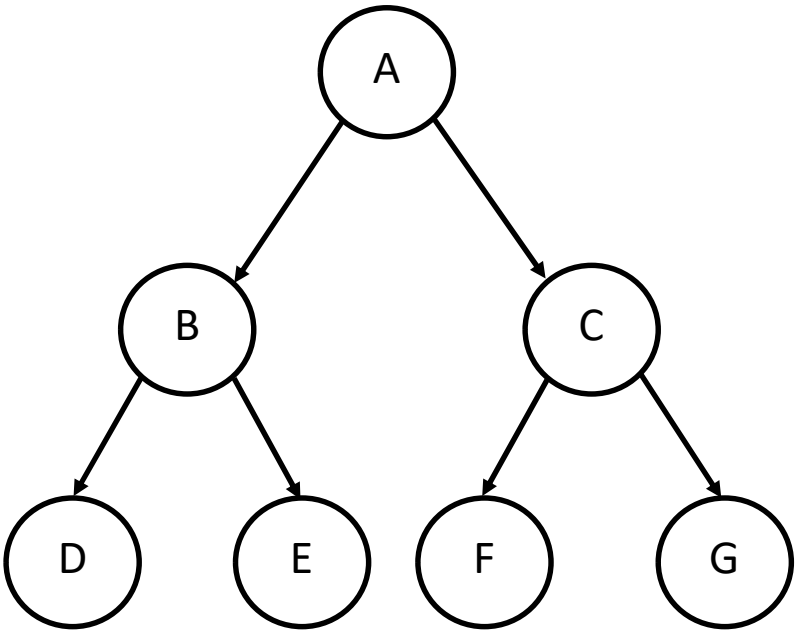
CIS* 2520
Lab 04

1) Binary Tree Overview

Binary trees are the next data structure we will be exploring. Unlike arrays and linked lists, which are linear data structures, binary trees are hierarchal data structures. A binary tree node is made of three elements: a data element; a pointer to a left child node; and a pointer to a right child node.

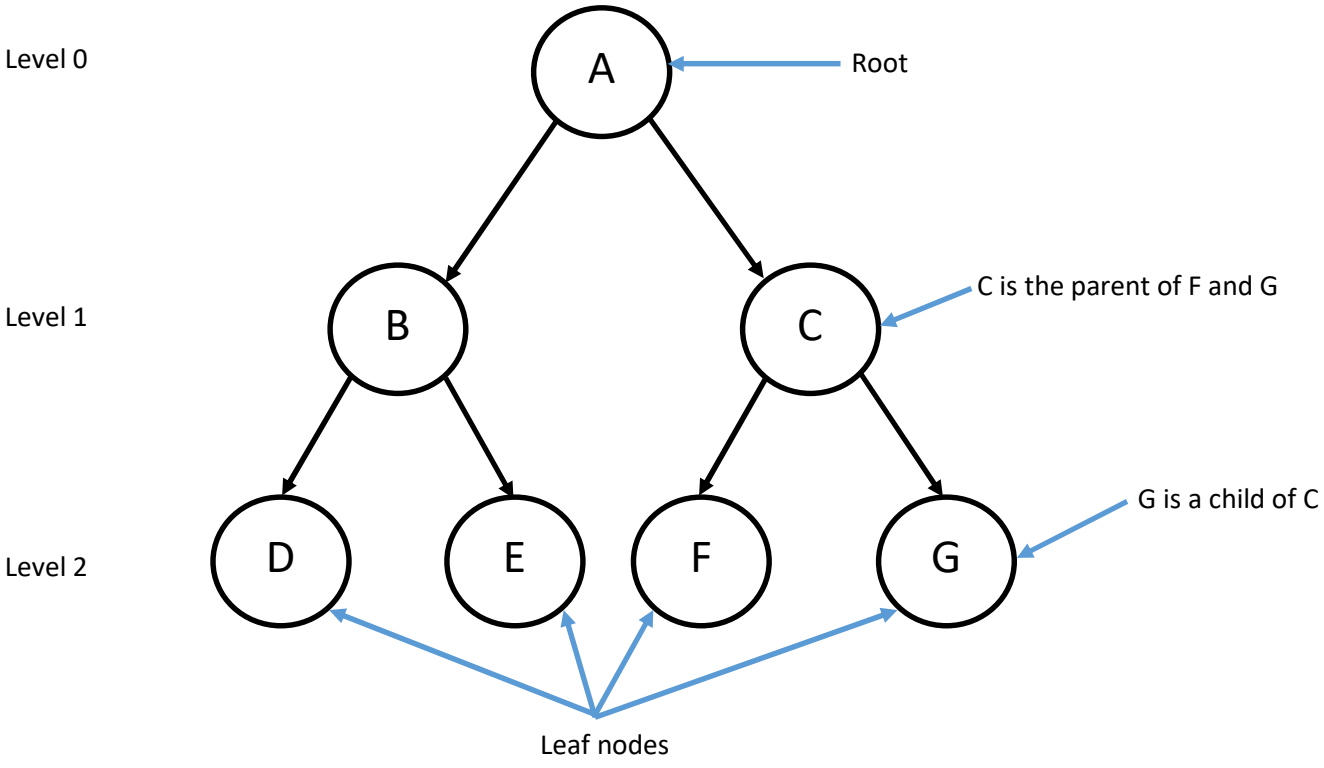
Advantages of trees:

- Store hierarchal information.
- Provide moderate access/search of nodes (faster than linked lists but worse than arrays).
- Provide moderate add/delete of nodes (faster than arrays but slower than linked lists).
- Similar to linked lists, dynamically allocated so there is no upper bound on how many elements it can hold (unlike arrays).



2) Tree Terminology

Root – The node at the top of the tree is called the root. There can only be one root per tree.
Child – The elements directly under a node are referred to as children or child nodes.
Parent – The elements directly above a given node is referred to as the parent or parent node.
Leaf – A node with no children.
Level – Refers to the generation of the node. The root node will be at level 0, its children will be at level 1, and so forth.
Height – The number of edges between a given node and the deepest leaf.
Depth – The number of edges between the root and the deepest leaf.
Subtree – The descendants (all the children) of a given node.
Traversing – Passing through the nodes in a particular order.
Path – The sequence of nodes along the edges of a tree.
Full Binary Tree – A binary tree is considered full if every node has either zero or two children.
Complete Binary Tree – A binary tree is considered complete if all levels are completely filled except for possibly the last level, however, the last level should have all keys as left as possible.
Perfect Binary Tree – All nodes have two children, and all the leaf nodes are at the same level.
Unbalanced Tree – A tree is considered unbalanced when most of the nodes are on one side of the root. Individual subtrees may also be unbalanced.

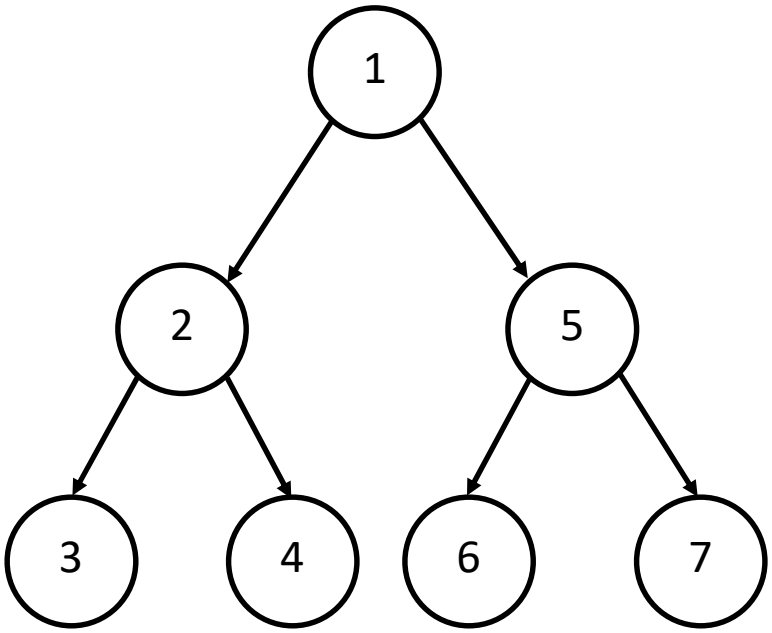


3) Traversing a Tree

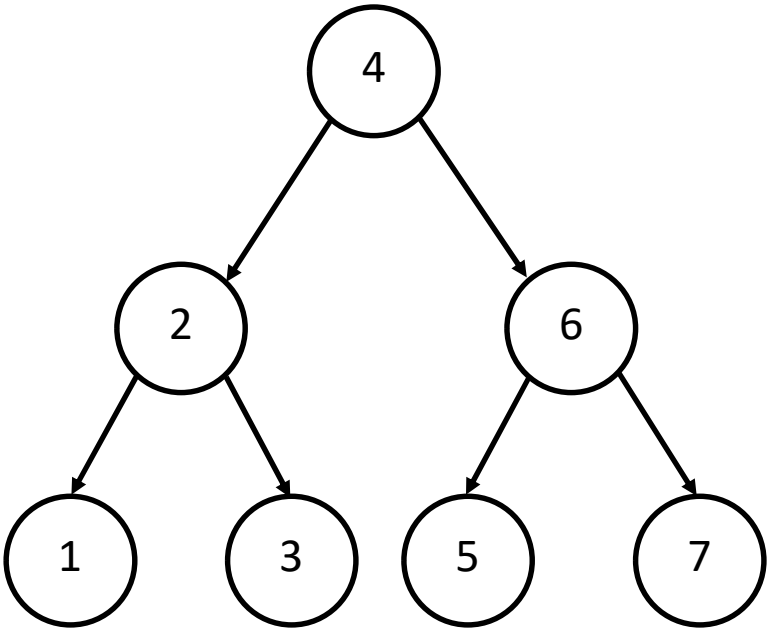
There are two different strategies to traverse binary trees, depth-first traversal and breath-first traversal.

The following are example of depth-first traversal:
Preorder Traversal – visit the parent node first, followed by the left then right child.
Inorder Traversal – visit the left child first, followed by the parent then the right child.
Postorder Traversal – visit the left child first, followed by the right child then the parent node.

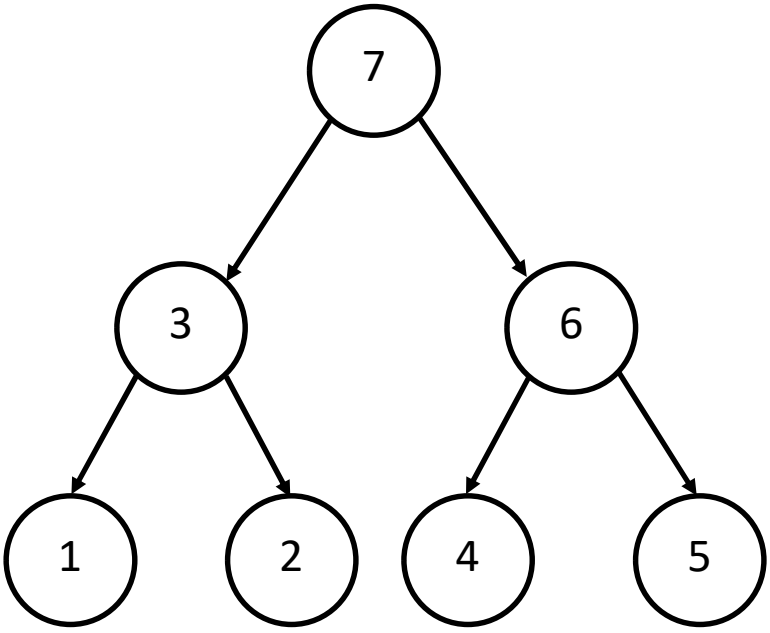
There is one example if breath-first traversal:
Level order traversal – visit nodes from top to bottom and from left to right.



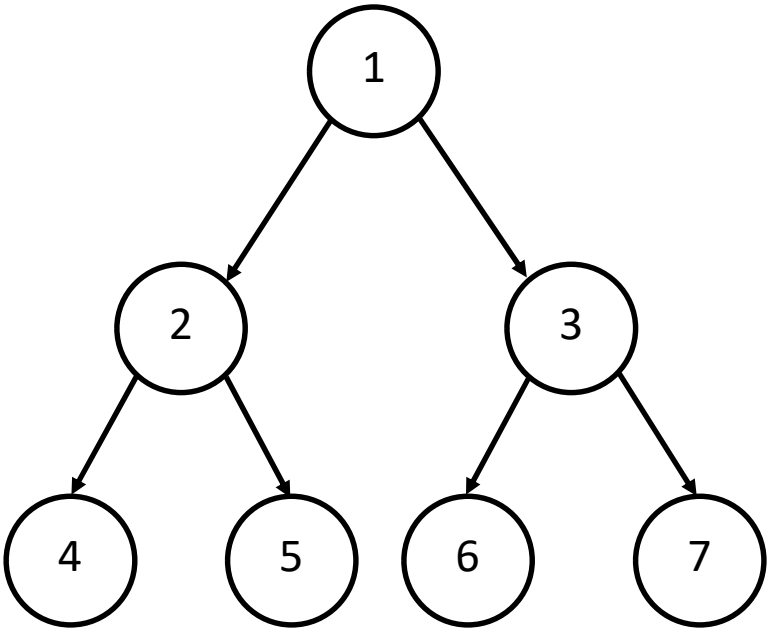
Preorder



Inorder

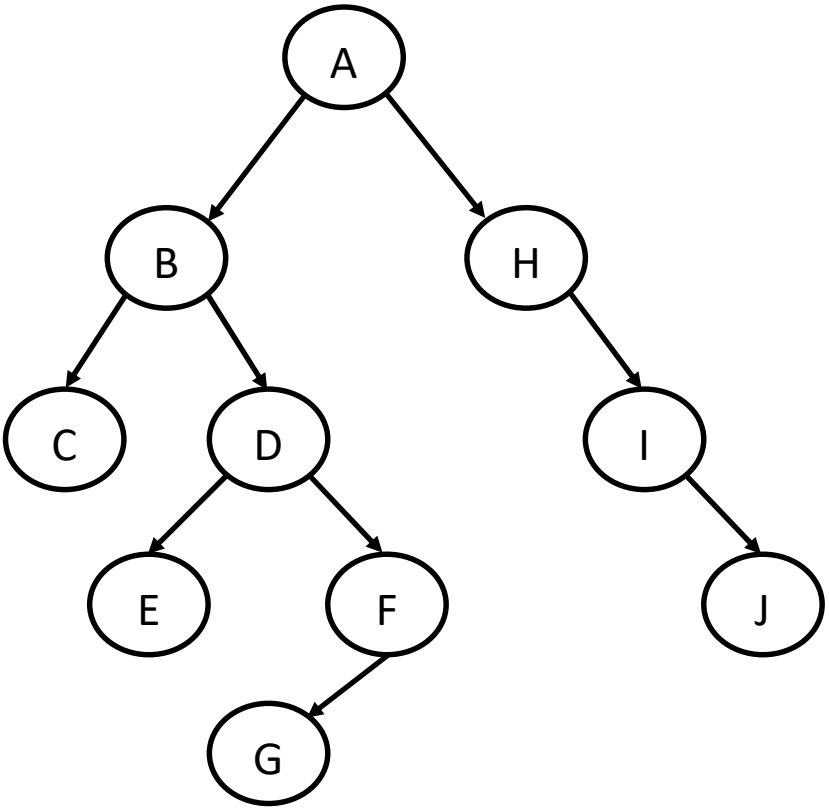


Postorder



Level Order

Practice: determine the order of nodes in each of the traversals mentioned above for the following tree:



4) Your task

For this lab, you will be building a binary tree using a series of integers and performing some functions on it. Your program should:

- Read input from a text file, one line at a time.
- Create a node per integer, storing each integer in the node's letter property.
- Connect each child node to a parent node, creating a tree.
- Implement the following operations on the created tree: createNode, insertNode, printPostorder, printPreorder, printInorder, size, and freeTree.

Part 1: Struct and typedef

You will be using the struct as defined below to build your binary tree program. Each node will hold data (of type int) called number, a pointer to the left child (of type Node) called left, and a pointer to the right child (of type Node) called right. For this lab, you can start to use typedef. Typedef will allow you to give a name to a user-defined data type.

Without typedef:

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

Using typedef:

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

Part 2: Initial setup

a. Create a new textfile called "input.txt"
b. In input.txt enter the following text "5\n3\n8\n2\n4\n7\n9", there should be a newline everywhere there is a \n so that every number is on its own line. Save and exit the file.
c. Create two files for your source code, binaryTree.c and main.c
d. Create a header file, binaryTree.h and define the struct Node, as shown above.
e. Include your header file in binaryTree.c and main.c as you have been doing for your assignments (you can add header guards for good practice).

Part 3: Makefile

a. Create a makefile to compile the program with our usual flags.
b. Add an "all:" and a "clean:" rule.
c. Use $(CFLAGS) and $(CC) variables.
d. Compile your program in two steps:

e. Compile source (.c) files into object (.o) files
f. Link the object files to create an executable file “tree”

Part 4: Read data from the input file

a. In your main.c, declare a file pointer to input.txt
b. Read the text in your input.txt file. You can use the fgets() function to read each line one by one.
c. You will need to convert the string into an integer (hint: atoi()).
d. You can verify you are reading the file correctly by outputting the text read.

Part 5: Create a function to create a new tree node.

```
Node *createNode(int data);
```

a. To create a node and set its data property to the integer read, you can create a helper function in binaryTree.c and call it in your main when needed (remember to define it in binaryTree.h).
b. The function should take in an integer.
c. Set the data field in your Node struct to integer passed in and set the left and right pointers in the Node struct to null.

Part 7: Create the root node.

a. In main.c (where you have the reading input functionality) you will begin building your binary tree.
b. To keep track of your tree, create a new node called root and initialize it to null.
c. Read the first number from the file, and using this number and createNode(), create a new node and assign the returned pointer to root.

Part 8: Create an insert function.

```
void insertNode(Node* root, Node* toInsert);
```

a. insertNode() should take in the root node and a node to insert.
b. Check to make sure the root and toInsert are not null.
c. If the integer of the toInsert node is greater than or equal to the root (or parent node) then it should insert the node as the right child. If the integer of the toInsert node is less than the root (or parent node) then it should insert the node as the left child. (Hint: you may want to use recursion in this function)
d. Your code should look something like this:

```
if(data to insert >= root data)
     if(right child of root is not null)
          insertNode(right child of root, toInsert)
     else
          root->right = toInsert
```

```
else if(data to Insert < root data)
     if(left child of root is not null)
          insertNode(left child of root, toInsert)
     else
          root->left = toInsert
```

Part 9: Create the rest of the tree

a. In your main, loop through the input.txt file, using your createNode() and insertNode() functions, create the rest of the tree. (Hint: this will be similar to how you created root).

Part 8: Create a printPostorder function

```
void printPostorder(Node* root);
```

a. Check to make sure the root is not null.
b. Print the data in the tree using postorder traversal.
c. This function should take the root node and output to stdout.

Part 8: Create a printPreorder function

```
void printPreorder(Node* root);
```

a. Check to make sure the root is not null.
b. Print the data in the tree using preorder traversal.
c. This function should take the root node and output to stdout.

Part 8: Create a printInorder function

```
void printInorder(Node* root);
```

a. Check to make sure the root is not null.
b. Print the data in the tree using inorder traversal.
c. This function should take the root node and output to stdout.

Part 9: Create a size() function that determines the number of nodes in the tree.

```
int size(Node* root);
```

a. Check to make sure the root is not null, if it is null, return -1.
b. This function should take the root node and return the number of nodes in the tree.

Part 12: Create a freeTree() function that determines the height of a given node.

```
void freeTree(Node* root);
```

a. This function should take in a node.

b. Check to make sure the root is not null.
c. If the node is not null, you can call freeTree() with the left and right child and then free the current node.
d. Verify that your freeTree is not leaking memory using valgrind.