ECE 220 Computer Systems & Programming

Lecture 23 – Trees: traversal and search

ECE ILLINOIS

ILLINOIS

Tree Data Structure

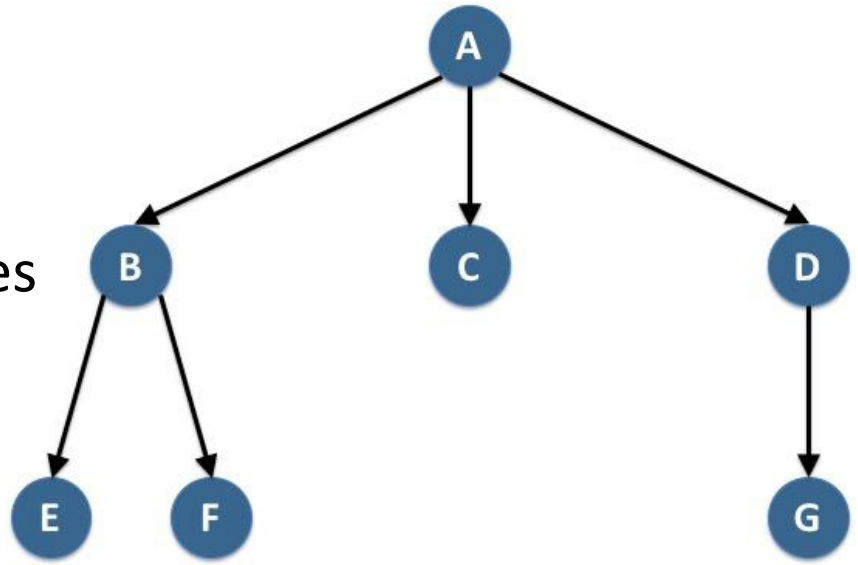
Array, linked list, stack, queue – linear data structures

Tree: A data structure that captures hierarchical nature of relations between data elements using a set of linked nodes. Nodes are connected by edges. It's a ***nonlinear*** data structure.

Tree Terminology:

root, internal node, external node (leaf),

parent, child, sibling, height, depth



The **depth** of a node is the number of edges from the node to the tree's root node.
A root node will have a depth of 0.

The **height** of a node is the number of edges on the *longest path* from the node to a leaf.
A leaf node will have a height of 0.

Common Operations on Tree:

- Locate an item
- Add a new item at a particular place
- Delete an item
- Remove a section of a tree (pruning)
- Add a new section to a tree (grafting)

Manually Creating a simple tree:

```
typedef struct nodeTag
{
    int data;
    struct nodeTag* left;
    struct nodeTag* right;
} t_node;
```

```
int main()
{
    /* manually create a simple tree */
    t_node *tree = NULL;
    tree = NewNode(10);
    tree->left = NewNode(5);
    tree->right = NewNode(-2);
    tree->left->left = NewNode(23);

    TraverseTree(tree);
    FreeTree(tree);
}
```

```
t_node* NewNode(int data)
{
    t_node* node;

    if ((node = (t_node *)malloc(sizeof(t_node))) != NULL)
    {
        node->data = data;
        node->left = NULL;
        node->right = NULL;
    }

    return node;
}
void TraverseTree(t_node *node)
{
    if (node != NULL)
    {
        printf("Node %d (address %p, left %p, right %p)\n",
                node->data, node, node->left, node->right);

        TraverseTree(node->left);
        TraverseTree(node->right);
    }
}
```

```
void FreeTree(t_node *node)
{
    if (node != NULL)
    {
        FreeTree(node->left);
        FreeTree(node->right);
        free(node);
    }
}
```

```
[ubhowmik@linux-a2 SourceCode]$ ./tree_basics
Node 10 (address 0x1476010, left 0x1476030, right 0x1476050)
Node 5 (address 0x1476030, left 0x1476070, right (nil))
Node 23 (address 0x1476070, left (nil), right (nil))
Node -2 (address 0x1476050, left (nil), right (nil))
```

Binary Tree

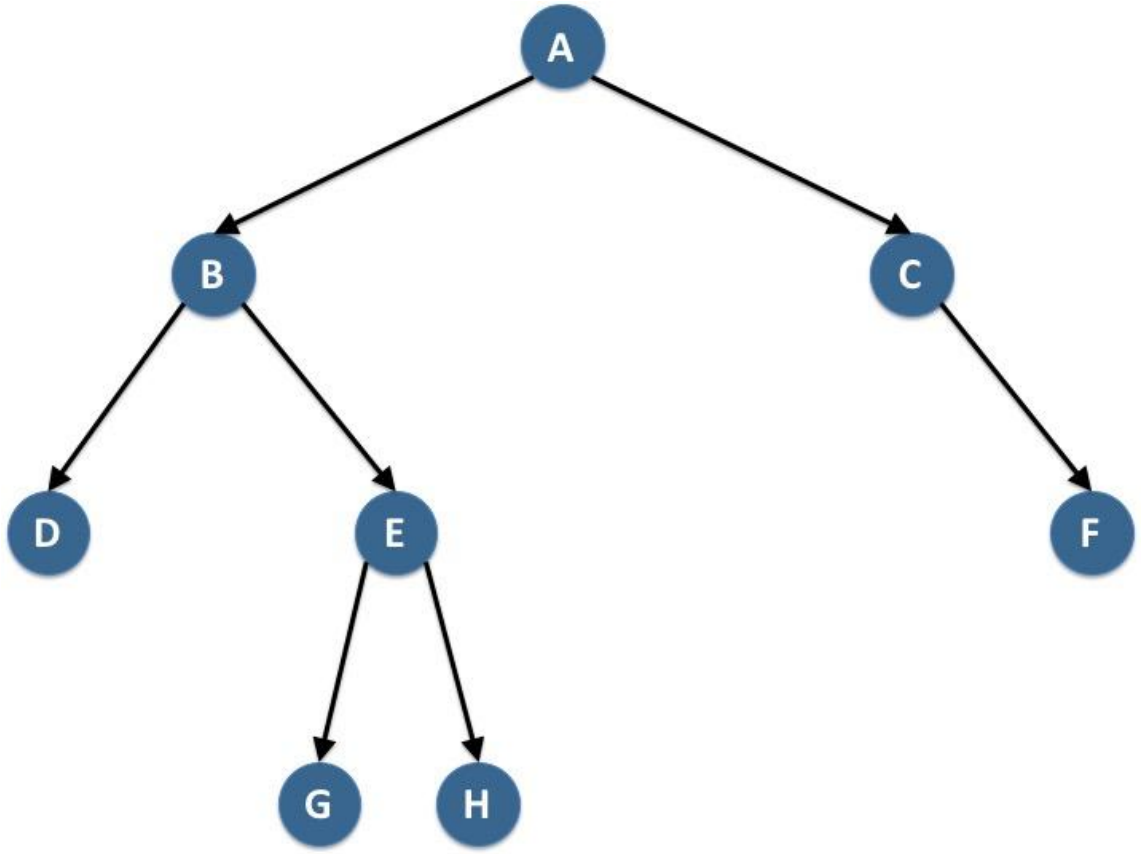
- Each node has at most 2 children – left child and right child

What is the height of the tree?

What is the depth of node E?

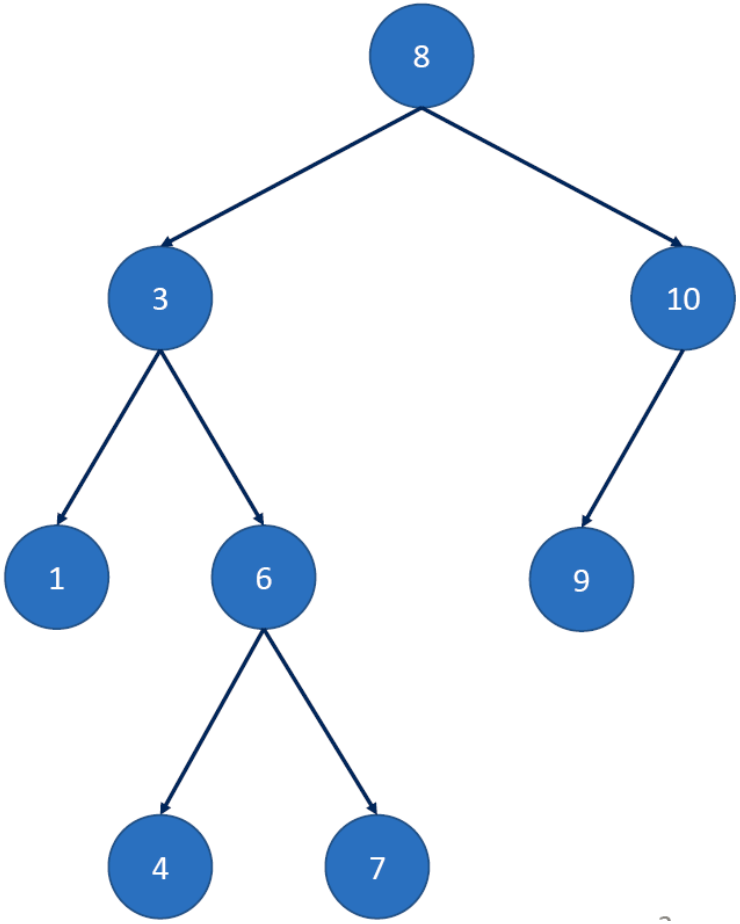
What is the height of node E?

Which nodes are leaves?



Binary Search Tree

- Data of nodes on the **left subtree** is **smaller** than the data of parent node
- Data of nodes on the **right subtree** is **larger** than the data of parent node
- Both left and right subtrees must also be BST
- Data in each node is unique



Insert a new node in the right place (BST)

```
t_node* InsertNode(t_node *node, int data)
{
    printf("Call InserNode-- node addree:%p, data:%d\n", node, data);
    //base case : Found a right place to insert the node.
    if(node ==NULL){
        node = NewNode(data);
        return node;
    }
    // recursive case: Traverse either to the left (new data is smaller)
        // or the right (new data is larger)
    else{
        if(data < node->data)
            node->left = InsertNode(node->left , data);
        else
            node->right = InsertNode(node->right , data);
        return node;
    }
}
```

Binary Search Tree

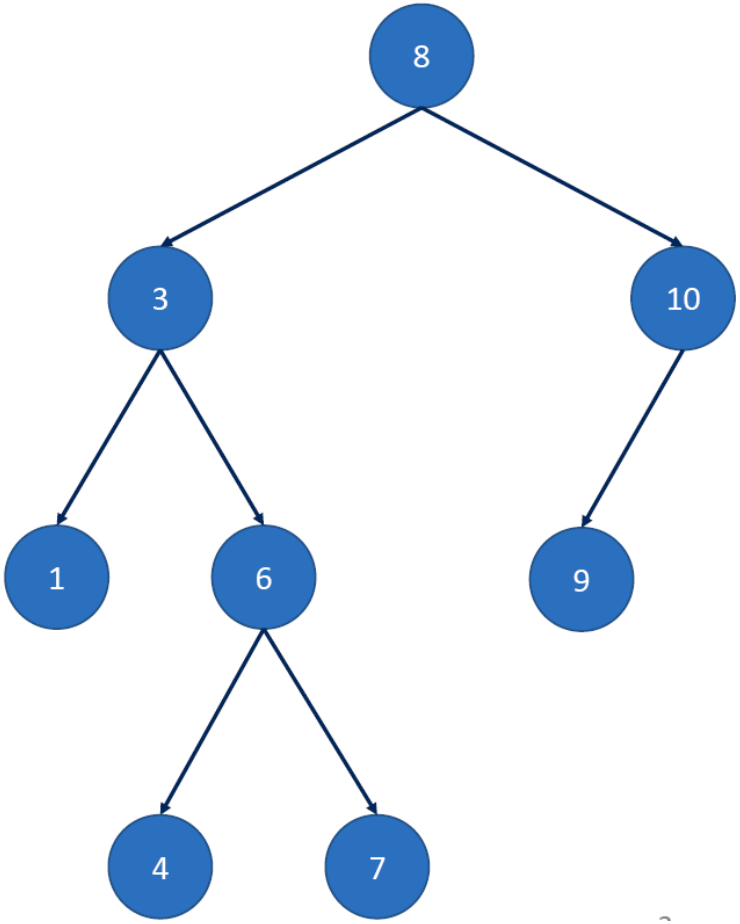
- Data of nodes on the **left subtree** is **smaller** than the data of parent node
- Data of nodes on the **right subtree** is **larger** than the data of parent node
- Both left and right subtrees must also be BST
- Data in each node is unique

What is the sequence of access for

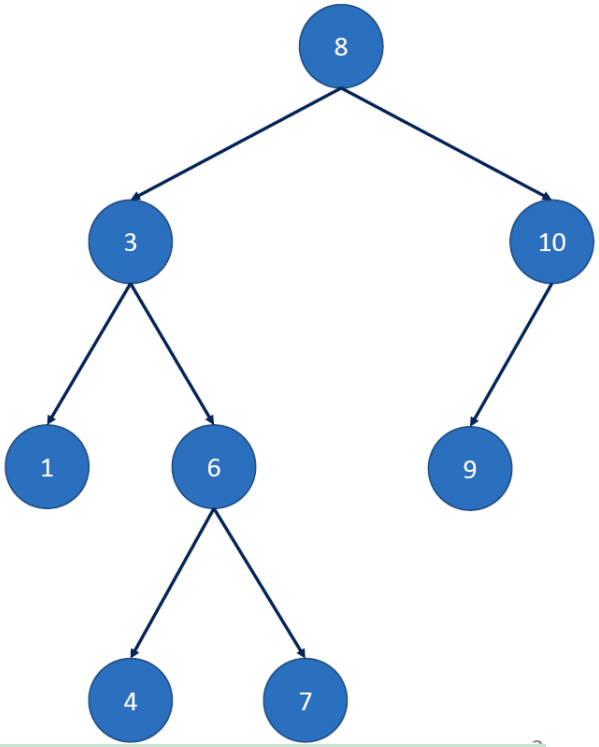
1. post-order traversal?

2. pre-order traversal?

3. in-order traversal?



Traverse a BST (postorder)

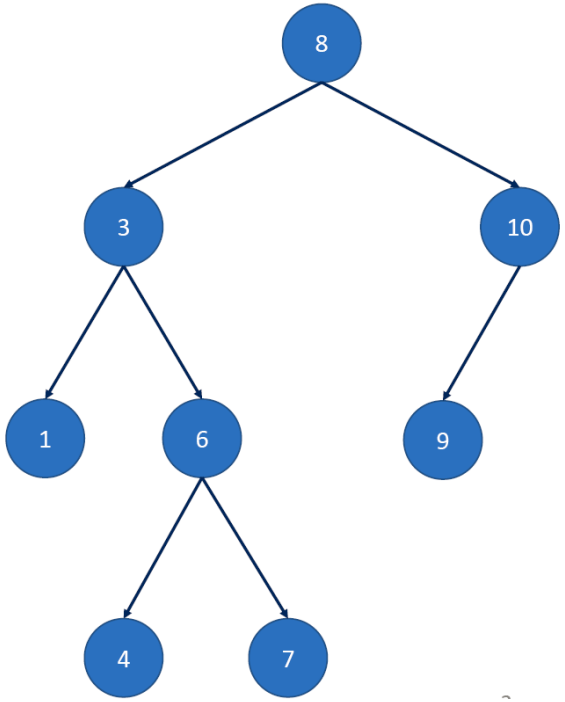



```
void postOrderTraversal(t_node *node)
{
    if (node != NULL)
    {
        postOrderTraversal(node->left);
        postOrderTraversal(node->right);
        printf("Node %d (address %p, left %p, right %p)\n",
                node->data, node, node->left, node->right);
    }
}
```

1 4 7 6 3 9 10 8

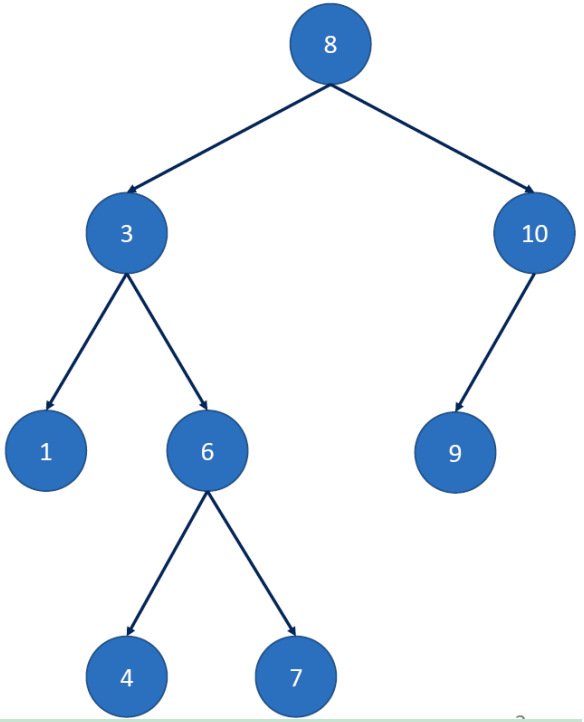
Traverse a BST (preOrder)

```
/* Pre-order
   Display the data part of the current node
   Traverse the left subtree by recursively calling the pre-order function
   Traverse the right subtree by recursively calling the pre-order function
*/
void preOrderTraversal(t_node *node)
{
    if (node != NULL)
    {
        printf("Node %d (address %p, left %p, right %p)\n",
                node->data, node, node->left, node->right);
        preOrderTraversal(node->left);
        preOrderTraversal(node->right);
    }
}
```



8 3 1 6 4 7 10 9

Traverse a BST (inOrder)




```
void inOrderTraversal(t_node *node)
{
    if (node != NULL)
    {
        inOrderTraversal(node->left);
        printf("Node %d (address %p, left %p, right %p)\n",
                node->data, node, node->left, node->right);
        inOrderTraversal(node->right);
    }
}
```

1 3 4 6 7 8 9 10

Search for a Node in BST

```
t_node* BSTSearch(t_node *node, int key)
{
    // base case
    // 1. no match
    if(node == NULL)
        return NULL;
    // 2. yes match
    if(node->data == key){
        printf("Found the key %d\n", key);
        return node;
    }
    // recursive case: traverse either to the left
    //or the right
    if(key < node->data)
        return BSTSearch(node->left, key);
    else
        return BSTSearch(node->right, key);
}
```

Finding Minimum and Maximum:

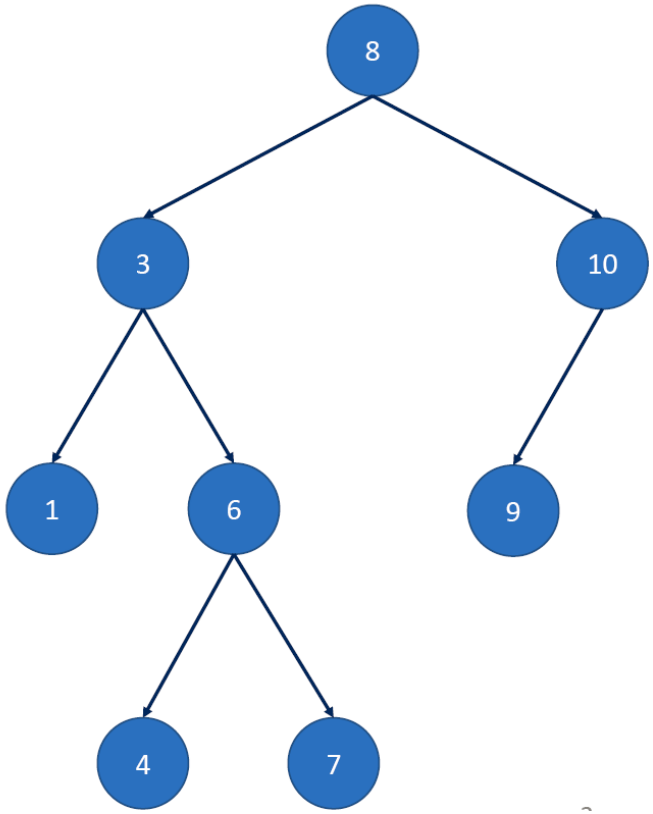
```
t_node* FindMin(t_node *node)
{
    //base case
    if(node->left == NULL)
        return node;
    //recursive case
    else
        return FindMin(node->left);
}

t_node* FindMax(t_node *node)
{
    //base case
    if(node->right == NULL)
        return node;
    //recursive case
    else
        return FindMax(node->right);
}
```

FreeTree:

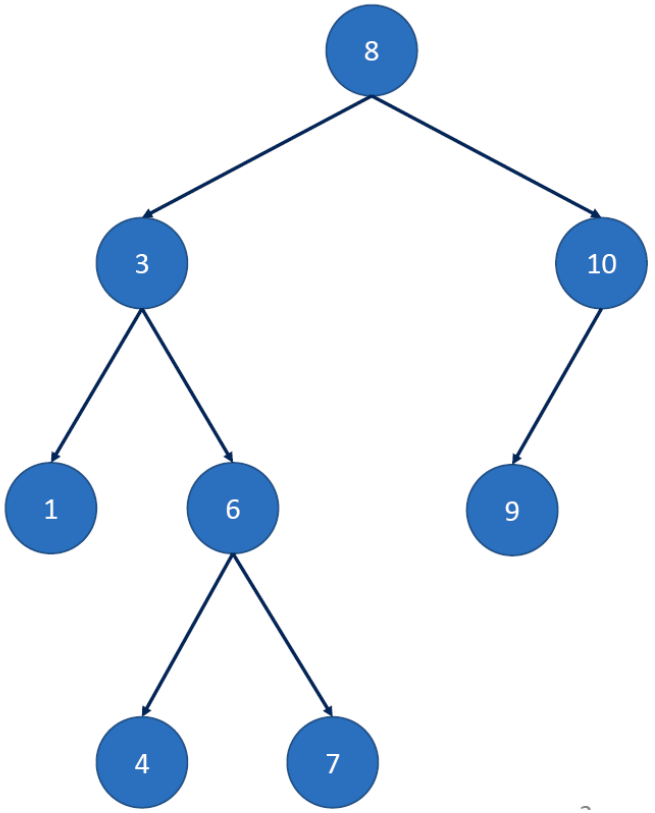
```
void FreeTree(t_node *node)
{
    // Base case
    if(node ==NULL)
        return;
    // Recursive case
    else{
        FreeTree(node->left);
        FreeTree(node->right);
        printf("Free node of %d\n ", node->data);
        free(node);

    }
}
```



Height of BST

```
int getHeight(t_node *node)
{
    int lh, rh;
    //base
    if(node == NULL)
        return -1;
    //recursive
    else{
        lh = getHeight(node->left);
        rh = getHeight(node->right);
        if(lh>rh)
            return lh + 1;
        else
            return rh + 1;
    }
```



Breadth First Search (BFS)

- Start at the root node and explores all neighboring nodes first. Then for each of these nearest nodes, it explores their unexplored neighbor nodes and so on. A queue data structure is used to carry out the search.

  Suitable for finding shortest path in a graph - GPS application.

Steps:

1. Enqueue the root node.
2. Dequeue the node and check it -if the sought element is found, done.

Otherwise, enqueue any direct childs that have not been tested.

3. Repeat step#2.

Code on Github: BST_search_BFS_DFS.c

```c
// BFS Search
t_node* BFS(t_node *node, int data)
{
    item *queue = NULL;
    t_node *i;

    if (node != NULL)
    {
        enqueue(&queue, node);
    }

    while ((i = dequeue(&queue)) != NULL)
    {
        if (i->data == data) break; /* found it! */

        if (i->left != NULL) enqueue(&queue, i->left);
        if (i->right != NULL) enqueue(&queue, i->right);
    }

    /* free the queue */
    while (dequeue(&queue) != NULL); //Pay attention to the ; termination//

   return i;
}
```

```c
typedef struct nodeTag
{
    int data;
    struct nodeTag *left;
    struct nodeTag *right;
} t_node;

typedef struct itemTag item;

struct itemTag
{
    t_node *data;
    item *nextItem;  /* pointer
};
```

```
void enqueue(item **head, t_node *data)
{
    item *newitem = NULL;

    if (*head == NULL)
    {
        newitem = (item *)malloc(sizeof(item));

        /* copy data */
        newitem->data = data;
        newitem->nextItem = *head;

        *head = newitem;

        return;
     }

     enqueue(&(*head)->nextItem, data);
}
```

```
t_node* dequeue(item **head)
{
    item *removed;
    t_node *data;
    if (*head == NULL) return NULL;
    /* get data */
    data = (*head)->data;
    /* remove node from list */
    removed = *head;
    *head = (*head)->nextItem;
    free(removed);
    return data;
}
```

Depth First Search (DFS)

Start at the root node and explores as far as possible along each branch, going deeper and deeper in the tree.

- When a leaf node is reached, the algorithm backtracks to the parent node and checks its children nodes.
- Can be implemented as a recursive algorithm.

  (The algorithm used in slide#6, “Search for a Node in BST,” is DFS)

BST (Binary Search Tree) implemented using C++

- Create a binary search tree and perform:
  - Insertion
  - Search a node
  - Count no. of nodes
  - Find height of the tree
  - Traverse the tree (inorder) and
    store the nodes in a vector
  - Delete tree
  - Delete a particular node!

BST (class)

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

```
class bst{
    public:
        bst();
        ~bst();
        void insert(int data);
        node *search(int data);
        void inorder();
        int countnodes();
        int getHeight();

    private:
        void delete_tree(node *root);
        void insert(int data, node *root);
        node *search(int data, node *root);
        void inorder(node *root, vector<int> &v);
        int countnodes(node *root);
        int getHeight(node *root);
        node *root;
};
```

BST (main)

```
int main(){
    cout<<"build a binary search tree"<<endl;
    bst tree1;
    tree1.insert(30);
    tree1.insert(20);
    tree1.insert(10);
    tree1.insert(15);
    tree1.insert(50);
    tree1.insert(12);
```

BST (main cont.)

```
cout<<"total number of nodes in this tree: "<<tree1.countnodes()<<endl;

//searching a node;

int node_data=6;
cout<<"Searching a node with data_val:"<<node_data<<endl;

node *x;
x=tree1.search(node_data);

if (x)
    cout<<"Node Found. The address is: "<<x<<endl;
else
        cout<<"Node does not exist"<<endl;
```

BST (main cont.)

```
//Calculating the height of the Tree

cout<<"The height of the tree is:"<<tree1.getHeight()<<endl;

//Inorder Traversing of the Tree
tree1.inorder();

    return 0;
}
```

BST (constructor & insertion)

Constructor:

```
bst::bst(){
    root = NULL;
}
```

```
void bst::insert(int data){
    if(root == NULL){
        root = new node;
        root->data = data;
        root->left = root->right = NULL;
        return;
    }
    else
        insert(data, root);
}
```

BST (insertion)

```
void bst::insert(int data){
    if(root == NULL){
        root = new node;
        root->data = data;
        root->left = root->right = NULL;
        return;
    }
    else
        insert(data, root);
}
```

```
void bst::insert(int data, node *root){
    if(data == root->data)
        return; //data already exists in BST
    else if(data < root->data){
        if(root->left == NULL){
            root->left = new node;
            root->left->data = data;
            root->left->left = root->left->right = NULL;
            return;
        }
        else
            return insert(data, root->left);
    }
    else{
        if(root->right == NULL){
            root->right = new node;
            root->right->data = data;
            root->right->left = root->right->right = NULL;
            return;
        }
        else
            return insert(data, root->right);
    }
}
```

BST (search)

```
node * bst::search(int data){
    if (root == NULL || root->data == data)
        return root;
    else
        return search(data, root);
}
```

```
node * bst::search(int data, node *root){
    if(root == NULL)
        return NULL;
    if(data == root->data)
        return root;
    else if(data < root->data)
        return search(data, root->left);
    else
        return search(data, root->right);
}
```

BST (countnodes)

```
int bst::countnodes(){
    if (root == NULL)
        return 0;
    else
        return countnodes(root);
}
```

```
int bst::countnodes(node *root){
    if(root == NULL)
        return 0;
    else
        return 1+countnodes(root->left)+countnodes(root->right);
}
```

BST (getHeight)

```
int bst::getHeight(){
    if (root == NULL)
        return 0;
    else
        return getHeight(root);
}
```

```
int bst::getHeight(node *root){
    int lh,rh;

// base case: Reached to NULL
    if(root == NULL)
    return -1;

// recursive case: Calculate the height of the left-subtree and
//   the right-subtree, and take the bigger one.

   else{
    lh = getHeight(root->left);
    rh = getHeight(root->right);
   if(lh>rh)
    return lh+1;
   else
    return rh+1;
}

}
```

BST (inorder traversing)

```
void bst::inorder(){
    if (root == NULL)
        cout<<"empty tree"<<endl;
    else{
        cout<<"inorder traversal of this binary search tree"<<endl;
        vector<int> v;
        inorder(root, v);
```

```
void bst::inorder(node *root, vector<int> &v){
    if(root != NULL){
        inorder(root->left, v);
        v.push_back(root->data);
        inorder(root->right, v);
    }
}
```

BST (inorder traversing)

```
void bst::inorder(node *root, vector<int> &v){
    if(root != NULL){
        inorder(root->left, v);
        v.push_back(root->data);
        inorder(root->right, v);
    }
}
```

BST (inorder traversing)

```
void bst::inorder(){
    if (root == NULL)
        cout<<"empty tree"<<endl;
    else{
        cout<<"inorder traversal of this binary search tree"<<endl;
        vector<int> v;
        inorder(root, v);

            //Traversing vector using iterator

            /*
                for(auto it=v.begin(); it!=v.end(); it++){
                    cout<<"node:"<<*it<<endl;
                }

            */

            //Traversing vector without using iterator

               for(int it=0; it<size(v); it++){
                   cout<<"node:"<<v[it]<<endl;
            }
        }
    }
```

BST (~bst) – delete BST

```
bst::~bst(){
    cout<<"delete a binary search tree "<<endl;
    delete_tree(root);
}
```

```
void bst::delete_tree(node *root){
    if(root != NULL){
        delete_tree(root->left);
        delete_tree(root->right);
        cout<<"node deleted: "<<root->data<<endl;
        delete root;
    }
}
```

BST (~bst) – delete BST

```
void bst::delete_tree(node *root){
    if(root != NULL){
        delete_tree(root->left);
        delete_tree(root->right);
        cout<<"node deleted: "<<root->data<<endl;
        delete root;
    }
}
```

Delete a Node from BST

```cpp
// C++ program to implement optimized delete in BST.
#include <bits/stdc++.h>
using namespace std;

struct Node {
        int key;
        struct Node *left, *right;
};

// A utility function to create a new BST node
Node* newNode(int item)
{
        Node* temp = new Node;
        temp->key = item;
        temp->left = temp->right = NULL;
        return temp;
}
```

```cpp
/* A utility function to insert a new node with given key in
* BST */
Node* insert(Node* node, int key)
{
        /* If the tree is empty, return a new node */
        if (node == NULL)
                return newNode(key);

        /* Otherwise, recur down the tree */
        if (key < node->key)
                node->left = insert(node->left, key);
        else
                node->right = insert(node->right, key);

        /* return the (unchanged) node pointer */
        return node;
}
```

```cpp
// Driver Code
int main()
{
        root = insert(root, 30);
        root = insert(root, 20);
        root = insert(root, 40);
        root = insert(root, 70);
        root = insert(root, 60);

        printf("Original BST: ");
        inorder(root);

        printf("\n\nDelete a Leaf Node: 20\n");
        root = deleteNode(root, 20);
        printf("Modified BST tree after deleting Leaf Node:\n");
        inorder(root);

        printf("\n\nDelete Node with single child: 70\n");
        root = deleteNode(root, 70);
        printf("Modified BST tree after deleting single child Node:\n");
        inorder(root);

        printf("\n\nDelete Node with both child: 50\n");
        root = deleteNode(root, 50);
        printf("Modified BST tree after deleting both child Node:\n");
        inorder(root);

        return 0;
}
```

Ref: https://www.geeksforgeeks.org/deletion-in-binary-search-tree/

```
/* Given a binary search tree and a key, this function
deletes the key and returns the new root */
Node* deleteNode(Node* root, int k)
{
        // Base case
        if (root == NULL)
                return root;

        // Recursive calls for ancestors of
        // node to be deleted
        if (root->key > k) {
                root->left = deleteNode(root->left, k);
                return root;
        }
        else if (root->key < k) {
                root->right = deleteNode(root->right, k);
                return root;
        }

        // We reach here when root is the node
        // to be deleted.

        // If one of the children is empty
        if (root->left == NULL) {
                Node* temp = root->right;
                delete root;
                return temp;
        }
        else if (root->right == NULL) {
                Node* temp = root->left;
                delete root;
```

```
// If both children exist
else {
        Node* succParent = root;
        // Find successor
        Node* succ = root->right;
        while (succ->left != NULL) {
                succParent = succ;
                succ = succ->left;
        }
        // Delete successor. Since successor
        // is always left child of its parent
        // we can safely make successor's right
        // right child as left of its parent.
        // If there is no succ, then assign
        // succ->right to succParent->right
        if (succParent != root)
                succParent->left = succ->right;
        else
                succParent->right = succ->right;
        // Copy Successor Data to root
        root->key = succ->key;
        // Delete Successor and return root
        delete succ;
        return root;
}
void inorder(Node* root)
{
        if (root != NULL) {
                inorder(root->left);
                printf("%d ", root->key);
                inorder(root->right);
        }
}
```