DATA STRUCTURE - BINARY SEARCH TREE

http://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm Copyright © tutorialspoint.com

A binary search tree BST is a tree in which all nodes follows the below mentioned properties −

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

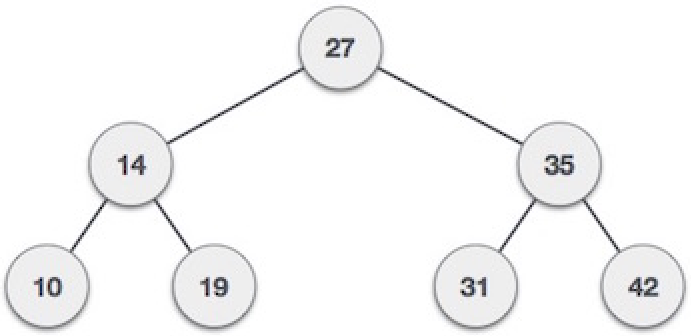
Thus, a binary search tree BST divides all its sub-trees into two segments; *left* sub-tree and *right* sub-tree and can be defined as −

```
left_subtree (keys)  ≤  node (key)  ≤  right_subtree (keys)
```

Representation

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

An example of BST −



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

Basic Operations

Following are basic primary operations of a tree which are following.

- **Search** − search an element in a tree.
- **Insert** − insert an element in a tree.
- **Preorder Traversal** − traverse a tree in a preorder manner.
- **Inorder Traversal** − traverse a tree in an inorder manner.
- **Postorder Traversal** − traverse a tree in a postorder 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 search. Start search from root node then if data is less than key value, search element in left subtree otherwise search element in right subtree. Follow the same algorithm for each node.

```
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 search from root node then if data is less than key value, search empty location in left subtree and insert the data. Otherwise search empty location in right subtree and insert the data.

```
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;
            }
         }
      }
   }
}
```

Loading [MathJax]/jax/output/HTML-CSS/jax.js