

## Lecture-18

### Binary Search Tree (BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

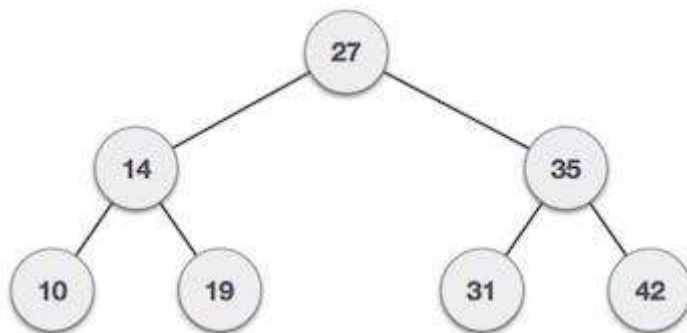
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

$$\text{left\_subtree (keys)} \leq \text{node (key)} \leq \text{right\_subtree (keys)}$$

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree –

- **Search** – Searches an element in a tree.
- **Insert** – Inserts an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {  
    int data;  
    struct node *leftChild;  
    struct node *rightChild;  
};
```

### 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

```
struct node* search(int data){  
    struct node *current = root;  
    printf("Visiting elements: ");  
  
    while(current->data != data){  
  
        if(current != NULL) {  
            printf("%d ",current->data);  
  
            //go to left tree  
            if(current->data > data){  
                current = current->leftChild;  
            } //else go to right tree  
            else {  
                current = current->rightChild;  
            }  
  
            //not found
```

```

        if(current == NULL){
            return NULL;
        }
    }
}

return current;
}

```

### **Insert Operation**

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.

### **Algorithm**

```

void insert(int data) {
    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;

    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty
    if(root == NULL) {
        root = tempNode;
    } else {
        current = root;
        parent = NULL;
    }
}

```

```
while(1) {  
    parent = current;  
  
    //go to left of the tree  
    if(data < parent->data) {  
        current = current->leftChild;  
        //insert to the left  
  
        if(current == NULL) {  
            parent->leftChild = tempNode;  
            return;  
        }  
    } //go to right of the tree  
    else {  
        current = current->rightChild;  
  
        //insert to the right  
        if(current == NULL) {  
            parent->rightChild = tempNode;  
            return;  
        }  
    }  
}  
}
```