ECE 220 Computer Systems & Programming

Lecture 21 – Trees: traversal and search

Tree Data Structure

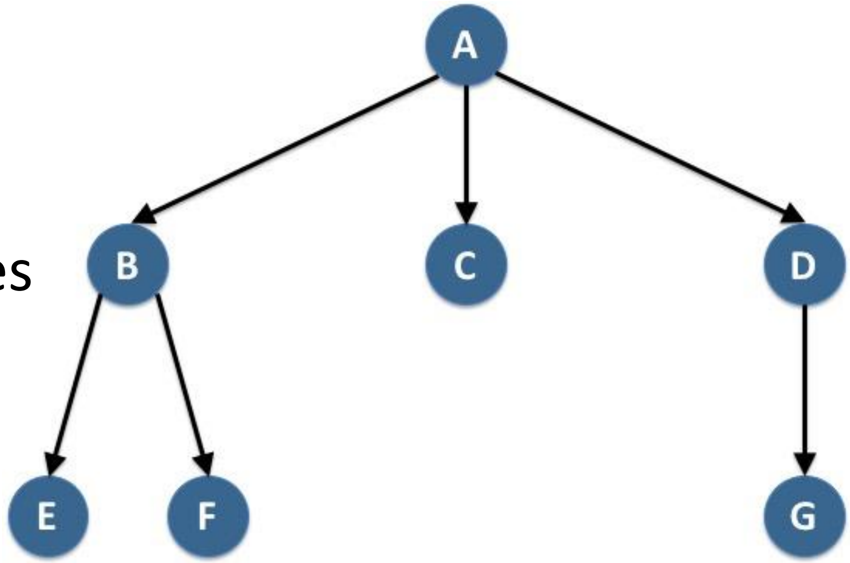
Array, linked list, stack, queue – linear data structures

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);
}
```

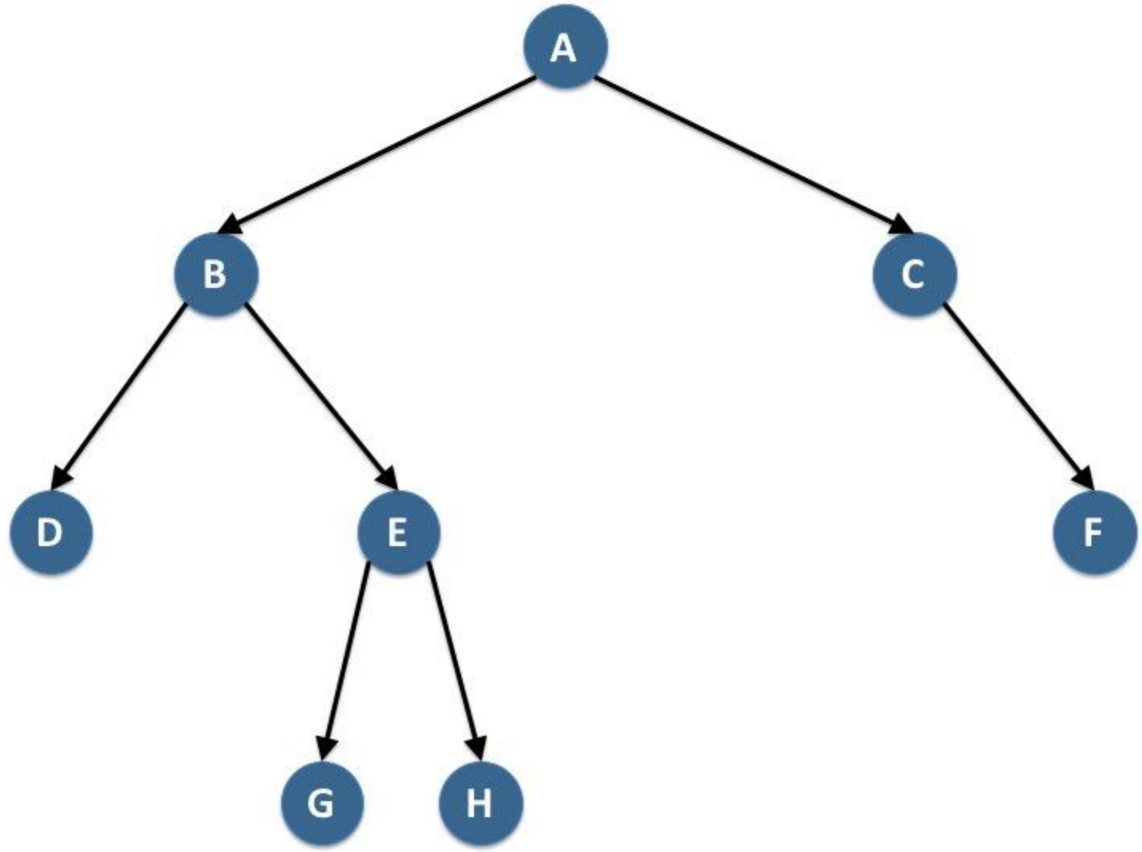
```c
t_node* NewNode(int data)
{
    t_node* node;

    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);
    }
}
```

```c
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?

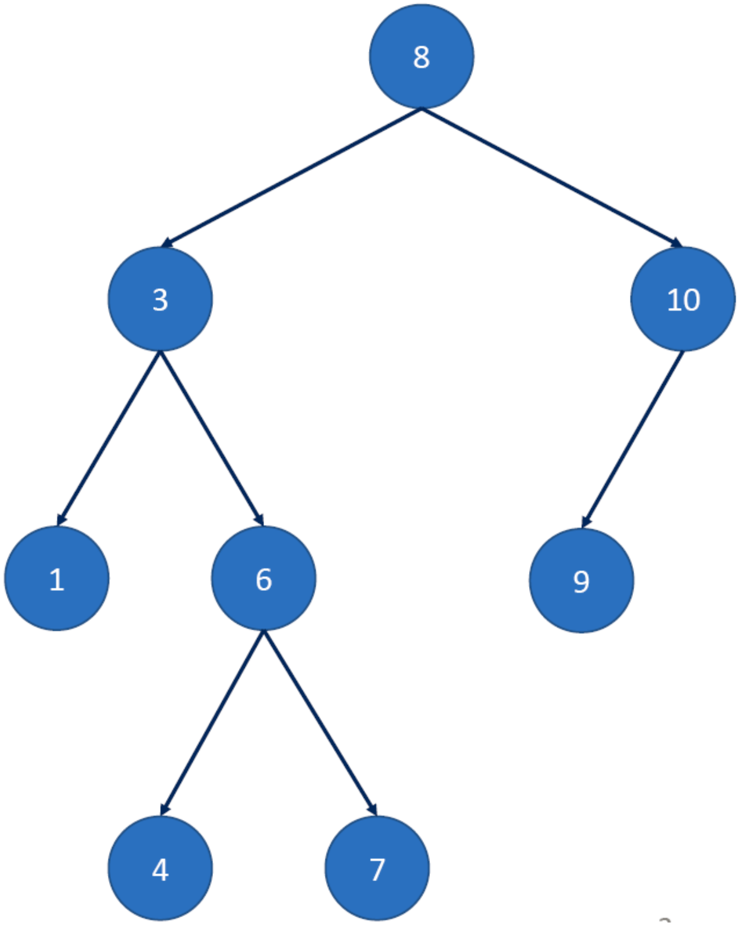
What is the depth of node E?

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



What is the sequence of access for

1. pre-order traversal?

2. in-order traversal?

3. post-order traversal?

http://visualgo.net/bst.html

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;
    }
}
```

Search for a Node in BST

```c
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);
}
```

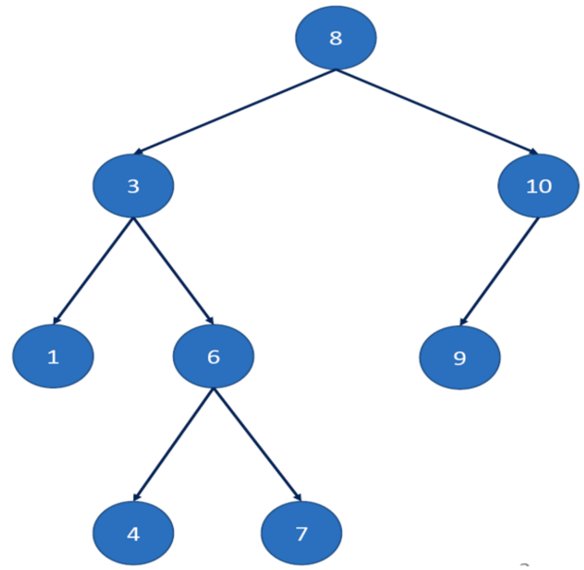
Traverse a BST (preOrder)

```
/* 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);
    }
}
```

Traverse a BST (inOrder)

```c
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);
    }
}
```

```c
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);
    }
}
```



InOrder Traverse:

		L-Null->(return)	R-Null->(return)	L-Null->(return)	R-Null->(return)		
L-Null->(return)	R-Null->(return)	4	Print[4]; –> 4 X	7	Print[7];-> 7 X		
1	Print [1]; –> 1 X	6		Print[6]	Return; -> 6 X		
3		Print[3]				Return;->3 X	
8							Print[8]
2nd Part:							
	L-Null->(return)	R-Null->(return)					
	9	Print[9];-> 9 X	R-Null->(return)				
	10		print[10];->10X				
8	Print[8]			8 X			

X=teardown; [-> 1 X, indicates tear down node 1 after return from the right(R), and so on for -> xx]

Print : 1->3->4->6->7->8->9->10

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);
    }
}
```

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
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);
}
```

```c
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)