# CSCI-2270 Data Structures Recitation 2 Instructors: Hoenigman/Zagrodzki/Zietz

**Learning Objectives:**

- Binary Trees
- Binary search trees
- Insertion and Search
- Traversals

## 1 Trees

**Definition:** A *tree* is a finite set of one or more nodes such that

- There is a specially designated node called *root*.
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1,\ T_2,\ ...,\ T_n$ where each of these sets is a tree. $T_1,\ T_2,\ ...,\ T_n$ are called *subtrees* of the root.

Few more things to know-

- *degree* - The number of subtrees of a node is called its *degree.*
- *terminal nodes or leaf node*- Nodes that have degree 0.
- *degree of a tree*- Maximum of the degrees of all the nodes.
- *Ancestors* of a node are all the nodes along the path from root to that node.
- *Level* Level of the root is assumed to be *1* or *O*. If a node is at level $l$ then its children are at level $l+1$.

## 2 Binary Tree

In binary tree no node has degree more than two. A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called left subtree and right subtrees.

```
        struct node{
            int key;
            node* left;
            node* right;
        }
```

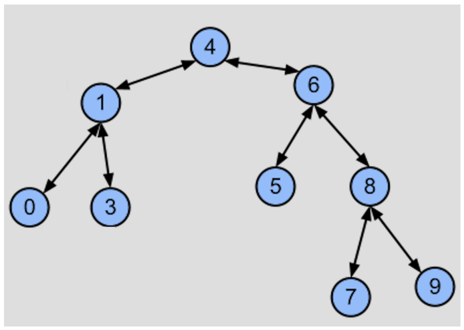

Figure 1: A Binary search tree

## 3 Binary Search Trees

A binary search tree is a binary tree (refer to figure 2). It may be empty. If not empty then it satisfies the following properties:

1. Each node has exactly one key and the keys in the tree are distinct.
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

### 3.1 Insert into the Binary tree

Remember the property of binary tree. Every value in the left subtree is smaller than the value at root and every value at right subtree is larger than the value at the root. So when we are inserting a new node in the tree we should maintain that property. Hence we should first compare the value of to be inserted with the current node. If the value is smaller than the value at the current node, move on to left subtree. If value is greater move on to the right subtree. Figure 3.1 depicts the process of inserting 2 in the tree presented in figure 2.

Here is the algorithm. Note depending on your implementation you may need to pass the node to the function and return a node from function

```
         insertNode( node* root , int data)
        {

            if(root == NULL)
                create a new node;

            if(data< root->key)
                insert the node at the left subtree;

            else if(data > root->key)
               insert the node at right subtree;

        }
```

### 3.2 Search in the tree

Remember searching in an array or linked list. We need to traverse the whole array to check presence of a particular element. Hence it may take $O(n)$ time. But using the binary search tree's property searching for a key in a binary search tree can be done in $O(logn)$ time. So the logic is given in the following pseudocode. Note depending on your implementation needs you may like to pass a node to the function. You may need to return the bool variable indicating the status of the presence or you may need to return the node pointer where you have found the data.

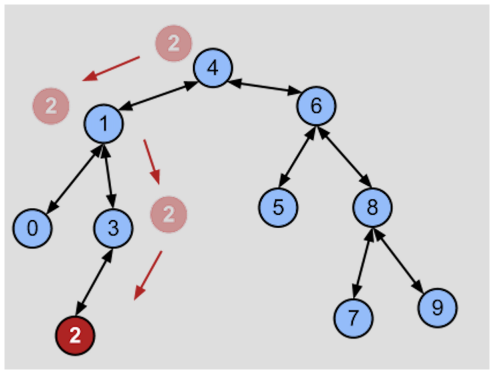

Figure 2: Insertion in a binary search tree

```
        searchKey( node* root , int data)
        {

            if(root == NULL)
                data is not present in the tree

            if(root->key ==  data )
                data is found in the tree;


            if( root->key < data )
                keep searching in the left subtree;

            if  (root->key > data)
                keep searching the right subtree;
        }
```

## 4 Traversal

There will be many situations when you may need to traverse the tree. For example you may like to print out the node values of the tree. There are many ways to traversing a tree. Here we will discuss three of them.

### 4.1 in-order traversal

In this traversal we will first print out the whole left subtree. We will print the root once we are done with the left subtree. Then we will move on to the right subtree. At each node we will follow this procedure. So-

1. print the left subtree of current node.
2. print the current node.
3. print the right subtree of current node.

So the in-order traversal for the tree given in Figure 2 is - `0, 1, 3, 4, 5, 6, 7, 8, 9`. **Can you see any interesting property here?**

```
        void inorderTraversal(node* root)
        {
            if(root!= NULL)
            {
                inorderTraversal(root->left);// traverse the left subtree
                cout << root->key; // process the root.
                inorderTraversal(root->right); // traverse the right subtree
            }
        }
```

### 4.2 Pre-order traversal

In this traversal we will process the root first. Then we will move on to left subtree. After that we will process the right subtree. So the order of processing is root- left- right. The pre-order traversal for the tree presented in Figure 2 will be- `4, 1, 0, 3, 6, 5, 8, 7, 9`.

```
    void preorderTraversal(node* root)
    {
        if(root!= NULL)
        {
            cout << root->key; // process the root.
            preorderTraversal(root->left);// traverse the left subtree
            preorderTraversal(root->right); // traverse the right subtree
        }
    }
```

### 4.3 Post-order traversal

In this traversal root will be processed at last. So first we will process left subtree. Then we will move on to right subtree. Once we are done with both left subtree and right subtree we will process the root. At each node of the tree we will follow this procedure. The post-order traversal for the tree presented in Figure 2 will be- `0, 3, 1, 5, 7, 9, 8, 6, 4`.

```
    void postorderTraversal(node* root)
    {
        if(root!= NULL)
        {
            postorderTraversal(root->left);// traverse the left subtree
            postorderTraversal(root->right); // traverse the right subtree
            cout << root->key; // process the root.
        }
    }
```

One nice application of post order traversal will be destroying the tree. Before we destroy the root we should destroy the left and right child of the root. We can do this processing following post order traversal.

## 5 Exercise

1. Download the cpp files.
2. Implement insert node function.
3. Implement search key function.

## 6 A small quiz

1. Where can you find the minimum value in a BST?
2. Where can you find the maximum value in a BST?
3. Let the root is at $0^{th}$ level. What is the maximum number of nodes that can be there at $i^{th}$ level?