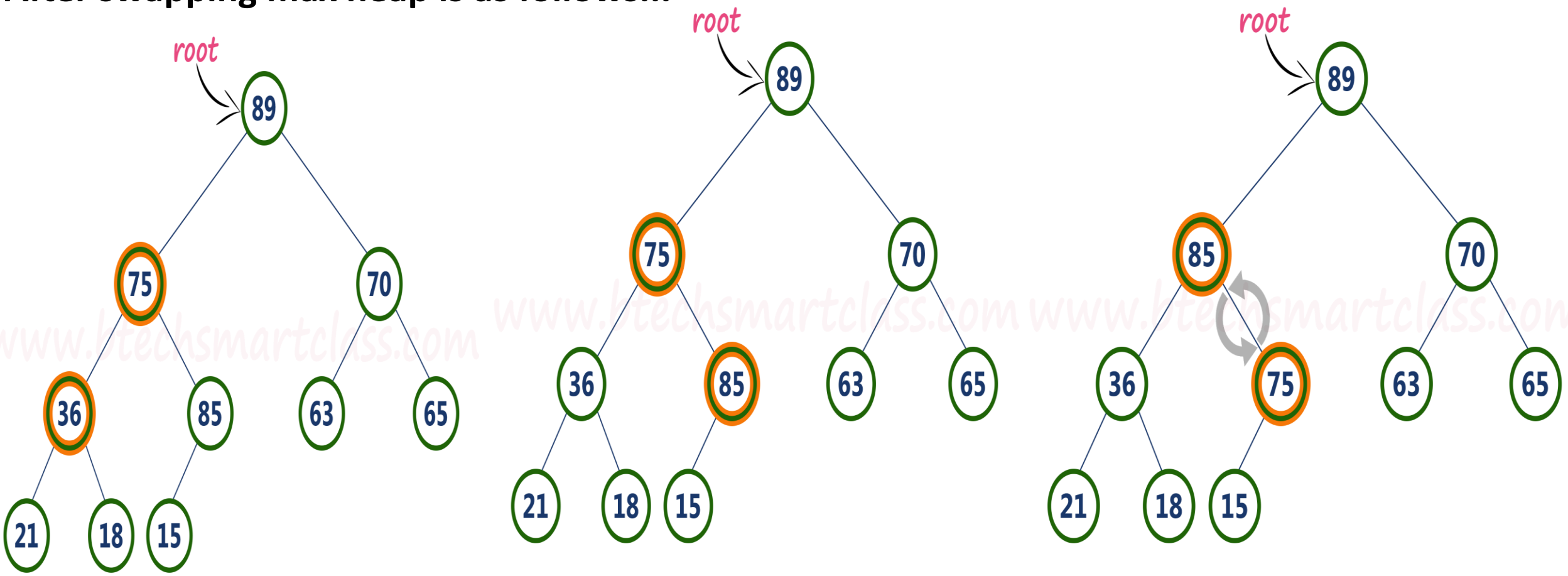


Step 5: Now, again compare 75 with its left child (36).

Here, node with value 75 is larger than its left child. So, we compare node with value 75 is compared with its right child 85.

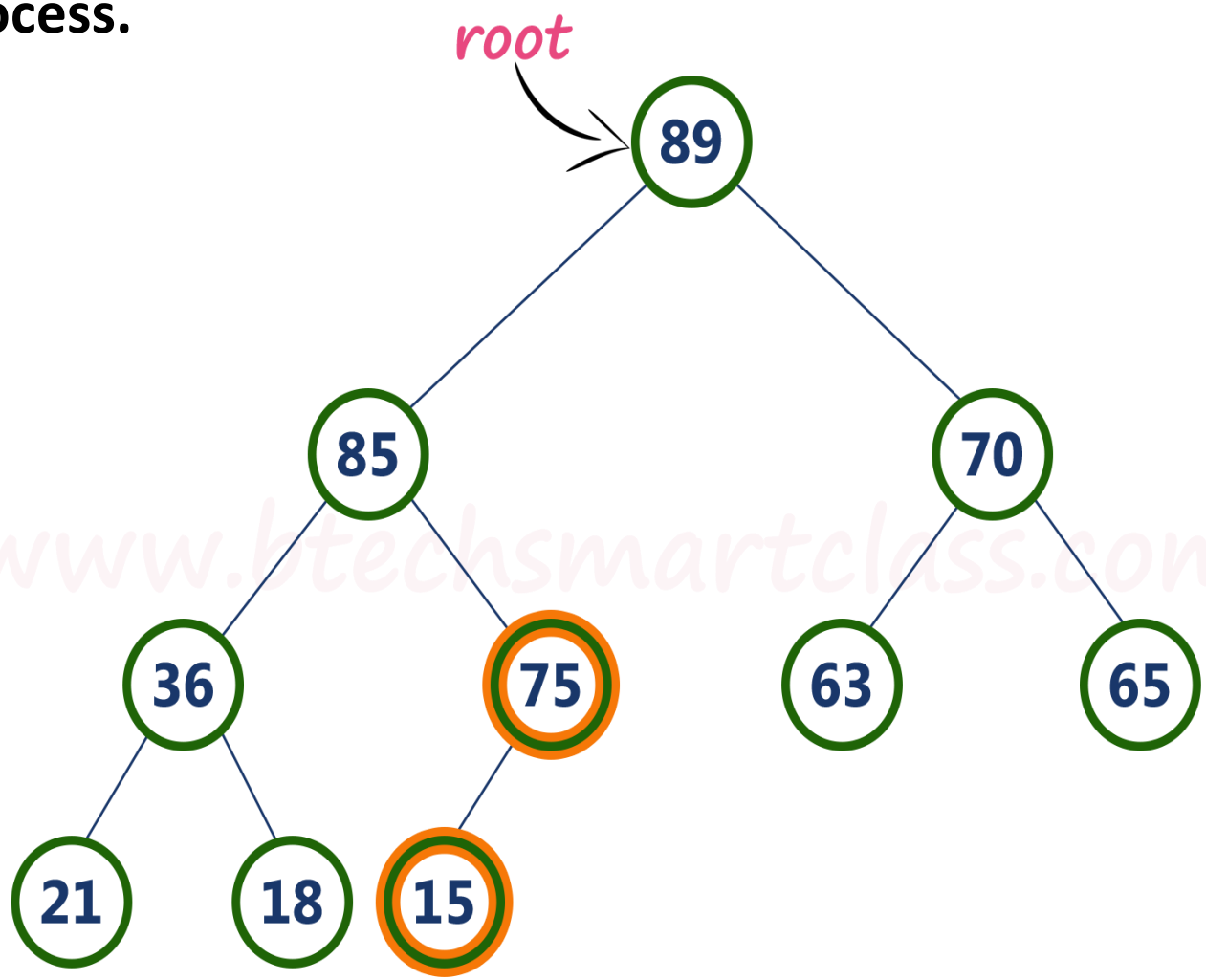
Step 6: Here, node with value 75 is smaller than its right child (85). So, we swap both of them.

After swapping max heap is as follows...



Step 7: Now, compare node with value 75 with its left child (15).

Here, node with value 75 is larger than its left child (15) and it does not have right child. So we stop the process.



Module 2 - Advanced Data Structures :

Introduction
B/B+ tree
Red-Black Trees
Heap operations 2 - MIN Heap
Implementation of queue using heap
Topological sort
Analysis of All problems

Advanced Data Structures and Analysis of Algorithms
ITCDLO 5011

Last session :

Heap operations 1 - MAX Heap

Current session :

Heap operations 2 - MIN Heap

1. What is a MIN Heap ?

2. How to create a MIN / MAX Heap from elements stored in array ?

3. Given the position of the parent in array , how to find index of its left and right children in the array and vice versa ?

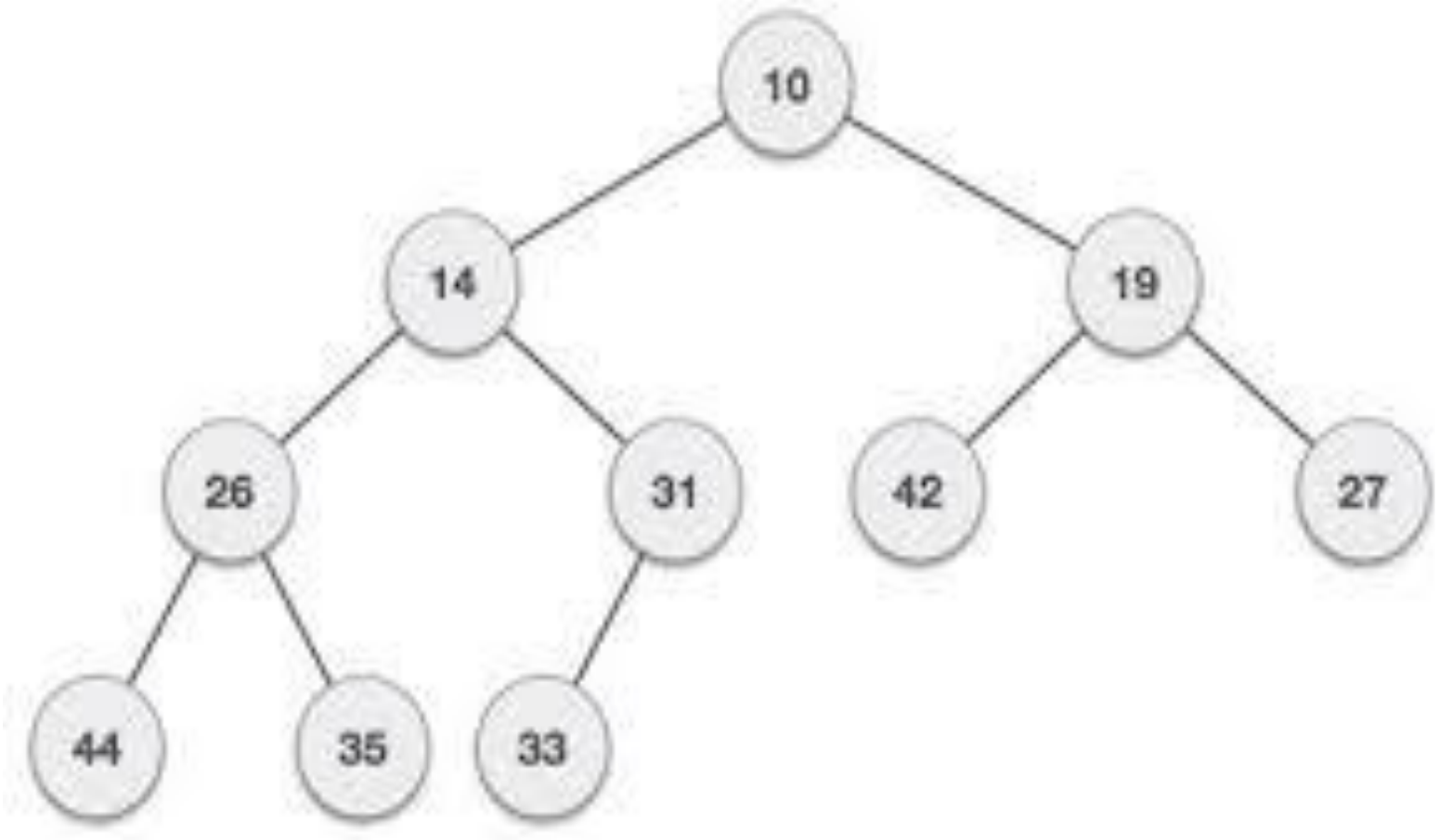
4. How to insert a node in MIN Heap ?

5. How to delete a node from MIN Heap ?

What is a MIN Heap ?

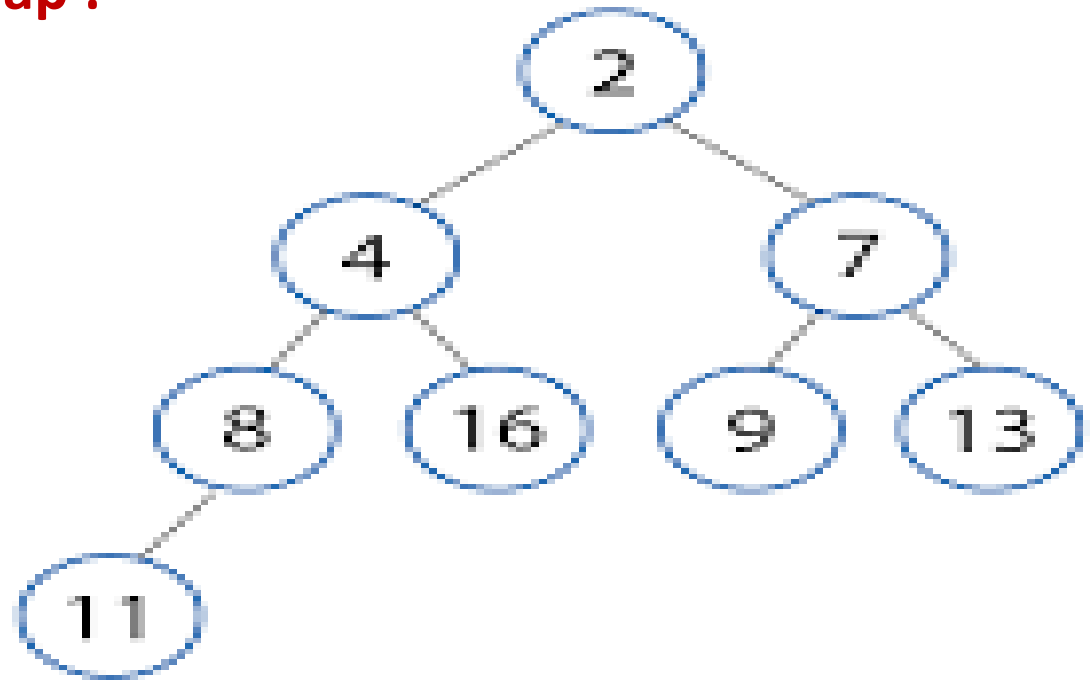
A MIN Heap : is a complete binary tree such that

- the data contained in each node is less than (or equal to) the data in that node's children.



How to create a MIN / MAX Heap from elements stored in array ?

A MIN Heap :



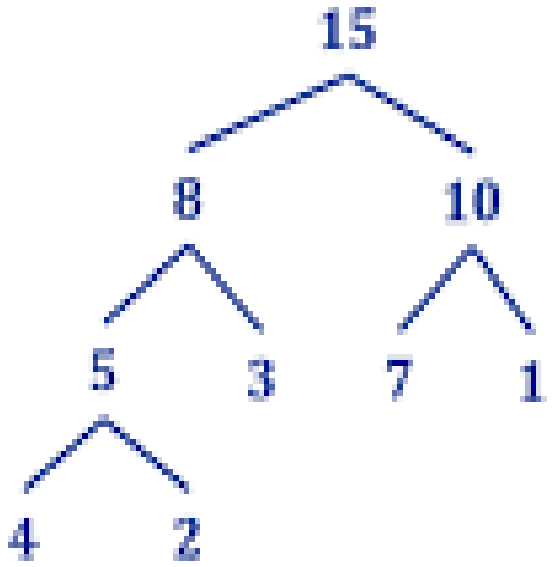
0	1	2	3	4	5	6	7	8
nil	2	4	7	8	16	9	13	11

Storage of a heap :

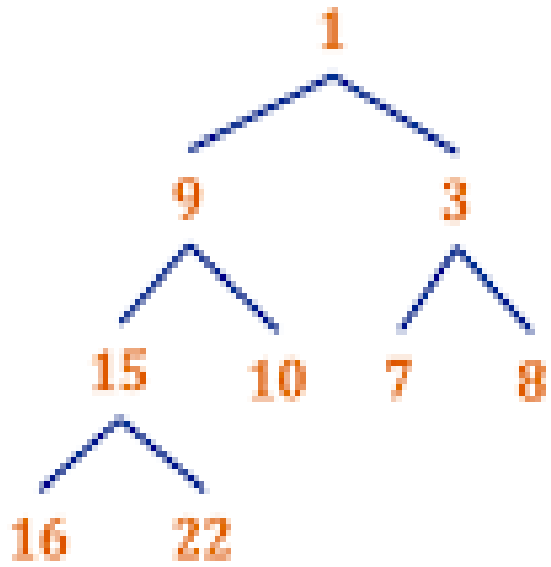
Use an array to hold the data.

Store the root in position 1.

We won't use index 0 for this implementation.



Max-Heap



Min-Heap

i =	1	2	3	4	5	6	7	8	9
	15	8	10	5	3	7	1	4	2

1	2	3	4	5	6	7	8	9
1	9	3	15	10	7	8	16	22

For any node in position i,

its left child (if any) is in position 2i

its right child (if any) is in position 2i + 1

its parent (if any) is in position i/2 (use integer division)

Inserting into a min-heap :

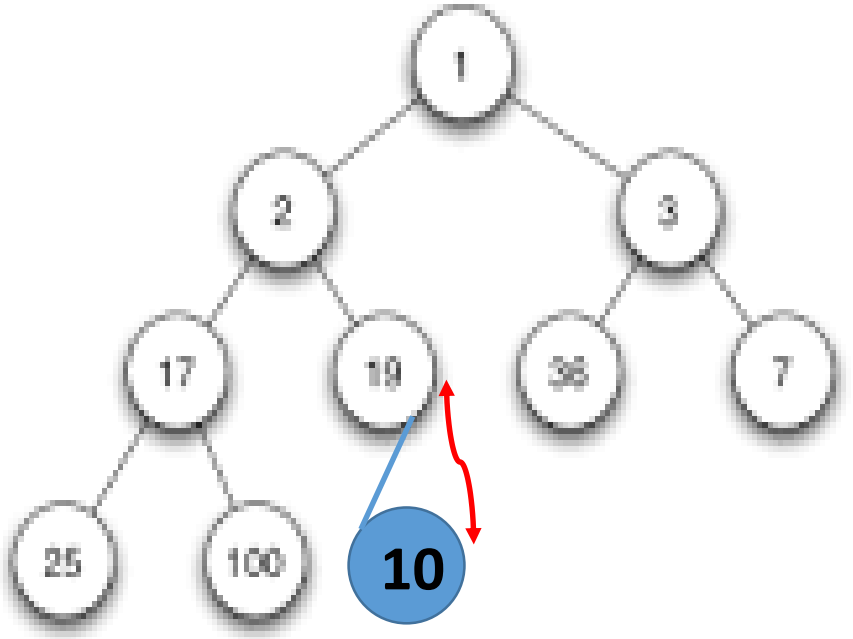
Place the new element in the next available position in the array.

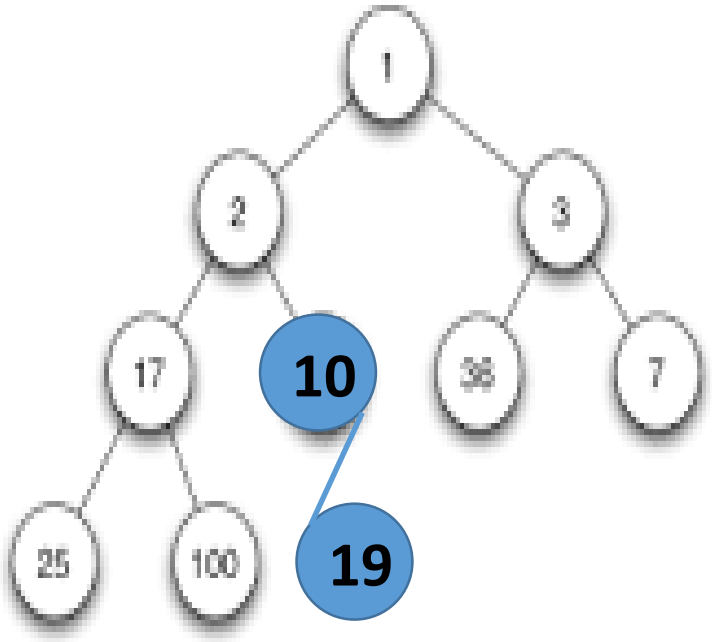
Compare the new element with its parent.

If the new element is smaller, than swap it with its parent.

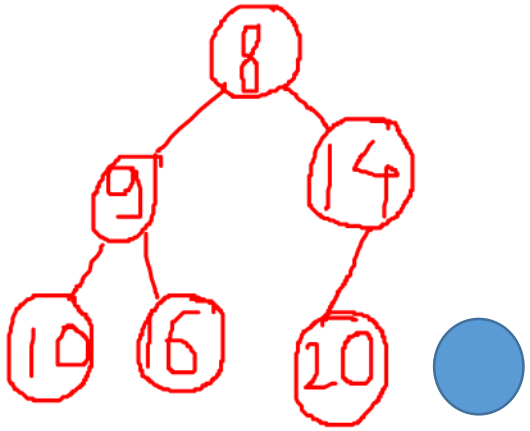
Continue this process until either

- **the new element's parent is smaller than or equal to the new element, or**
- **the new element reaches the root (index 0 of the array)**



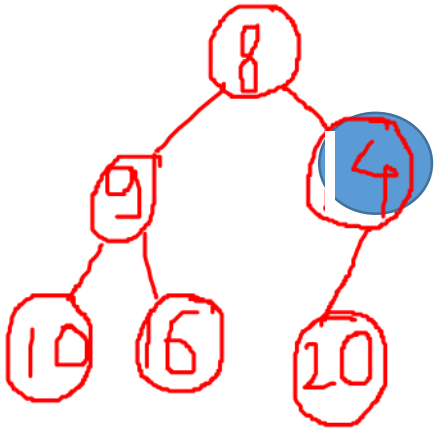


Inserting into a min-heap : Eg. Insert element 4 in the MIN Heap (a) given below.



(a)

(b)



(C)

(d)

Deletion from a MIN heap :

Place the root element in a variable to return later.

Remove the last element in the deepest level and move it to the root.

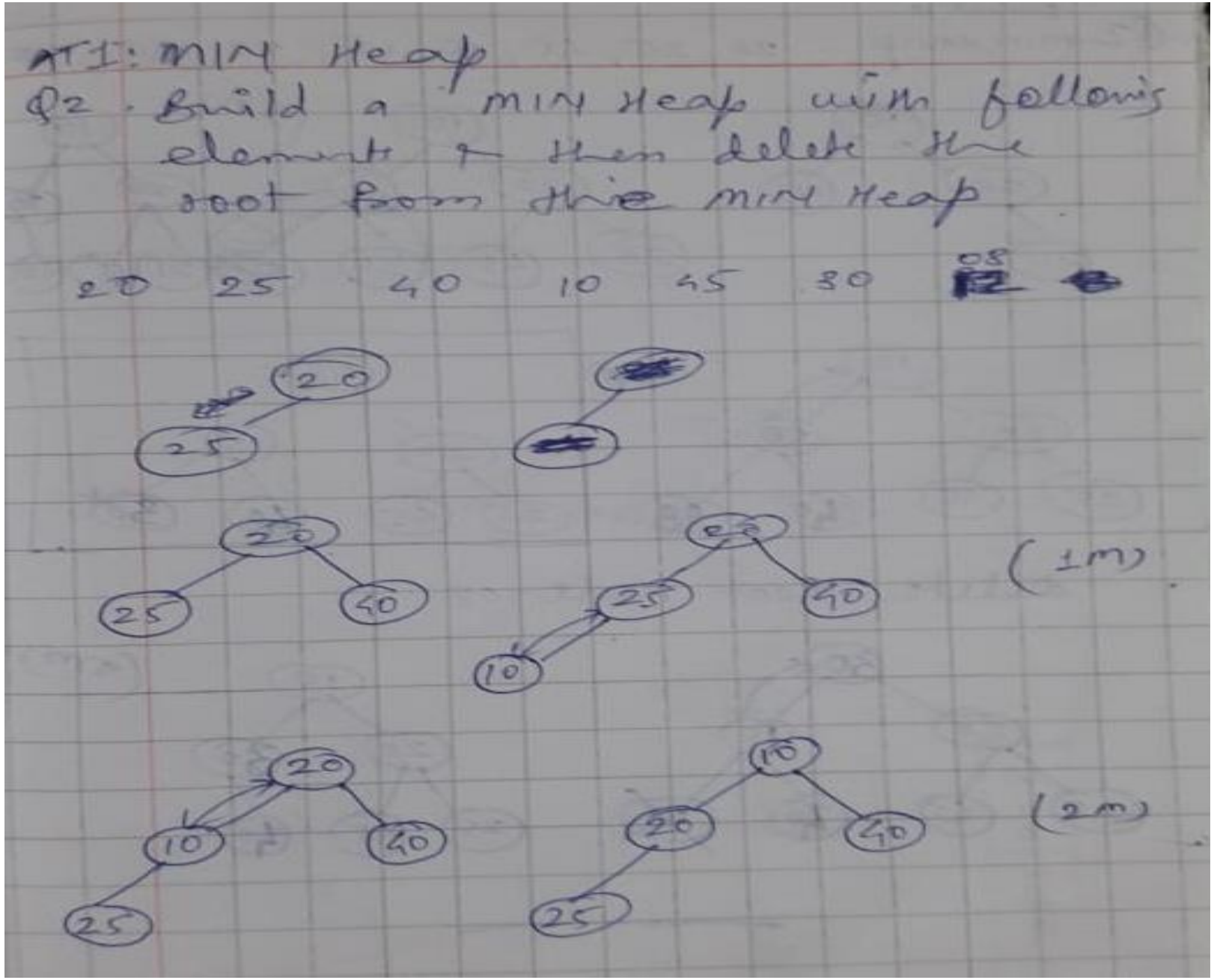
While the moved element has a value greater than at least one of its children, swap this value with the smaller-valued child.

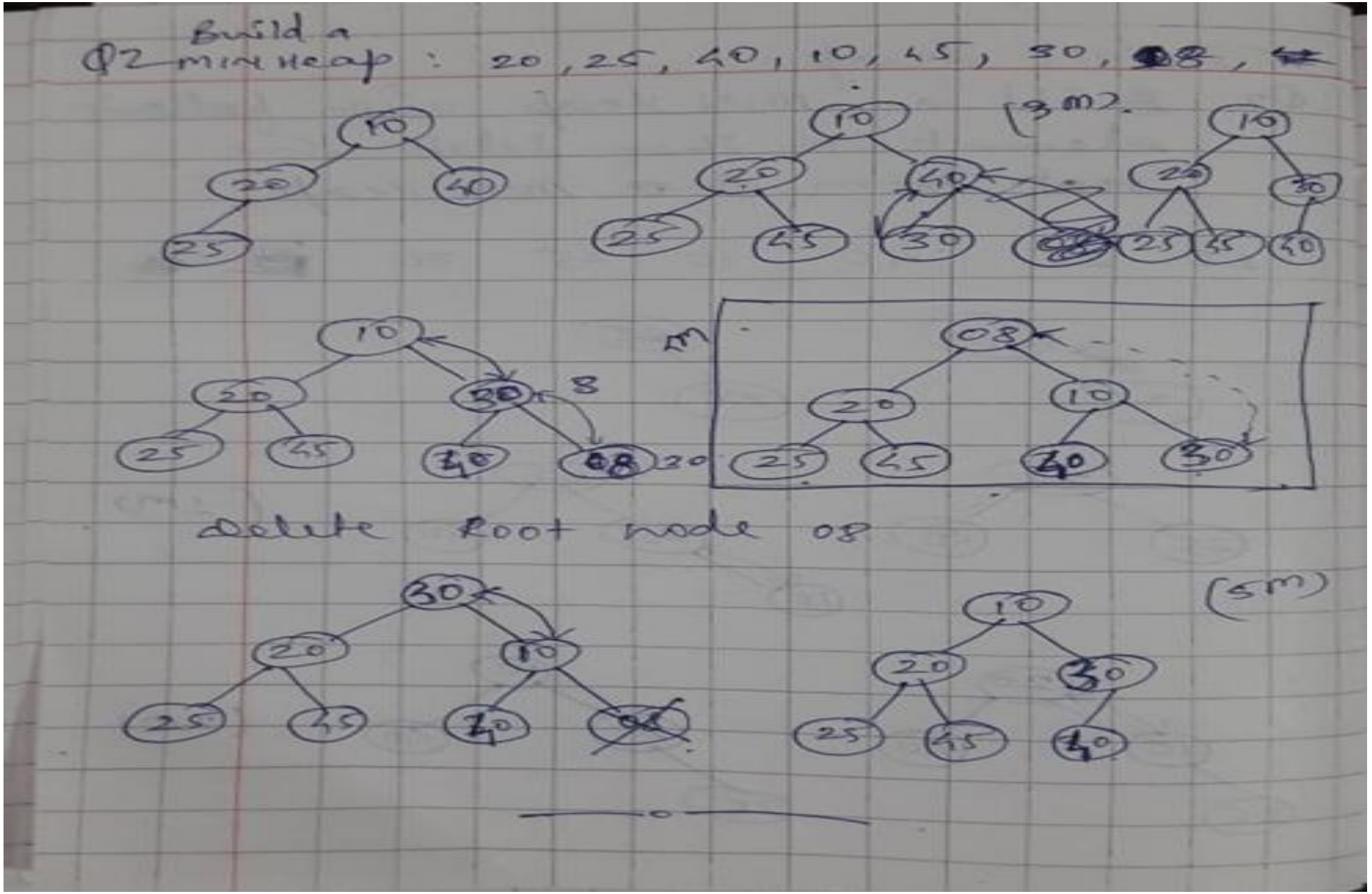
Return the original root that was saved.

Deletion from a MIN heap : **Eg. Build a MIN Heap with array**

2	6	3	8	7	9	5

and then Delete element 2 from that MIN heap .


AT1: MIN Heap
Q2. Build a MIN Heap with following elements & then delete the root from the MIN Heap.
20 25 40 10 45 30
(1m)
(2m)


Q2 Build a min Heap : 20, 25, 40, 10, 45, 30, 08, 50
(3m)
Delete Root node 08
(5m)

Summary MIN Heap :

What is a MIN Heap ?

How to create a MIN / MAX Heap from elements stored in array ?

Given the position of the parent in array , how to find index of its left and right children in the array and vice versa ?

How to insert a node in MIN Heap ?

How to delete a node from MIN Heap ?

Applications of Heaps :

Sorting an array using Heap Sort algorithm.

Implementation of Priority queues.

Implementation of priority queue using Heap :

Queue is the first-in first-out data structure.

One important variation of a queue is called a priority queue.

A priority queue acts like a queue in that you de-queue an item by removing it from the front.

However, in a priority queue the logical order of items inside a queue is determined by their priority.

The highest priority items are at the front of the queue and the lowest priority items are at the back.

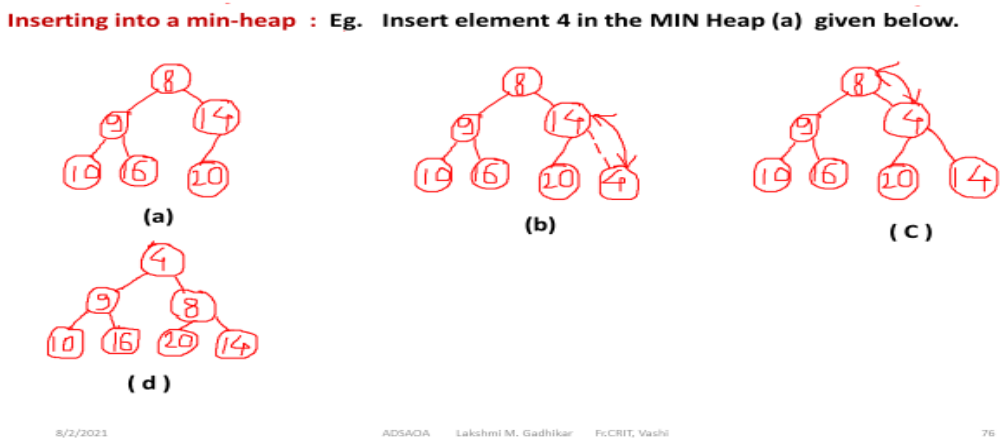
Thus when you en-queue an item on a priority queue, the new item may move all the way to the front.

Applications of Priority Queue:

4	8	15	19	22	10

4	8	10	15	19	22

1) CPU Scheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3) All queue applications where priority is involved.



Different ways to implement a priority queue are :

Lists (array) using sorting functions and
Heaps.

Implementation using Lists :

Every time we insert an element in a list , we need to sort it in priority order.
Inserting into a list takes O(n) time and sorting a list takes O(n log n) time.

Implementation using Heaps :

Implementation of a priority queue using a binary heap is more advantageous over list implementation because a binary heap will allow us both enqueue and dequeue items in O(log n).

For the insert operation in heap , we start by adding a value to the end of the array (constant time, assuming the array doesn't have to be expanded)

then we swap values up the tree until the order property has been restored.

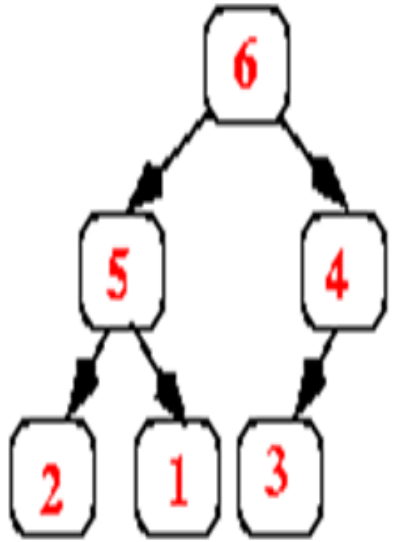
In the worst case, we follow a path all the way from a leaf to the root (i.e., the work we do is proportional to the height of the tree).

Because a heap is a balanced binary tree, the height of the tree is O(log N), where N is the number of values stored in the tree.

The removeMax operation is similar: in the worst case, we follow a path down the tree from the root to a leaf. Again, the worst-case time is O(log N).

Implementing priority queues using heaps :

The standard approach of implementing priority queues using heaps is to use an array (or an ArrayList), starting at position 1 (instead of 0), where each item in the array corresponds to one node in the heap:



The root of the heap is always in array[1].
Its left child is in array[2].
Its right child is in array[3].
In general, if a node is in array[k], then
its left child is in array[k*2], and
its right child is in array[k*2 + 1].
If a node is in array[k], then its parent is in array[k/2]
(using integer division, so that if k is odd, then the result is truncated; e.g., 3/2 = 1).

Above example shows both the conceptual heap (the binary tree), and its array representation.

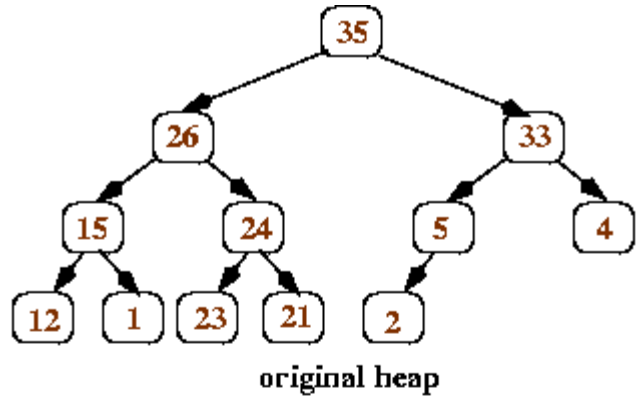
When a new value is inserted into a priority queue, we need to:
Add the value so that the heap still has the order and shape properties.

Step 1 : Add the new value at the end of the array; that corresponds to adding it as a new rightmost leaf in the tree (or, if the tree was a complete binary tree, i.e., all leaves were at the same depth d, then that corresponds to adding a new leaf at depth d+1).

Step 1 above ensures that the heap still has the shape property; however, it may not have the order property.
We can check that by comparing the new value to the value in its parent.

If the parent is smaller, we swap the values, and we continue this check-and-swap procedure up the tree until we find that the order property holds, or we get to the root.

Insert the value 34 into a heap:



original heap

Removing an item from a priority queue implemented using heap is same as removing an item from max or min heap as done before.

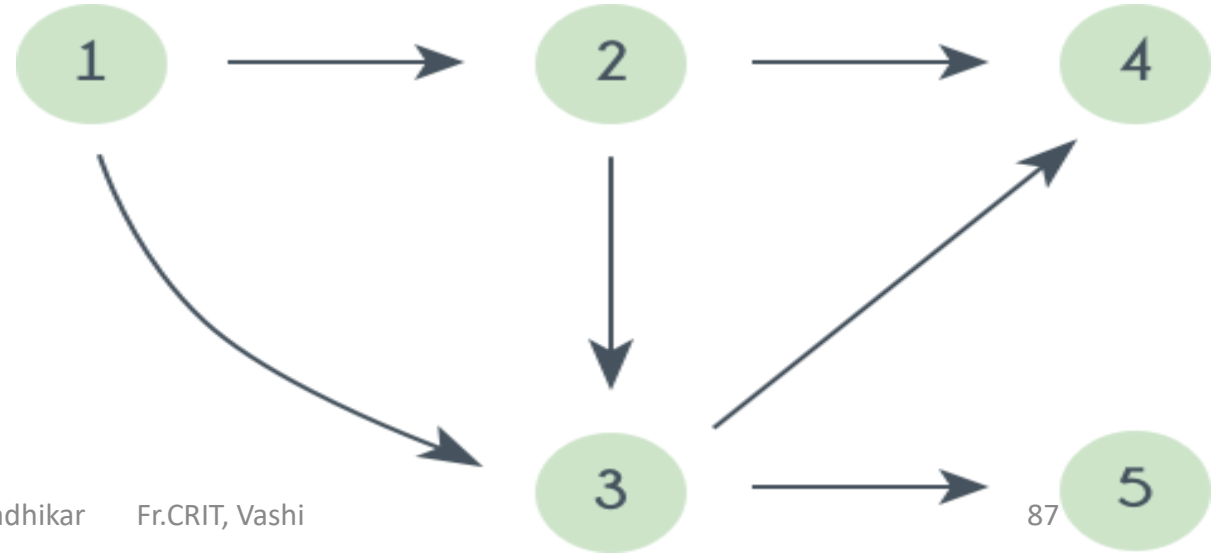
Write a note on topological sorting. (10M SH-18)

Topological sorting : for Directed Acyclic Graph (DAG) is a linear ordering of vertices v1,v2,v3 …. … , vn such that if there is a directed edge going out from vertex vi to vj , then vertex vi comes before vertex vj in the ordering.

Topological Sorting for a graph is not possible if the graph is not a DAG.
For example, a topological sorting of the following graph is “1 2 3 4 5”.

There can be more than one topological sorting for a graph.
For example, another topological sorting of the following graph is “1 2 3 5 4”.

The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Two methods to find topological Ordering are :

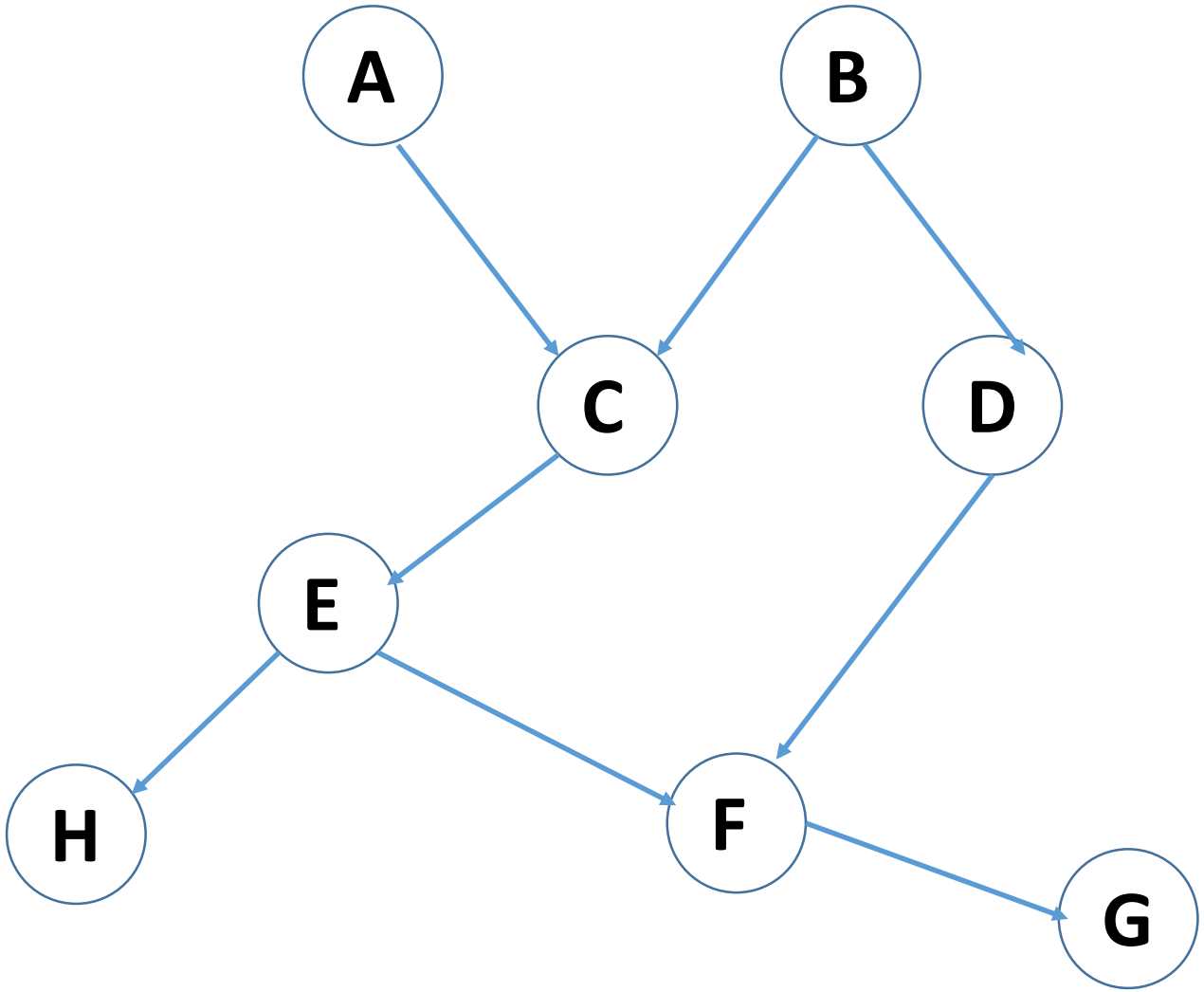
1. DFS method
2. Adjacency matrix method

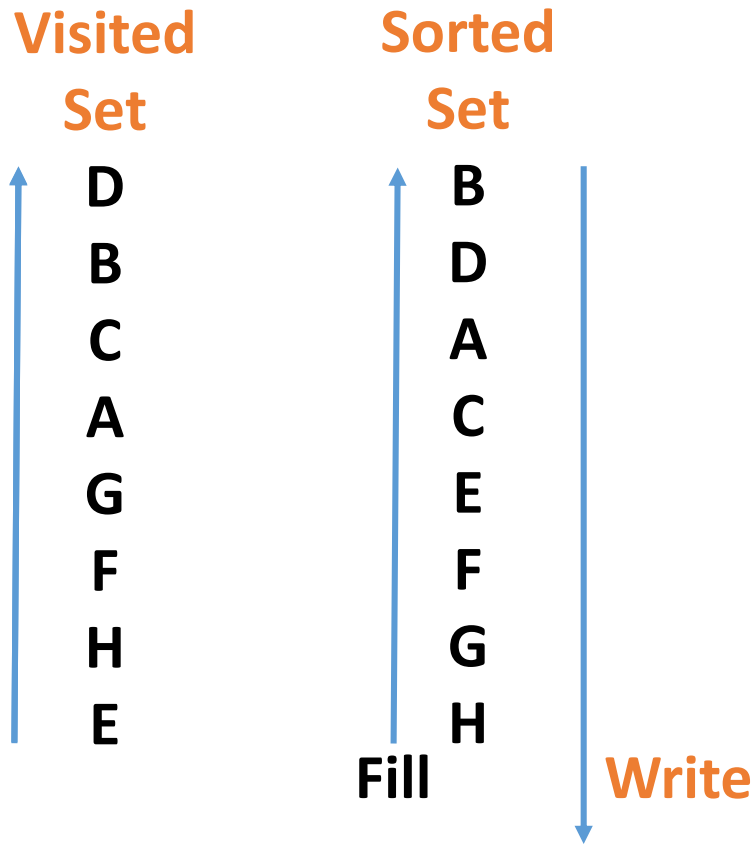
1. DFS method :

1. Create two sets 1. Visited set 2. Sorted set
2. Select any node from where to start say E and put it in Visited set at the bottom.
3. Traverse all the nodes one by one that can be reached from E.
   Eg. H and F
   Select one of them say H and Put it in Visited set.
4. Check whether there are any outgoing edges from this selected node H.
   If YES, Put them in Visited set
   If NO, Put that node Eg. H in the Sorted set.
5. Go back to the node from where you had started. Eg. E
6. Repeat above steps till all nodes in the graph are visited.

After F, you can visit either A or B. You can choose any one.

Topological Sorting - 1. DFS method : Multiple Topological orders are possible.





Topological Ordering :

B, D, A, C, E, F, G, H

2. Adjacency matrix method to find topological Ordering :

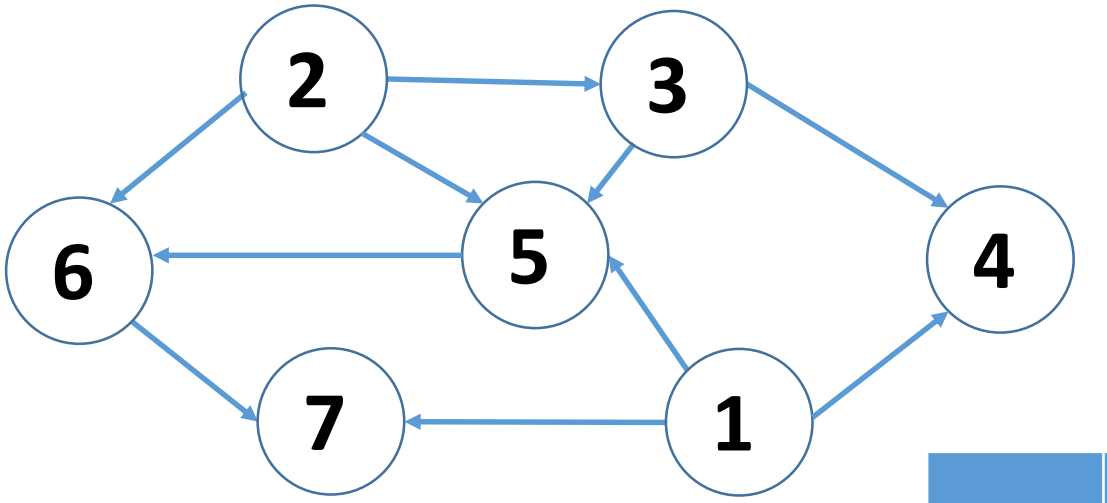
1. Create an adjacency matrix with the node numbers written in both rows and columns.

2. For each row Eg. 1 in the following graph, if there is an outgoing edge from 1 to nodes 4, 5, 7, then mark 1 in the columns corresponding to the nodes 4,5,7 respectively.

3. Select any node with an in-degree = 0, i.e. the node which does not have any incoming edges.
4. Delete its row from the adjacency matrix and write that node in the topological order.

5. Now, select the next node with in-degree = 0. This is done by selecting the column Eg. Column 2, which is empty after deleting row 1 in above example.
   All other columns have at least one 1 in them.
6. So, Delete its row (Eg. 2) from the adjacency matrix and write that node 2 in the topological order.
7. Repeat above steps till all nodes in the graph are visited and added in the topological order.

Topological Sorting - 2. Adjacency matrix method :

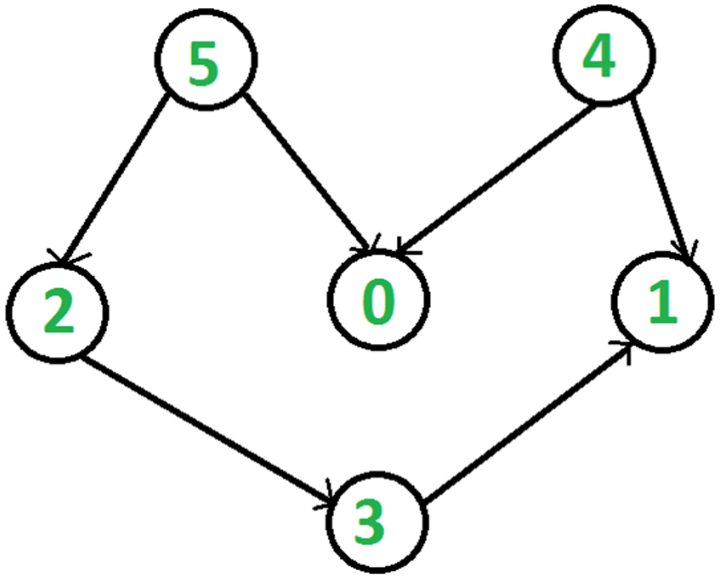


Topological Ordering :

{1, 2} , 3, {4, 5}, 6, 7

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

Topological Sorting - 2. Adjacency matrix method : 4 5 2 3 1 0 , 5 4 2 3 1 0 , {4,5} , {0,2} , 3, 1



Topological Ordering :

{4,5} , {0,2} , 3, 1

	0	1	2	3	4	5
0						
1						
2				1		
3		1				
4	1	1				
5	1		1			

END OF MODULE 2