ST Implementation And Operations With

Detailed Tutorial on Binary Search Tree (BST) In C++ Including Operations, C++ Implementation, Advantages, and Example Programs:

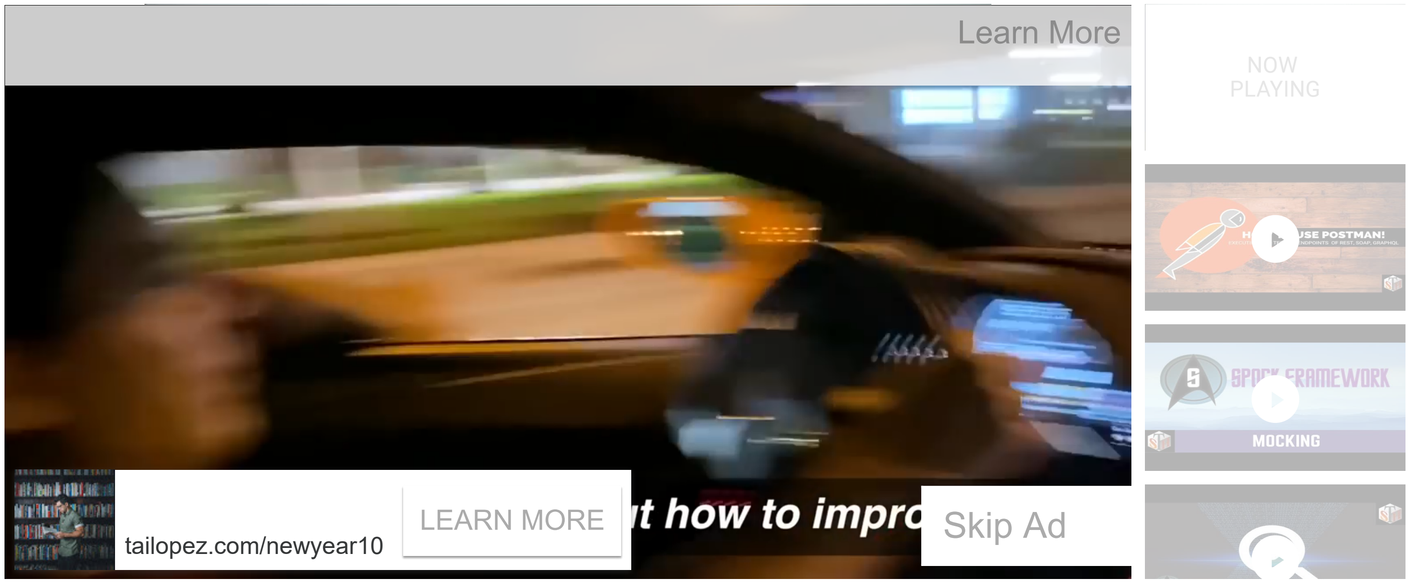
A Binary Search Tree or BST as it is popularly called is a binary tree that fulfills the following conditions:

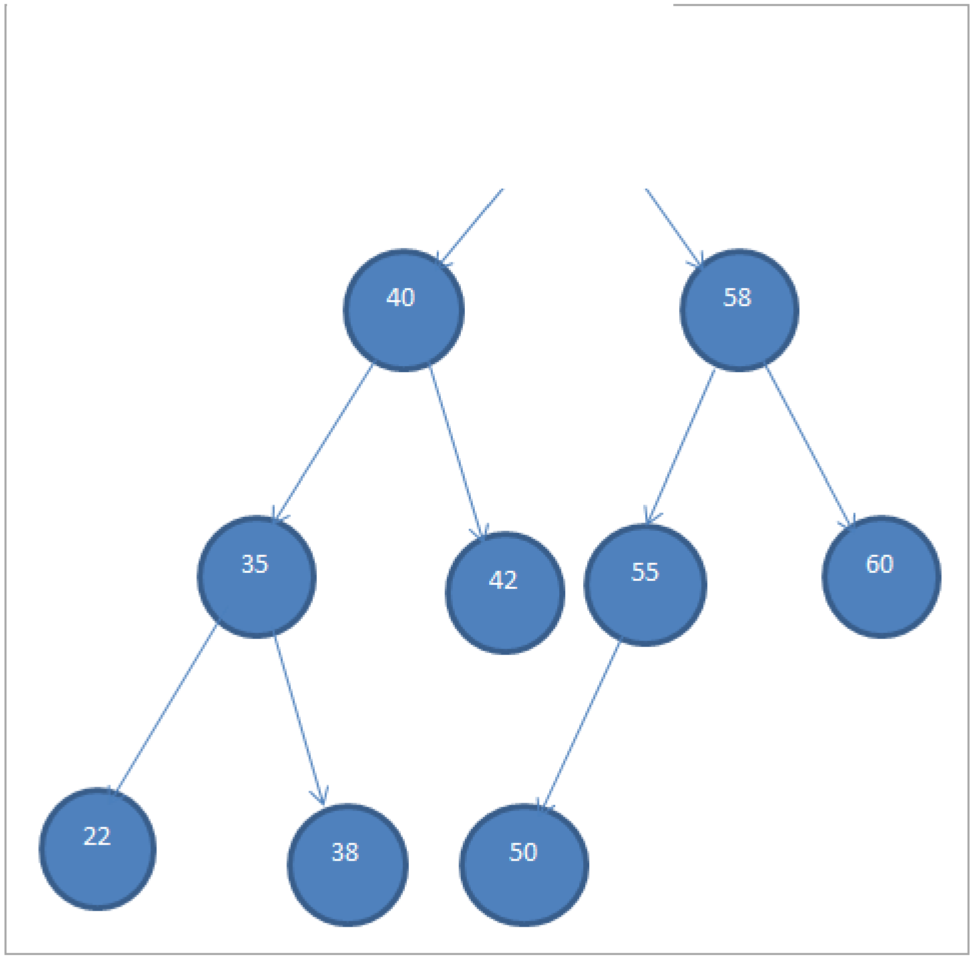
1. The nodes that are lesser than the root node which is placed as left children of the BST.
2. The nodes that are greater than the root node that is placed as the right children of the BST.
3. The left and right subtrees are in turn the binary search trees.

This arrangement of ordering the keys in a particular sequence facilitates the programmer to carry out operations like searching, inserting, deleting, etc. more efficiently. If the nodes are not ordered, then we might have to compare each and every node before we can get the operation result.

=> **Check The Complete C++ Training Series Here.**

FEATURED VIDEOS





Binary Search Trees are also referred to as “Ordered Binary Trees” because of this specific ordering of nodes.

subtree has nodes that are less than the root i.e. 45 while the right subtree has

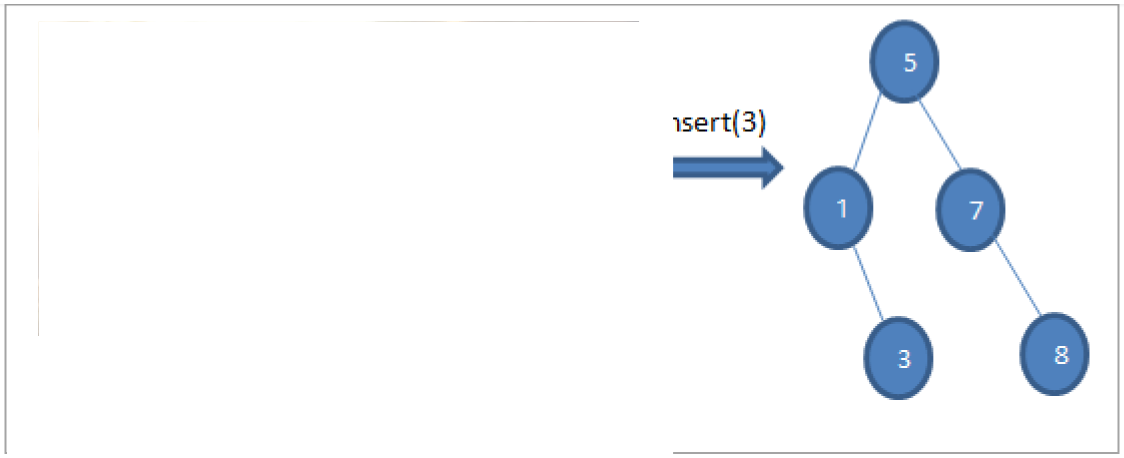
BST.

search tree.

rt operation is given below.

```
Insert(data)
Begin
          If node == null
                     Return createNode(data)
          If(data >root->data)
                    Node->right = insert(node->left,data)
          Else If(data < root->data)
                    Node->right = insert(node>right,data)
          Return node;
end
```

As shown in the above algorithm, we have to ensure that the node is placed at the appropriate position so that we do not violate the BST ordering.



ꜱ, we make a series of insert operations. After comparing the key to be inserted with the root node, the left or right subtree is chosen for the key to be inserted as a leaf node at the appropriate position.

#2) Delete

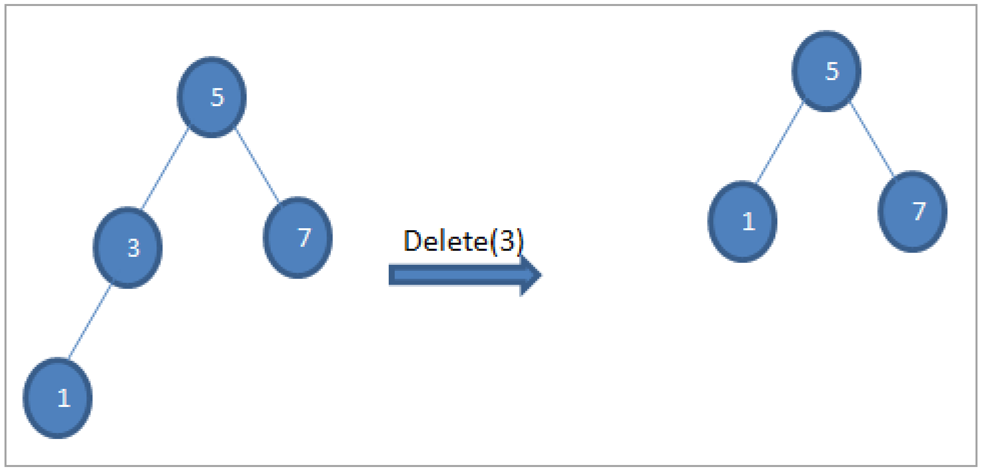
Delete operation deletes a node that matches the given key from BST. In this operation as well, we have to reposition the remaining nodes after deletion so that the BST ordering is not violated.

Hence depending on which node we have to delete, we have the following cases for deletion in BST:

#1) When the node is a Leaf Node

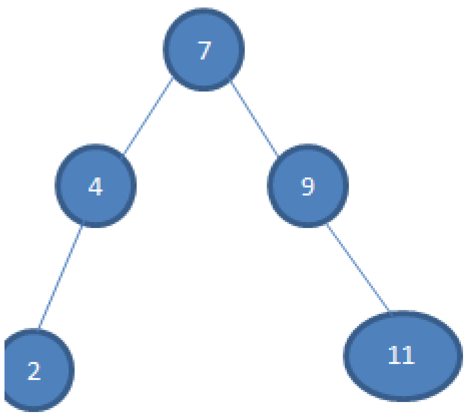
When the node to be deleted is a leaf node, then we directly delete the node.

When the node to be deleted has only one child, then we copy the child into the node and delete the child.



#3) When the node has Two Children

If the node to be deleted has two children, then we find the inorder successor for the node and then copy the inorder successor to the node. Later, we delete the inorder successor.



wo children, we first find the inorder successor for that node to be deleted. We find the inorder successor by finding the minimum value in the right subtree. In the above case, the minimum value is 7 in the right subtree. We copy it to the node to be deleted and then delete the inorder successor.

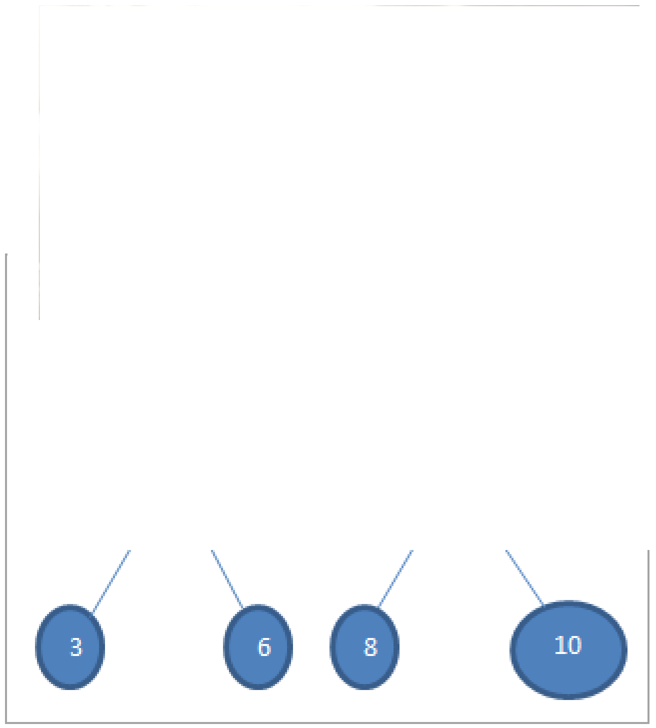
#3) Search

The search operation of BST searches for a particular item identified as “key” in the BST. The advantage of searching an item in BST is that we need not search the entire tree. Instead because of the ordering in BST, we just compare the key to the root.

If the key is the same as root then we return root. If the key is not root, then we compare it with the root to determine if we need to search the left or right subtree. Once we find the subtree, we need to search the key in, and we recursively search for it in either of the subtrees.

Following is the algorithm for a search operation in BST.

```
Search(key)
Begin
      If(root == null || root->data == key)
```



If we want to search a key with value 6 in the above tree, then first we compare the key with root node i.e. if (6==7) => No if (6<7) =Yes; this means that we will omit the right subtree and search for the key in the left subtree.

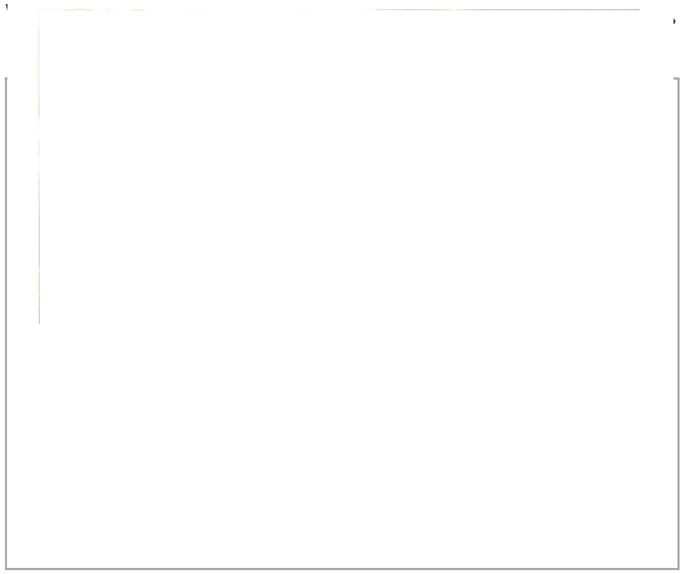
Next we descent to the left sub tree. If (6 == 5) => No.

If (6 < 5) => No; this means 6 >5 and we have to move rightwards.

If (6==6) => Yes; the key is found.

#4) Traversals

We have already discussed the traversals for the binary tree. In the case of BST as well, we can traverse the tree to get inOrder, preorder or postOrder sequence. In fact, when we traverse the BST in Inorder01 sequence, then we get the sorted sequence.



The traversals for the above tree are as follows:

Inorder traversal (lnr): 3 5 6 7 8 9 10
Preorder traversal (nlr): 7 5 3 6 9 8 10
PostOrder traversal (lrn): 3 6 5 8 10 9 7

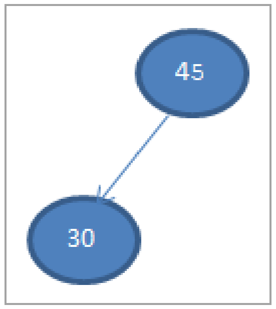
Illustration

Let us construct a Binary Search Tree from the data given below.

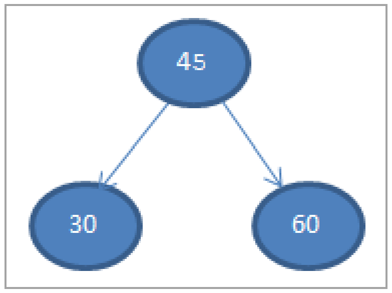
45 30 60 65 70

Let us take the first element as the root node.

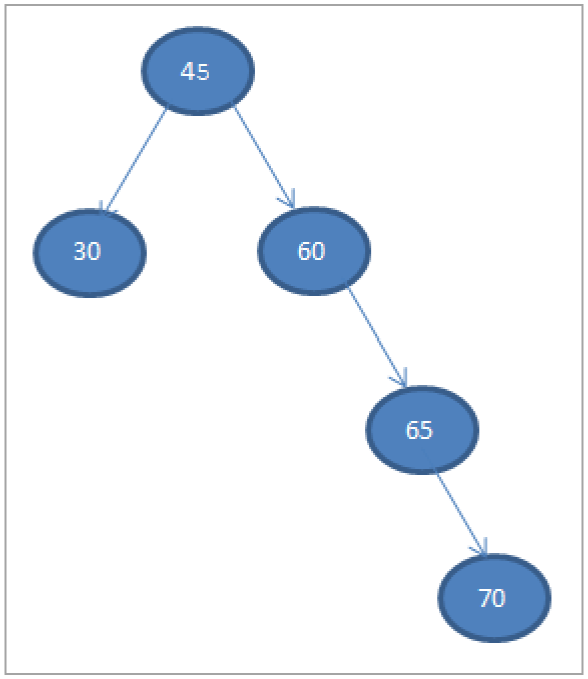
ıta according to the definition of Binary Search tree i.e. if data is less than the
: child and if the data is greater than the parent node, then it will be the right



#3) 60



#5) 70



When we perform the inorder traversal on the above BST that we just constructed, the sequence is as follows.

Home Resources FREE EBooks QA Testing ▾ Courses ▾ Automation ▾ Types Of Testing ▾ Tutorials ▾

ılements arranged in ascending order.

ation C++

sing C++ implementation.

```
//declaration for new bst node
struct bstnode
{
int data;
struct bstnode *left, *right;
};

// create a new BST node
struct bstnode *newNode(int key)
{
struct bstnode *temp =  new struct bstnode();
temp->data = key;
temp->left = temp->right = NULL;
return temp;
}

// perform inorder traversal of BST
void inorder(struct bstnode *root)
{
if (root != NULL)
    {
inorder(root->left);
cout<<root->data<<" ";
inorder(root->right);
    }
}
```

```
ıode* node, int key)

ıode
:ey);

ıe proper place to insert new node

:ey);

 key);



ılue
ıct bstnode* node)
struct bstnode* current = node;

    //search the leftmost tree
while (current && current->left != NULL)
current = current->left;

return current;
}
  //function to delete the node with given key and rearrange the root
struct bstnode* deleteNode(struct bstnode* root, int key)
{
    // empty tree
if (root == NULL) return root;

    // search the tree and if key < root, go for lefmost tree
if (key < root->data)
root->left = deleteNode(root->left, key);

    // if key > root, go for rightmost tree
else if (key > root->data)
root->right = deleteNode(root->right, key);

    // key is same as root
else
    {
        // node with only one child or no child
if (root->left == NULL)
```

```
get successor and then delete the node
le(root->right);

sor's content to this node


essor
root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
// main program
int main()
{
    /* Let us create following BST
              40
             /  \
          30     60
                   \
                    65
                      \
                       70*/
struct bstnode *root = NULL;
root = insert(root, 40);
root = insert(root, 30);
root = insert(root, 60);
root = insert(root, 65);
root = insert(root, 70);

cout<<"Binary Search Tree created (Inorder traversal):"<<endl;
inorder(root);

cout<<"\nDelete node 40\n";
root = deleteNode(root, 40);
```

:

rch Tree:

30 60 65 70

In the above program, we output the BST in for in-order traversal sequence.

Advantages Of BST

#1) Searching Is Very Efficient

We have all the nodes of BST in a specific order, hence searching for a particular item is very efficient and faster. This is because we need not search the entire tree and compare all the nodes.

We just have to compare the root node to the item which we are searching and then we decide whether we need to search in the left or right subtree.

#2) Efficient Working When Compared To Arrays And Linked Lists

When we search an item in case of BST, we get rid of half of the left or right subtree at every step thereby improving the performance of search operation. This is in contrast to arrays or linked lists in which we need to compare all the items sequentially to search a particular item.

/hen compared to other data structures like linked lists and arrays.

follows:

dexing in database applications.
cts like a dictionary.
efficient searching algorithms.
quire a sorted list as input like the online stores.
ession using expression trees.

Conclusion

Binary search trees (BST) are a variation of the binary tree and are widely used in the software field. They are also called ordered binary trees as each node in BST is placed according to a specific order.

Inorder traversal of BST gives us the sorted sequence of items in ascending order. When BSTs are used for searching, it is very efficient and is done within no time. BSTs are also used for a variety of applications like Huffman's coding, multilevel indexing in databases, etc.

=> Read Through The Popular C++ Training Series Here.

Home Resources FREE EBooks QA Testing ▾ Courses ▾ Automation ▾ Types Of Testing ▾ Tutorials ▾

Recommended Reading

- Binary Tree Data Structure In C++
- AVL Tree And Heap Data Structure In C++
- B Tree And B+ Tree Data Structure In C++
- Basic Input/Output Operations In C++
- Basic I/O Operations in Java (Input/Output Streams)
- Trees In C++: Basic Terminology, Traversal Techniques & C++ Tree Types
- File Input Output Operations In C++
- Pointers And Pointer Operations In C++

Home Resources FREE EBooks QA Testing ▾ Courses ▾ Automation ▾ Types Of Testing ▾ Tutorials ▾

About SoftwareTestingHelp

Helping our community since 2006! Most popular portal for Software professionals with **100 million+ visits!** You will absolutely love our tutorials on Software Testing, Development, Software Reviews and much more!

Selenium Tutorials
QTP/UFT Tutorials
Quality Center QC Tutorials
LoadRunner Tutorials
JMeter Tutorials
JIRA Tutorials
VBScript Tutorials
Best Test Management Tools
Unix Tutorials
DevOps Tutorials
JAVA Tutorials
Python Tutorials
Free C++ Tutorials
101+ Interview Questions

Home Resources FREE EBooks QA Testing ▾ Courses ▾ Automation ▾ Types Of Testing ▾ Tutorials ▾

Home Resources FREE EBooks QA Testing ▾ Courses ▾ Automation ▾ Types Of Testing ▾ Tutorials ▾