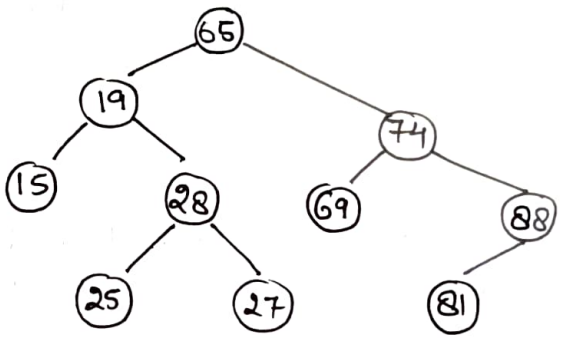
BINARY SEARCH TREE

→ Binary Tree T is termed Binary Search Tree (or Binary Sorted Tree) if each node N of T satisfies the following property :-

The value of N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N.



→ There are 4 operations in a Binary Search Tree :-

(a). Searching Data.

(b). Inserting Data.

(c). Deleting Data.

(d). Traversing the Tree.

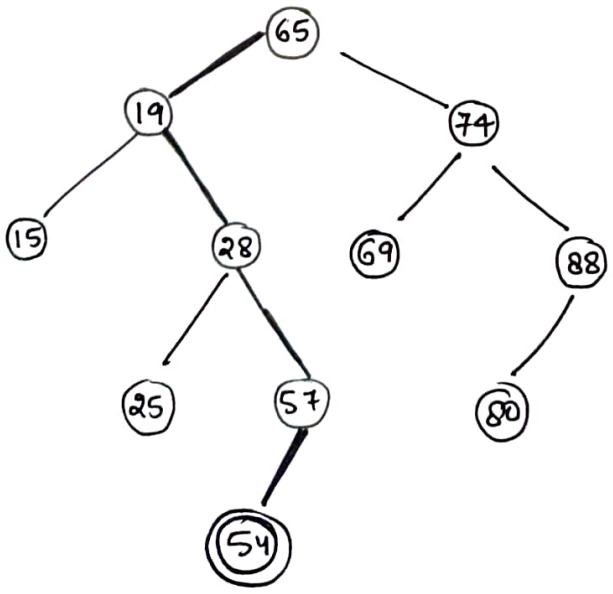
Searching a Binary Search Tree

- Searching data in the Binary Search Tree is much faster than searching data in arrays or a linked lists. This is why where frequent searching is requirement or need to be performed, this Data Structure is used to store data.

- Suppose, in a Binary Search Tree, T, the ITEM the item to be search. We will assume that the tree is represented using a linked structure.

- We start from the root node R. Then, if ITEM is less than the value in the root node R, we proceed to its left child; if ITEM is greater than the value in the R node, we proceed to its right child.

  The process continued till the ITEM is not found. or we reach dead end. → that is the leaf node.



54 to be search.

ALGORITHM Search_BST

Input: ITEM is the data that has to be searched.

output: If found then pointer to the node containing data ITEM else a message.

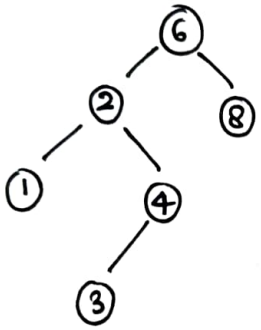
Data Structure: linked structure of the BinaryTree.
Pointer to the root node is ROOT.

Steps

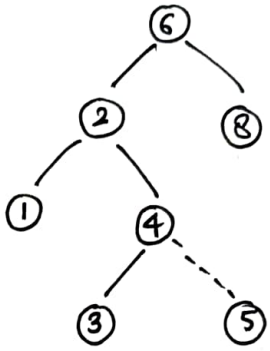
1. ptr = ROOT , flag = FALSE // start from the root.
2. While (ptr ≠ NULL) and (flag = FALSE) do
3. Case: ITEM < ptr → DATA // Goto Left Sub-tree
4. ptr = ptr → LCHILD
5. Case: ptr → DATA = ITEM
6. flag = TRUE
7. Case: ITEM > ptr → DATA // Goto RIGHT Sub-tree.
8. ptr = ptr → RCHILD
9. Endcase
10. EndWhile
11. If (flag = TRUE) then // Search is succesful
12. Print "ITEM has found at the node", ptr
13. Else
14. Print " ITEM does not exists : Search is unsuccesful".
15. Endif
16. Stop.

③ Inserting a node in a Binary Search Tree

- It is one more step than the searching Operation.
- To insert a node with data, say ITEM, into a tree, the tree is required to be searched starting from the root node.
- If ITEM is found, do nothing, otherwise ITEM is to be Inserted at the dead end where the search halts.



(a). Before Insertion



(b) Search find the location where 5 should be Inserted.

Insertion of 5 in a Binary Tree. Search proceeds starting from the ⑥-②-④ then halts when it finds the right child is null. (dead end). This simply means that if 5 occurs, then it should have occured on the right part of the node.4.

ALGORITHM Insert-BST.

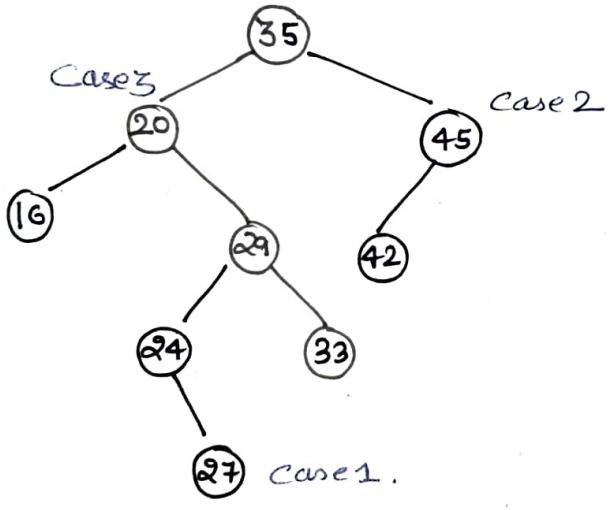
```
1.  ptr = ROOT , flag = FALSE
2.  While (ptr≠NULL) and (flag=FALSE) do
3.      Case: ITEM < ptr→DATA   // GOTO LEFT SUBTREE
4.          ptr1 = ptr
5.          ptr = ptr→LCHILD
6.      Case: ITEM > ptr→DATA   // GOTO RIGHT SUBTREE
7.          ptr1 = ptr
8.          ptr = ptr→RCHILD
9.      Case: ~~ITEM =~~ ptr→DATA = ITEM     // Node exists
10.         flag = TRUE
11.         Print "ITEM already exists".
12.         Exit
13.     Endcase
14. EndWhile
15. If (ptr=NULL) then  // Insert when the search halts at the dead end
16.         new = GetNode (NODE)
17.         new→DATA = ITEM     // Avail a node & then initialize it.
18.         new→LCHILD = NULL
19.         new→RCHILD = NULL
20.         If (ptr1→DATA < ITEM) then
21.             ptr1→RCHILD = new
22.         Else
23.             ptr1→LCHILD = new
24.         Endif
25. Endif
26. Stop.
```

Deleting a node from a Binary Search Tree.

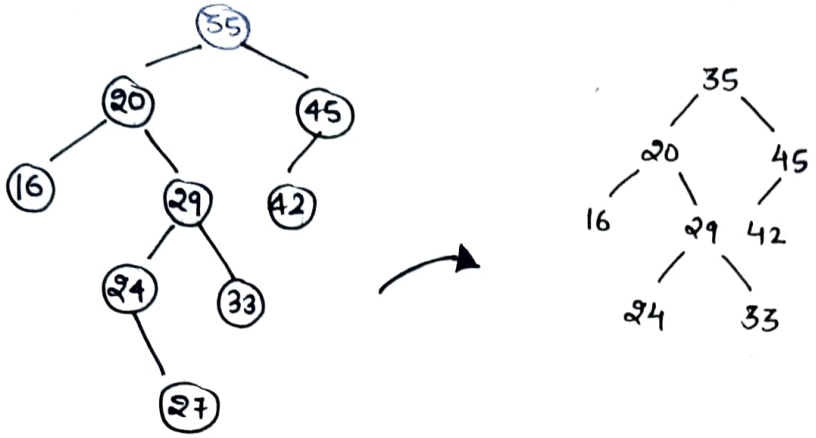
→ Suppose T is a BST, & ITEM is the information given which has to be deleted from T, if that exists in the Tree.

→ Suppose N be the node which contains the information ITEM. Let us assume PARENT(N) denotes the Parent node of N and SUCC(N) denotes the INORDER SUCCESSOR of the node N (inorder successor means the node which comes after N during the Inorder traversal of T).

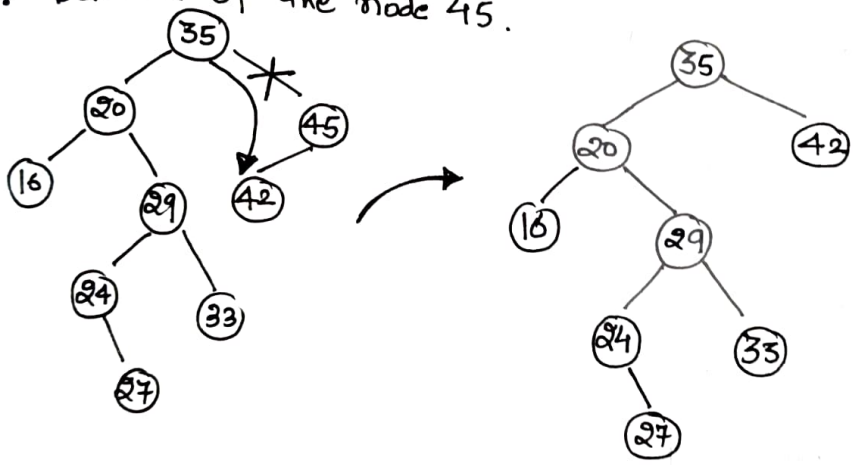
Tree → N is the leaf node.
Tree → N has exactly one child.
Tree → N has two child.



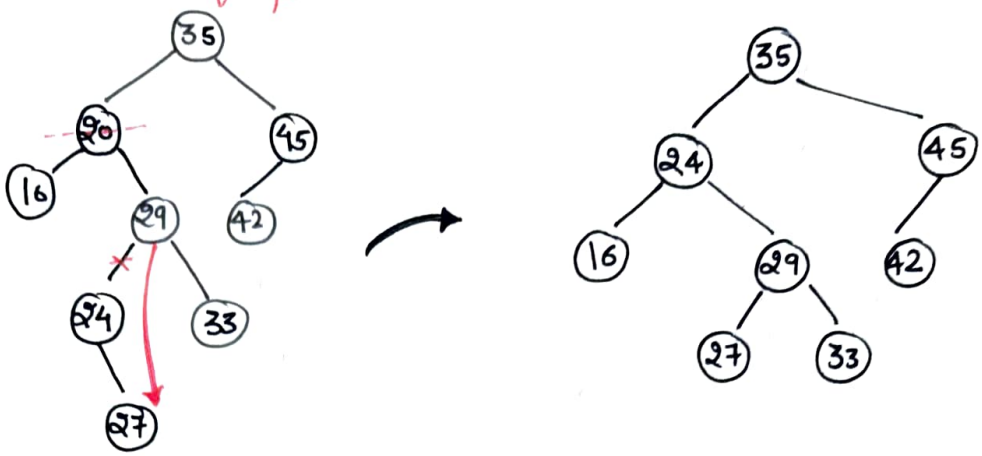
Case 1 :- Deletion of the node 27



Case 2 : Deletion of the node 45.



Case3: Deletion of node 20.



Step1:

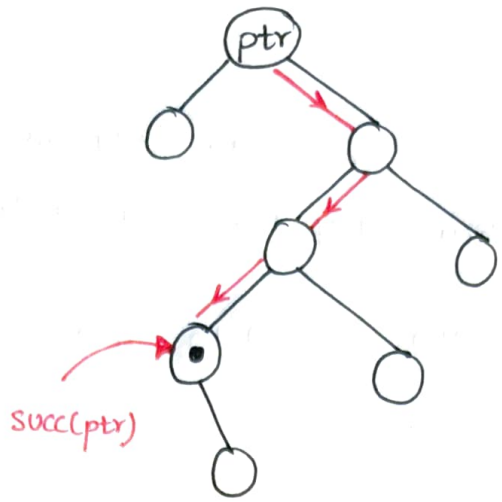
1. ptr = ROOT , flag = FALSE
2. While (ptr ≠ NULL) and (flag = FALSE) do // Step to find the location of the node.
3. Case: ITEM < ptr → DATA
4. parent = ptr
5. ptr = ptr → LCHILD
6. Case: ITEM > ptr → DATA
7. parent = ptr.
8. ptr = ptr → RCHILD
9. Case: ptr → DATA = ITEM
10. flag = TRUE
11. End Case
12. End While
13. If (flag = FALSE) then // when node doesn't exists
14. Print "ITEM does not exists: No deletion"
15. Exit
16. End If

```
/* DECIDE THE CASE OF DELETION */
17. If (ptr→LCHILD = NULL) and (ptr→RCHILD = NULL) then   // Node has no child
18.     case = 1
19. Else
20.     If (ptr→LCHILD ≠ NULL) and (ptr→RCHILD ≠ NULL) then   // Node has both a left & Right child
21.         case = 3
22.     Else
23.         case = 2
24.     Endif
25. Endif.

/* DELETION : CASE 1 */
26. If ((case = 1) then
27.     If (parent→LCHILD = ptr) then      // If the node is a left child
28.         parent → LCHILD = NULL         // Set the pointer of its parent
29.     Else
30.         parent → RCHILD = NULL
31.     Endif
32.     Returns Node(ptr)   // Return deleted node to the memory bank.
33. Endif.

/* DELETION : CASE 2 */
34. If (case = 2) then
35.     If ( parent → LCHILD = ptr ) then
36.         If ( ptr → LCHILD = NULL) then
37.             parent → LCHILD = ptr → RCHILD
38.         Else
39.             parent → LCHILD = ptr → LCHILD
```

```
40.         Endif
41.      Else
42.         If (parent →RCHILD = ptr) then
43.            if ( ptr →LCHILD = NULL) then
44.               parent →RCHILD = ptr→ RCHILD
45.            Else
46.               parent →RCHILD = ptr→ LCHILD
47.            EndIf
48.         Endif
49.      Endif
50.   ReturnNode(ptr)
51. Endif.
      /* DELETION : CASE 3 */
52.   If (case=3)
53.      ptr1 = succ(ptr) // Find the inorder successor of the node
54.      item1 = ptr1 →DATA
55.      Delete_BST (item1) // Delete the Inorder Successor.
56.      ptr→DATA = item1 // Replace the data with data of the inorder
                            successor.
57. Endif
58 Stop
```



Note that we assume the function succ(ptr) which return pointer to the Inorder Successor of the node ptr.

It can be verified that the inorder successor of ptr always occurs in the right subtree of ptr, and the inorder successor of ptr does not have a left child.

ALGORITHM SUCC.

Input: Pointer to a node PTR whose inorder successor is to be found.

output: Pointer to the inorder successor of ptr.

Data Structure: Linked Structure of Binary Tree.

Algo. Succ.

Step

1. ptr1 = PTR → RCHILD // Move to the right subtree
2. if (ptr1 ≠ NULL) then // If the right subtree is not empty
3. while (ptr1 → LCHILD ≠ NULL) do // Move to the leftmost end
4. ptr1 = ptr1 → LCHILD
5. Endwhile
6. Endif
7. Return (ptr1) // Return the pointer to the Inorder successor
8. Stop