Lab 13: Trees

1.1 Information

Topics: Trees

Turn in: This is another on-paper "lab". Turn in a text file, PDF file, scanned or photographed images.

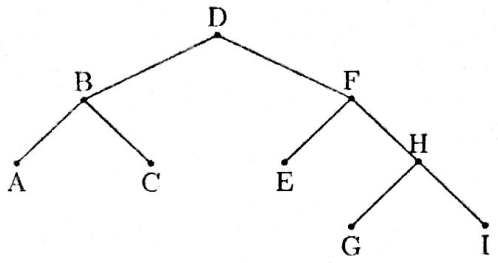
1.2 About: Efficiency

Every data structure has trade-offs. For example, it is faster to access items at some position in an array than it is to do so in a linked list. However, if the array is unsorted, there is not a good efficient way to search the array to find some specific data. If we were to build a sorted array, we would have to decide *when* to do the sorting...

- During Insert - Locate the proper place for the new data
- After Insert - Insert to the end, then re-sort the array

Either way, the act of inserting data into the array will slow down the process, since we would need to either shift n items over to make room for the new item, or perform a sorting algorithm; neither way is as efficient as simply putting data in the array at the end, but it might make *access time* more efficient.

When selecting a data structure to use, part of what we need to consider is what we're designing and how the data structure will be used - will we do a lot of inserts? Will we do a lot of accesses? If we're inserting data often but not reading that data very much, a structure like a Linked List or Unsorted Array might be fine, with $O(1)$ time for the "push" function. If we don't do insertions very often, but need to read the data frequently, it would be better to keep our data sorted.



Smaller ⟷ Larger

Trees, especially Binary Search Trees (which we will talk about on its own), are a type of structure where we can make a compromise. Specifically for a **Binary Search Tree**, insert and access are both $O(log\ n)$ on average, because as we traverse through the tree, each step we're removing *half* of the search space.

We will return to Binary Search Trees later, but for now let's go over the terminology associated with Trees.

1.3 Intro to Trees

Tree: A collection of **Nodes (or vertices)** and **Edges**.

Edge: A path that connects two Nodes together. If we have N nodes, then there are $N - 1$ edges.

Nodes: A vertex in the tree, usually associated with some data.

Root Node: The source Node of the tree; it has no parents. Each Tree has one Root Node, usually drawn at the top. All other Nodes descend from the Root Node.

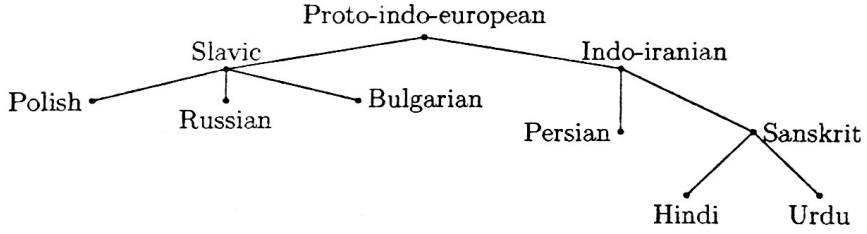
Leaf Node: A Node with no children.

Node Family: We use family terminology to talk about how Nodes are related to each other.

- **Parent node:** Given some Node n, n's parent is the Node immediately above n. in the path between n and the root node. Each Node can have only 0 or 1 parent.
- **Ancestor node:** Given some Node n, an Ancestor of n is any Node along the path from n to the root node.
- **Child node:** Given some Node n, n's child is a Node that comes immediately below it in the tree. Node n lies in the path from its child to the root node. Each Node can have 0 or more children. With a Binary Search Tree, a Node can have 0, 1, or 2 children.
- **Descendant node:** Given some Node n, a Descendant is a Node that comes below it in the tree, where the Node n lies in the path from that descendant to the root node.
- **Sibling node:** Given some Node n, a Sibling of n is another Node where n and that Sibling share the same Parent node.

Question 1 ____ / 4

For the given tree:



a. What are all the (listed) ancestors of *Russian*?
Slavic - Proto-Indo-European

b. What are all the (listed) descendants of *Indo – iranian*?
Sanskrit - Hindi - Urdu

c. What are all the (listed) siblings of *Polish*?
Slavic - Indo-Iranian - Polish

d. What are all the (listed) leaves of the tree?
Polish - russian - Bulgarian - Persian - Hindi - urdu

Path: A path between two nodes, n_a and n_b, is a series of connecting edges between these two nodes.

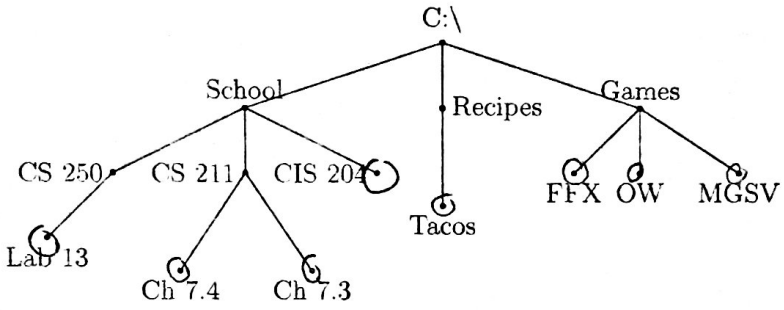
Path Length: The Length of a Path is the amount of edges in the path.

Node Depth: Given any node n, the Depth of n is the *length* of the path between n and the root.

Node Height: Given any node n, the Height of n is the longest path from n to a leaf. Leaves have a height of 0.

Question 2 ____ / 4

For the given tree:



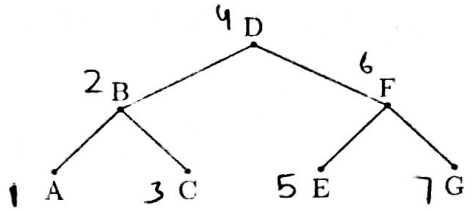
a. Write out all the Nodes in the path from *Lab 13* to *C:*.
CS 250 - School - C:\

b. What is the length of the path from *Lab 13* to *C:*? 3

c. Find the Depths and Heights for the following:

Node	Depth	Height
Lab 13	3	0
Ch 7.3	3	0
Tacos	2	0
MGSV	2	0
CS 211	2	1
School	1	2

1.3.1 Traversals



Since a Tree is not a linear structure, what order do you display its contents? There are three main methods you will see for traversing through a tree. Each of these are recursive, beginning at the root node. Once the end of a path is reached (by hitting a leaf), the recursion causes it to step back upwards through the tree.

Pre-order traversal Begin at the Root r node of some Tree/Subtree...

1. Process r
2. Traverse left, if available
3. Traverse right, if available

With the above tree, we process nodes as such:

D - B - A - C - F - E - G
4 2 1 3 6 5 7

In-order traversal Begin at the Root r node of some Tree/Subtree...

1. Traverse left, if available
2. Process r
3. Traverse right, if available

With the above tree, we process nodes as such:

A - B - C - D - E - F - G
1 2 3 4 5 6 7

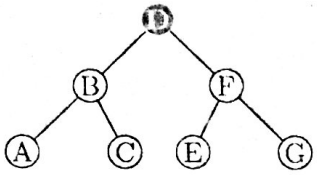
Post-order traversal Begin at the Root r node of some Tree/Subtree...

1. Traverse left, if available
2. Traverse right, if available
3. Process r

With the above tree, we process nodes as such:

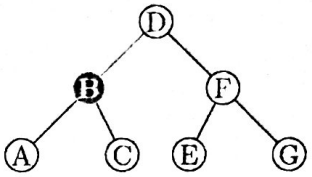
A - C - B - E - G - F - D
1 3 2 5 7 6 4

Step-by-step pre-order illustration:



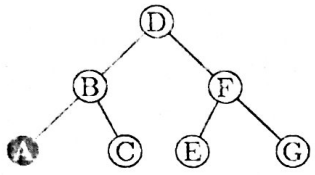
1. Begin at D. Process "D" then go left.

Output: `D`



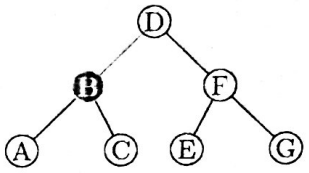
2. Process "B" then go left.

Output: `D B`



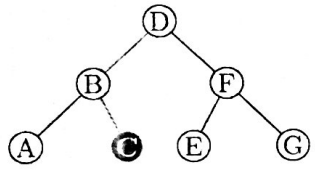
3. Process "A"; no more children, return (back to B).

Output: `D B A`



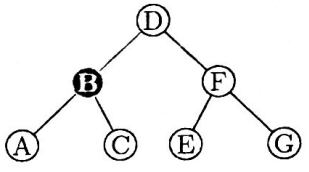
4. Go to right child.

Output: `D B A`



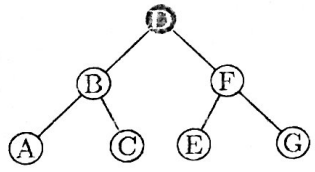
5. Process "C"; no more children, return (back to B).

Output: `D B A C`



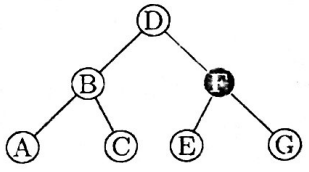
6. Done with left and right subtrees, return (back to D).

Output: `D B A C`



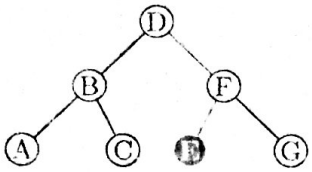
5. Go to right child.

Output: `D B A C`



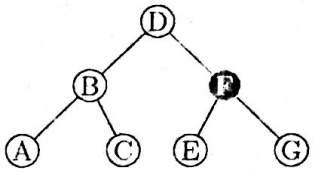
6. Process "F" then go left.

Output: `D B A C F`



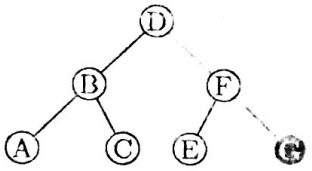
7. Process "E" then return (back to F).

Output: **D B A C F E**



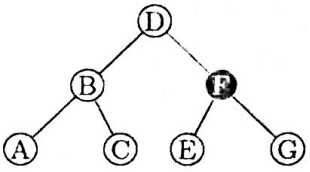
8. Go to right child.

Output: **D B A C F E**



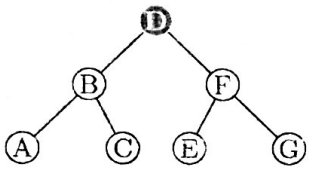
9. Process "G" then return (to F).

Output: **D B A C F E G**



10. Return (to D).

Output: **D B A C F E G**



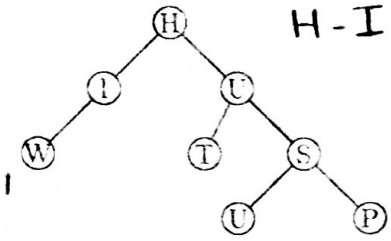
11. Finished.

Output: **D B A C F E G**

Hopefully these steps help you better visualize the recursive nature of tree traversal.

Question 3 ____ / 2

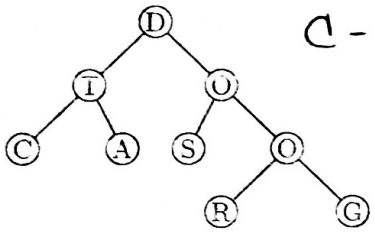
Traverse the following tree using **pre-order** traversal. Write out each Node as you "process" it.



H-I-W-U-T-S-U-P

Question 4 ____ / 2

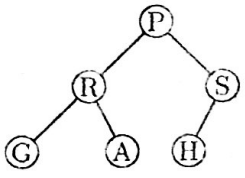
Traverse the following tree using **post-order** traversal. Write out each Node as you "process" it.



C-A-T-S-R-G-O-O-D

Question 5 ____ / 2

Traverse the following tree using **in-order** traversal. Write out each Node as you "process" it.



G-R-A-P-H-S

Question 6 ____ / 3

We have a Node class declared like this:

```
template <typename T>
struct Node
{
    T data;
    Node* ptrLeft;
    Node* ptrRight;
};
```

And we have nodes already declared, **nodeA, nodeB, nodeC, nodeD**. They store 'A', 'B', 'C', and 'D', respectively. Write some basic code to create a tree with these nodes. One will be a root. For each Node, the item to its *left* should be lower in value ($A < B$), and the item to its *right* should be greater ($C > B$).

```
Struct Node* root= NewNode('B')
root
root->ptrLeft= newNode('A')
root->ptrRight= newNode('C')
root->ptrRight->ptrRight= newNode('D')
```