Unit-2

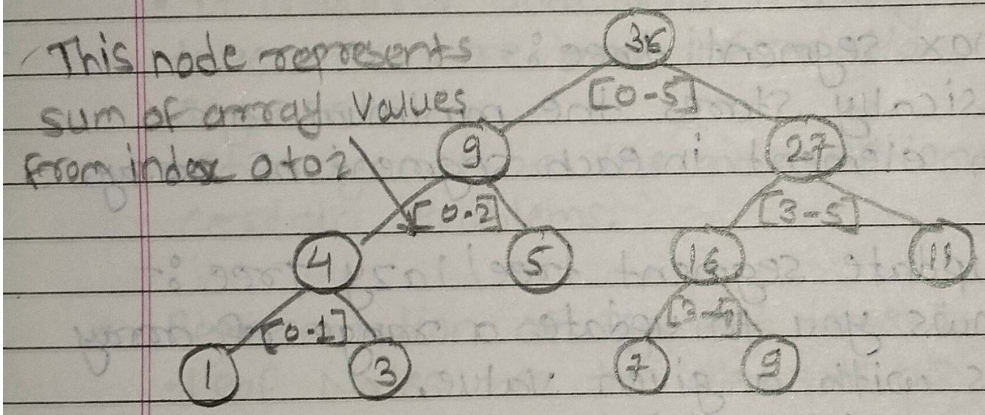
Advanced Trees

Page No.
Date

Que- Explain segment tree with example, advantages, Disadvantages and application.

⇒

Segment tree :-
- A segment tree is a data structure used to effectively query and update ranges of array members.
- It's typically implemented as a binary tree, with each node representing a segment or range of array element.



segment Tree for input array {1, 3, 5, 7, 9, 11}

Characteristics of segment Tree :-
- A segment tree is a binary tree with a leaf node for each element in the array.
- Each internal node represents a segment or a range of elements in the array.
- The root node represents the entire array.
- Each node stores information about the segment it represents, such as the sum or minimum of the elements in the segment.

- Types of segment tree :-
  - Based on the type of information stored in the nodes, the tree can be divided into the following few types :-

  1] Sum segment tree :-
  - It stores the sum of elements in each segment of the array and is used in a problem where efficient answer queries are needed about the sum of elements.

  2] min/max segment tree :-
  - It basically stores the minimum or maximum element in each segment of array.

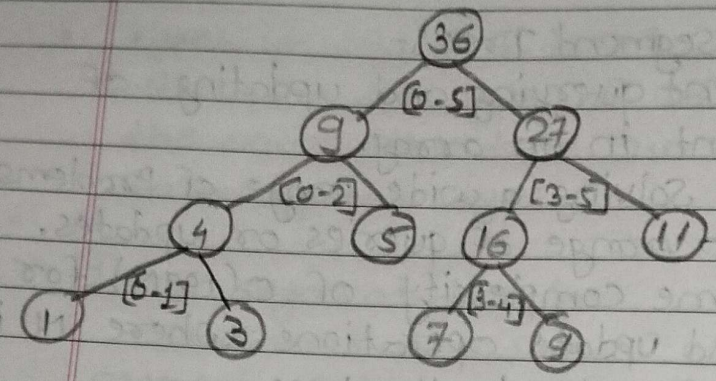
  3] Range update segment tree/ Lazy Tree :-
  - It allows you to update a range of array elements with a given value.
  - It is used to efficiently update a range of array element.

- Applications of segment tree :-
  - In finding the sum or minimum or maximum of a range of elements in an array.
  - In finding the number of elements in a range that satisfy a certain condition, such as being greater than a certain value.
  - It can be used in several problems of computational geometry.

- Advantages of segment Tree:-
  - It allow efficient querying and updating of ranges of element in an array.
  - It can help in solving a wide range of problems that involve a range of queries or updates.
  - It has a time complexity of O(logN) for both query and update operations where N is the number of nodes in the tree.

- Disadvantages of segment Tree:-
  - It requires additional memory to store the tree structure and information about the segments.
  - It may not be the most optional solution for certain problems.

- Representation of segment trees:-
  - Leaf Nodes are the elements of the input array.
  - Each internal node represents some merging of the leaf nodes.
    - The merging may be different for different problems. For this problem, merging is sum of leaf nodes under a node.
  - An array representation of tree is used to represent segment Trees. For each node at index i, the left child is at index (2*i+1), right child at (2*i+2) and the parent is at (L(i-1)/2J)



segment Tree = {1, 3, 5, 7, 9, 11}

- This structure enables fast query and update operations with time complexity of O(log n).
- A segment Tree is a versatile data structure used in computer science and data structure that allows efficient querying and updating of intervals or segments of an array.
- The divide and conquer strategy here dictate that during traversal along some path from top to bottom in the tree, at each level the sample space is halved.

- Implementation of segment Tree :-
  The 'buildTree()' function builds a segment tree using the array 'v' and stores it into the array 'tree'.
  - At each level of recursion, the range is divided into two halves.
  - Then recursively the left and right children nodes are built for the first half and the other half of the range respectively.

Page No.
Date

- The current node will then hold the sum of its children's values. The base condition is the case of a leaf, which is when the size of the range is one.

Suppose an array arr = [1,2,3]. It will be stored in the tree in 3 layers of nodes / blocks:

[[3+3=6]]
[[1+2=3], [3]]
[[1], [2], [3]]

Page No.
Date

Que- Explain Binary Indexed Tree with example, Advantages, disadvantages and application.

⇒

Binary Indexed Tree :-

- Binary indexed Tree is a data structure that is used to efficiently calculate the prefix sum of a series (array) of numbers.
- It is also known as the Fenwick tree as "Peter Fenwick" introduced it to the world through his paper.

• Introduction to Fenwick Tree :-

- When answering to multiple prefix sum (sum of array elements from 0 to an index x i.e. a[0] + a[1] + ... + a[x] queries of an array, the first idea that comes to our mind is to use a prefix_sum array.
- For Example :-

arr = | 3 | 1 | 5 | -1 | 8 | 2 |

Prefix_sum = | 3 | 4 | 9 | 8 | 16 | 18 |

- A binary indexed tree popularly known as the Fenwick tree is a data structure that maintains the cumulative frequencies of the array elements at each of its nodes.
- one of the best and simple use cases can be calculating the prefix sum of an array in which values are mutable (i.e. values can be changed) logarithmic time complexity.

Page No.
Date

- Basic idea of Fenwick Tree:-
  - The basic idea behind Binary Indexed Tree is that we can store the sum of a particular range of elements at an index of the newly formed bit array.
  - which will be used later to calculate Prefix-sum in the fewer number of steps.
  - The idea that "every integer can be represented as the sum of powers of 2" is used extensively in the formation of the Fenwick tree.
  - using this fact we can calculate the - Prefix\-sum- of an array in a logarithmic time complexity.

- What is Fenwick Tree?
  - Fenwick Tree or Binary Indexed Tree is a data structure used to calculate range queries along with updating the elements of the array, such that each query or update takes logarithmic time complexity. It is used to calculate Prefix sums or running total of values up to any index.

- Construct the Fenwick tree?
  - To construct a Fenwick tree in linear time, we can follow these steps:

1) Initialize the Fenwick tree with all values set to 0.
2) Iterate through each element in the original array from left to right.
3) For each element, add its value to the next

immediate cell in the tree that is responsible for the current element. This cell is located at the index i + g(i), where i is the index of the current element and g(i) is the range of responsibility of i.

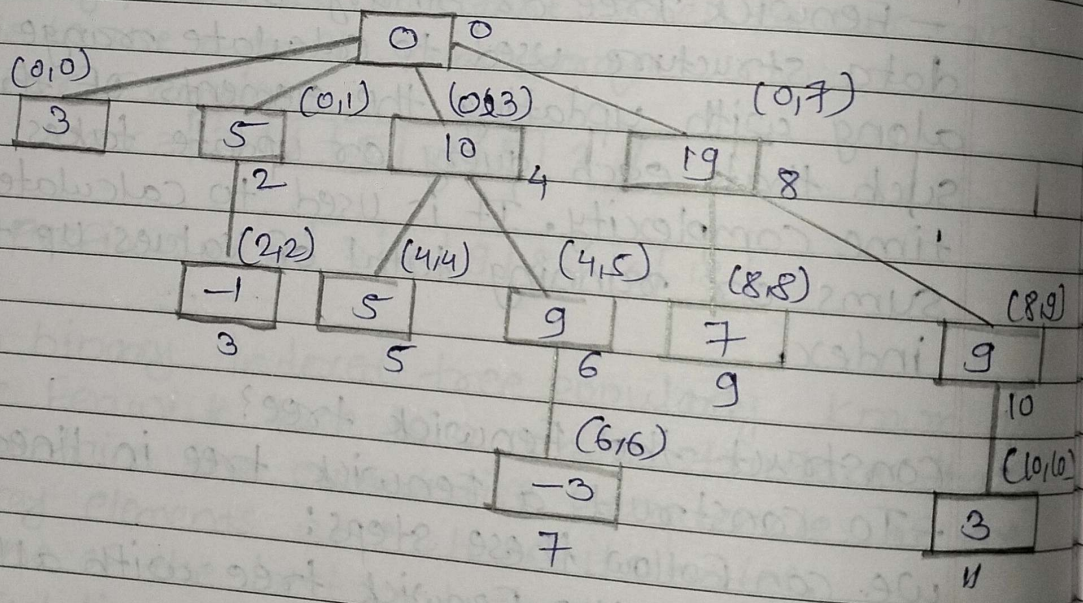
4) If the parent cell (i + g(i)) is out of bounds, ignore this update.

- Example:-

~~3 5 4 10 15 19 16 19 2~~

Arr	3	2	-1	6	5	4	-3	3	7	2	3

Sum Arr	3	5	4	10	15	19	16	19	20	28	31



Page No.
Date

- operations of Fenwick Tree :-
- The two main operations supported by Fewick tree are -

1] getSum(x) - This operation returns the sum of first x elements of the array i.e.
a[0] + a[1] + --- + a[x]

2] update(x, val) - This operations updates the value of a at index x i.e. a[x] = val.

1] get sum(x) operation :-
- getsum(x) operation returns sum of array elements from 0th index to x^{th} index i.e. a[0] + a[1] + --- a[x].
- steps involved to calculate it are :-
a] Initialize answer, ans with 0.
b] Increase x by 1 (To maintain 1-based indexing)
c] while x is greater than 0
  - ans += bit[x]
  - x -= (x & (-x))
d] Return ans.

2] Update(x, val) operation :-
- update(x, val) operation updates the value of array a at index x to val.
- After updating, we also need to update bit array at those indices which get impacted by the element at index x.

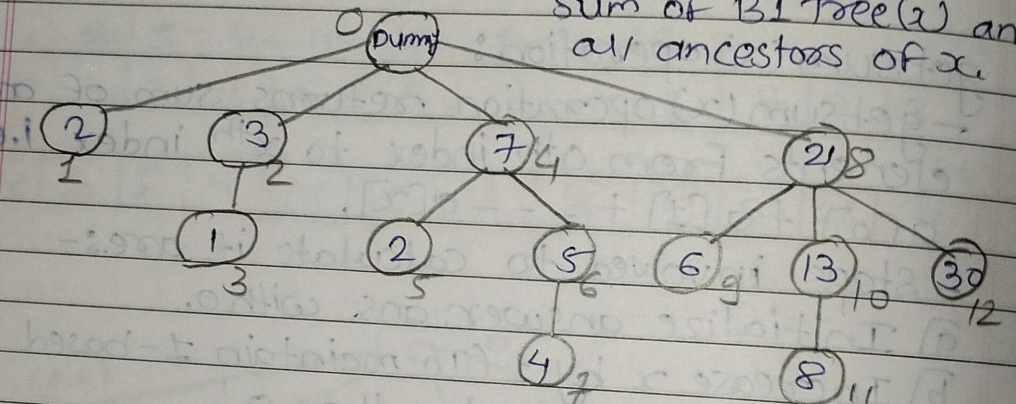
• steps involved in that are -

a) Find the change happening at the index x,
   δ val = Val - a[x]

b) update the value at index x i.e. a[x] = val,

c) while x ≤ n
   - bit[x] += δ val
   - x += (x & (-x))

• getsum(x) operation :- In getsum(x), we return sum of BITree(x) and all ancestors of x.

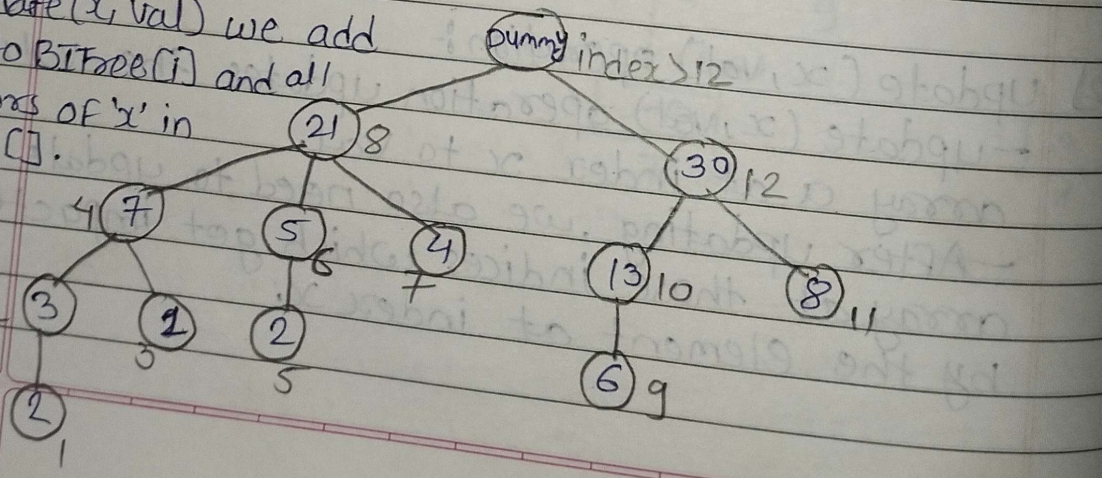


Input Array: {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9}
0 1 2 3 4 5 6 7 8 9 10 11

BI Tree Array: {2, 3, 1, 7, 2, 5, 4, 21, 6, 13, 8, 30}
1 2 3 4 5 6 7 8 9 10 11 12

• update(x, val) :-

In update(x, val) we add 'val' to BITree[i] and all ancestors of 'x' in BITree[].



Page No.
Date

- Advantages :-

1) Efficient Range queries :- BIT allows for quick calculation of cumulative sum queries within a given range in O(logn) time complexity.

2) Efficient updates :-
-updates to elements in the array can be done efficiently in O(logn) time complexity.

3) Space Efficiency :- BIT requires relatively low memory overhead compared to other data structures like segment trees.

4) Simplicity :- The implementation of BIT is simpler compared to segment trees, making it easier to understand and code.

- Disadvantages :-

1) Limited Functionality :- BIT supports only certain types of range queries (cumulative sum being the most common).
2) update complexity :- Although updates are relatively efficient, they require understanding of bitwise operations, which can be challenging for some programmers.
3) Readability :- While BIT implementations are simpler than segment trees, they may still be less intutive to understand for those ufamiliar with the concept.

Applications :-

1] Prefix sum queries
2] Frequent updates and queries
3] Inversion counting
4] Computing Histogram statistics
5] Coordinate compression.

Page No.
Date

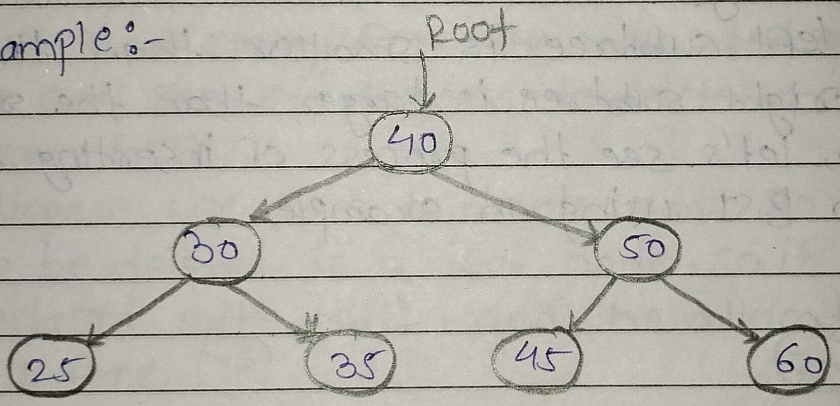
Que- Explain Binary search Tree with example also its operations, Advantage, disadvantage and application.

⟹

Binary search Tree:-

- A binary search tree follows some order to arrange the elements.
- In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node.
- This rule is applied recursively to the left and right subtrees of the root.

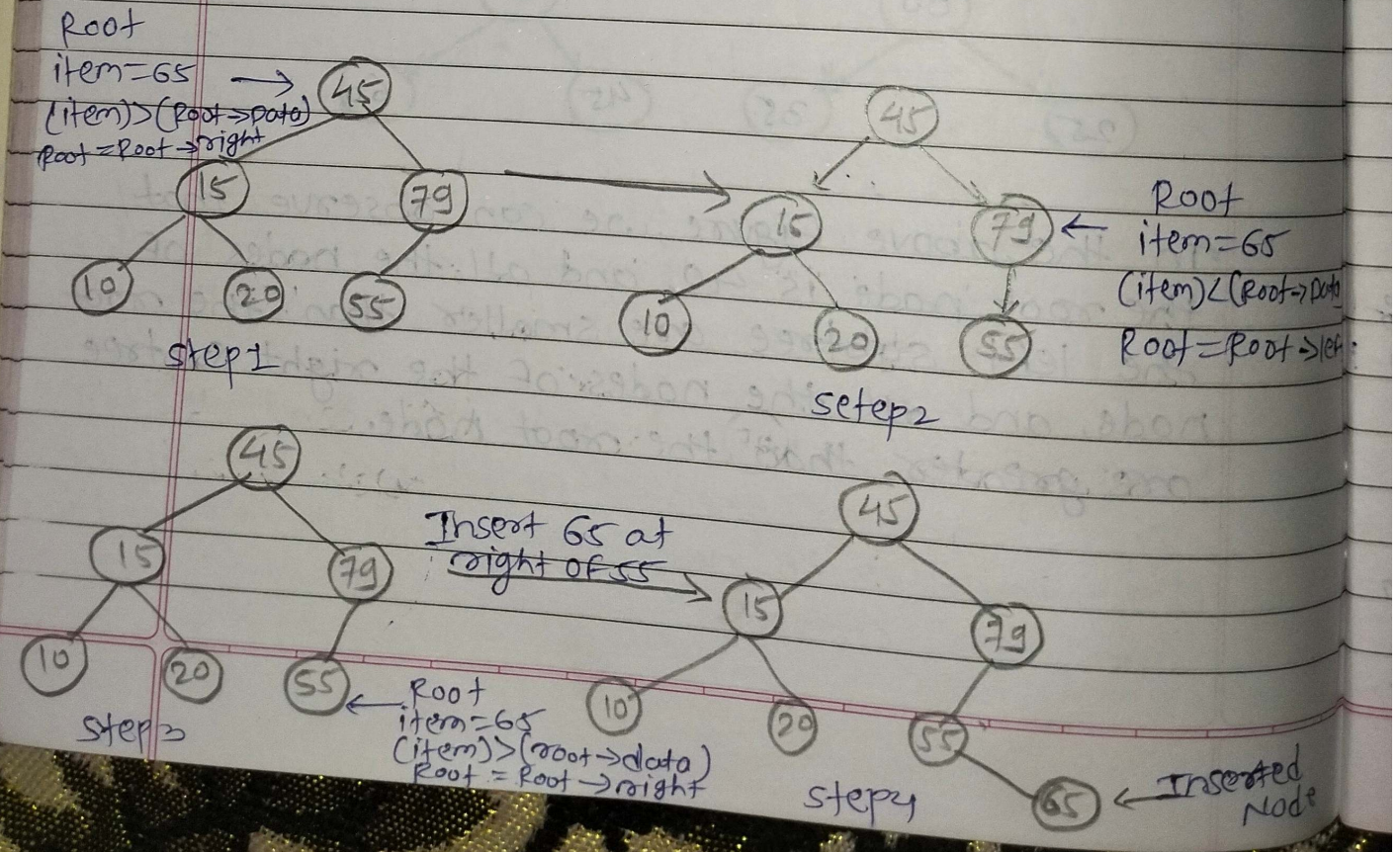
Example:-



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

- operations:-

1] Insertion in BST:-
- A new key in BST is always inserted at the leaf.
- To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree.
- Else, search for the empty location in the right subtree and insert the data.
- Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root and right subtree is larger than the root.
- Now, let's see the process of inserting a node into BST using an example.



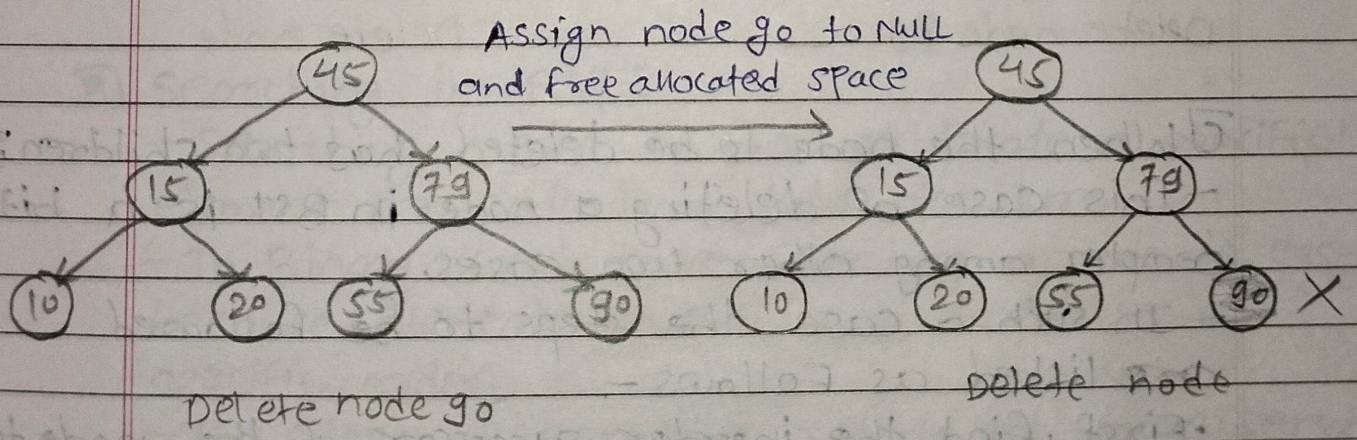
Page No.
Date

2) Deletion in BST :-

- In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated.
- To delete a node from BST, there are three possible situations occur -
  - The node to be deleted is the leaf node
  - The node to be deleted has only one child
  - The node to be deleted has two children.

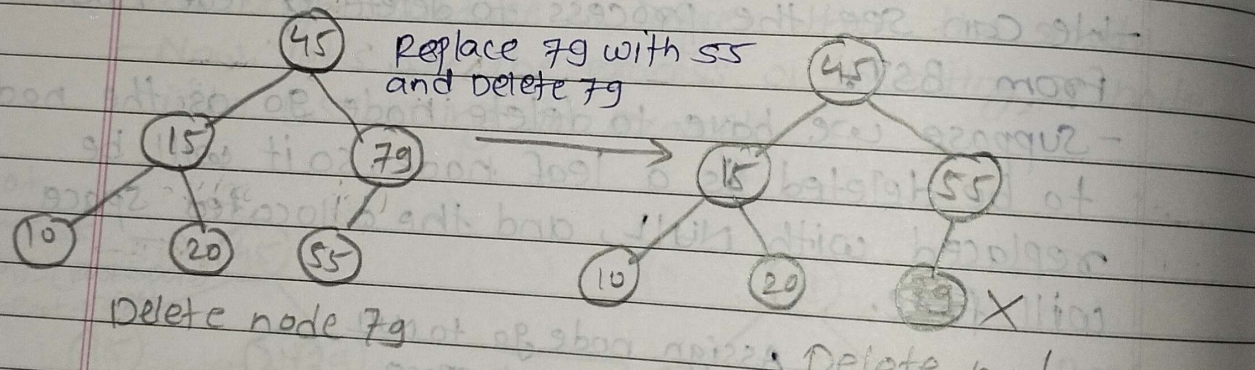
a) When the node to be deleted is the leaf node :-

- It is the simplest case to delete a node in BST.
- Here, we have to replace the leaf node with NULL and simply free the allocated space.
- We can see the process to delete a leaf node from BST.
- Suppose we have to delete node 90 as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



b] When the node to be deleted has only one child :-

- In this case, we have to replace the target node with its child, and then delete the child node.
- It means that after replacing the target node with its child node, the child node will now contain the value to be deleted.
- So we simply have to replace the child node with NULL and free up the allocated space.
- Suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.
- So, the replaced node 79 will now be a leaf node that can be easily deleted.

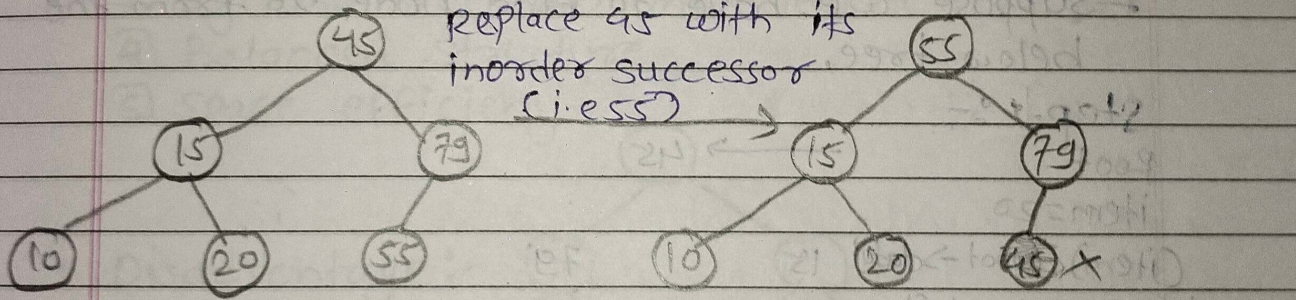


c] When the node to be deleted has two children :-

- This case of deleting a node in BST is a bit complex among other two cases.
- In such a case, the steps to be followed are listed as Follows -
  - First, find the inorder successor of the node to be deleted.
  - After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.

- And at last, replace the node with NULL and free up the allocated space.

- The inorder successor is required when the right child of the node is not empty.
- we can obtain the inorder successor by finding the minimum element in the right child of the node.
- Suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor.
- Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



3) Searching in BST:-
- Searching means to find or locate a specific element or node in a data structure.
- In Binary search tree, searching a node is easy because elements in BST are stored in a specific order.
- The Steps of searching a node in Binary search tree are listed as follows:-

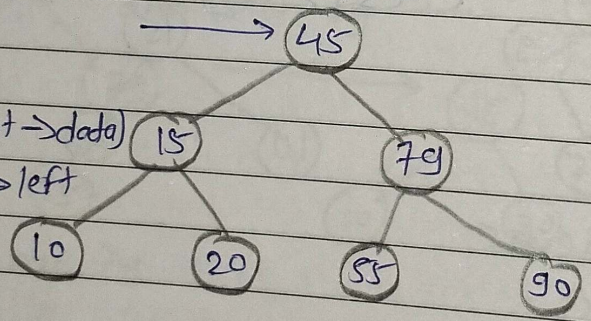
Time complexity :- O(log n)
space complexity :- O(n)

1) First, compare the element to be searched with the root element of the tree.
2) If root is matched with the target element, then return the node's location.
3) If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4) If it is larger than the root element, then move to the right subtree.
5) Repeat the above procedure recursively until the match is found.
6) If the element is not found or not present in the tree, then return NULL.

- suppose we have to find node 20 from the below tree.

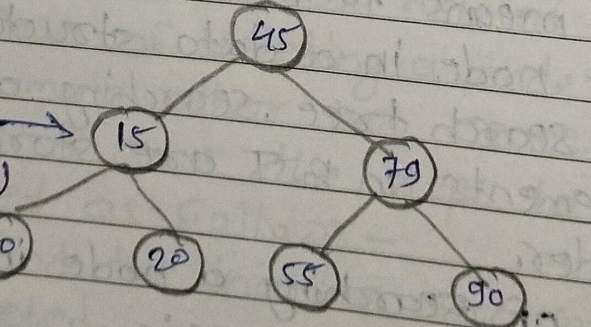
Step-1 :-

Root
item = 20
(item) < (root -> data)
Root = Root -> left

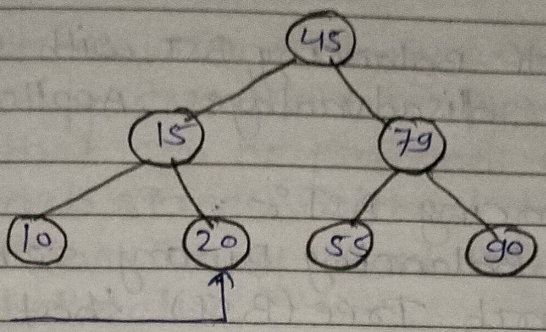


Step 2 :-

Root
item = 20
(item) > (root -> data)
Root = Root -> Right



Step3:-



Root
item=20
(item)=(root->data)
return Root

- Advantages:-
1) Efficient searching
2) ordered structure
3) Dynamic insertion and deletion
4) Balanced structure
5) space efficieny

- Disadvantage:-
1) Not self-balancing
2) worst-case time complexity
3) memory overhead
4) Not suitable for large datasets
5) Limited functionality.

- Application:-
1) Searching
2) Sorting
3) Range queries
4) Data Storage
5) Databases
6) computer graphics
7) Artificial intelligence.

Page No.
Date

Que- Explain Self-Balancing BST with example also Advantages, disadvantages, Applications.

⇒

Self-Balancing BST :-

-A self-Balancing Binary search Tree is a Binary search Tree (BST) that inevitably attempts always to maintain its height to a minimum, ensuring that its operations will maintain a worst-case time complexity of $O(\log_2 n)$

-A Binary Tree with height h can have at most $2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{(h+1)} - 1$ nodes.

$n \leq 2^{h+1} - 1$

$h \geq \lceil \log_2 (n+1) - 1 \rceil \geq \lfloor \log_2 n \rfloor$

-Hence, with repect to the self-Balancing Binary search Trees, the minimum height must be equal to the Floor value of $\log_2 n$.

-A binary tree is said to be balanced if the height of left and right sub-trees of every node differs by either -1, 0, or +1. This value is called the Balance Factor.

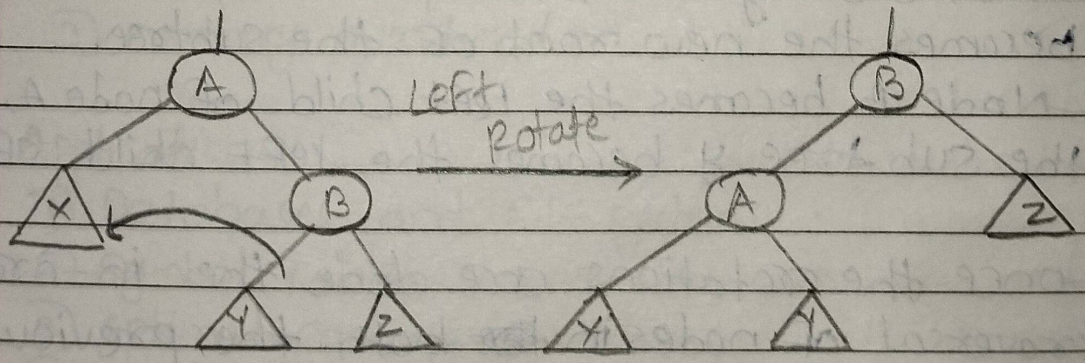
Balance Factor =

Height of the Left subtree - Height of Right subtree.

Page No.
Date

- How do self-Balancing Binary search Trees balance?
  - The following are the two types of rotation operations that can be performed to balance the Binary search Trees without breaching the property of the Binary search Tree:
    1] left Rotation
    2] Right Rotation

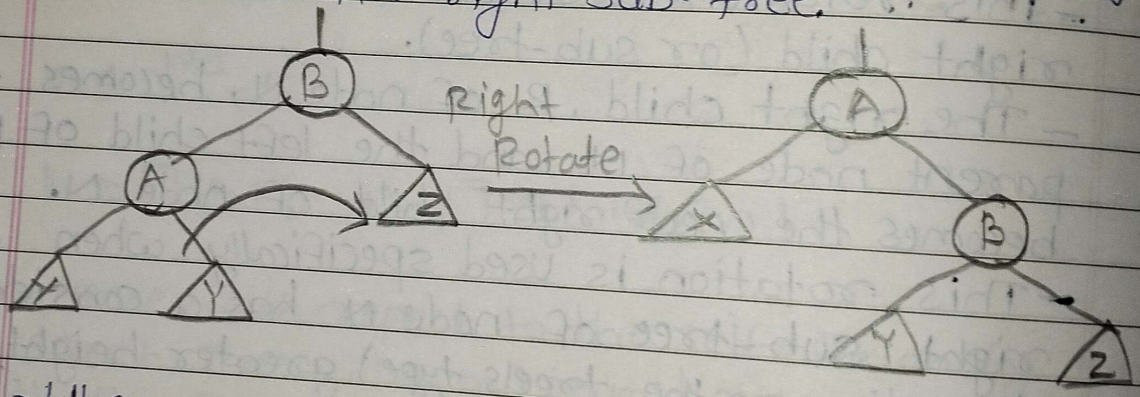
1] left Rotation:-
- The left Rotation is one of the operations of the Binary search Tree where we push a node, N down to left in order to balance the tree.
- This rotation assumes that node N has the right child (or sub-tree).
- The right child, R of node N, becomes the parent node of N, and the left child of R becomes the new right child of node N.
- This rotation is used specifically when the right sub-tree of node N has a considerably (depending on the tree's type) greater height than its left sub-tree.



- When we left rotate about node A, node B becomes the new root of the subtree. Node A becomes the left child of node B, and the sub-tree, y becomes the right child of node A.

2) Right Rotation :-

- The Right Rotation is another operation of the Binary Search Tree where we push a node, N, down to the right in order to balance the tree.
- This rotation assumes that node N has a left child (or sub-tree).
- The left child L, of the node N, becomes the parent node of N, and the right child of L becomes N's new left child.
- This rotation is specifically used when the left sub-tree of node N has a considerably (depending on the tree's type) greater height than than its right sub-tree.



- When we right rotate about node B, node A becomes the new root of the subtree.
- Node B becomes the left child of node A, and the sub-tree y becomes the left child of node B.
- Once the rotations are done, the in-order traversal of nodes in both the previous and final trees will be the same, and the Binary Search Tree property will retain.

Page No.
Date

- Types of self-Balancing Binary search Tree:-
  1) 2-3 Tree
  2) Red-Black Tree
  3) AA Tree
  4) AVL Tree
  5) B Tree
  6) Scapegoat Tree
  7) Splay Tree
  8) Tango Tree
  9) treap
  10) weight Balanced Trees.

- Advantages:-
  1) Efficient operations
  2) Predictable Performance
  3) versatility

- Disadvantage:-
  1) overhead
  2) complex Implementation
  3) memory overhead

- Application:-
  1) Databases and file systems
  2) Symbol Tables
  3) optimal caching
  4) Dynamic set operations
  5) Concurrency control.

Page No.
Date

Que- Explain Red Black Tree with example, also operations, Advantages, Disadvantages and Applications

⇒

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.
- Red Black Trees are self-balancing, meaning that the tree adjusts itself automatically after each insertion or deletion operation.
- It uses a simple but powerful mechanism to maintain balance, by coloring each node in the tree either red or black.

- 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.

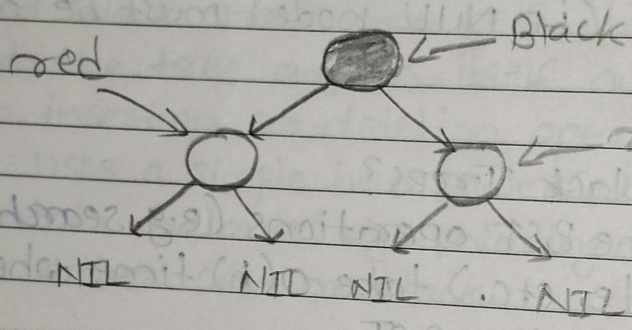
- Rules That Every Red-Black Tree Follows:-
  1) Every node has a color either red or black.
  2) The root of the tree is always black.
  3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
  4) Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
  5) Every leaf (i.e NULL node) must be colored BLACK.

- Why Red-Black Trees?
  - Most of the BST operations (e.g. search, max, min, insert, delete etc.) take O(h) time where h is the height of the BST.
  - The cost of these operations may become O(n) for a skewed Binary tree.
  - If we make sure that the height of the tree remains O(logn) after every insertion and deletion, then we can guarantee an upper bound of O(logn) for all these operations.
  - The height of a Red-Black tree is always O(logn) where n is the number of nodes in the tree.

  Search — O(logn)
  Insert — O(logn)
  Delete — O(logn)

- characteristics of Red Black Tree:
  - The root node is always black and each node can be either black or red.
  - Every leaf node of the red-black tree is black.
  - The children of red nodes are black
  - The number of black nodes will be the same for every simple path from the root to the descendant leaf node.

Example:-



- operations of Red-Black Trees:-

1) Insertion operation:-
- Insertion operation of a Red-Black tree follows the same insertion algorithm of a binary search tree.
- The elements are inserted following the binary search property and as an addition, the nodes are color coded as red and black to balance the tree according to the red-black tree properties.

- Follow the procedure given below to insert an element into a red-black tree by maintaining both binary search tree and red black tree properties.

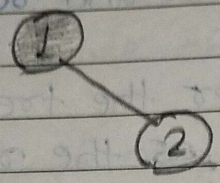
Case-1:- check whether the tree is empty; make the current node as the root and color the node black if it is empty.

Case-2:- But if the tree is not empty, we create a new node and color it red. Here we face two different cases -

- If the parent of the new node is a black colored node, we exit the operation and tree is left as it is.
- If the parent of this new node is red and the color of the parent's sibling is either black or if it does not exist, we apply a suitable rotation and recolor accordingly.
- If the parent of this new node is red and color of the parent's sibling is red, recolor the parent, the sibling and grandparent nodes to black. The grandparent is recolored only if it is not the root node; if it is the root node recolor only the parent and the sibling.

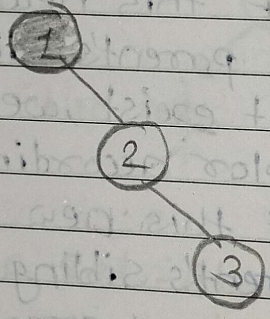
Example:-
Let us construct an RB Tree for the first 7 integer.
Step 1:- The tree is checked to be empty so the First node added is a root and colored black.

Step2:- the tree is not empty so we create a new node and add the next integer with color red.

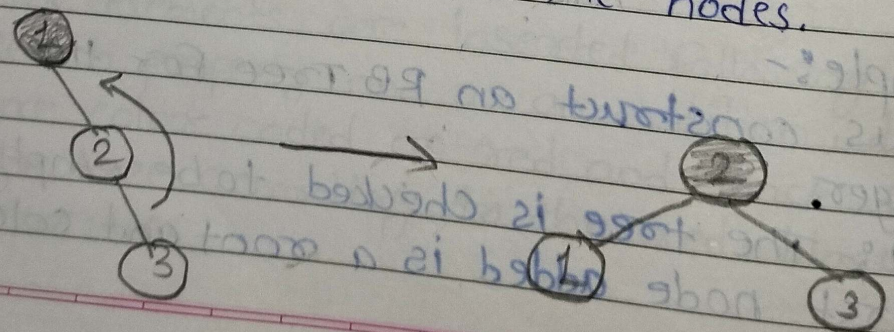


The nodes do not violate the binary search tree and RB tree properties, hence we move ahead to add another node.

The tree is not empty; we create a new red node with the next integer to it. But the Parent of the new node is not a black colored node.

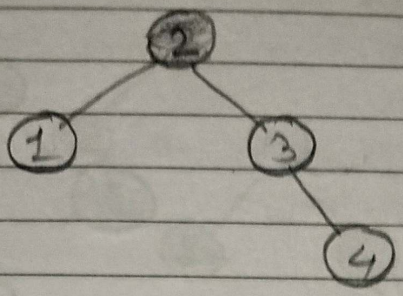


The tree right now violates both the binary search tree and RB tree properties; since Parent's sibling is NULL, we apply a suitable rotation and recolor the nodes.

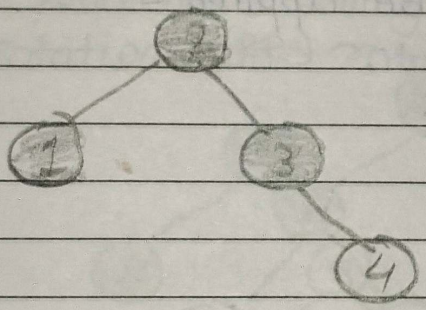


Page No.
Date

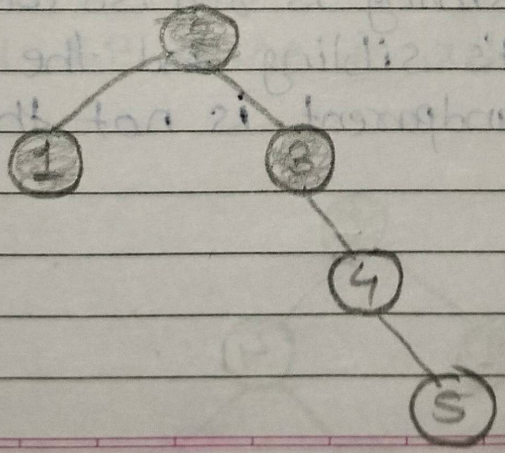
Now that the RB Tree property is restored, we add another node to the Tree -



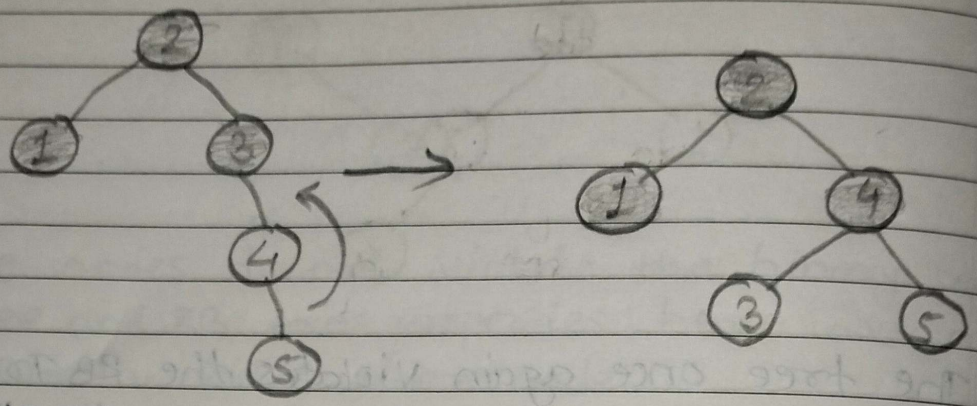
The tree once again violates the RB Tree balance property, so we check for the parent's sibling node color, red in this case, so we just recolor the Parent and the sibling.



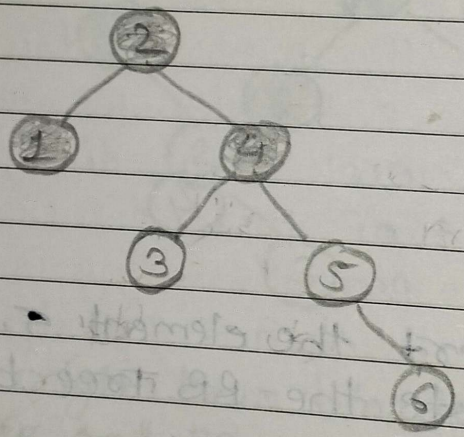
We next insert the element 5, which makes the tree violate the RB Tree balance property once again.



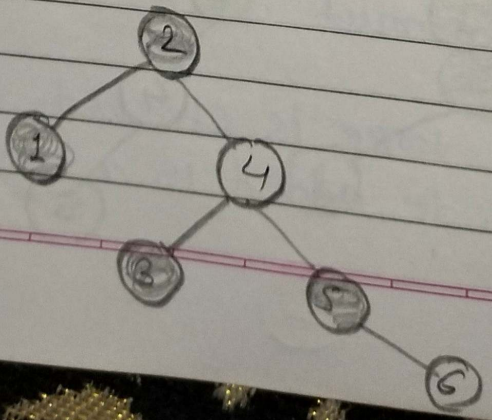
And since the sibling is NULL, we apply suitable rotation and recolor.



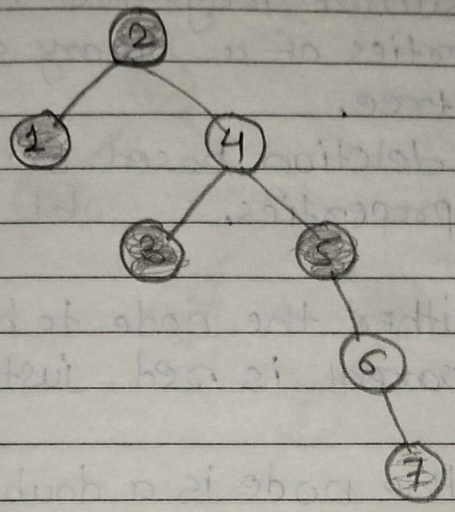
Now, we insert element 6, but the RB Tree property is violated and one of the insertion cases need to be applied -



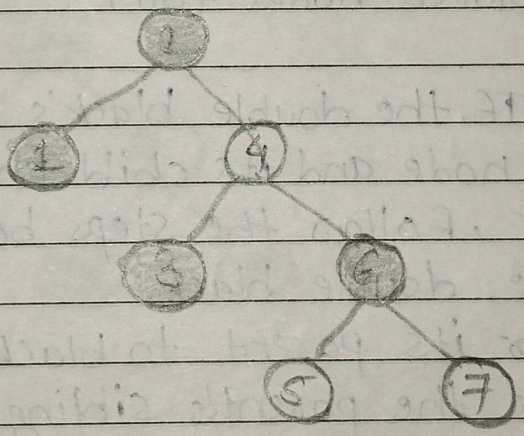
The parent's sibling is red, so we recolor the parent, parent's sibling and the grandparent node since the grandparent is not the root node.



Now, we add the last element 7, but the Parent node of this new node is red.



Since the parent's sibling is NULL, we apply suitable rotations (RR) rotation.



The Final RB Tree is achieved.

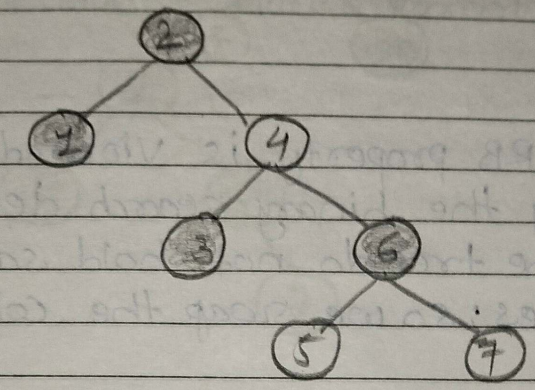
2) Deletion operation:-
- The deletion operation on red black tree must be performed in such a way that it must restore all the properties of a binary search tree and a red black tree.
- we perform deletion based on the binary search tree properties.

case-1:- If either the node to be deleted or the the node's parent is red, just delete it.

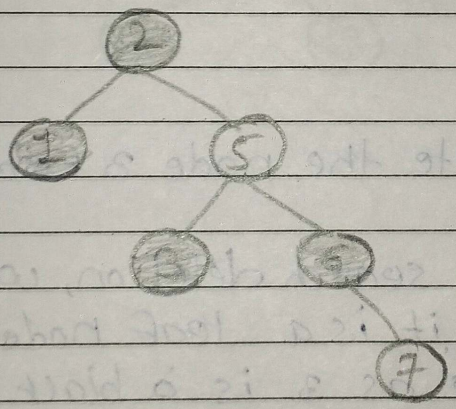
case-2:- If the node is a double black, just remove the double black (double black occurs when the node to be deleted is a black colored leaf node, as it adds up the NULL nodes which are considered black colored nodes too)

case-3:- If the double black's sibling node is also a black node and its child nodes are also black in color, Follow the steps below -
- Remove double black
- Recolor its parent to black
- Recolor the parent's sibling with red
- If double black node still exists, we apply other cases.

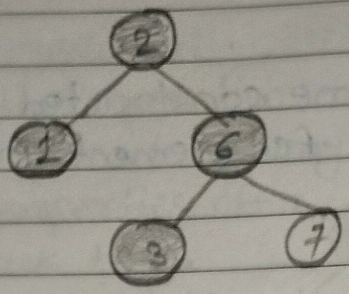
- Example:
considering the same constructed Red-Black Tree above, let us delete few elements from the tree.



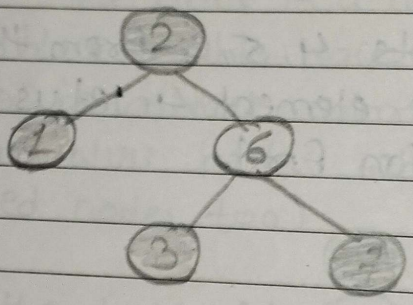
Delete elements 4, 5, 3 from the tree.
To delete the element 4, let us perform the binary search deletion first.



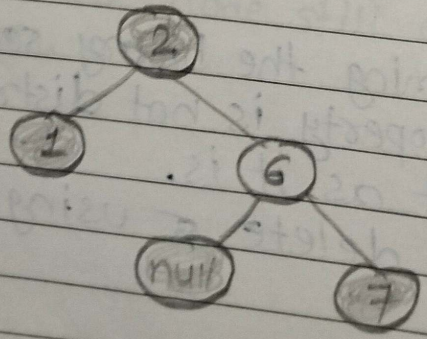
After performing the binary search deletion, the RB Tree property is not distribed, therefore the tree is left as it is.
- Then, we delete 5 using the binary search deletion.



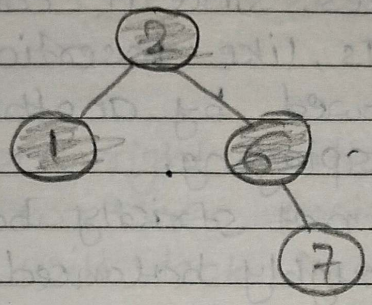
But the RB property is violated after performing the binary search deletion i.e. all the paths in the tree do not hold same number of black nodes; so we swap the colors to balance the tree.



Then, we delete the node 3 from the tree obtained -
Applying binary search deletion, we delete node 3 normally as it is a leaf node. And we get a double node as 3 is a black colored node.



we apply case 3 deletion as double black's sibling node is black and its child nodes are also black. Here we remove the double black, recolor the double black's parent and sibling.



All the desired nodes are deleted and the RB Tree property is maintained.

Page No.
Date

Que- Explain Splay Trees with example and operation

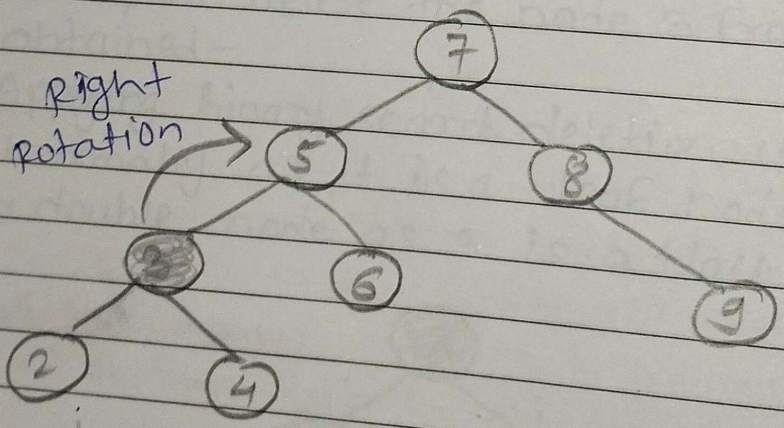
Splay Tree :-

- Splay tree are the altered versions of the Binary search Trees, since it contains all the operations of BSTs, like insertion, deletion and searching, followed by another extended operation called splaying.
- Splay trees are not strictly balanced trees, but they are roughly balanced tree.

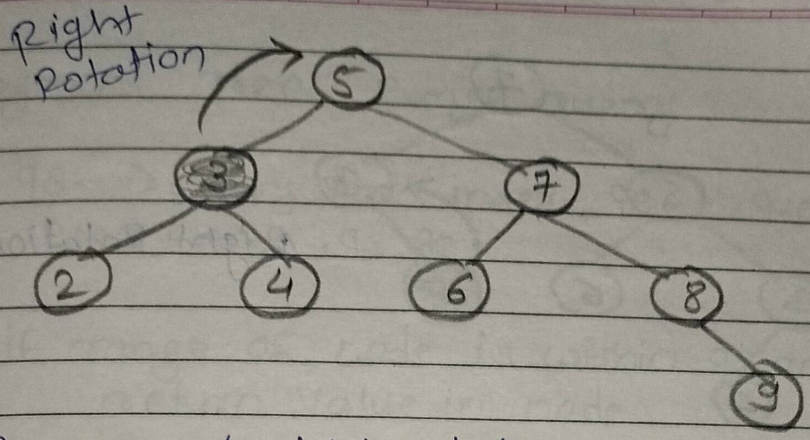
Rotations :-

1) Zig - Zig rotation :-

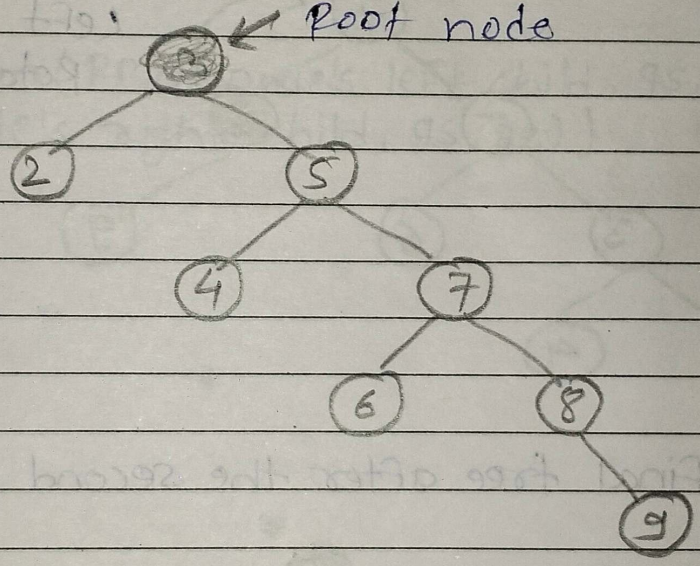
- The Zig - Zig rotations are performed when the operational node has both parent and a grand-parent. The node is rotated two places towards its right.



The First rotation will shift the tree to one position right.-

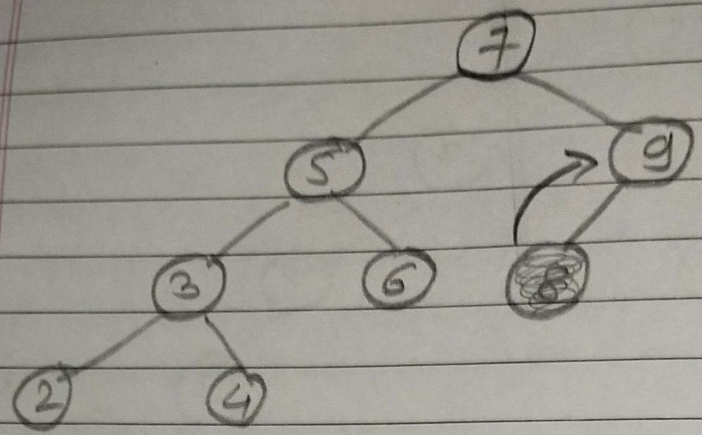


- The second right rotation will once again shift the node for one position. The final tree after the shift will look like this -



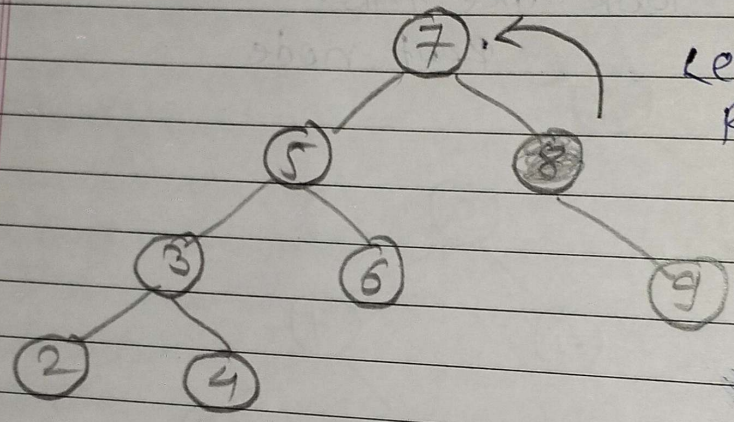
2) Zig-Zag rotation :-

- The zig-zag rotations are performed when the operational node has both a parent and a grandparent. But the difference is the grandparent, Parent and child are in LRL format. The node is rotated first towards its right followed by left.



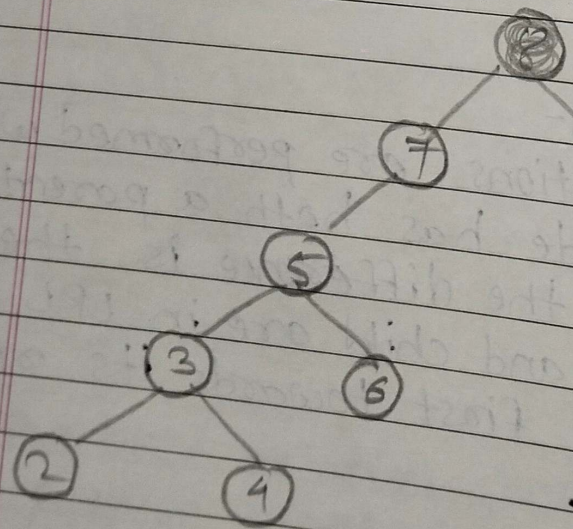
Right Rotation

After the first rotation, the tree is-



Left Rotation

The final tree after the second rotation-



Page No.
Date

que- segment tree Range query

⇒

```
// qs --> query start index, qe --> query end index
int RMQ (node, qs, qe)
{
  if range of node is within qs and qe
      return value in node
  else if range of node is completely outside
     qs and qe
        return INFINITE
  else
  return min(RMQ(node's left child, qs, qe),
   RMQ (node's right child, qs, qe))
}
```