**UNIT 2**

**(2 MARKS)**

**1.Describe the properties of Red-Black tree**.

Red-Black tree is a binary search tree in which every node is colored with either red or black. It is a type of self-balancing binary search tree. It has a good efficient worst case running time complexity.

**Properties of Red Black Tree:**

The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties –

1. **Root property:** The root is black.

2. **External property:** Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.

3. **Internal property:** The children of a red node are black. Hence possible parent of red node is a black node.

4. **Depth property:** All the leaves have the same black depth.

5. **Path property:** Every simple path from root to descendant leaf node contains same number of black nodes.

The result of all these above-mentioned properties is that the Red-Black tree is roughly balanced.

**2.Explain left rotation in RB tree.**

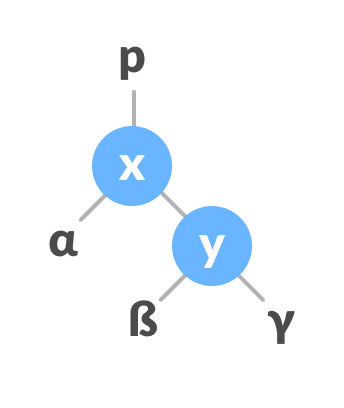
Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

There are two types of rotations:

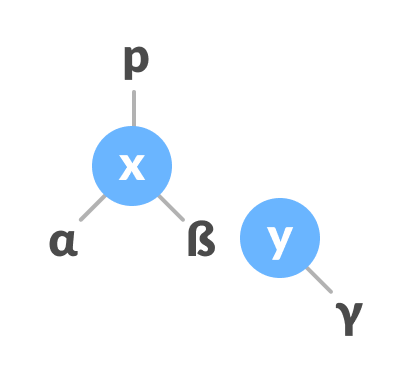
**Left Rotate**

In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

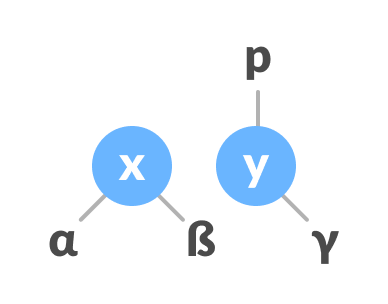
**Algorithm**

1. Let the initial tree be:

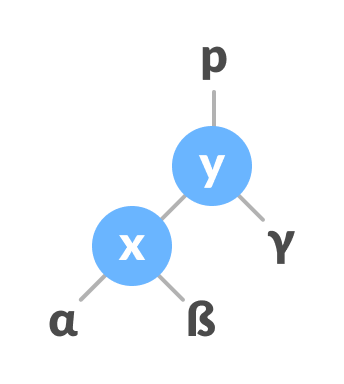
Initial tree

1. If y has a left subtree, assign x as the parent of the left subtree of y.

Assign x as the parent of the left subtree of y

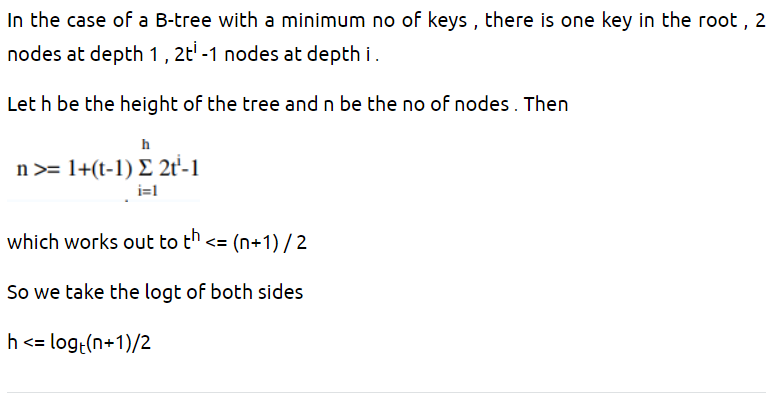
1. If the parent of x is NULL, make y as the root of the tree.
2. Else if x is the left child of p, make y as the left child of p.
3. Else assign y as the right child of p.

Change the parent of x to that of y

1. Make y as the parent of x.

Assign y as the parent of x.

**3.Prove that if n>=1, then for any n-key B-Tree of height h and minimum degree t >=2, h<= log t ((n +1)/2).**



**4.** **Discuss the properties of binomial trees(Heap).**

Binomial Heap is an extension of [Binary Heap](https://www.geeksforgeeks.org/binary-heap/)that provides faster union or merge operation with other operations provided by Binary Heap. *A Binomial Heap is a collection of Binomial Trees*.

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one the leftmost child of the other. A Binomial Tree of order k the has following properties.

* It has exactly 2k nodes.
* It has depth as k.
* There are exactly kaiCi nodes at depth i for i = 0, 1, . . . , k.
* The root has degree k and children of the root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

**5.Difference between Complete Binary Tree and full Binary Tree?**

|  |  |
| --- | --- |
| **Full Binary tree** | **Complete Binary tree** |
| In a full binary tree every node has either 0 or 2 child nodes. | In a complete binary tree, all levels are completely filled except possibly the last level. |
| There is no particular sequence for filling in nodes. | All nodes should be on the left. This means that there is no node that has a right child but not a left child. |
| There is no necessity for all leaf nodes to be at the same level. | All leaf nodes must be at the same level. |

**6.Discuss Skip list and its operations.**

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

## **Skip List Basic Operations**

There are the following types of operations in the skip list.

* **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
* **Deletion operation:** It is used to delete a node in a specific situation.
* **Search Operation:** The search operation is used to search a particular node in a skip list.

**7.Discuss the properties of binomial trees.**

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one the leftmost child of the other. A Binomial Tree of order k the has following properties.

* It has exactly 2k nodes.
* It has depth as k.
* There are exactly kaiCi nodes at depth i for i = 0, 1, . . . , k.
* The root has degree k and children of the root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

**8.Write down the properties of Fibonacci Heap.**

Fibonacci Heap - A Fibonacci heap is defined as the collection of rooted-tree in which all the trees must hold the property of Min-heap. That is, for all the nodes, the key value of the parent node should be greater than the key value of the parent node:

**Properties of Fibonacci Heap:**

1. It can have multiple trees of equal degrees, and each tree doesn't need to have 2^k nodes.
2. All the trees in the Fibonacci Heap are rooted but not ordered.
3. All the roots and siblings are stored in a separated circular-doubly-linked list.
4. The degree of a node is the number of its children. Node X -> degree = Number of X's children.
5. Each node has a mark-attribute in which it is marked TRUE or FALSE. The FALSE indicates the node has not any of its children. The TRUE represents that the node has lost one child. The newly created node is marked FALSE.
6. The potential function of the Fibonacci heap is F(FH) = t[FH] + 2 \* m[FH]
7. The Fibonacci Heap (FH) has some important technicalities listed below:
   1. min[FH] - Pointer points to the minimum node in the Fibonacci Heap
   2. n[FH] - Determines the number of nodes
   3. t[FH] - Determines the number of rooted trees
   4. m[FH] - Determines the number of marked nodes
   5. F(FH) - Potential Function.

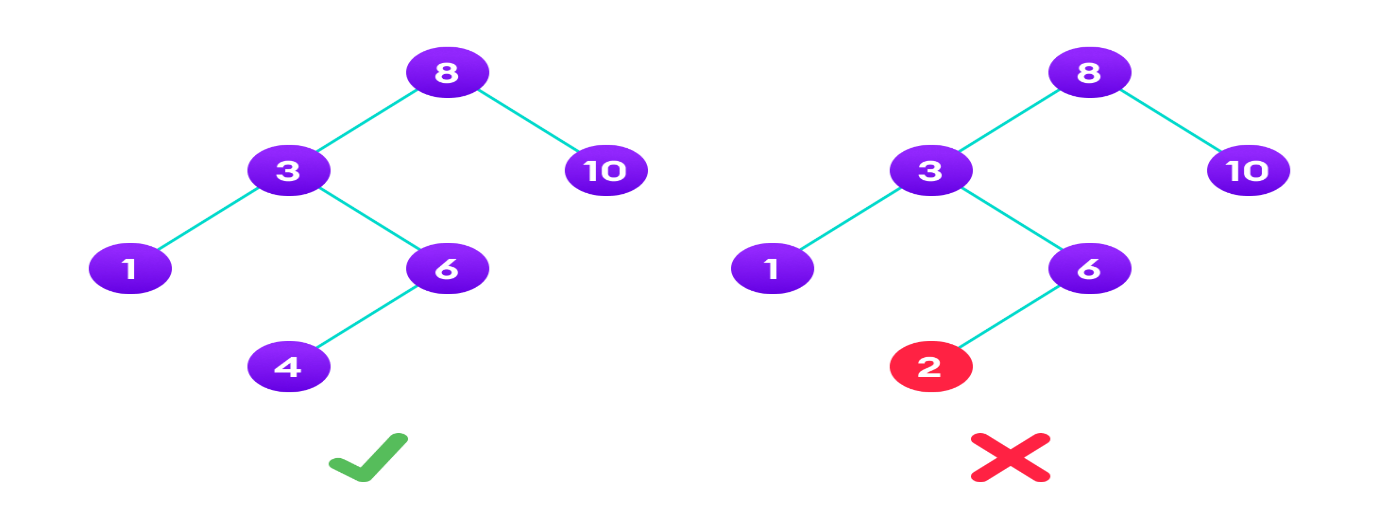
**9.Write short note on Binary Search Tree.**

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

* It is called a binary tree because each tree node has a maximum of two children.
* It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular [binary tree](https://www.programiz.com/data-structures/trees) is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties

A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

**10.Prove that maximum degree of any node in an n node binomial tree is log n.**

* 1. **MARKS)**

**1.Write the properties of Red-Black Tree. Illustrate with an example, how the keys are inserted in an empty red-black tree.**

**The red-Black tree** is a binary search tree. The prerequisite of the red-black tree is that we should know about the binary search tree. In a binary search tree, the values of the nodes in the left subtree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node.

**Properties of Red-Black tree**

* It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
* This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
* In the Red-Black tree, the root node is always black in color.
* In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
* If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.
* Every path from a node to any of its descendant's NIL node should have same number of black nodes.

**Insertion in Red Black tree**

**The following are some rules used to create the Red-Black tree:**

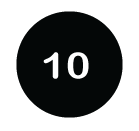
1. If the tree is empty, then we create a new node as a root node with the color black.
2. If the tree is not empty, then we create a new node as a leaf node with a color red.
3. If the parent of a new node is black, then exit.
4. If the parent of a new node is Red, then we have to check the color of the parent's sibling of a new node.

4a) If the color is Black, then we perform rotations and recoloring.

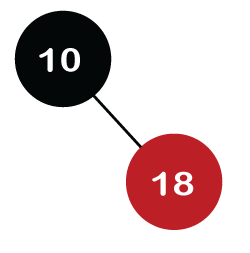
4b) If the color is Red then we recolor the node. We will also check whether the parents' parent of a new node is the root node or not; if it is not a root node, we will recolor and recheck the node.

**Example:10, 18, 7, 15, 16, 30, 25, 40, 60**

**Step 1:** Initially, the tree is empty, so we create a new node having value 10. This is the first node of the tree, so it would be the root node of the tree. As we already discussed, that root node must be black in color, which is shown below:

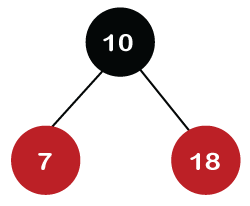


**Step 2:** The next node is 18. As 18 is greater than 10 so it will come at the right of 10 as shown below.



We know the second rule of the Red Black tree that if the tree is not empty then the newly created node will have the **Red** color. Therefore, node 18 has a Red color, as shown in the below figure:Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. In the above figure, the parent of the node is black in color; therefore, it is a Red-Black tree.

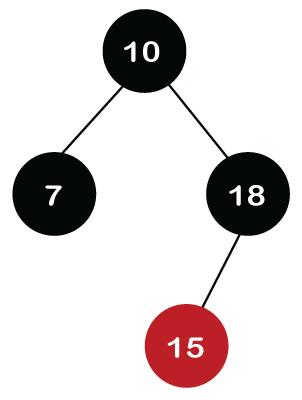
**Step 3:** Now, we create the new node having value 7 with Red color. As 7 is less than 10, so it will come at the left of 10 as shown below.



Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. As we can observe, the parent of the node 7 is black in color, and it obeys the Red-Black tree's properties.

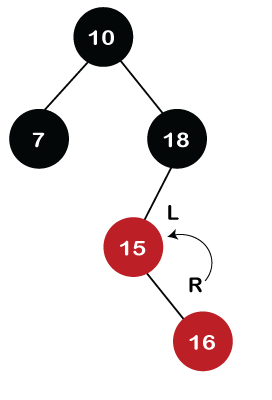
**Step 4:** The next element is 15, and 15 is greater than 10, but less than 18, so the new node will be created at the left of node 18. The node 15 would be Red in color as the tree is not empty.

The above tree violates the property of the Red-Black tree as it has Red-red parent-child relationship. Now we have to apply some rule to make a Red-Black tree. The rule 4 says that ***if the new node's parent is Red, then we have to check the color of the parent's sibling of a new node.*** The new node is node 15; the parent of the new node is node 18 and the sibling of the parent node is node 7. As the color of the parent's sibling is Red in color, so we apply the rule 4a. The rule 4a says that we have to recolor both the parent and parent's sibling node. So, both the nodes, i.e., 7 and 18, would be recolored as shown in the below figure.

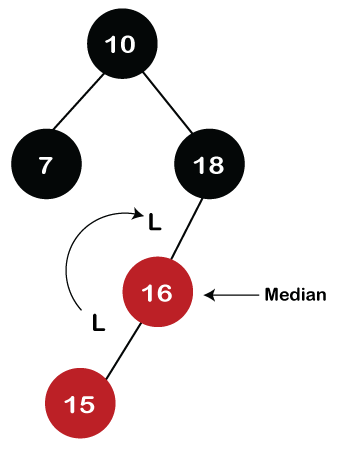


We also have to check whether the parent's parent of the new node is the root node or not. As we can observe in the above figure, the parent's parent of a new node is the root node, so we do not need to recolor it.

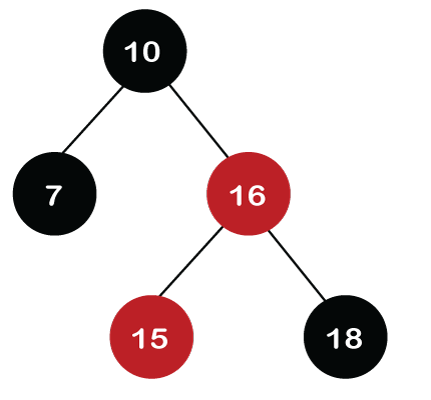
**Step 5:** The next element is 16. As 16 is greater than 10 but less than 18 and greater than 15, so node 16 will come at the right of node 15. The tree is not empty; node 16 would be Red in color, as shown in the below figure:



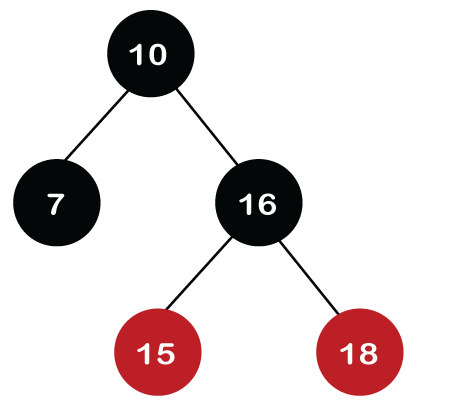
In the above figure, we can observe that it violates the property of the parent-child relationship as it has a red-red parent-child relationship. We have to apply some rules to make a Red-Black tree. Since the new node's parent is Red color, and the parent of the new node has no sibling, so rule **4a** will be applied. The rule **4a** says that some rotations and recoloring would be performed on the tree.Since node 16 is right of node 15 and the parent of node 15 is node 18. Node 15 is the left of node 18. Here we have **an LR** relationship, so we require to perform two rotations. First, we will perform left, and then we will perform the right rotation. The left rotation would be performed on nodes 15 and 16, where node 16 will move upward, and node 15 will move downward. Once the left rotation is performed, the tree looks like as shown in the below figure:



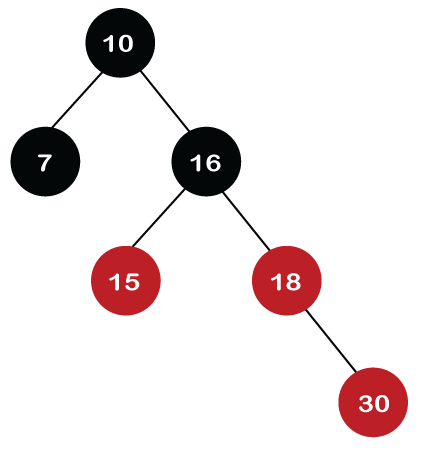
In the above figure, we can observe that there is **an LL** relationship. The above tree has a Red-red conflict, so we perform the right rotation. When we perform the right rotation, the median element would be the root node. Once the right rotation is performed, node 16 would become the root node, and nodes 15 and 18 would be the left child and right child, respectively, as shown in the below figure.



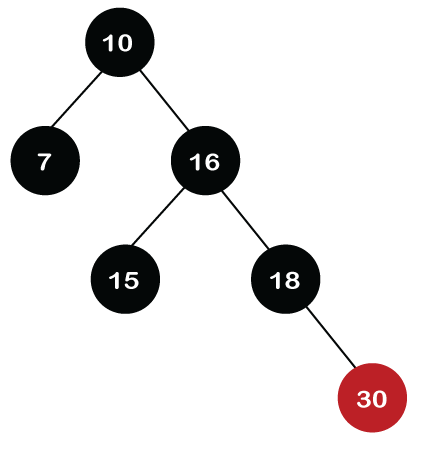
After rotation, node 16 and node 18 would be recolored; the color of node 16 is red, so it will change to black, and the color of node 18 is black, so it will change to a red color as shown in the below figure:



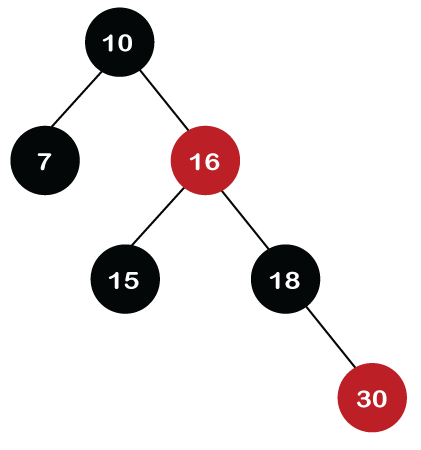
**Step 6:** The next element is 30. Node 30 is inserted at the right of node 18. As the tree is not empty, so the color of node 30 would be red.



The color of the parent and parent's sibling of a new node is Red, so rule 4b is applied. In rule 4b, we have to do only recoloring, i.e., no rotations are required. The color of both the parent (node 18) and parent's sibling (node 15) would become black, as shown in the below image.

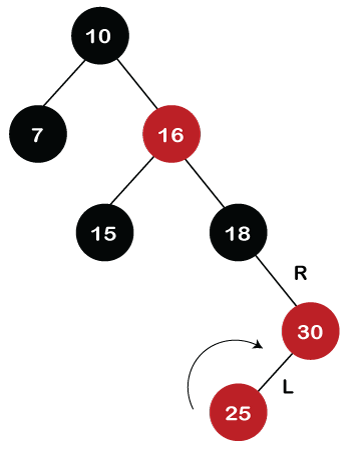


We also have to check the parent's parent of the new node, whether it is a root node or not. The parent's parent of the new node, i.e., node 30 is node 16 and node 16 is not a root node, so we will recolor the node 16 and changes to the Red color. The parent of node 16 is node 10, and it is not in Red color, so there is no Red-red conflict.

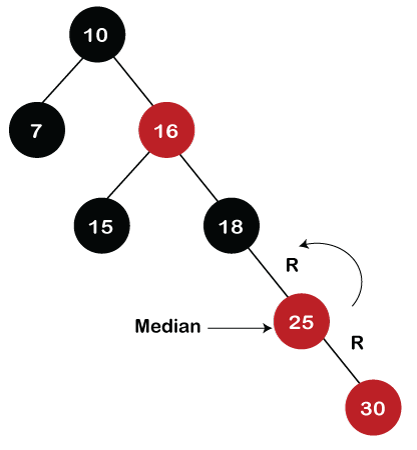


**Step 7:** The next element is 25, which we have to insert in a tree. Since 25 is greater than 10, 16, 18 but less than 30; so, it will come at the left of node 30. As the tree is not empty, node 25 would be in Red color. Here Red-red conflict occurs as the parent of the newly created is Red color.

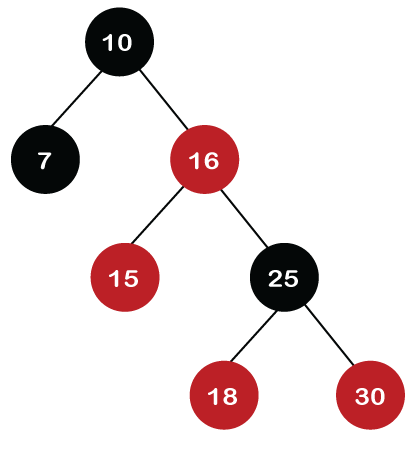
Since there is no parent's sibling, so rule 4a is applied in which rotation, as well as recoloring, are performed. First, we will perform rotations. As the newly created node is at the left of its parent and the parent node is at the right of its parent, so the RL relationship is formed. Firstly, the right rotation is performed in which node 25 goes upwards, whereas node 30 goes downwards, as shown in the below figure.



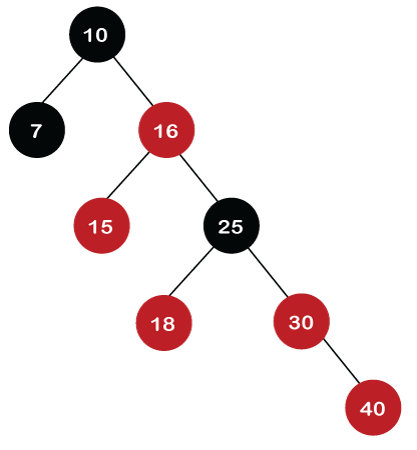
After the first rotation, there is an RR relationship, so left rotation is performed. After right rotation, the median element, i.e., 25 would be the root node; node 30 would be at the right of 25 and node 18 would be at the left of node 25.

* 

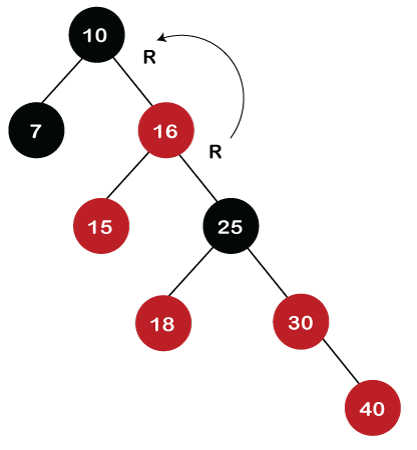
Now recoloring would be performed on nodes 25 and 18; node 25 becomes black in color, and node 18 becomes red in color.



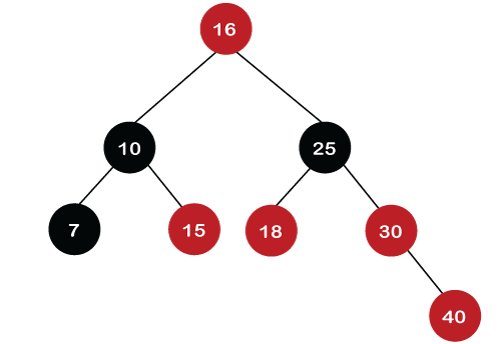
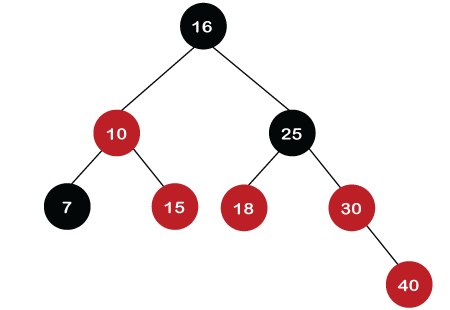
**Step 8:** The next element is 40. Since 40 is greater than 10, 16, 18, 25, and 30, so node 40 will come at the right of node 30. As the tree is not empty, node 40 would be Red in color. There is a Red-red conflict between nodes 40 and 30, so rule 4b will be applied.



As the color of parent and parent's sibling node of a new node is Red so recoloring would be performed. The color of both the nodes would become black, as shown in the below image.After recoloring, we also have to check the parent's parent of a new node, i.e., 25, which is not a root node, so recoloring would be performed, and the color of node 25 changes to Red.After recoloring, red-red conflict occurs between nodes 25 and 16. Now node 25 would be considered as the new node. Since the parent of node 25 is red in color, and the parent's sibling is black in color, rule 4a would be applied. Since 25 is at the right of the node 16 and 16 is at the right of its parent, so there is an RR relationship. In the RR relationship, left rotation is performed. After left rotation, the median element 16 would be the root node, as shown in the below figure.

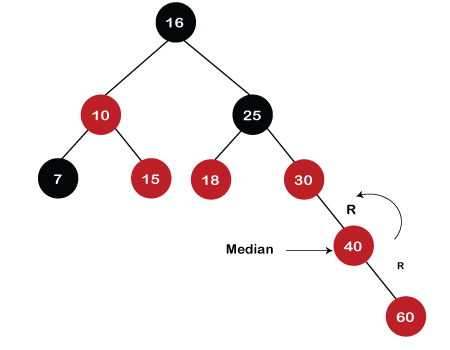


After rotation, recoloring is performed on nodes 16 and 10. The color of node 10 and node 16 changes to Red and Black, respectively as shown in the below figure.

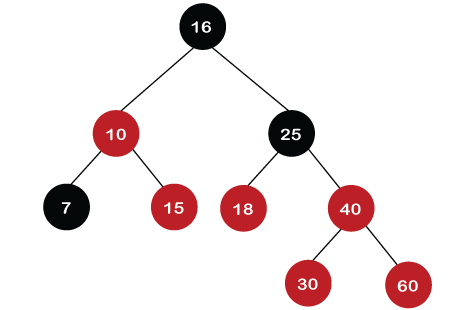
  


**Step 9:** The next element is 60. Since 60 is greater than 16, 25, 30, and 40, so node 60 will come at the right of node 40. As the tree is not empty, the color of node 60 would be Red.

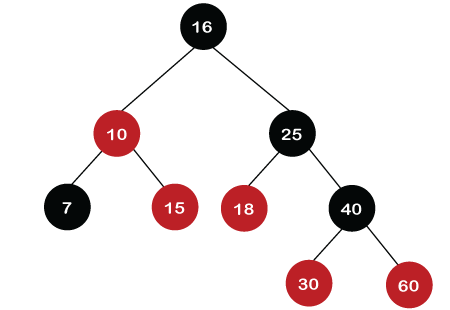
As we can observe in the above tree that there is a Red-red conflict occurs. The parent node is Red in color, and there is no parent's sibling exists in the tree, so rule 4a would be applied. The first rotation would be performed. The RR relationship exists between the nodes, so left rotation would be performed.



When left rotation is performed, node 40 will come upwards, and node 30 will come downwards, as shown in the below figure:



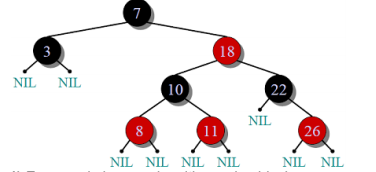
After rotation, the recoloring is performed on nodes 30 and 40. The color of node 30 would become Red, while the color of node 40 would become black.



The above tree is a Red-Black tree as it follows all the Red-Black tree properties.

2.**Discuss the various cases for insertion of key in red-black tree for given sequence of key in an empty red-black tree- {15,13,12,16,19,23,5,8}. Also show that a red-black tree with n internal nodes has height at most 2lg(n+1).**

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



**Red Black Tree Insertion Rules**

1-If tree is empty, create new node as root node with color black

2-If tree is not empty, create new node as leaf node with color red

3-If parent of new node is black then exit

4-If parent of new node is red, then check the color of parents sibling of new node

a.  If color is black or null then do suitable rotation & recolor

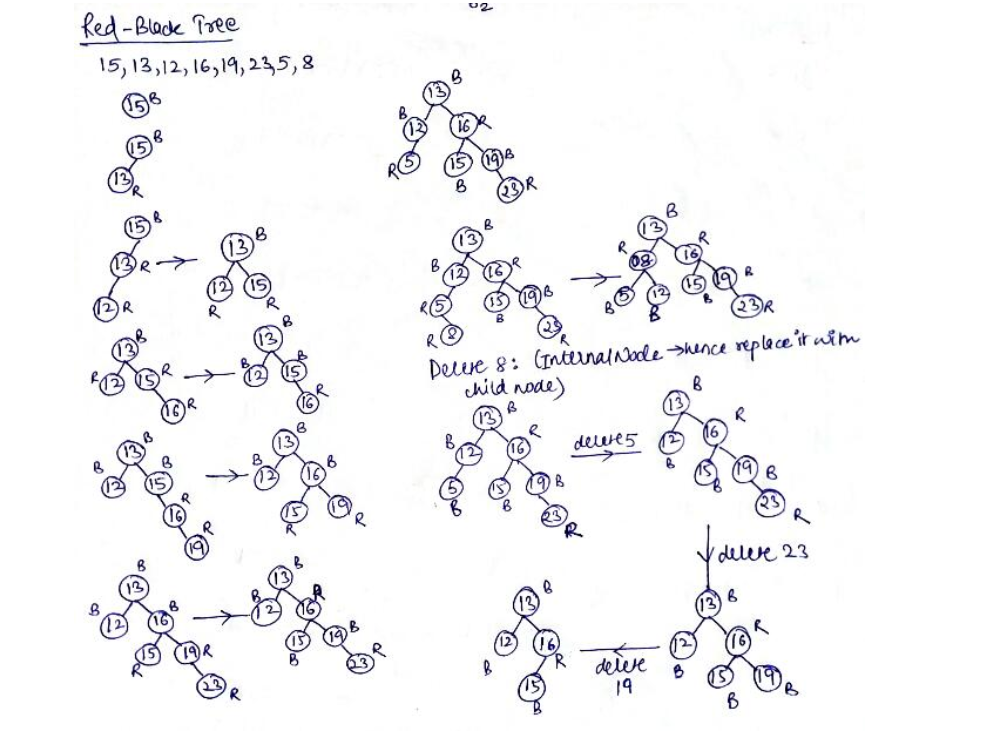
b.  If color is red then recolor & also check if parents parent of new node is not root node then recolor it & recheck

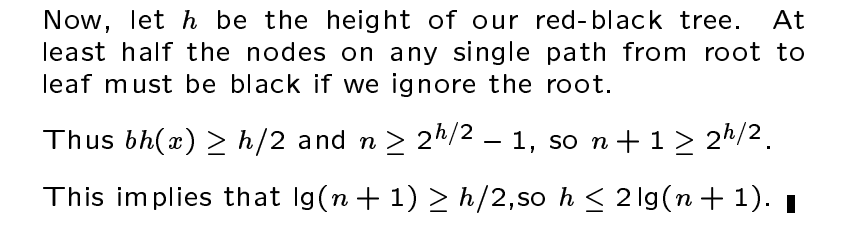
**Insertion:**

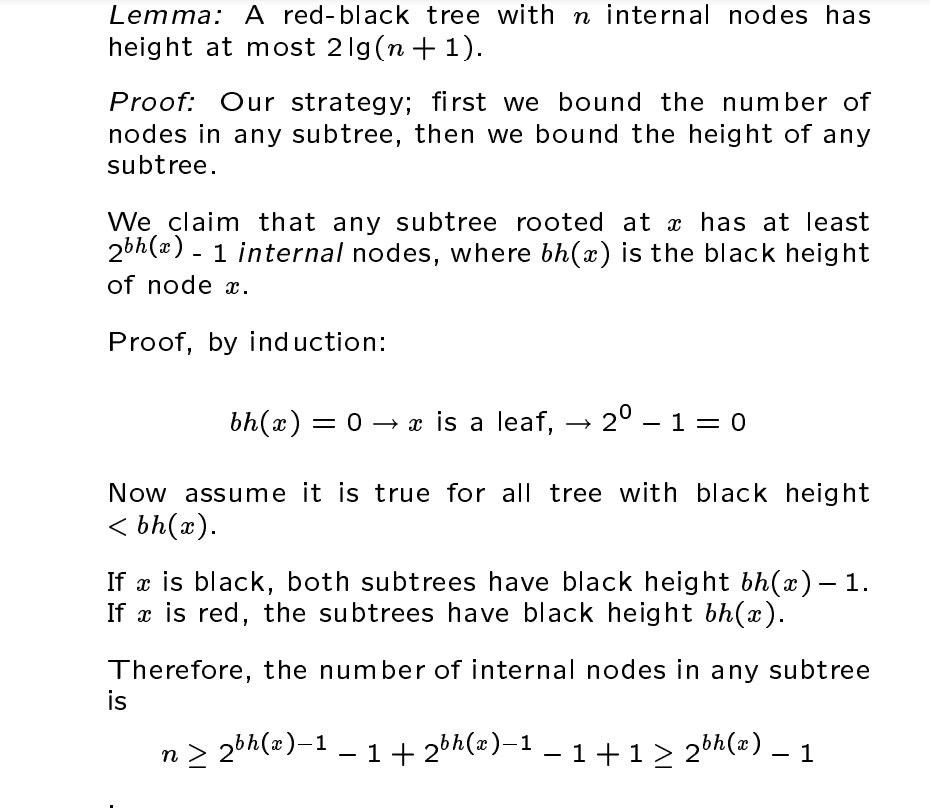
1. Insert the new node the way it is done in Binary Search Trees.
2. Color the node red
3. If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.
4. In Red black tree if imbalancing occurs then for removing it two methods are used that are:

a) Recoloring

b) Rotation





**2. 3. Explain insertion in Red-Black tree. Show steps for inserting 1,2,3,4,5,6,7,8&9 into empty RB tree.**

A **red–black tree** is a kind of **self-balancing binary search tree** in computer science. 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.

**To add an element to a Red Black Tree, we must follow this algorithm:**

1) Check whether tree is Empty.

2) If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.

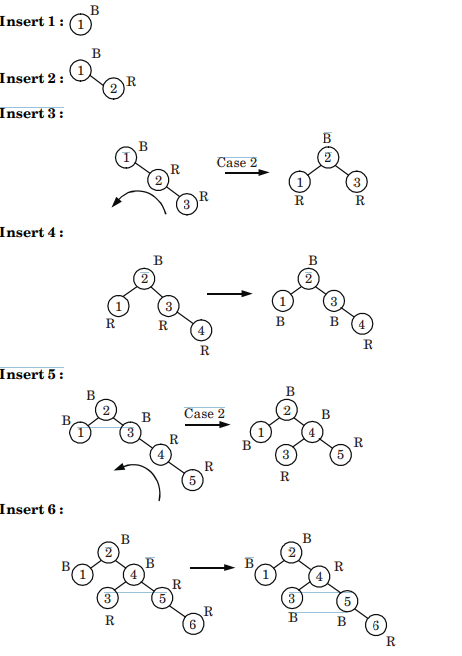
3) If tree is not Empty then insert the newNode as a leaf node with Red color.

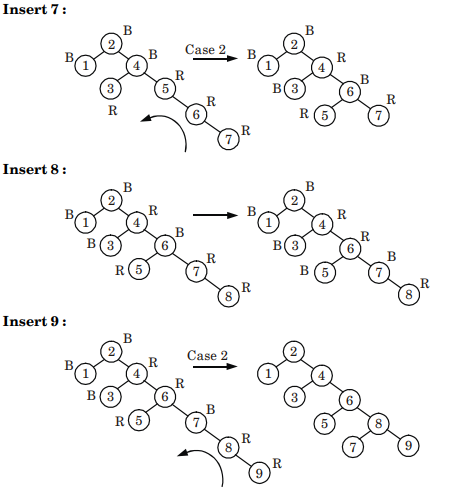
4) If the parent of newNode is Black then exit from the operation.

5) If the parent of newNode is Red then check the color of parent node's sibling of newNode.

6) If it is Black or NULL node then make a suitable Rotation and Recolor it.

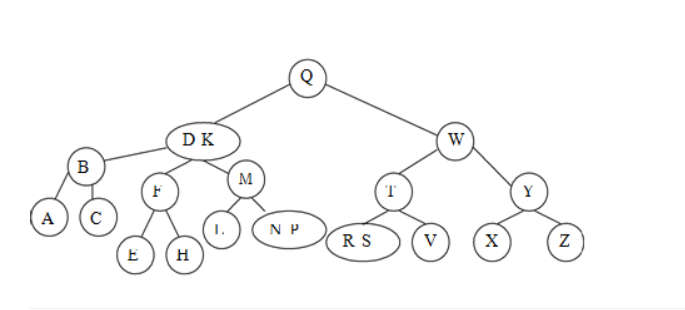
7) If it is Red colored node then perform Recolor and Recheck it. Repeat the same until tree becomes Red Black Tree.





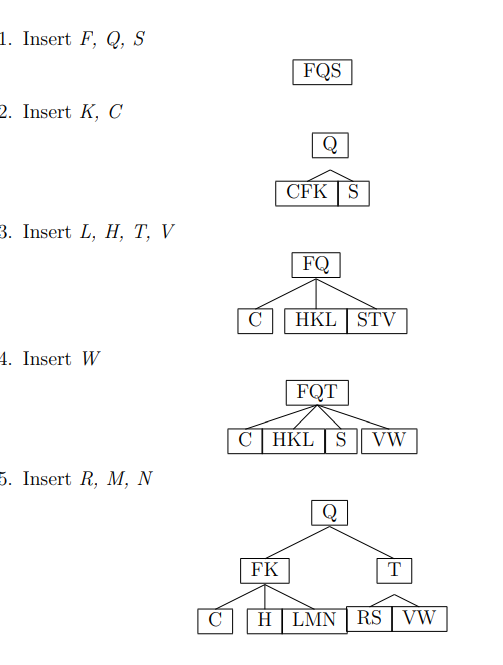
4.**Insert the elements 8, 20, 11, 14, 9, 4, 12 in a Red-Black Tree and delete 12, 4, 9, 14 respectively.**

Show the results of inserting the keys F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E in order into an empty B-tree. Use t=3, where t is the minimum degree of B- tree.



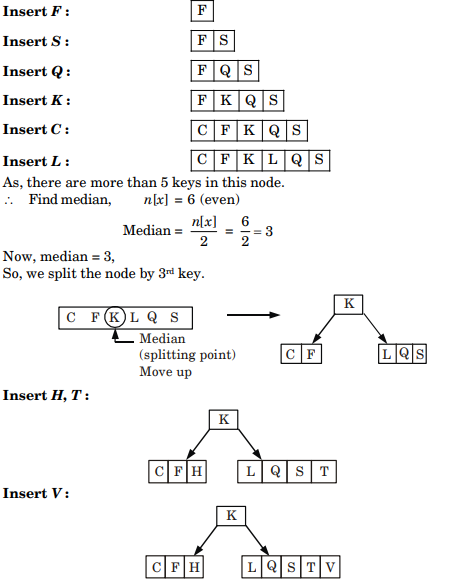
5.**How B-tree differs with other tree structures. Insert the following information F,S,Q,K,C,L,H,T,V,W,M,R,N into an empty B-tree with minimum degree 2.**

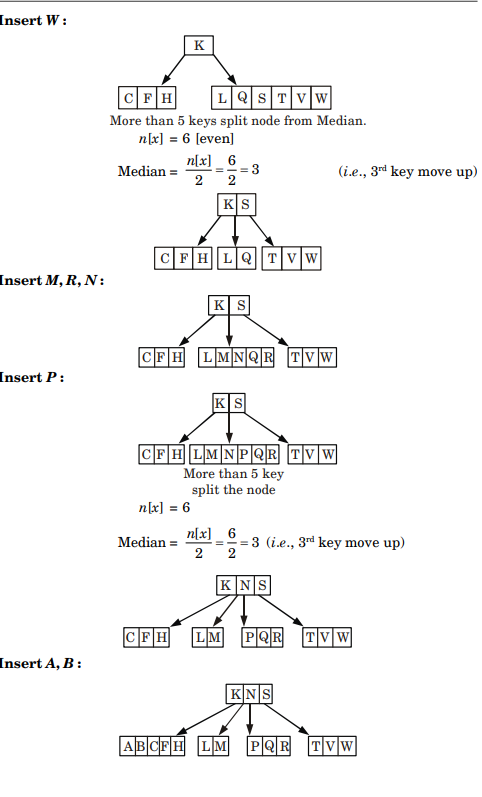
B-Tree : B-Tree is known as a self-balancing tree as its nodes are sorted in the inorder traversal. Unlike the binary trees, in B-tree, a node can have more than two children. B-tree has a height of logM N (Where 'M' is the order of tree and N is the number of nodes). The B-tree generalizes the binary search tree, allowing for nodes with more than two children. Unlike other self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as databases and file systems.

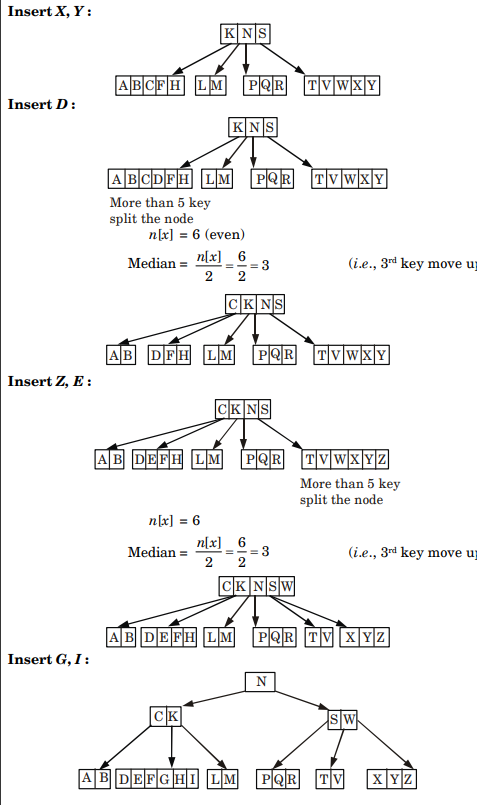
****

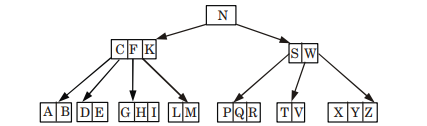
**6.Show the results of inserting the keys F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E in order into an empty B-tree. Use t=3, where t is the minimum degree of B- tree.**

Assume that t = 3 2t – 1 = 2 × 3 – 1 = 6 – 1 = 5 and t – 1 = 3 – 1 = 2 So, maximum of 5 keys and minimum of 2 keys can be inserted in a node. Now, apply insertion process as

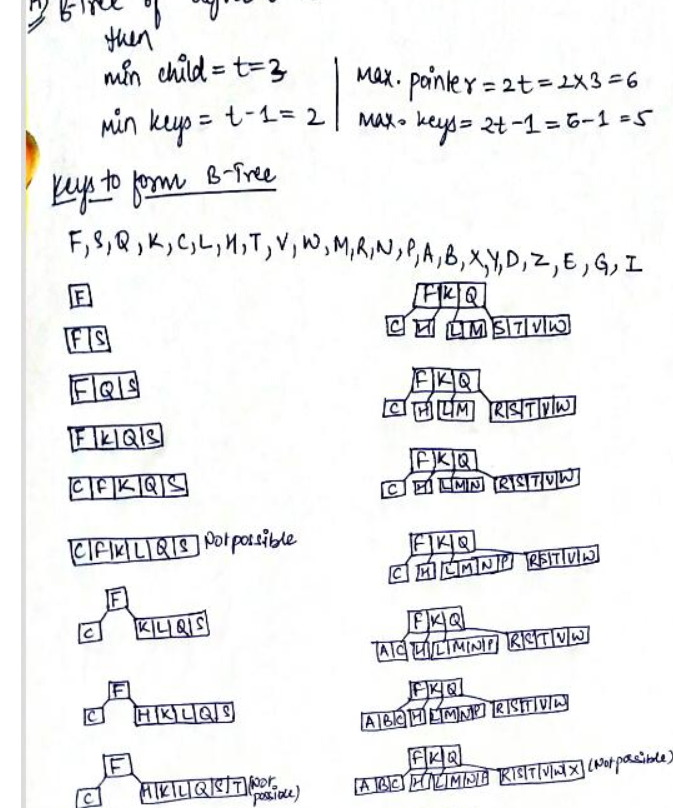


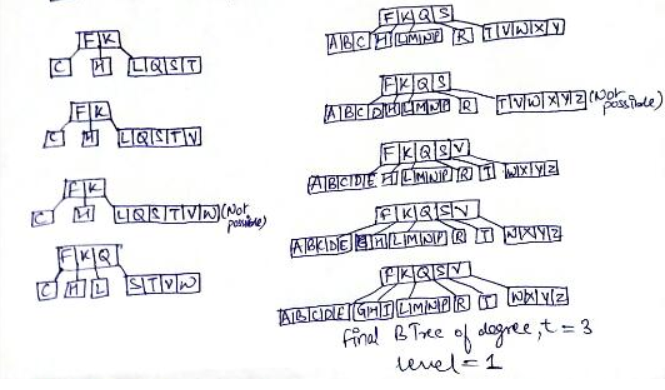






**7.Insert the following information F,S,Q,K,C,L,H,T,V,W,M,R,N,P,A,B,X,Y,D,Z,E,G,I. Into an empty R-tree with degree t=3.**

****

****

**8.Discuss the various cases for insertion of key in red-black tree for given sequence of key in an empty red-black tree- 5, 16, 22, 25, 2, 10, 18, 30, 50, 12, 1.**

**9.Insert the following element in an initially empty RB-Tree. 12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now Delete 23 and 81.**

**10.Define a B-Tree of order m. Explain the searching operation in a B-Tree.**

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

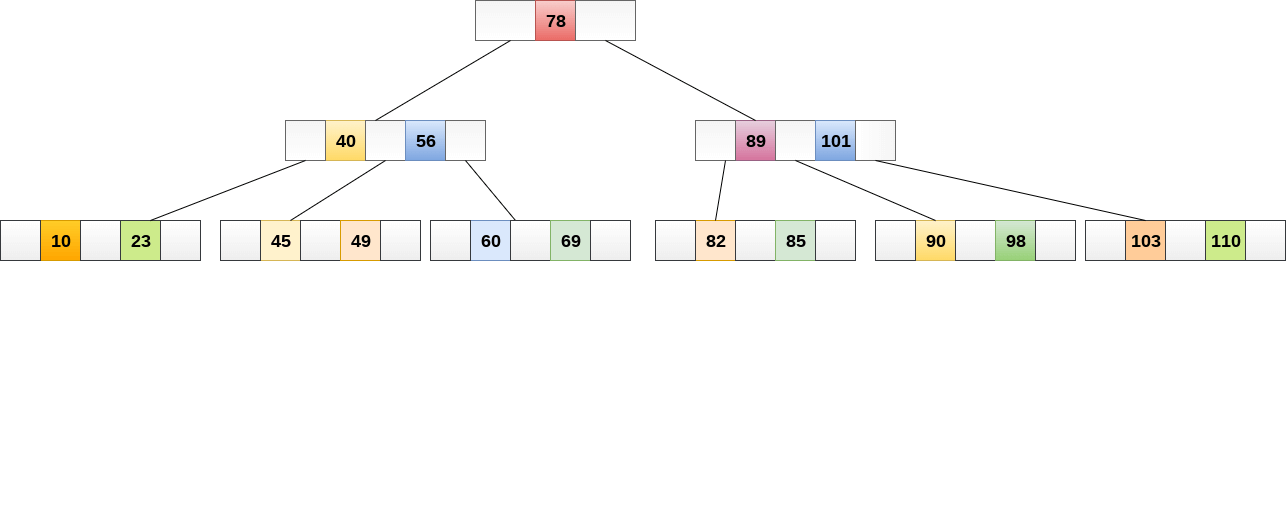
1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

**Searching Operations:**

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



**11.Discuss the advantages of using B-Tree. Insert the following Information 86, 23, 91, 4, 67, 18, 32, 54, 46, 96, 45 into an empty B-Tree with degree t=2 and delete 18, 23 from it.**

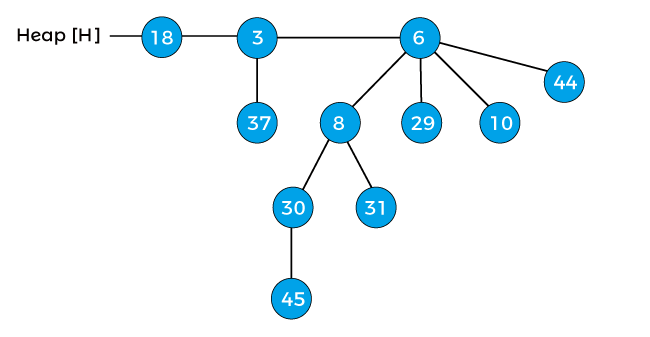
**12.What is Binomial Heap? Write down the algorithm for Decrease key operation in Binomial Heap also write its time complexity.**

## A binomial heap can be defined as the collection of binomial trees that satisfies the heap properties, i.e., min-heap. The min-heap is a heap in which each node has a value lesser than the value of its child nodes. Mainly, Binomial heap is used to implement a priority queue. It is an extension of binary heap that gives faster merge or union operations along with other operations provided by binary heap.

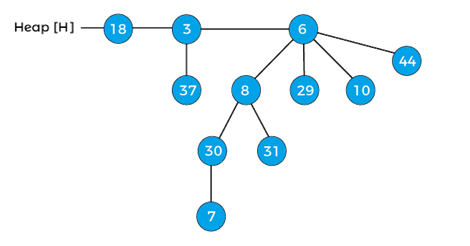
## **Decreasing a key**

## Now, let's move forward to another operation to be performed on binomial heap. Once the value of the key is decreased, it might be smaller than its parent's key that results in the violation of min-heap property. If such case occurs after decreasing the key, then exchange the element with its parent, grandparent, and so on until the min-heap property is satisfied.

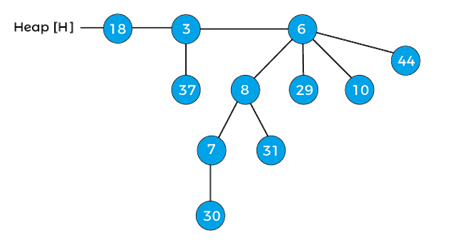
Let's understand the process of decreasing a key in a binomial heap using an example. Consider a heap given below -



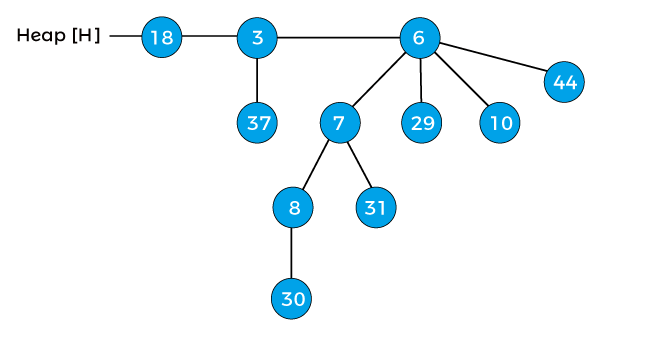
Decrease the key 45 by 7 of the above heap. After decreasing 45 by 7, the heap will be -



After decreasing the key, the min-heap property of the above heap is violated. Now, compare 7 wits its parent 30, as it is lesser than the parent, swap 7 with 30, and after swapping, the heap will be -



Again compare the element 7 with its parent 8, again it is lesser than the parent, so swap the element 7 with its parent 8, after swapping the heap will be -



Now, the min-heap property of the above heap is satisfied. So, the above heap is the final heap after decreasing a key.

**Time Complexity:**

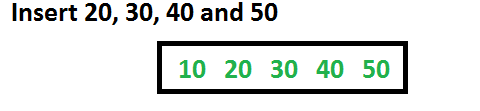
Decreases the value of the key. The time complexity of this operation is O(log N). If the decreased key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

**13.Using minimum degree ‘t’ as 3, insert following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90 in an initially empty B-Tree. Give the number of nodes splitting operations that take place.**

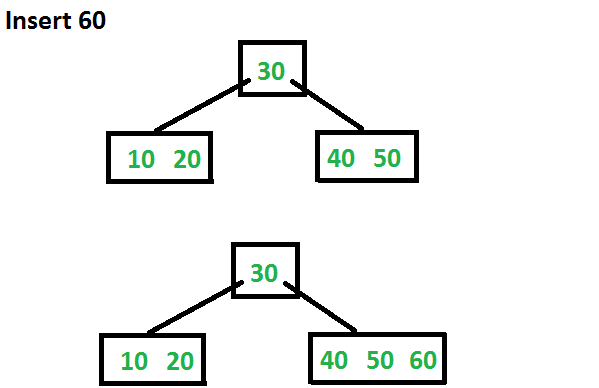
1.tree of minimum degree ‘t’ as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.  
Initially root is NULL. Let us first insert 10.



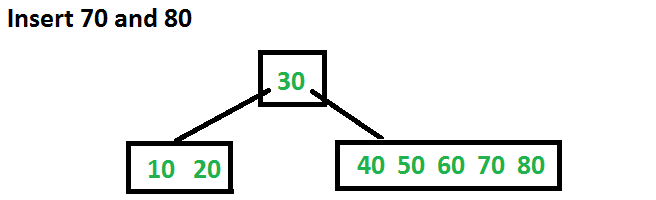
2.Let us now insert 20, 30, 40 and 50. They all will be inserted in root because the maximum number of keys a node can accommodate is 2\*t – 1 which is 5.



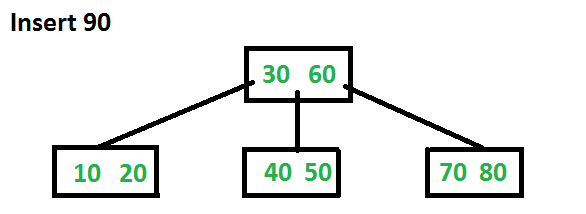
3.Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child. 



4.Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.



5.Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.



**14.Insert the following keys in a *2-3-4 B Tree*: 40, 35, 22, 90, 12, 45, 58, 78, 67, 60 and then delete key 35 and 22 one after other**

**15.Explain the algorithm to delete a given element in a binomial Heap. Give an example for the same.**

A binomial heap can be defined as the collection of binomial trees that satisfies the heap properties, i.e., min-heap. The min-heap is a heap in which each node has a value lesser than the value of its child nodes. Mainly, Binomial heap is used to implement a priority queue. It is an extension of binary heap that gives faster merge or union operations along with other operations provided by binary heap.

### Properties of Binomial heap

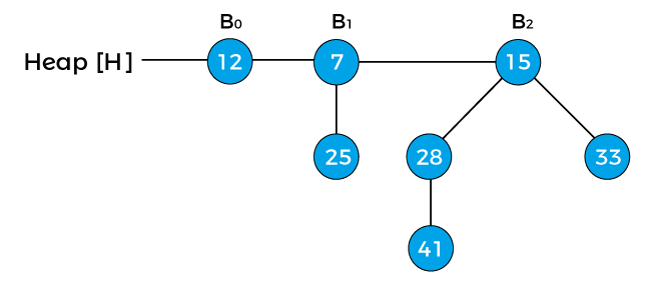
There are following properties for a binomial heap with **n** nodes -

* Every binomial tree in the heap must follow the **min-heap** property, i.e., the key of a node is greater than or equal to the key of its parent.
* For any non-negative integer k, there should be atleast one binomial tree in a heap where root has degree k.

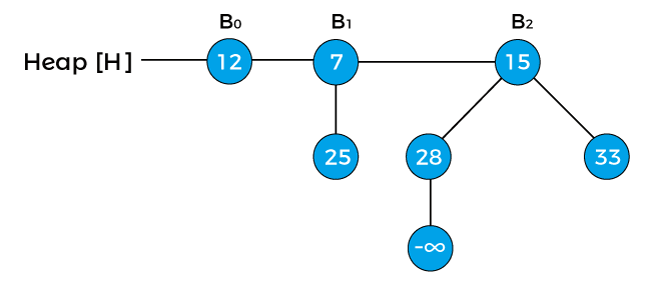
The first property of the heap ensures that the min-heap property is hold throughout the heap. Whereas the second property listed above ensures that a binary tree with **n** nodes should have at most **1 + log2 n** binomial trees, here **log2** is the binary logarithm.

## **Deleting a node from the heap**

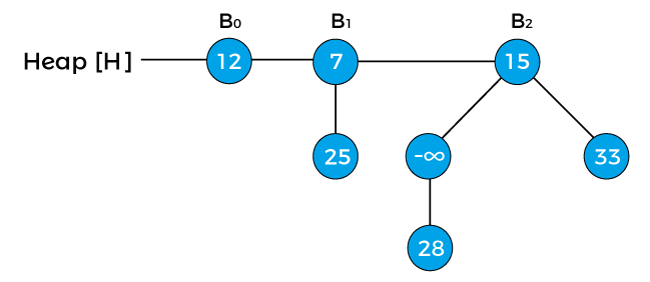
To delete a node from the heap, first, we have to decrease its key to negative infinity (or -∞) and then delete the minimum in the heap. Now we will see how to delete a node with the help of an example. Consider the below heap, and suppose we have to delete the node 41 from the heap -



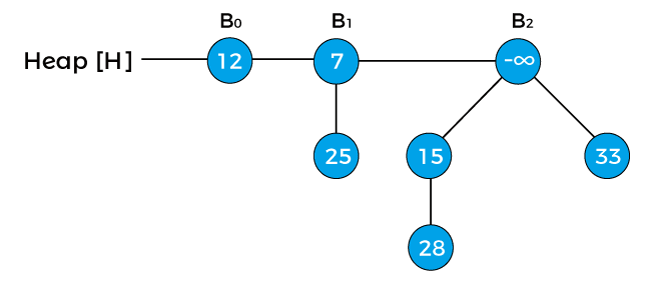
First, replace the node with negative infinity (or -∞) as shown below -



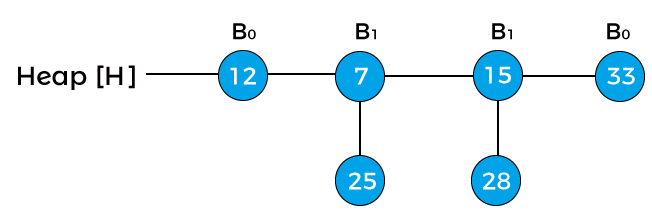
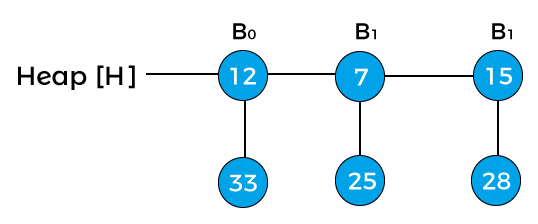
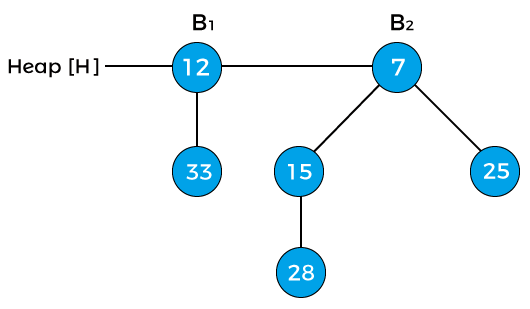
Now, swap the negative infinity with its root node in order to maintain the min-heap property.



Now, again swap the negative infinity with its root node.



The next step is to extract the minimum key from the heap. Since the minimum key in the above heap is -infinity so we will extract this key, and the heap would be:

The above is the final heap after deleting the node 4

**TIME COMPLEXITY:**

**Deleting a node : O(log n)**

**16.Explain and write an algorithm for union of two binomial heaps and write its time complexity.**

It is the most important operation performed on the binomial heap. Merging in a heap can be done by comparing the keys at the roots of two trees, and the root node with the larger key will become the child of the root with a smaller key than the other. The time complexity for finding a union is O(logn). The function to merge the two trees is given as follows -

1. function merge(a,b)
2. if a.root.key ? b.root.key
3. return a.add(b)
4. else
5. return b.add(a)

To perform the union of two binomial heaps, we have to consider the below cases:

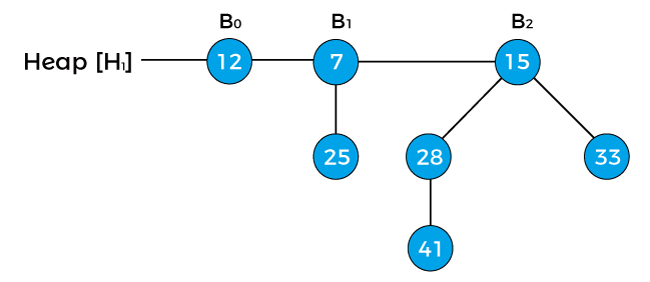
**Case 1:** If degree[x] is not equal to degree[next x], then move pointer ahead.

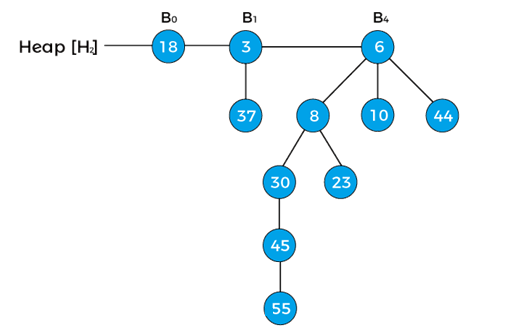
**Case 2:** if degree[x] = degree[next x] = degree[sibling(next x)] then,Move the pointer ahead.

**Case 3:** If degree[x] = degree[next x] but not equal to degree[sibling[next x]] and key[x] < key[next x] then remove [next x] from root and attached to x.

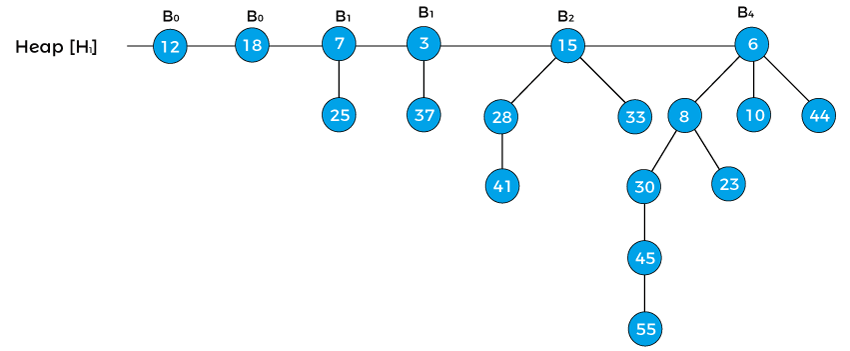
**Case 4:** If degree[x] = degree[next x] but not equal to degree[sibling[next x]] and key[x] > key[next x] then remove x from root and attached to [next

example. Consider two binomial heaps –





We can see that there are two binomial heaps, so, first, we have to combine both heaps. To combine the heaps, first, we need to arrange their binomial trees in increasing order.



In the above heap first, the pointer x points to the node 12 with degree B0, and the pointer next[x] points the node 18 with degree B0. Node 7 with degree B1 is the sibling of 18, therefore, it is represented as sibling[next[x]].

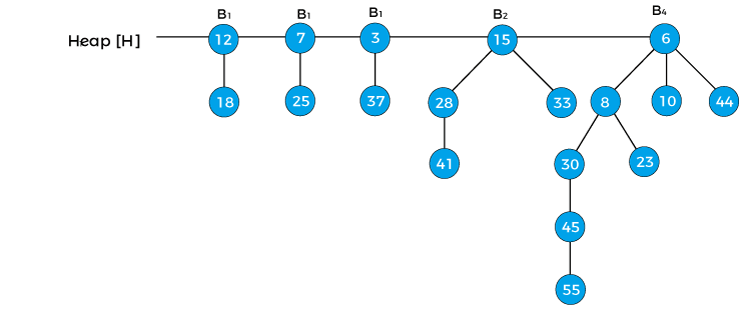
Now, first apply Case1 that says **'if degree[x] ≠ degree[next x] then move pointer ahead'** but in the above example, the degree[x] = degree[next[x]], so this case is not valid.

Now, apply Case2 that says **'if degree[x] = degree[next x] = degree[sibling(next x)] then Move pointer ahead'.** So, this case is also not applied in the above heap.

Now, apply Case3 that says **' If degree[x] = degree[next x] ≠ degree[sibling[next x]] and key[x] < key[next x] then remove [next x] from root and attached to x'.** We will apply this case because the above heap follows the conditions of case 3 -

degree[x] = degree[next x] ≠ degree[sibling[next x]] {as, B0 = B0¬ ≠ B1} and key[x] < key[next x] {as 12 < 18}.

So, remove the node 18 and attach it to 12 as shown below -



**x = 12, next[x] = 7, sibling[next[x]] = 3, and degree[x] = B1, dgree[next[x]] = B1, degree[sibling[next[x]]] = B1**

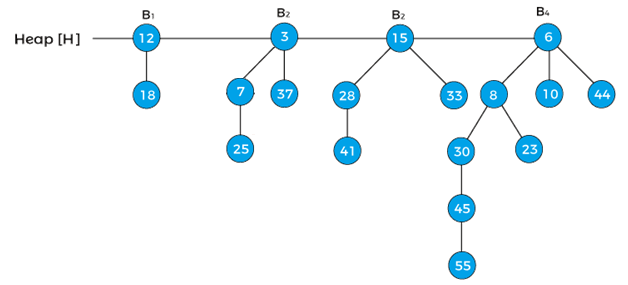
Now we will reapply the cases in the above binomial heap. First, we will apply case 1. Since x is pointing to node 12 and next[x] is pointing to node 7, the degree of x is equal to the degree of next x; therefore, case 1 is not valid.

Here, case 2 is valid as the degree of x, next[x], and sibling[next[x]] is equal. So, according to the case, we have to move the pointer ahead.

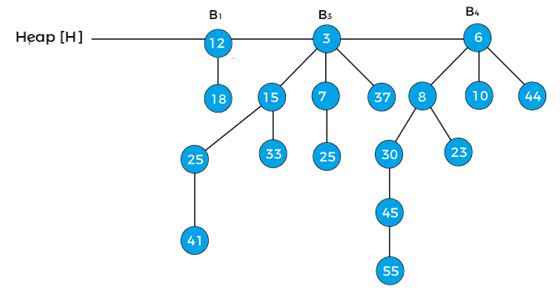
Therefore, **x = 7, next[x] = 3, sibling[next[x]] = 15,** and **degree[x] = B1, dgree[next[x]] = B1, degree[sibling[next[x]]] = B2**

Now, let's try to apply case 3, here, first condition of case3 is satisfied as degree[x] = degree[next[x]] ≠ degree[sibling[next[x]]], but second condition (key[x] < key[next x]) of case 3 is not satisfied.

Now, let's try to apply case 4. So, first condition of case4 is satisfied and second condition (key[x] > key[next x]) is also satisfied. Therefore, remove x from the root and attach it to [next[x]].



Now, the pointer x points to node 3, next[x] points to node 15, and sibling[next[x]] points to the node 6. Since, the degree of x is equal to the degree of next[x] but not equal to the degree[sibling[next[x]]], and the key value of x is less than the key value of next[x], so we have to remove next[x] and attach it to x as shown below -



Now, x represents to the node 3, and next[x] points to node 6. Since, the degree of x and next[x] is not equal, so case1 is valid. Therefore, move the pointer ahead. Now, the pointer x points the node 6. The B4 is the last binomial tree in a heap, so it leads to the termination of the loop. The above tree is the final tree after the union of two binomial heaps.

**Time Complexity:**

binomial heap is the collection of binomial trees, and every binomial tree satisfies the min-heap property. It means that the root node contains a minimum value. Therefore, we only have to compare the root node of all the binomial trees to find the minimum key. The time complexity of finding the minimum key in binomial heap is **O(logn).**

**17.Explain the algorithm to extract the minimum elements in a binomial Heap. Give an example for the same.**

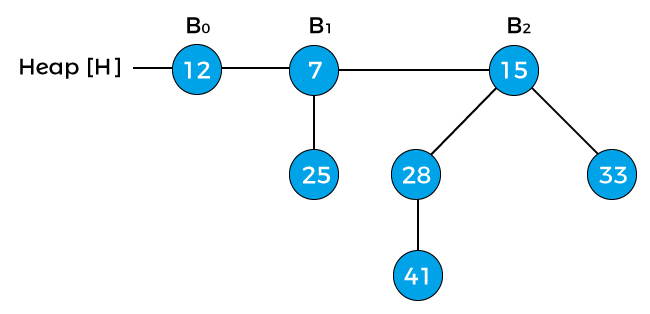
*A Binomial Heap is a collection of Binomial Trees* The main operation in Binomial Heap is a union(), all other operations mainly use this operation.

**Extracting(H):** This operation also uses a union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally, we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.

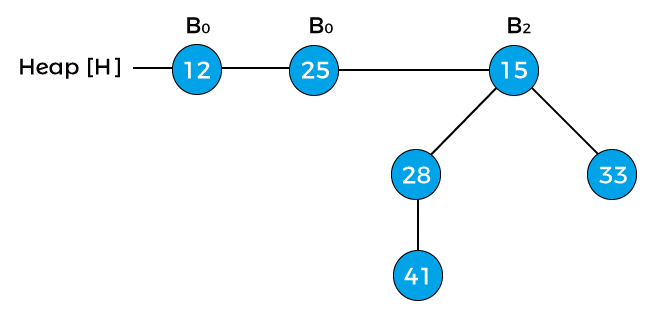
## **Extracting the minimum key**

It means that we have to remove an element with the minimum key value. As we know, in min-heap, the root element contains the minimum key value. So, we have to compare the key value of the root node of all the binomial trees. Let's see an example of extracting the minimum key from the heap.

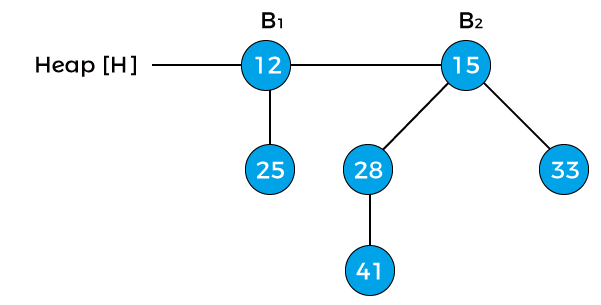
Suppose the heap is -



Now, compare the key values of the root node of the binomial trees in the above heap. So, 12, 7, and 15 are the key values of the root node in the above heap in which 7 is minimum; therefore, remove node 7 from the tree as shown in the below image -



Now, the degree of node 12 and node 25 is B0, and the degree of node 15 is B2. Pointer x points to the node 12, next(x) points to the node 25, and sibling(next(x)) points to the node 15. Since the degree of x is equal to the degree of next(x) but not equal to the degree of sibling(next(x)). Value of pointer x is less than the pointer next(x), so node 25 will be removed and attached to node 12 as shown in the below image -



Now, the degree of node 12 is changed to B1. The above heap is the final heap after extracting the minimum key.

**18.What are the various differences in Binomial and Fibonacci Heap? Explain.**

| **Feature** | **Binomial Heap** | **Fibonacci Heap** |
| --- | --- | --- |
| Structure | Collection of binomial trees | Collection of min-heap-ordered trees |
| Insertion | O(log n) | O(1) amortized |
| Deletion | O(log n) | O(log n) |
| Decrease-key | O(log n) | O(1) amortized |
| Merging | O(log n) | O(1) amortized |
| Space Efficiency | Requires more space due to binomial trees | Requires more space due to additional properties |
| Key Advantage | Efficient decrease-key operations | Extremely fast insertion and merging operations |
| Applications | Priority queues, graph algorithms | Advanced algorithms, specifically decrease-key ops |

**19.What is Fibonacci Heap? Discuss the application of Fibonacci Heap.**

A Fibonacci heap is a data structure used for implementing priority queues. It is a type of heap data structure, but with several improvements over the traditional binary heap and binomial heap data structures.The key advantage of a Fibonacci heap over other heap data structures is its fast amortized running time for operations such as insert, merge and extract-min, making it one of the most efficient data structures for these operations. The running time of these operations in a Fibonacci heap is O(1) for insert, O(log n) for extract-min and O(1) amortized for merge.A Fibonacci heap is a collection of trees, where each tree is a heap-ordered multi-tree, meaning that each tree has a single root node with its children arranged in a heap-ordered manner. The trees in a Fibonacci heap are organized in such a way that the root node with the smallest key is always at the front of the list of trees.

In a Fibonacci heap, when a new element is inserted, it is added as a singleton tree. When two heaps are merged, the root list of one heap is simply appended to the root list of the other heap. When the extract-min operation is performed, the tree with the minimum root node is removed from the root list and its children are added to the root list.One unique feature of a Fibonacci heap is the use of lazy consolidation, which is a technique for improving the efficiency of the merge operation. In lazy consolidation, the merging of trees is postponed until it is necessary, rather than performed immediately. This allows for the merging of trees to be performed more efficiently in batches, rather than one at a time.

**Application:**

Fibonacci heaps are used to implement the priority queue element in Dijkstra's algorithm, giving the algorithm a very efficient running time. Fibonacci heaps have a faster amortized running time than other heap types. Fibonacci heaps are similar to binomial heaps but Fibonacci heaps have a less rigid structure.

**20.What is skip list? Explain the Search operation in Skip list with suitable example also write its algorithm.**

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

## **Skip List Basic Operations**

There are the following types of operations in the skip list.

**Insertion operation:** It is used to add a new node to a particular location in a specific situation.

**Deletion operation:** It is used to delete a node in a specific situation.

**Search Operation:** The search operation is used to search a particular node in a skip list.

**Algorithm of searching operation**

1. Searching (L, SKey)
2. a = L → header
3. loop invariant: a → key level down to 0 **do**.
4. **while** a → forward[i] → key forward[i]
5. a = a → forward[0]
6. **if** a → key = SKey then **return** a → value
7. **else** **return** failure

Example where we want to search for key 17.

