# Computer Science 1MD3

Lab 4 – An Introduction to Graphs and Trees

A graph is much like a roadmap but instead of cities being connected by roads you have nodes connected by edges. There is a specific type of graph called a tree which has very distinct characteristics. It is the purpose of this lab to introduce trees and the methods to traverse through them.

GRAPHS

A graph is a collection of nodes which are connected together by edges.

(1.1)   (1.2)   (1.3)   (1.4)
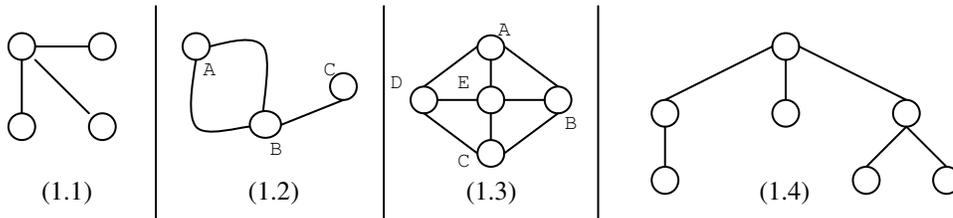
Are all examples of graphs.

TREE

A *walk* in a graph is to move from one node to another node along an edge. In graph 2, A-B-C would be a legal walk, but C-A-B would be an illegal walk. If a walk ends at a node which is connected to the node that you started from this is called a *cycle*. In graph 1.2, A-B-A would be a cycle, and in graph 1.3, A-B-C-D-A would be as cycle as well as A-E-B-A, and so on.

A tree is a graph which does not contain any cycles, graph 1.1 and graph 1.4 are examples of trees.

There is also some terminology which goes along with tree graphs. A position in the tree (◯) is called a *node*, the node at the top is called the *root* node, and the nodes at the bottom are called the *terminal* or *leaf* nodes. We will also refer to proper *ancestors* and *descendants* which are respectively all nodes above or below a given node. Immediate ancestors and descendants, or a position one edge up or down from a node, is a *parent* or a *child* to that node. Finally, the *depth* of a tree is the number of nodes in the longest walk that may be taken from the root to any leaf.
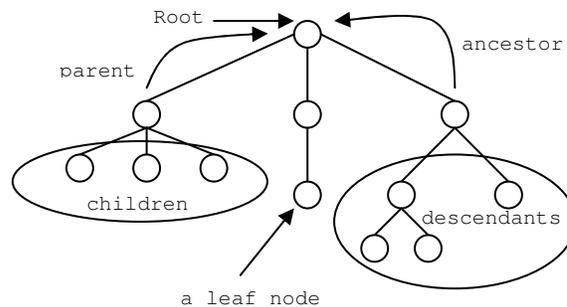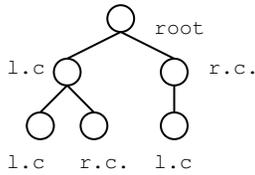

Graph (2.1)

The depth of Graph 2.1 is four.

BINARY TREES

A *binary tree* is a tree such that every node in the tree has at most two children. These children are called the *left-child* (l.c.) and *right-child* (r.c.), if a node only has one child it is the *left-child*.

There are two standard methods for storing binary trees in computers. The first uses a recursive structure, and the second an array. The contrast of these two methods is the structure method uses a lot less memory whereas the array implementation is much easier to program. Lets investigate these two methods.

<u>structure implementation</u>

```
struct node {
        int val;
        node *lc;
        node *rc;
}
```

This definition will make a lot more sense after we have done linked lists. However this is the general idea: you declare a pointer `tree` which points to the root node. The root node has a value, and also two pointers to its left and right child, which have a similar structure. The leafs nodes will have left and right child pointing to nil.
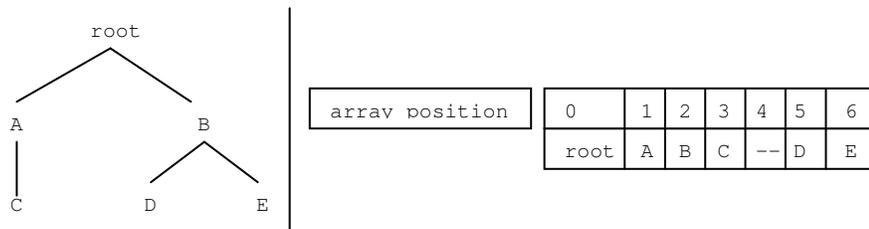
Consider the following tree and its structure implementation.
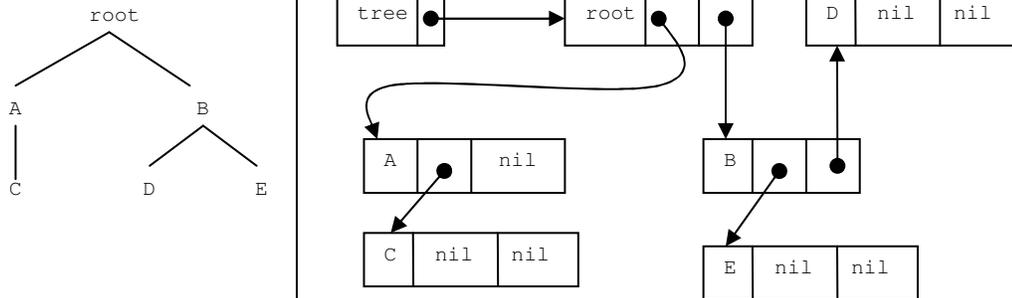
There is some C syntax for traversing the tree which we will cover when we do linked lists.

<u>array implementation</u>

An array implementation is much simplier.

| array position	0	1	2	3	4	5	6
	root	A	B	C	--	D	E

The nodes are put into the array *breadth first* meaning that you just go left to right down each level inserting into the array the value of the node. We may also observe that the left child of array position `x`, is at `2*x+1` and its corresponding right child is at `2*x+2`.

TREE TRAVERSAL

A traversal is the same thing as a walk, and there are a lot ways we can traverse a graph. For all the following examples we will use this graph.

Tree 1.2 demonstrates a traversal which some other traversals use to define themselves. The traversal is as this: imagine we walk around the outside of the tree, starting at the root, moving counterclockwise, and staying as close to the tree as possible; the path we have is like the one in Tree 1.2.
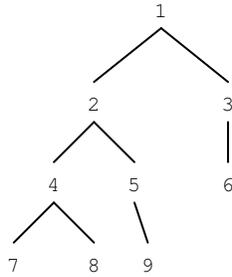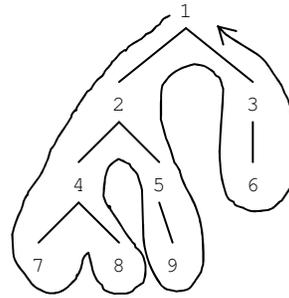
Tree 1.1   Tree 1.2

Breadth First Search:
List a node as you pass left to right down every level of the graph

$$1-2-3-4-5-6-7-8-9$$

Depth First Search:
Traverse the tree as in Tree 1.2
List a node the first time you pass it.

$$1-2-4-7-8-5-9-3-6$$

Preorder Listing
Traverse the tree as in Tree 1.2
List a node the first time we pass it.

$$1-2-4-7-8-5-9-3-6$$

Inorder Listing
Traverse the tree as in Tree 1.2
List an interior node the second time we pass it and leaf nodes the first time we pass it.

$$7-8-4-2-9-5-1-6-3$$
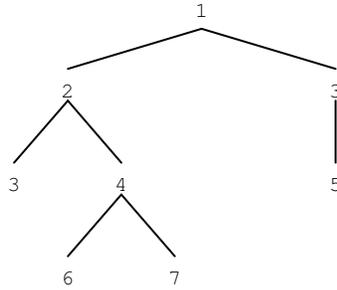
Postorder Listing
Traverse the tree as in Tree 1.2.
List a node the last time we pass it.

$$7-8-4-9-5-2-6-3-1$$

You should familiarize yourselves with these trees for they will become very important in binary search tress.

Self-Test Problems

1.) Produce every listing possible for the following graph:

2.) What is the array representation of the graph presented in 1?

3.) What is the linked list representation of the graph presented in 1?

4.) In your own words, define graph.