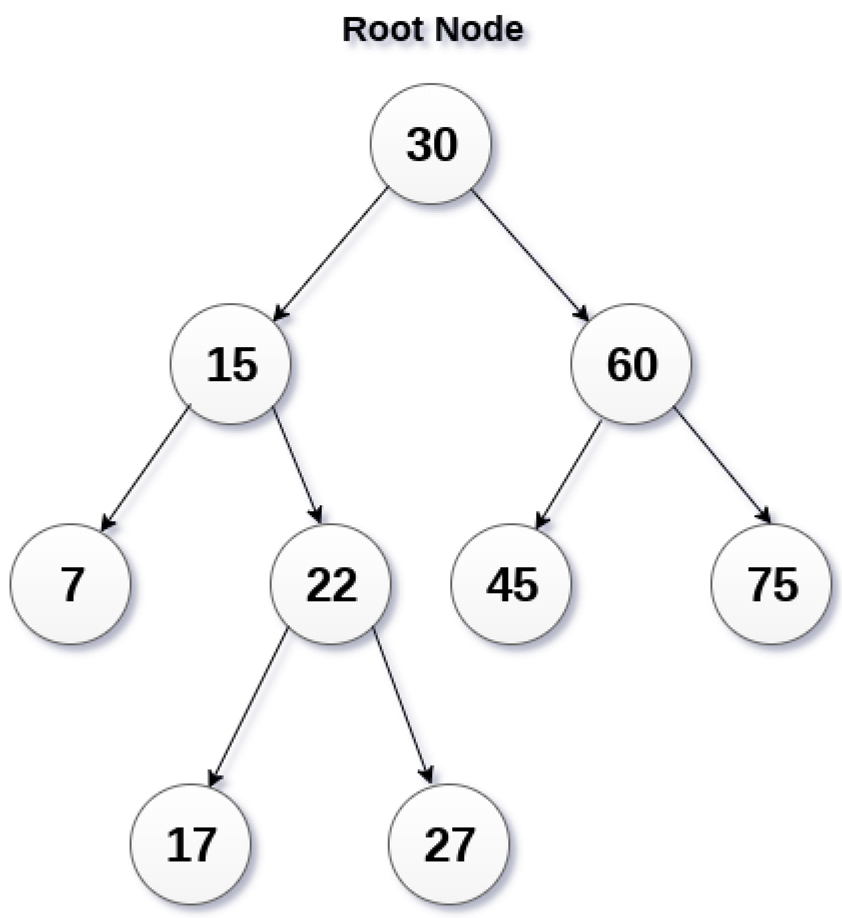
DATA STRUCTURES

UNIT-3

Binary Search Trees

Binary Search Tree

- Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
- In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
- Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
- This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

Advantages of using Binary Search Tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.

2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2 n$) time. In worst case, the time it takes to search an element is 0(n).

3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

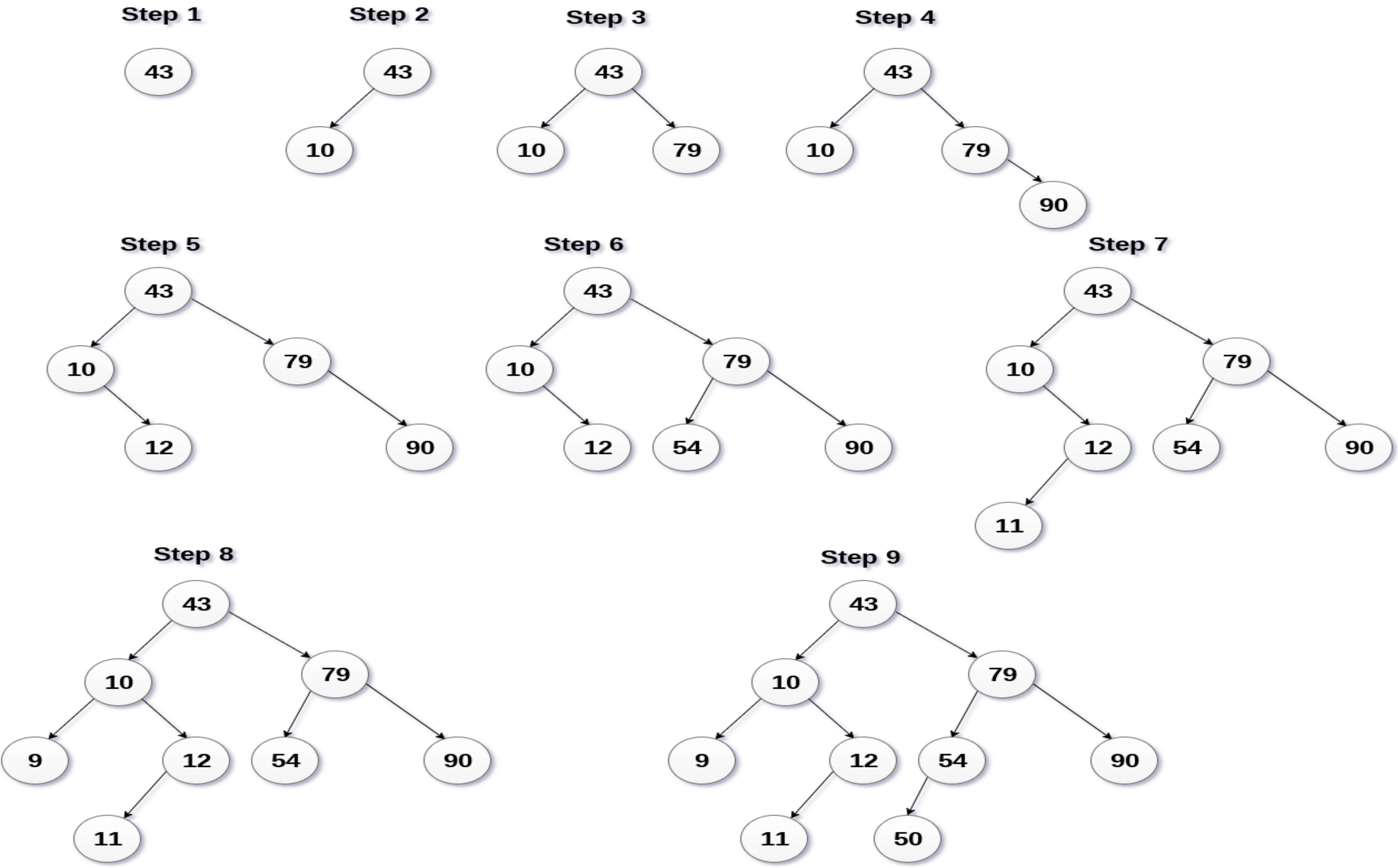
Create the binary search tree

Create the binary search tree using the following data elements.
43, 10, 79, 90, 12, 54, 11, 9, 50

1.Insert 43 into the tree as the root of the tree.

2.Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.

3.Otherwise, insert it as the root of the right of the right sub-tree.



Binary search Tree Creation

Operations on Binary Search Tree

Operation	Description
Searching in BST	Finding the location of some specific element in a binary search tree.
Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

Searching in BST

- Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order.

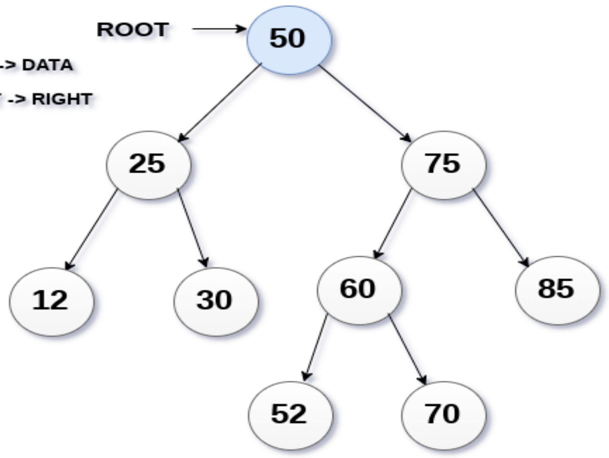
- Process:
  - Compare the element with the root of the tree.
  - If the item is matched then return the location of the node.
  - Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
  - If not, then move to the right sub-tree.
  - Repeat this procedure recursively until match found.
  - If element is not found then return NULL.

Example search in BST

Item = 60

ITEM > ROOT -> DATA

ROOT = ROOT -> RIGHT

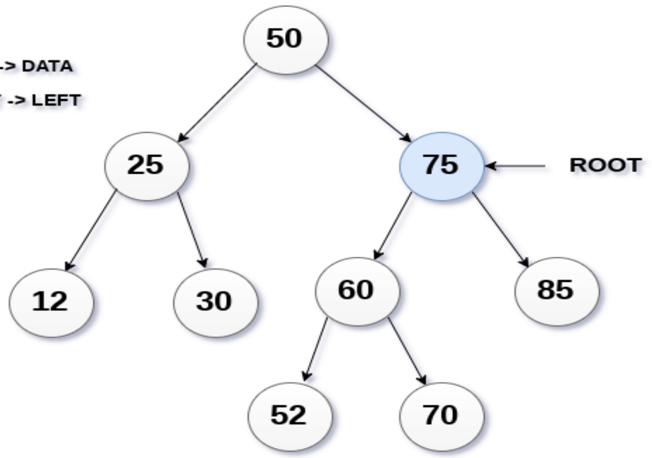


STEP 1

Item = 60

ITEM < ROOT -> DATA

ROOT = ROOT -> LEFT

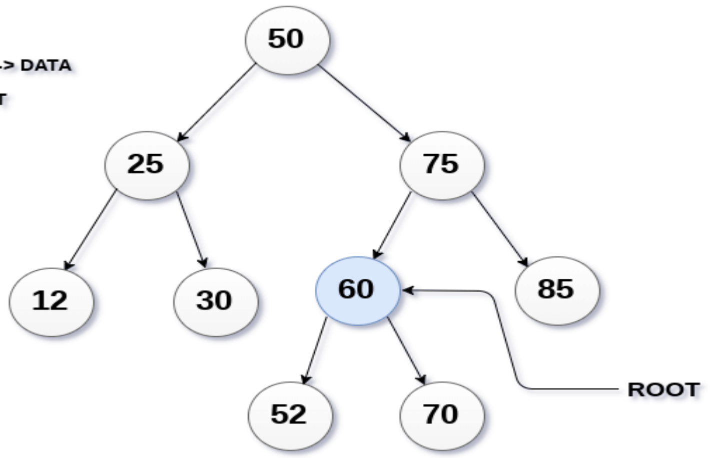


STEP 2

Item = 60

ITEM = ROOT -> DATA

RETURN ROOT



STEP 3

Algorithm Searching in BST

```
Algorithm Search (ROOT, ITEM)
{
    IF ROOT = NULL
          ?
    ELSE IF ITEM==ROOT -> DATA
                 ?
          ELSE   IF item < ROOT -> DATA
                              ?
                     ELSE
                              ?
}
```

Algorithm Searching in BST

```
Algorithm Search (ROOT, ITEM)
{
    IF ROOT = NULL
          Write not found; Return;
    ELSE IF ITEM==ROOT -> DATA
                 Write element is found.
          ELSE   IF item < ROOT -> DATA
                        search(ROOT -> LEFT, ITEM)
                 ELSE
                        search(ROOT -> RIGHT,ITEM)
}
```

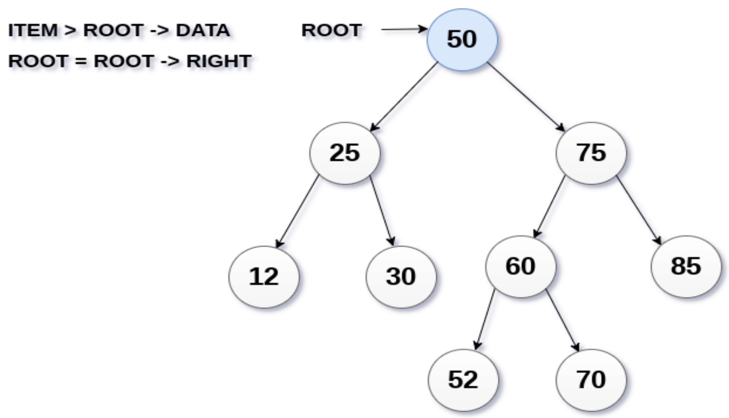
Insertion in BST

- Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

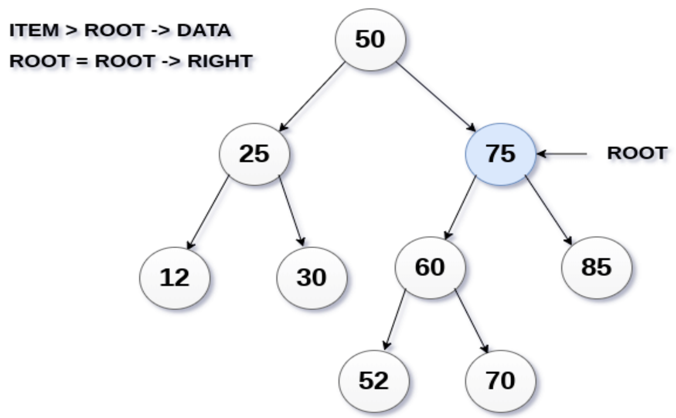
- Process of Insertion:
  - Allocate the memory for tree.
  - Set the data part to the value and set the left and right pointer of tree, point to NULL.
  - If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
  - Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
  - If this is false, then perform this operation recursively with the right sub-tree of the root.

Example for Insertion into BST

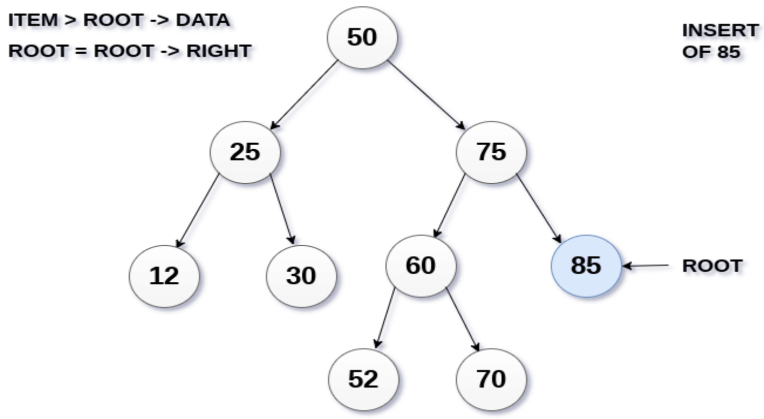
Item = 95



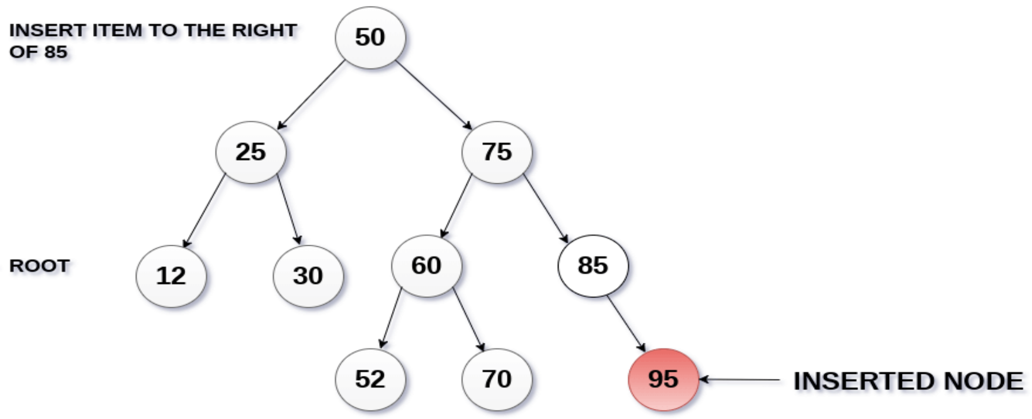
STEP 1



STEP 2



STEP 3



STEP 4

Algorithm for Insertion into BST

```
Algorithm Insert (root, item)
{
    IF root = NULL
    {

        ?
    }
    else
    {
        IF item < root -> data
            ?
        else
            ?
    }
}
```

Algorithm for Insertion into BST

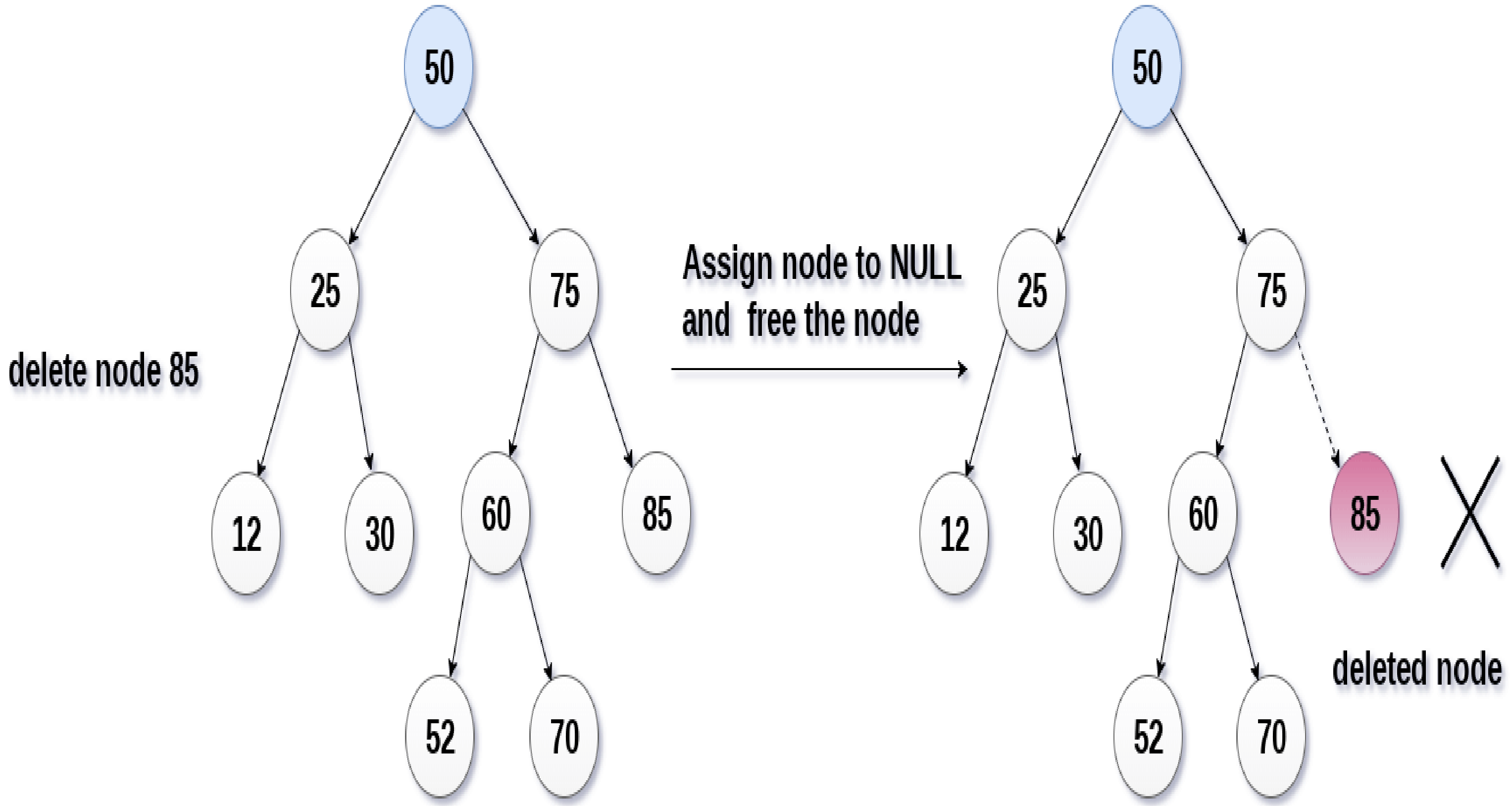
```
Algorithm Insert (root, item)
{
    IF root = NULL
    {
        Allocate memory to the new_node;
        new_node->data = item;
        new_node->left = new_node->right-> = NULL;
        root = new_node;
    }
    else
    {
        IF item < root -> data
            Insert(root->left, item);
        else
            Insert(root->right, item);
    }
}
```

Deletion in BST

- Delete function is used to delete the specified node from a binary search tree.
- However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate.
- There are three situations of deleting a node from binary search tree.
  - The node to be deleted is a leaf node
  - The node to be deleted has only one child.
  - The node to be deleted has two children.

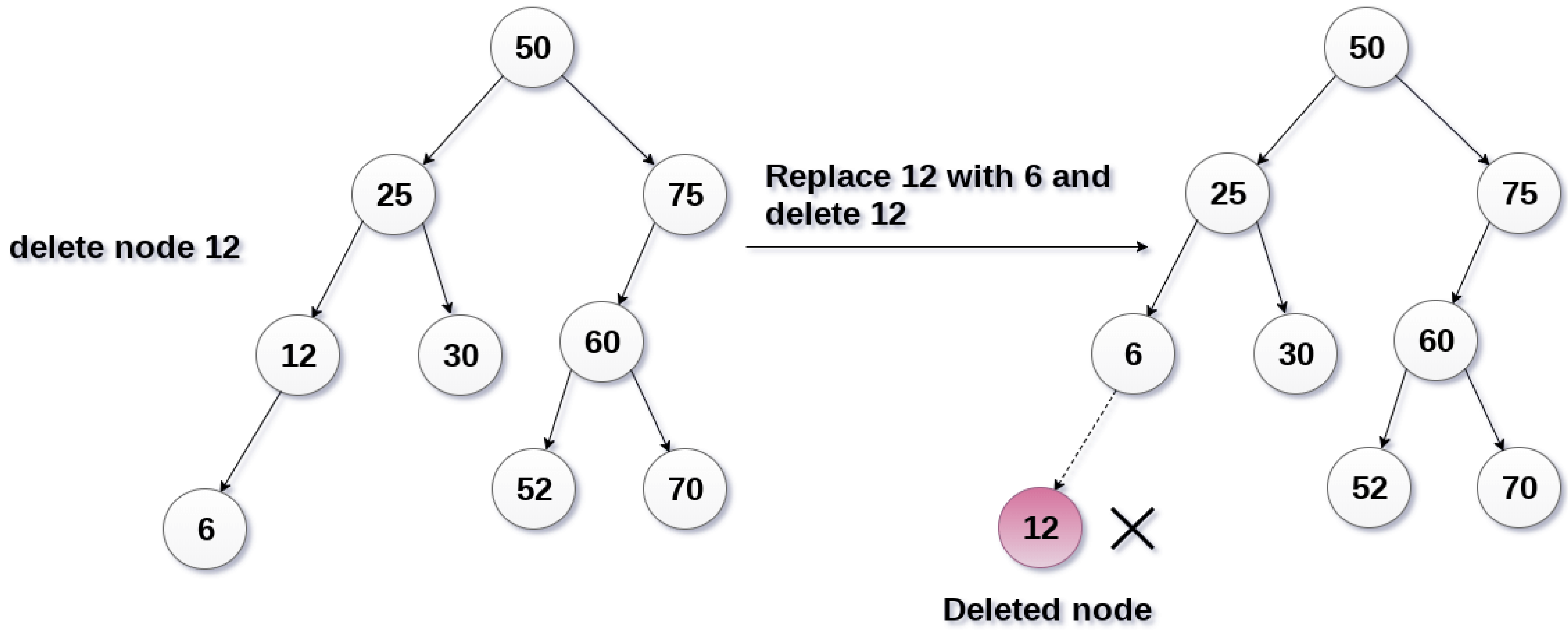
The node to be deleted is a leaf node

- It is the simplest case, in this case, replace the leaf node with the NULL and simply free the allocated space.



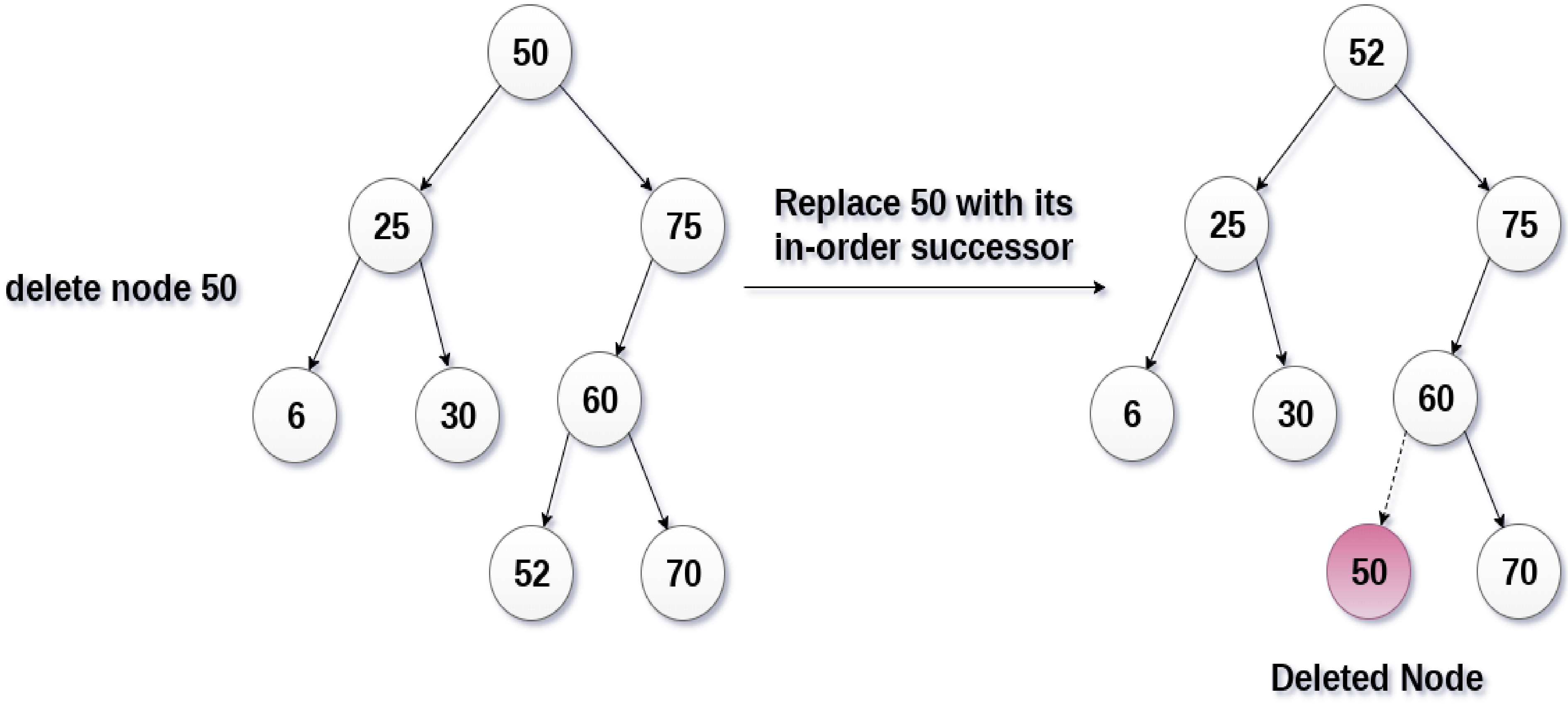
The node to be deleted has only one child

- In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.



The node to be deleted has two children.

The node which is to be deleted, is replaced with its in-order successor or predecessor After the procedure, replace the node with NULL and free the allocated space.



Algorithm for Deletion in BST

```
Algorithm deletion(Node *root, int item)
{
    Node* parent = NULL;
    Node* cur = root;
    locate (cur, item, parent);
    if (cur == NULL)
            ?
    if (cur->left == NULL && cur->right == NULL)
    {                                            // no child case
        if (cur != root)
         {
                  ?
         }
        else
                 ?
        free(cur);
    }
    Else if (cur->left && cur->right)//both child's case
        {
                          ?
         }
         else                    // single child case
         {
                          ?
         }
}
```

Algorithm for Deletion in BST

```
Algorithm deletion(Node *root, int item)
{
   Node* parent = NULL;
   Node* cur = root;
   locate (cur, item, parent);
   if (cur == NULL)
      write element not found; return;
   if (cur->left == NULL && cur->right == NULL)
   {                                    // no child case
      if (cur != root)
       {
         if (parent->left == cur)
            parent->left = NULL;
         else
            parent->right = NULL;
       }
      else  { root = NULL; }
      free(cur);
   }
Else if (cur->left && cur->right)//both childs case
     {
     Node* succ  = findMax (cur->left);
      int val = succ->data;
      deletion(root, succ->data);
      cur->data = val;
     }
     else                    // single child case
     {
         Node* child = (cur->left)? cur- >left: cur->right;
         if (cur != root)
         {
             if (cur == parent->left)
                  parent->left = child;
             else
                  parent->right = child;
         }
          else  {    root = child; }
     }
}
```

Supporting Algorithms for Deletion in BST

Algorithm locate(Node* cur, int item, Node* parent)

```
{
    while (cur != NULL && cur->data != item)
    {
        parent = cur;
        if (item < cur->data)
            cur = cur->left;
        else
            cur = cur->right;
    }
}
```

Algorithm Node* findMax(Node* x)

```
{
    while(x->right != NULL)
     {
            x = x->right;
    }
    return x;
}
```

Pro’s and Con’s of BST

- **Advantages :**
  - we can always keep **the cost of insert(), delete(), search() to O(logN) where N is the number of nodes in the tree** - so the benefit really is that search can be done in logarithmic time which matters a lot when N is large.
  - We have an **ordering of keys stored in the tree**. Any time we need to traverse the increasing (or decreasing) order of keys, we just need to do the in-order (and reverse in-order) traversal on the tree.
  - We can implement order statistics with binary search tree - **Nth smallest, Nth largest element and no.of values in the specified range.** This is because it is possible to look at the data structure as a sorted array.

- **Disadvantages:**
  - The **BST may not be balanced(degenerate tree)** hence the cost of operations may not be logarithmic and degenerate into a linear search on an array.

Applications of BST

Binary Search Tree Applications

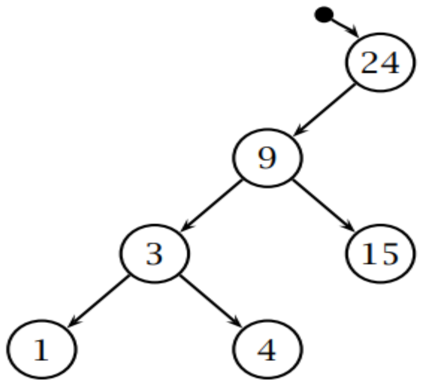
- BSTs are used for indexing and multi-level indexing in DATABASE.
- They are also helpful to implement various searching algorithms.
- It is helpful in maintaining a sorted stream of data.
- It is used to implement dictionary.
- It is used to implement Huffman Coding Algorithm.
- For managing virtual memory areas in Unix kernel.

Exercise on Binary Search Trees

- Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?
- ***a.*** 2, 252, 401, 398, 330, 344, 397, 363.
- ***b.*** 924, 220, 911, 244, 898, 258, 362, 363.
- ***c.*** 925, 202, 911, 240, 912, 245, 363.
- ***d.*** 2, 399, 387, 219, 266, 382, 381, 278, 363.
- ***e.*** 935, 278, 347, 621, 299, 362, 263, 363

Exercise on Binary Search Trees

- Consider the following binary search tree (BST).



- Question 1: List all the possible insertion orders (i.e., permutations) of the keys that could have produced this BST.
- Question 2: Draw the same BST after the insertion of keys: 6, 45, 32, 98, 55, and 69, in this order.
- Question 3: Draw the BST resulting from the deletion of keys 9 and 45 from the BST resulting from question 2.
- Question 4: Write at least three insertion orders (permutations) of the keys remaining in the BST after question 3 that would produce a binary search tree.

Exercise on Binary Search Trees

- Draw a binary search tree containing keys 8, 27, 13, 15, 32, 20, 12, 50, 29, 11, inserted in this order. Then, add keys 14, 18, 30, 31, in this order, and again draw the tree. Then delete keys 18, 29,12 and 27, in this order, and again draw the tree.

Useful links for Interview/GATE Preparation

- https://academyera.com/binary-search-tree-data-structures-psu-topic-wise-solved-questions
- https://www.btechonline.org/2013/01/gate-questions-data-structures-trees.html
- https://www.gatevidyalay.com/data-structures/