Becca Williams

Lab 13: Trees

1.1 Information

Topics: Trees

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

Make sure to notice the di erence between a binary tree and a binary search tree.

1.2 About: E ciency

Every data structure has trade-o s. 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 e cient way to search the array to nd some speci c data. If we were to build a sorted array, we would have to decide when to do the sorting...

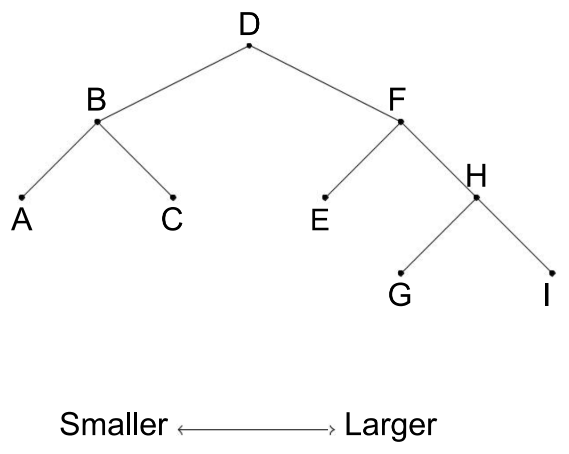
During Insert - Locate the proper place for the new data

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 e cient as simply putting data in the array at the end, but it might make access time more e cient.

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 ne, 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.



Trees, especially Binary Search Trees (which we will talk about on its own), are a type of structure where we can make a compromise. Speci cally 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.

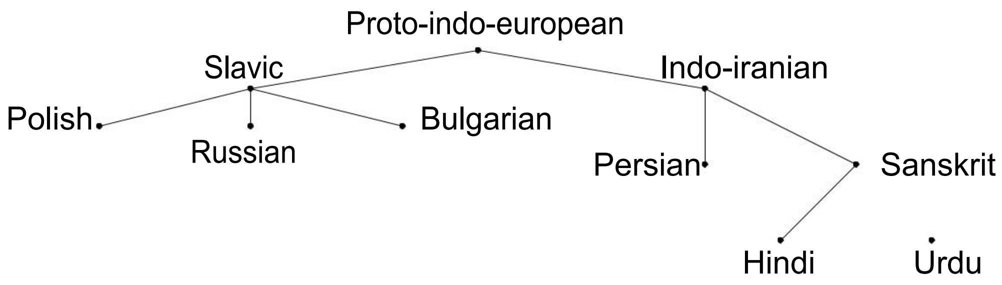
Child node: Given some Node n, n's child is a Node that comes immediately below it in the tree. Node n lies in the path from its child to the root node. Each Node can have 0 or more children. With a Binary Search Tree, a Node can have 0, 1, or 2 children.

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

b. What are all the (listed) descendants of Indo iranian?

   Persian, Hindi, Urdu, Sanskrit

c. What are all the (listed) siblings of P olish?
   Russian, Bulgarian

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.

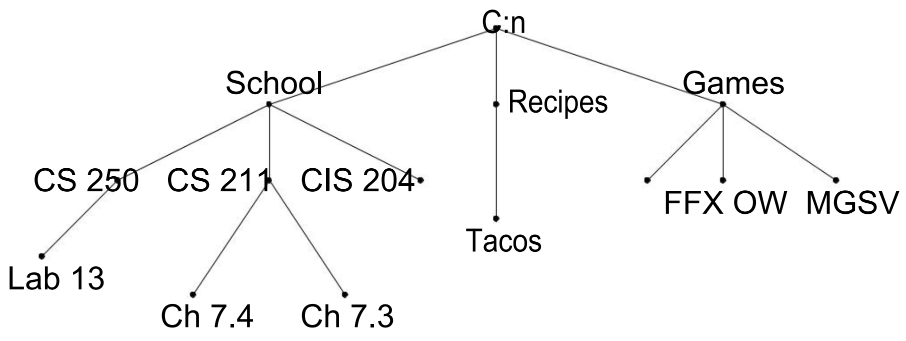
Path Length: The Length of a Path is the amount of edges in the path.

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:



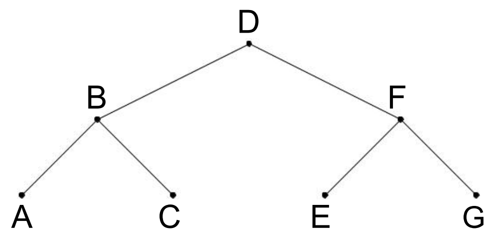
a. Write out all the Nodes in the path from Lab 13 to C:n.
   CS 250, School

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

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

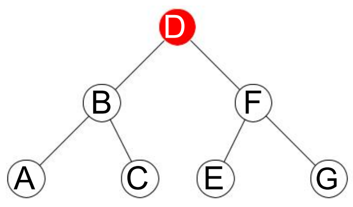
With the above tree, we process nodes as such:
A - B - C - D - E - F - G

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

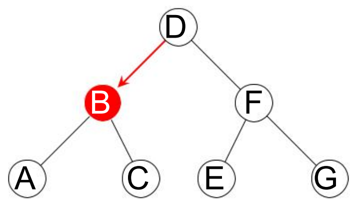
With the above tree, we process nodes as such:
A - C - B - E - G - F - D

Step-by-step pre-order illustration:



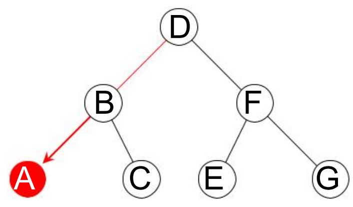
1. Begin at D. Process \D" then go left.

Output: D



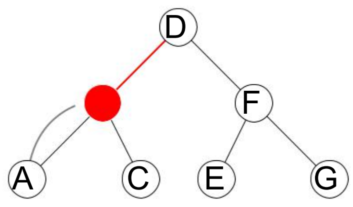
2. Process \B" then go left.

Output: D B



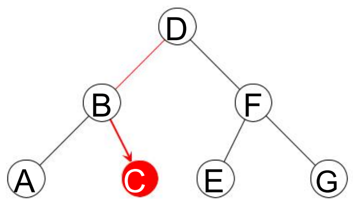
3. Process \A"; no more children, return (back to B).

Output: D B A

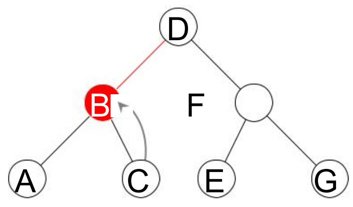


4. Go to right child.

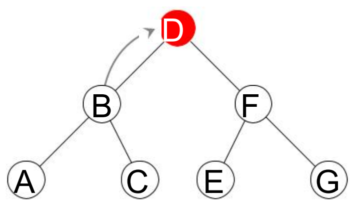
Output: D B A



5. Process \C"; no more children, return (back to B). Output: D B A C

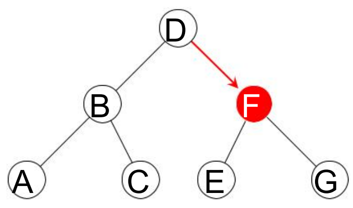


6. Done with left and right subtrees, return (back to D). Output: D B A C



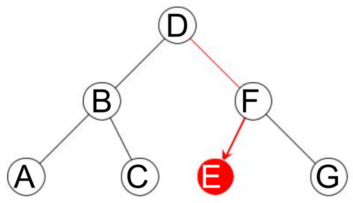
5. Go to right child.

Output: D B A C



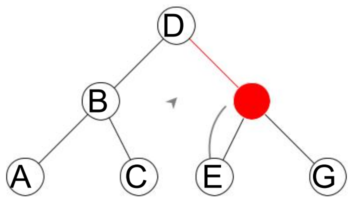
6. Process \F" then go left.

Output: D B A C F



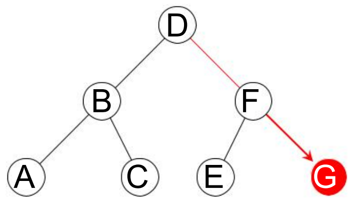
7. Process \E" then return (back to F).

Output: D B A C F E



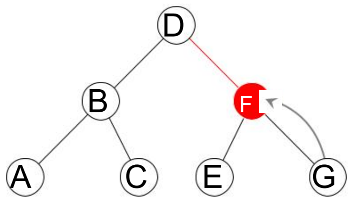
8. Go to right child.

Output: D B A C F E



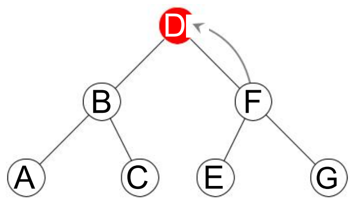
9. Process \G" then return (to F).

Output: D B A C F E G



10. Return (to D).

Output: D B A C F E G



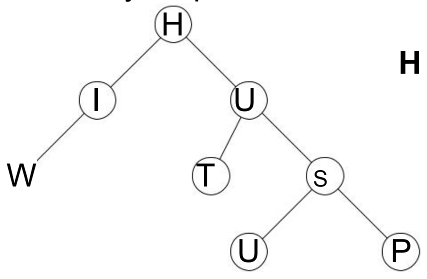
11. Finished.

Output: D B A C F E G

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

Question 3 ____ / 2

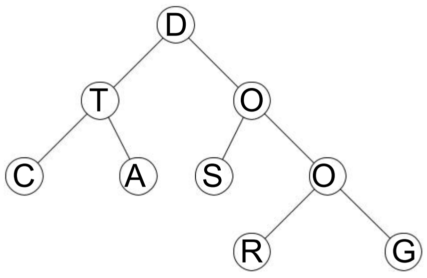
Traverse the following tree using pre-order traversal. Write out each Node as you \process" it.



H I W U T S U P

Question 4 ____ / 2

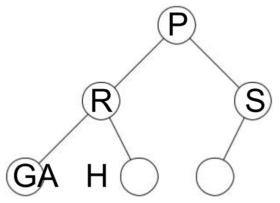
Traverse the following tree using post-order traversal. Write out each Node as you \process" it.



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

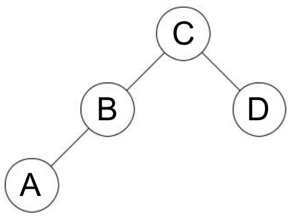
Question 6 ____ / 3

A binary search tree is a type of tree where each node can have 0, 1, or 2 children, but no more than 2. For any Node n, any nodes to the left of n are less than n. Similarly, any nodes to the right of n are greater than n.

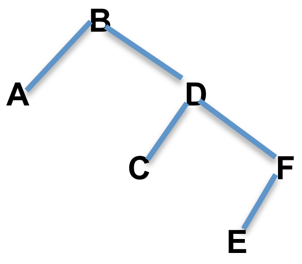
When the rst node is added to a binary tree, it becomes the root. When subsequent nodes are added, we traverse the tree, moving to the left or right until we nd an available space.

For the following, there is a list of nodes in the order added to a tree. Draw out the binary tree once all nodes are added.

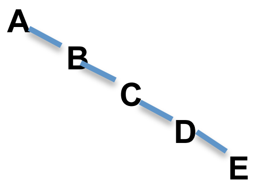
Example: Add: C, B, D, A



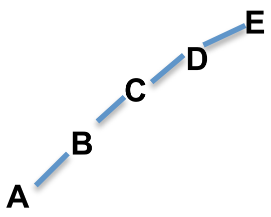
a. Add: B, A, D, C, F, E



c. Add: A, B, C, D, E



c. Add: E, D, C, B, A



Question 7 ____ / 1

For a binary search tree, we have a Node declared like this:

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

And the following variables are already declared:

```
Node<char> nodeA;
Node<char> nodeB;
Node<char> nodeC;
Node<char> nodeD;
```

For each tree generated by adding nodes in the given order, write out the code (or pseudocode) to relate each node together. (e.g. nodeA->ptrLeft = &nodeB;)

a. B, A, D, C

```
nodeB->ptrLeft = &nodeA;
nodeB->ptrRight = &nodeD;
nodeD->ptrLeft = nodeC;
```

b. A, B, C, D

```
nodeA->ptrRight = &nodeB;
nodeB->ptrRight = &nodeC;
nodeC->ptrRight = &nodeD;
```

c. D, C, B, A

```
nodeD->ptrLeft = &nodeC;
nodeC->ptrLeft = &nodeB;
nodeB->ptrLeft = &nodeA;
```

1.3.2 Thinking in subtrees

WIP

1.3.3 Balanced trees

The same set of data can be arranged in a binary search tree in di erent ways, depending on the order that items are added to it.

For example, lets add the numbers 1 through 7 to a binary tree in di erent orders.

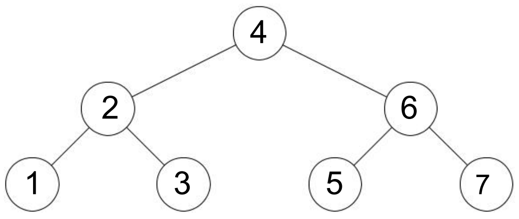


Figure 1.1: Add numbers: 4, 2, 1, 3, 6, 5, 7

Adding numbers like this gives us a nice binary search tree - it is not heavily skewed to one side or the other. This tree is actually balanced.

> \a binary tree is height balanced, or simply balanced, if the height of any node's right subtree di ers from the height of the node's left subtree by no more than 1."
>
> From Data Abstraction & Problem Solving with C++: Walls and Mirrors 7th ed, by Carrano and Henry, page 452

However, if we add these numbers in an order such that each new number is greater than the last one, everything gets added to one side of the tree:

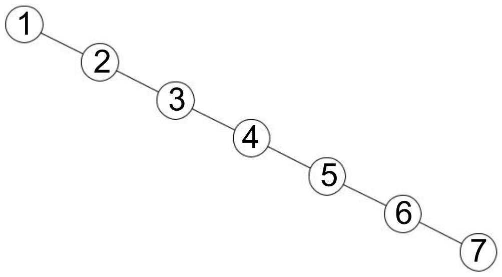


Figure 1.2: Add numbers: 1, 2, 3, 4, 5, 6, 7

This binary tree is no more e cient to search from a sorted linear array, because everything is o to one side.

1.3.4 Full trees

\In a full binary tree of height h, all nodes that are at a level less than h have two children each."

From Data Abstraction & Problem Solving with C++: Walls and Mirrors 7th ed, by Carrano and Henry, page 451

1.3.5 Complete trees

\A complete binary tree of height h is a binary tree that is full down to level h 1, with level h lled in from left to right. [...]

More formally, a binary tree T of height h is complete if

1. All nodes at level h 2 and above have two children each, and
2. When a node at level h 1 has children, all nodes to its left at the same level have two children each, and
3. When a node at level h 1 has one child, it is a left child

[...] Note that a full binary tree is complete."

From Data Abstraction & Problem Solving with C++: Walls and Mirrors 7th ed, by Carrano and Henry, page 451