Rainbow
PAGE:
DATE: / /

Assignment No. 5

Name: Bhavana Batna
Class: SE 09
Rollno.: 23107
Batch: E 09

1) Title: Assignmenment 5: Binary Search Tree

2) Aim: To implement a Binary Search Tree

3) Problem Statement: Implement binary search tree & perform following operations:
   a) Insert (Handle insertion of duplicate entry)
   b) Delete
   c) Search
   d) Display tree (Traversal)
   e) Display - Depth of tree
   f) Display - Mirror image
   g) Create a copy
   h) Display all parent nodes with their child nodes
   i) Display leaf nodes
   j) Display tree level wise.

4) Objective:
   1) To study Data structures & their implementations & applications.
   2) To learn different Searching & sorting techniques.
   3) To study some advanced data structures such as trees, graphs & tables.
   4) To learn different file organizations.
   5) To learn algorithm development & analysis of algorithms.

PAGE:
DATE: / /

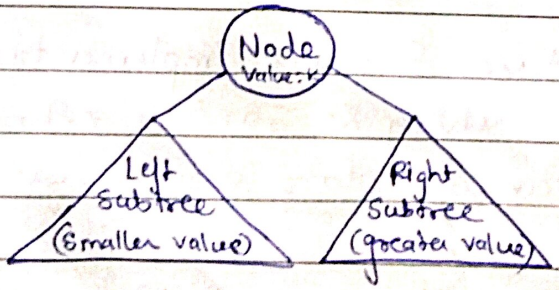
5) Outcome:

1) Analyze algorithms & to determine algorithm correctness & time efficiency class.
2) Implement abstract data type (ADT) & data structures for given application.
3) Design algorithms based on techniques like brute-force, divide & conquer, greedy, etc.
4) Solve problems using algorithmic design techniques & data structures.
5) Analyze of algorithms with respect to time & space complexity.

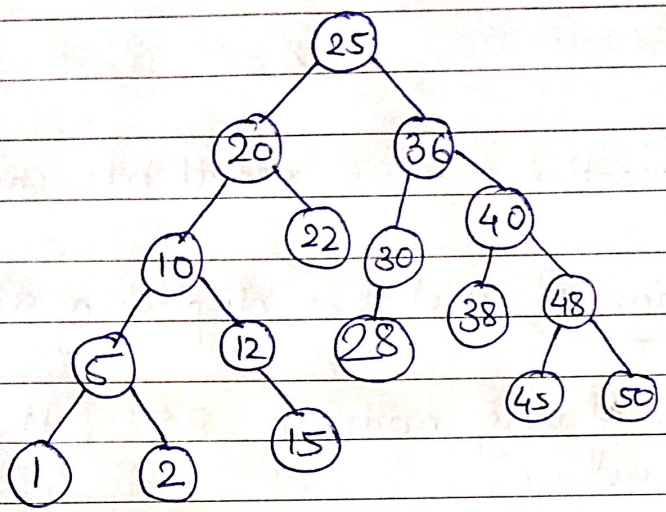
6) Theory:

1) Binary search Tree :

A binary tree in which the data of all the nodes in the left sub-tree of the root node is less than the data of the root & the data of all the nodes in the right sub-tree of the root node is more than the data of the root, is called as Binary search tree.



Example:



In this tree, left sub-tree of every node has smaller values while right sub-tree of every node has larger values than the root.

2) Applications of BST:

1) BST is used to implement multilevel indexing in database applications.
2) BST is also used used to implement constructs like dictionary.
3) BST can be used to implement various efficiency searching algorithms.
4) BST is also used in applications that require a sorted list as input like the online stores.
5) BSTs are also used to evaluate the expression using expression trees.

3) ADT BST:

Instances: Binary search tree is a binary tree in which value of left child is less than its parent node & value of right child is greater than its parent node.

Operations:

1) Create: Using this operation a binary search tree can be created.
2) Display: This operation is used for displaying all nodes of BST.
3) Insert: For insertion of any node in BST, this operation is useful.
4) Delete: Using this operation, any node from BST can be deleted.
5) Search: This function is used for searching any node from BST.

Node structure:

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

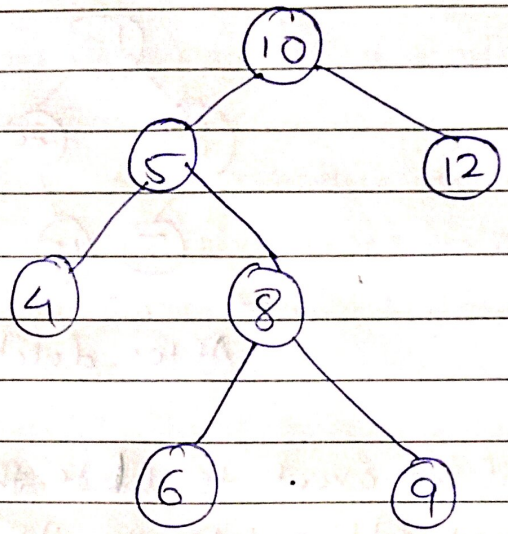
Create:

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

Search:

- Node to be searched is called key node.
- Key node is compared with each node starting from root node, if value of key node is greater then search on right sub-branch or on left sub-branch.
- If we reach leaf node & still not find the node then "node is not present in the tree".

Example:



In above tree if we want to search 9. Then compare 9 with root node 10. As 9 is less than 10 search on left sub-branch. Now 9 is greater than 5, so we will move on to right sub-branch. Now compare 9 with 8 but 9 is greater that 8 we will move on to right sub-branch. As the node we will get holds the value 9. Thus desired node can be searched.

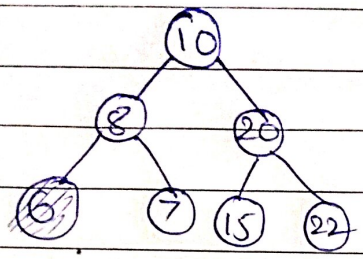
Deletion of element from binary tree:

For deletion of any node from binary search tree there are three cases:

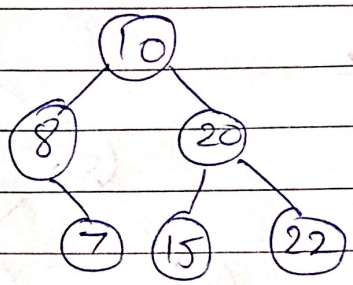
1) Deletion of leaf node.
2) Deletion node having one child
3) Deletion of node having two children.

- **Deletion of leaf node:**

This is the simplest deletion, in which we set left or right pointer of parent node as NULL.



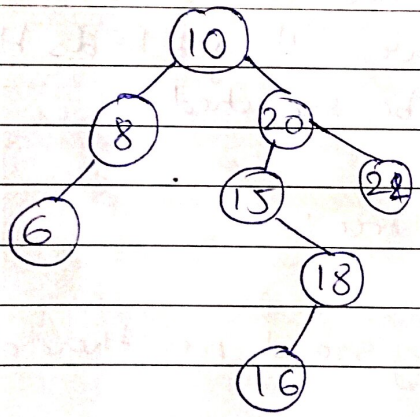
Before deletion



After deletion

In the above tree if we want to delete the node having value 6 then we set left pointer to its parent node as NULL. That is left pointer of node having value 8 is set to NULL.

- **Deletion of a node having one child:**



If we want to delete node 15, then we will simply copy node 18 at place of 15 and then set the node free. The inorder successor is always copied at position of node to be deleted.

Step 1: Find node to be deleted using search operation
Step 2: If it has only one child, then create a link between its parent & child nodes.
Step 3: Delete the node

- **The node having two children:**

Step -1: Find node to be deleted using search operation
Step -2: If it has two children, then find largest node in its left subtree

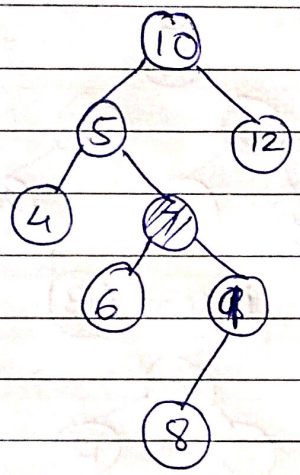
Step-3 : Swap both deleting node & node which found in above step.

Step-4 : Then check whether deleting node came to case 1 or case 2 else go to step 2

Step-5 : If it comes to case 1, then delete using case 1 or else with case 2

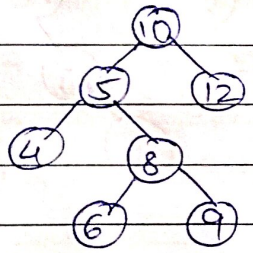
Step-6 : Repeat same process until node is deleted.

Eg :



Let us consider we want to delete node having value 7. We will find out in order successor of node 7. The inorder successor will be simply copied at location of node 7.

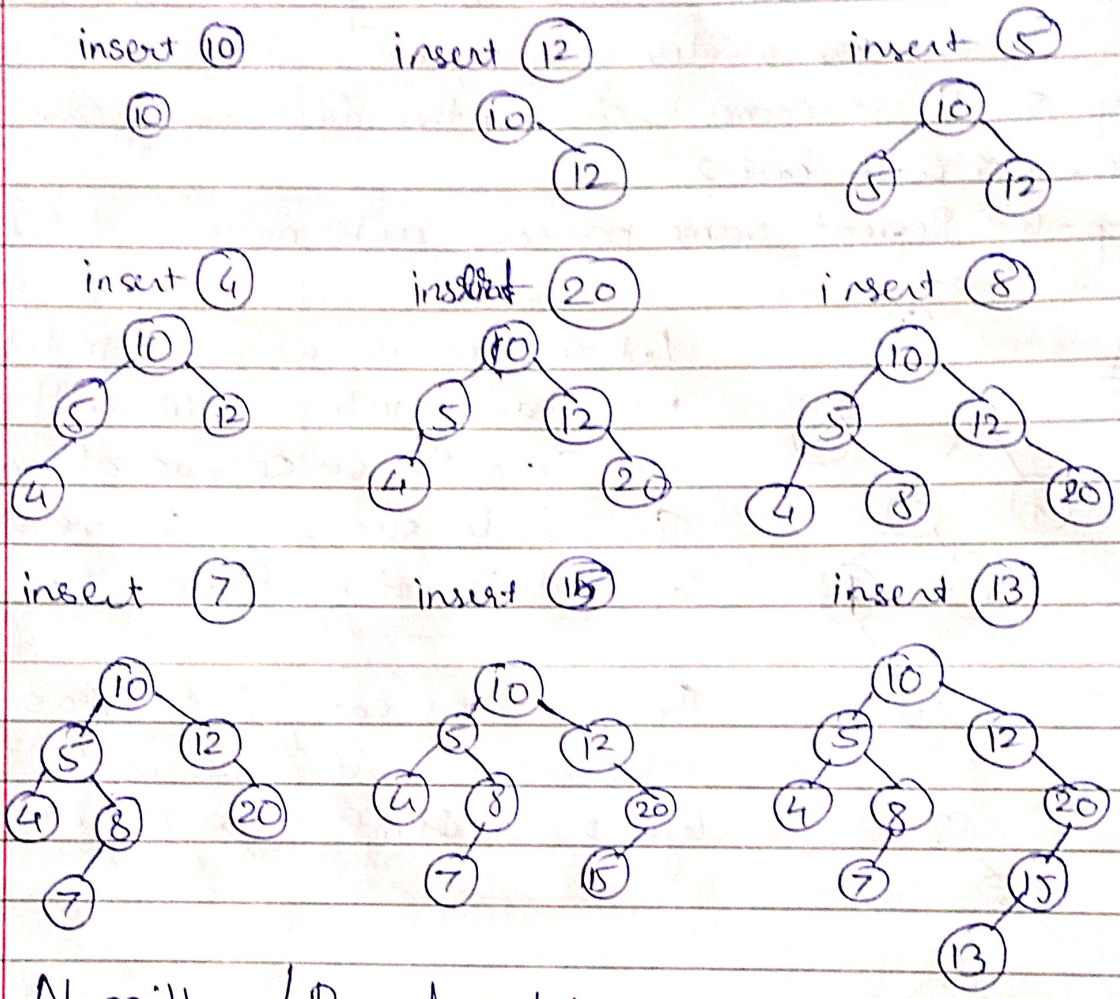
That means copy 8 at position where value of node is 7. Set left pointer of 9 as NULL.



Insertion of node in binary tree :

1. Read the value for node which is to be created, and store it in a node called New.
2. Initially if (root != NULL) then root = New then again read.
3. If (New -> value < root -> value) then attach New node as a left child of root otherwise attach New node as a right child of root.
4. Repeat above steps for constructing required binary search tree completely.

Example:
Construct a binary search tree by inserting: 10, 12, 5, 4, 20, 8, 7, 15 & 13.



7) Algorithm / Pseudocode:

1) BST creation Recursive:

```
Node * create (Node *p , int x)
  if p = NULL
    p = getNode()
    return p
  else
    if (x < p → data)
      if p → (child != NULL)
        p → lchild = create (p → lchild, x)
      else
        p → lchild = get node()
```

```
Else
    if p -> rchild != NULL
        p -> rchild = create (p -> rchild, n)
    else
        p -> rchild = getNode()
return p.
```

2) BST creation non-recursive:

```
Node *create (int num)
    Node *p = getNode(num)
    if root = NULL
        root = p
    else
        Node *temp = root, *parent
        while (temp ≠ NULL)
            parent = temp
        if temp -> data = num
            return NULL
        if temp -> data < num
            temp = temp -> right
        else
            temp = temp -> right
        End while
        If parent -> data > num
            parent -> left = p
        else
            parent -> right = p
    return p.
```

Rainbow
PAGE:
DATE: / /

3) BST search recursive:

```
Node * search (key, node* root)
p = root
if (key < p -> data)
p = search (key, p -> left)
else
if (key > p -> data)
p = search (key, p -> right)
search p.
```

4) BST search non-recursive:

```
Node * search (int num)
    Node* temp = root
    while temp ≠ NULL
        if temp -> data > num
            temp = temp -> left
        else
        if temp -> data < num
            temp = temp -> right
        else
            return temp.
        end while
        return NULL.
```

5) BST delete recursive:

```
Node * delete (Node * T, int num)
    if T = NULL
        return T
```

```
if num < T -> data
   T -> left = delete (T -> left, num)
if num > T -> right
   T -> right = delete (T -> right, num)
else
   Node * temp = T
   if T -> left = NULL
      T = T -> right
   free (temp)
   return (T)
else if  T -> right = NULL
      T = T -> left
      free (temp)
      return (T)
Temp = findmin (T -> right)
T -> data = temp -> data
T -> right = delete (T -> right, temp -> data)
Return (T).
```

6) Level Order Traversal:

```
Display Levelwise ()
 // create a queue
   enqueue (root)
   enqueue (NULL)
   while (q - size > 1)
      Node * curr = dequeue ()
      if current = NULL
         enqueue (NULL)
         print "\n"
      else
         if current -> left ≠ NULL
```

```
        enqueue (current → left)
    if current → right ≠ NULL
        enqueue (current → right)
    print "current → data.
    end if
    end while.
```

2) Depth of tree recursive:

```
int treeDepth (Node * T)
  if T = NULL
     return 0
  Return 1 + max (treeDepth (T → left), treeDepth (T → right))
```

Non-recursive:

```
int treeDepth (root)
    if root = NULL
    return 0
// create an empty queue for level order traversal.
  q.insert (root)
  height = 0
  while True
    node count = q.size ()
    if node count = 0
       return height
    height = height + 1
    while node count > 0
    temp = q.delete ()
    if temp → left ≠ NULL
```

```
q.insert (temp → left)
if temp → right ≠ NULL
    q.insert (temp → right)
Node count --
end while
end while
return height.
```

8) Mirror Image:

Recursive:

```
mirror Img (T)
  if T = NULL
      return
  temp = T → left
  T → left = T → right
  T → right = temp
  mirror Img (T → left)
  mirror Img (T → right)
```

Non-recursive:

```
mirror Img ( )
  // create an empty queue
 if root = NULL
     return
     q. insert (root)
     while (! q = is Empty)
         T = q. dequeue ( )
 if T → left = NULL && T → right == NULL,
```

continue

```
else if T -> left ≠ NULL & T -> right ≠ NULL
        Temp = T -> left
        T -> left = T -> right
        T -> right = Temp
        q.insert (T -> left)
        q.insert (T -> right)
else if T -> left = NULL
        T -> left = T -> right
        T -> right = NULL
        q.insert (T -> left)
else    T -> right = T -> left
        T -> left = NULL
        q.insert (T -> right)
end if
end while
```

9) Copy of tree:

Recursive:

```
Node * createcopy (T)
if T == NULL
   return NULL
// create a new node
   new Node -> left = createcopy (newNode -> left)
   new Node -> right = create copy (newNode -> rig
   Return newNode
```

10) Count number of leaf node, non leaf nodes:

a) leaf node:

```
int countleafNode (T)
    if (T = NULL)
      return 0
    if T → left = NULL & T → right == NULL
      return 1
    else
      return countleafNode (T → left) + countleafNod
                                        (T → right)
```

b) Non-leaf Nodes:

```
int countNonleafNode (T)
    if T = NULL or T → left = NULL & T → right = NULL
      return 0
    return 1+ count NonleafNode (T → left) +
              countNonleafNode (T → right)
```

II) Traversal:

a) Inorder:

```
inorder (T)
if T = NULL
  return
inorder (T → left)
print T → data
inorder (T → right)
```

b) Preorder:

```
preorder (T)
if T = NULL
return
print "T -> data"
preorder (T -> left)
preorder (T -> right)
```

c) Postorder:

```
postorder (T)
if T = NULL
  return
postorder (T -> left)
postorder (T -> right)
print "T -> data"
```

8) Test cases / validations:

Validations:

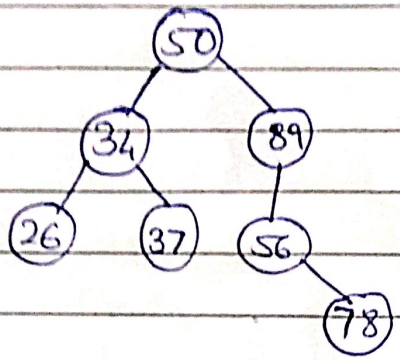
Valid key input for insertion, deletion, search operations.

Test cases:

1) Random input
2) Sorted input
3) Input for skewed tree concepts.

Rainbow
PAGE:
DATE: / /

Example:



insert (11) ⇒

Number of comparision = 3

insert (66) ⇒

Number of comparision = 4

7) Conclusion:

Binary search tree is a sorted binary tree whose internal nodes each store a key greater than all keys in the left subtree & less than those in right subtree. Using BST, we can perform various operations like searching, deleting, inserting effectively.