Binary Search Tree

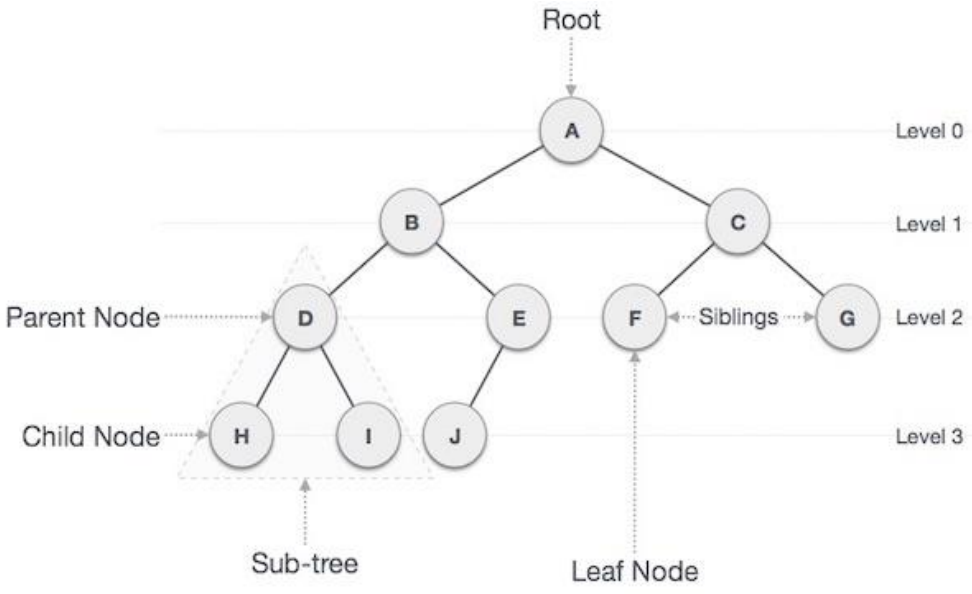
Binary Search Tree (BST), is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, we may have to compare every key to search a given key.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children.

A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

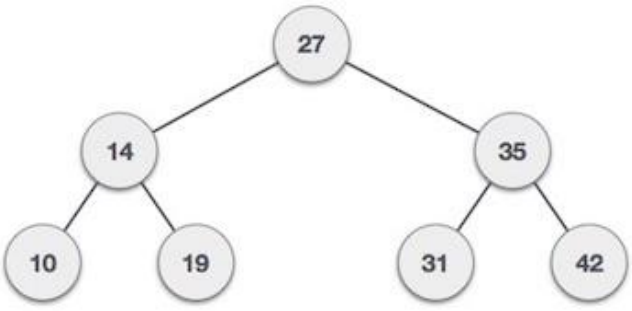


Important Terms

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



Binary Tree Node

A tree node has a data part and references to its left and right child nodes.

```
typedef int BType;

typedef struct node {
       BType data;
       struct node *leftchild;
       struct node *rightchild;

}bnode_t;
```

In a tree, all nodes share common construct.

Create a new BST node:

```
bnode_t *getBinaryNode(BType key)
{
       bnode_t *node = (bnode_t*)malloc(sizeof(bnode_t));
       node->leftchild = NULL;
       node->rightchild = NULL;
       node->data = key;
       return(node);

}
```

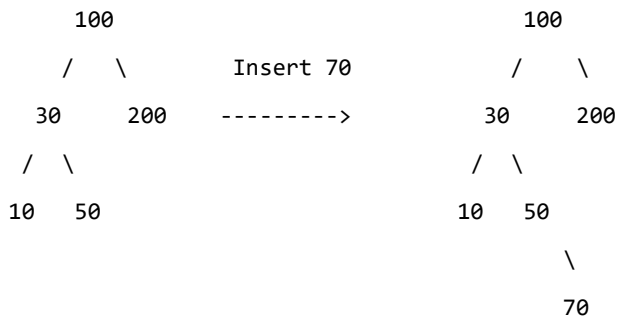
BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

Insert Operation

The first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data. In other words, once a leaf node is found, the new node is added as a child of the leaf node.



Algorithm

```
If root is NULL
   then create root node
Otherwise If root exists then
   compare the data with node.data
   while until insertion position is located
      If data is greater than node.data
         goto right subtree
      else   goto left subtree
   endwhile
   insert data
end If
```

Implementation

```
bnode_t * insert( bnode_t *root, BType key) {
	bnode_t *tempnode;
	bnode_t *current;
	bnode_t *parent;

	tempnode = getBinaryNode(key); //create a new node

	if (root == NULL) //if tree is empty, create root node
		root = tempnode;
	else {
		current = root;
		while (current!=NULL && current->data!=key) {
			parent = current;
			//go to left of the tree
			if (key < parent->data) {
				current = current->leftchild;
				//insert to the left
				if (current == NULL)
					parent->leftchild = tempnode;

			}
			//go to right of the tree
			else {
				current = current->rightchild;
				//insert to the right
				if (current == NULL)
					parent->rightchild = tempnode;

			}
		}
	}
	return(root);
}
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
   return root
else
   while data not found
      If data is greater than node.data
         goto right subtree
      else
         goto left subtree
      If data found
         return node
   endwhile
   return data not found
end if
```

Implementation

```
bnode_t *search(bnode_t *root, BType key) {
       bnode_t *current = root;
       printf("Visiting elements: ");

       while (current!=NULL && current->data != key) {
              printf("%d ", current->data);
              //go to left tree
              if (current->data > key)
                     current = current->leftchild;
              //else go to right tree
              else
                     current = current->rightchild;
       }
       return current;
}
```

Example: Write the recursive implementation of insert function.

```
bnode_t *insertRec(bnode_t *node, BType key){

	/* If the node is empty, return a new node */
	if (node == NULL) return getBinaryNode(key);
	else /* Recur down the tree */
		if (key < node->data)
			node->leftchild = insertRec(node->leftchild, key);
		else if (key > node->data)
			node->rightchild = insertRec(node->rightchild, key);
	return node;
}
```

Example: Write the recursive implementation of search function.

```
bnode_t *searchRec(bnode_t *root, BType key) {

	if (root == NULL || root->data == key)
		return(root);
	else {
		printf("%d ", root->data);
		if (root->data > key) //go to left tree
			return(searchRec(root->leftchild, key));
		else //else go to right tree
			return(searchRec(root->rightchild, key));
	}
}
```

Example: Write a function to find the minimum value of the given binary tree.

```
bnode_t *minValueNode(bnode_t * node){

	bnode_t * current = node;
	/* loop down to find the leftmost leaf */
	while (current->leftchild != NULL)
		current = current->leftchild;
	return current;
}
```

Example: Write a function to delete the given key from the given binary search tree.

```
bnode_t *deleteNode(bnode_t *root, BType key){

	bnode_t *temp;
	if (root == NULL) return root;
	// If the key to be deleted is smaller than the root's key,
	// then it lies in left subtree
	if (key < root->data)
		root->leftchild = deleteNode(root->leftchild, key);

	// If the key to be deleted is greater than the root's key,
	// then it lies in right subtree
	else   if (key > root->data)
		root->rightchild = deleteNode(root->rightchild, key);

	// if key is same as root's key, then This is the node
	// to be deleted
		else
		{
			// node with only one child or no child
			if (root->leftchild == NULL){
				temp = root->rightchild;
				free(root);
				return temp;
			}
			else if (root->rightchild == NULL){
				temp = root->leftchild;
				free(root);
				return temp;
			}

			// node with two children: Get the inorder successor (smallest
			// in the right subtree)
			temp = minValueNode(root->rightchild);

			// Copy the inorder successor's content to this node
			root->data = temp->data;

			// Delete the inorder successor
			root->rightchild = deleteNode(root->rightchild, temp->data);
		}
		return root;
}
```

Binary Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree:

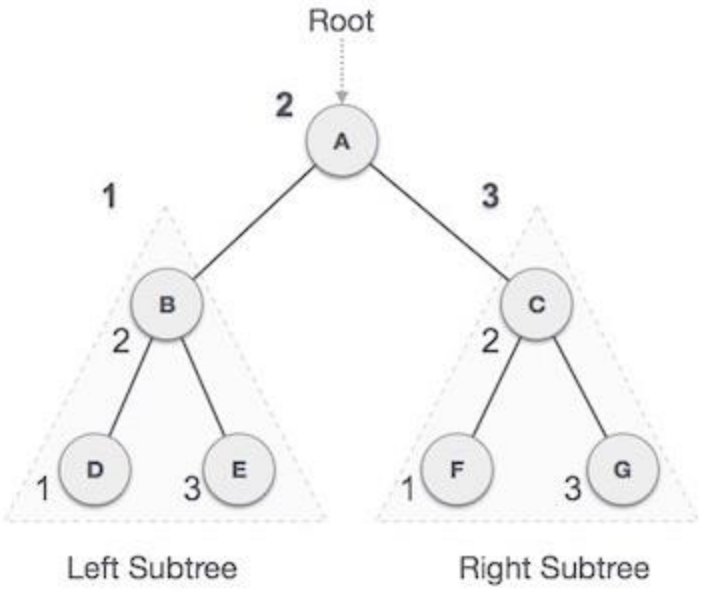
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be:

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

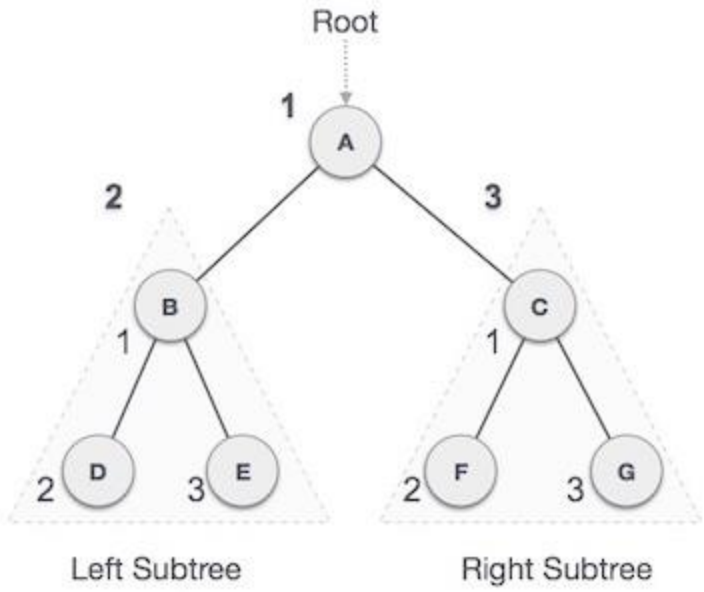
Until all nodes are traversed:
Step 1 – Recursively traverse left subtree.
Step 2 – Visit root node.
Step 3 – Recursively traverse right subtree.

Implementation

```
void inOrderTraversal(bnode_t *root) {
	if (root != NULL) {
		inOrderTraversal(root->leftchild);
		printf("%d ", root->data);
		inOrderTraversal(root->rightchild);
	}
}
```

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be:

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

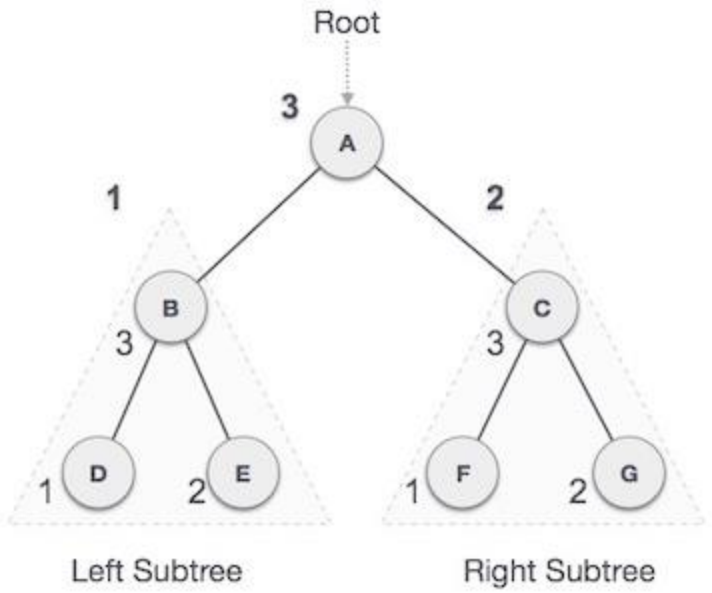
Until all nodes are traversed:
Step 1 – Visit root node.
Step 2 – Recursively traverse left subtree.
Step 3 – Recursively traverse right subtree.

Implementation

```
void preOrderTraversal(bnode_t *root) {
	if (root != NULL) {
		printf("%d ", root->data);
		preOrderTraversal(root->leftchild);
		preOrderTraversal(root->rightchild);
	}
}
```

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
Step 2 – Recursively traverse right subtree.
Step 3 – Visit root node.

Implementation

```
void postOrderTraversal(bnode_t *root) {
	if (root != NULL) {
		postOrderTraversal(root->leftchild);
		postOrderTraversal(root->rightchild);
		printf("%d ", root->data);
	}
}
```