Contents

Lab 11

Intro to Dictionaries

Group work: Group work is allowed for the labs, but each person must do their own coding and each person must turn in an assignment. Copying other peoples' code / code files is not allowed. Copied assignments will receive 0% for everybody involved.

Assignments must build: Assignments that don't build will automatically just be given a flat 50% score. I may still give feedback on what went wrong, and 50% is better than 0%, so turn in something - but ideally, make sure it builds.

Paper assignments: Type out your answers or print and write answers. Upload lab digitally (scanned/photographed).

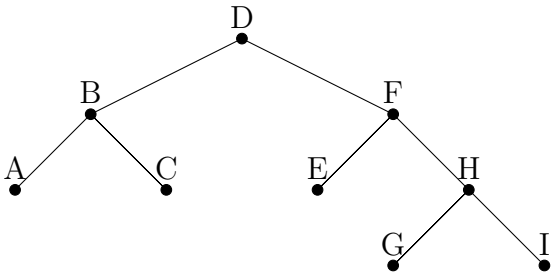
11.1 About: Efficiency

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 $\longleftrightarrow$ 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 $O(logn)$ 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.

11.2 Tree Terminology

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.

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 Height: Given any node n, the Height of n is the amount of edges in the path from node n to the **furthest-down leaf**.

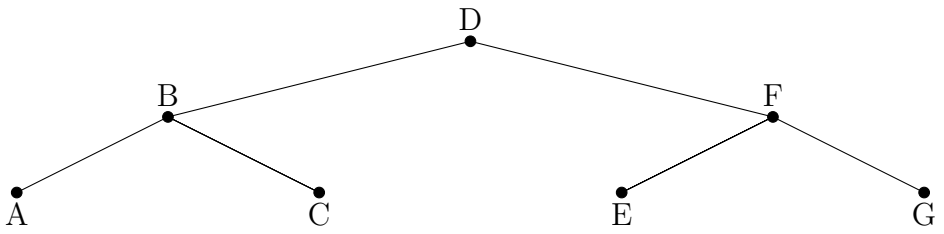
The height of an *empty tree* is 0.

The height of a tree with *only the root* is 1.

Node Depth: Given any node n, the Depth of n is the amount of edges from the **root** to the node.

11.3 Traversing a Binary Tree

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 of some tree.

```
Preorder( Node n ):
    Process n
    Preorder( n->left )
    Preorder( n->right )
```

Using this traversal, we would get the output: `DBACFEG`

In-order traversal:

```
Inorder( Node n ):
    Inorder( n->left )
    Process n
    Inorder( n->right )
```

Using this traversal, we would get the output: `ABCDEFG`

Post-order traversal:

```
Postorder( Node n ):
    Postorder( n->left )
    Postorder( n->right )
    Process n
```

Using this traversal, we would get the output: `ACBEGFD`

11.4 Pushing into a Binary Search Tree

Recall that a **Binary Search Tree** is a tree where:

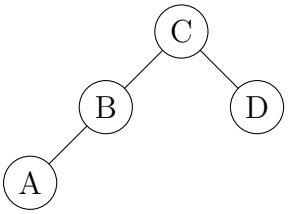
- Each node has 0, 1, or 2 children.
- The left child of n has a value less than n.
- The right child of n has a value greater than n.

When pushing a new item into a tree, there are two scenarios: Either the tree is empty and this is our first node, OR the tree is not empty and we need to find a place for the new node.

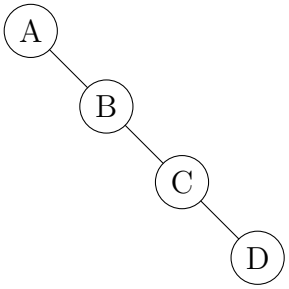
```
Push( data ):
    Create new node, set its data to the input data

    if tree is empty:
        Set root to point to the new node

    else:
        Keep moving left or right down the tree
        until an available spot is found.
```

Depending on the order items are added into a tree, the tree will look different:

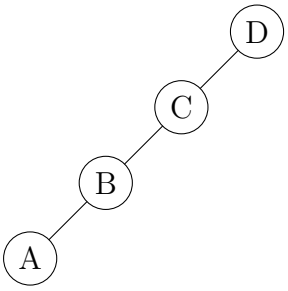
Push: C, B, D, A



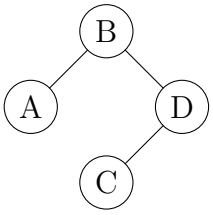
Push: A, B, C, D



Push: D, C, B, A



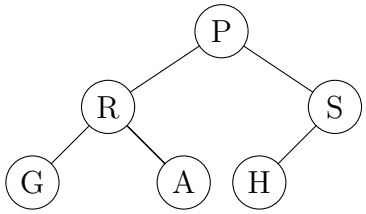
Push: B, D, A, C



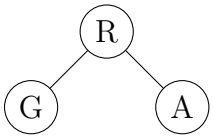
11.5 Subtrees

With any given tree, we can take a subtree of ones of its nodes. If we take some node n out of a tree and build a "subtree based at n", it will be a tree with n as the root, and include all descendants of n.

Example, given this tree:

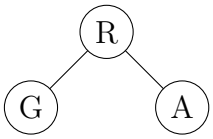


The **subtree at node** R is:

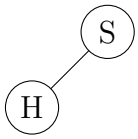


If we talk about a **left-subtree of node** n, we will take the subtree with $n->left$ as the root. The **right-subtree of node** n will be the subtree with $n->right$ as the root.

Left subtree

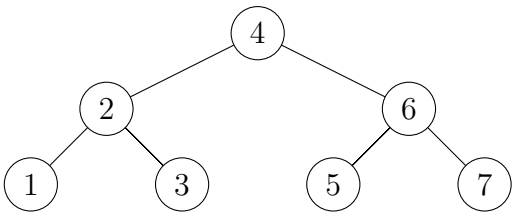


Right subtree



11.6 Balanced trees

The same set of data can be arranged in a binary search tree in different 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 different orders.

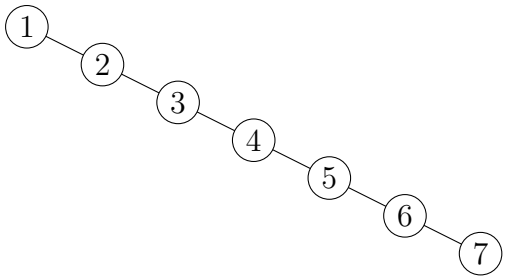


Add numbers: 4, 2, 1, 3, 6, 5, 7

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 differs 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:



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

This binary tree is no more efficient to search from a sorted linear array, because everything is off to one side.

11.6.1 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

11.6.2 Complete trees

"A complete binary tree of height h is a binary tree that is full down to level $h - 1$, with level h filled in from left to right. [...] More formally, a binary tree T of height h is complete if...

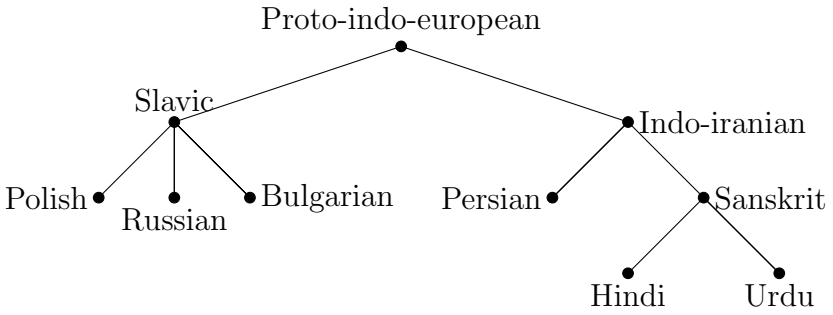
1. All nodes at level $h - 2$ and above have two children each, and
2. When a node at level $h - 1$ has children, all nodes to its left at the same level have two children each, and
3. When a node at level $h - 1$ has one child, it is a left child.

[...] 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

11.7 Questions

Question 1
Answer the questions about the given tree:



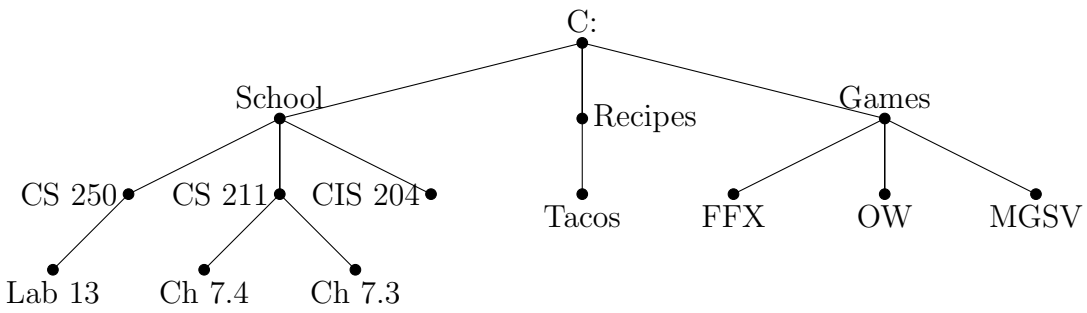
a. What are all (listed) ancestors of *Russian*?

a. What are all (listed) descendants of *Indo-iranian*?

a. What are all (listed) siblings of *Polish*?

a. What are all the leaves of the tree?

Question 2
Answer the questions about the given tree:



a. Write out all the Nodes in the path from **Lab 13** to **C:**.

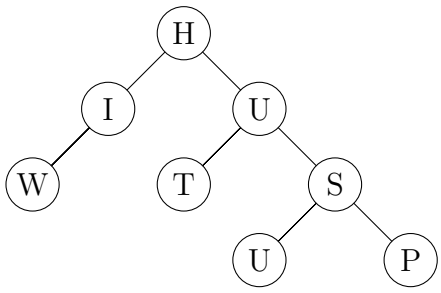
b. What is the length of the path from **Lab 13** to **C:**?

c. Find the Depth and Height of each of the following nodes:

Node	Lab 13	Ch 7.4	CS 250	CS 204	School
Depth					
Height					

Node	Recipes	Tacos	FFX	Games	C:
Depth					
Height					

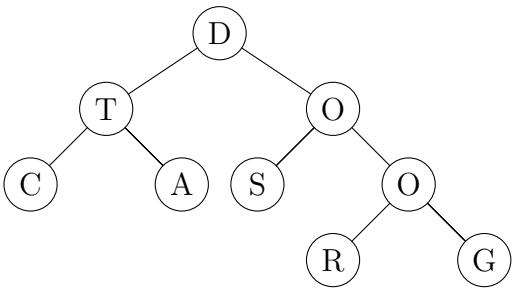
Question 3
Traverse the following tree using **pre-order traversal**. Write out each node as you "process" it.




```
Preorder( Node n ):
    Process n
    Preorder( n->left )
    Preorder( n->right )
```

```
Preorder( H ):
    Process H
    Preorder( H->left )
    ...
```

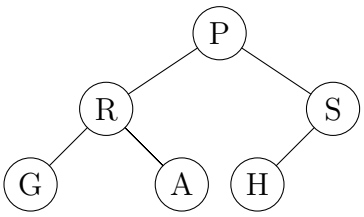
Question 4
Traverse the following tree using **post-order traversal**. Write out each node as you "process" it.




```
Postorder( Node n ):
    Postorder( n->left )
    Postorder( n->right )
    Process n
```

```
Postorder( H ):
    Postorder( H->left )
    ...
```

Question 5
Traverse the following tree using **in-order traversal**. Write out each node as you "process" it.




```
Inorder( Node n ):
    Inorder( n->left )
    Process n
    Inorder( n->right )
```

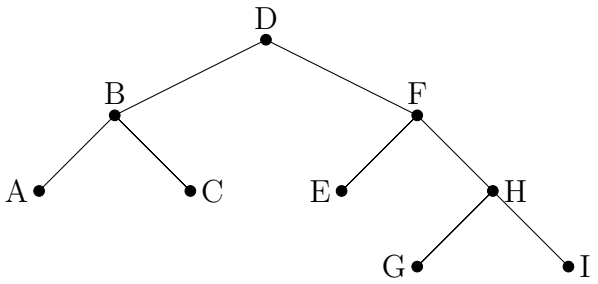
```
Inorder( H ):
    Inorder( H->left )
    ...
```

Question 6
Draw the binary search tree based on the series of items pushed.

a. Push: 4, 2, 3, 1

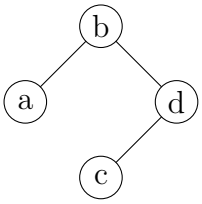
b. Push: 5, 3, 7, 2, 8, 1, 9

Question 7
For the given tree, draw the **right sub-tree** and the **left sub-tree**.



Question 8
Redraw the following trees as complete trees. Make sure to keep all nodes and edges; don't add new edges.

a.



b.