Lab 11: Introduction to Trees

Overview

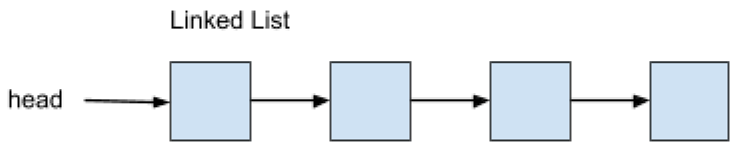
This lab will introduce the Tree data structure.

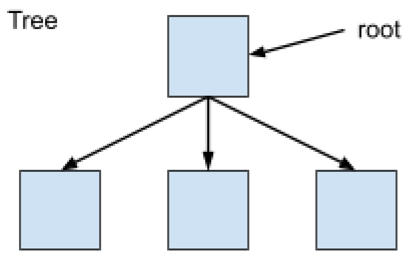
Trees

We have talked a lot about linear data structures in this class. This has included arrays, linked lists, and then abstract representations such as queues and stacks.

Now imagine that we want to move beyond linear structures to something hierarchical. For example, consider the file system of your computer. A single folder is able to hold multiple other files or folders. A folder within that folder can hold its own group of files or folders, and so on. Can we easily store this type of data using a linear structure like the linked list? No, linear structures don't lend themselves very well to hierarchical data where you can have multiple elements at the same level. You can extend linear structures with additional flags and options to manage this type of data, but let us consider an alternative, a new structure called a tree.

Trees are data structures which can be defined as a recursive collection of nodes. Each node in a tree contains a value and references to other nodes in the tree. The starting node of a tree is called the *root* node. Every node in a tree spans a *subtree*, which is the smaller-sized tree that has such a node as its root.





Tree variations are usually named after the number of references to other nodes (children) that each node holds. **Binary trees** contain nodes which store references to two children, commonly referred to as *left* and *right* children. The figure above shows an example of a **ternary tree** in which each node may have three children. **N-ary** trees are not assumed to carry a specific amount of children references.

Trees can also be further specialized to follow a certain goal. For example, **search trees** can be used for efficiently locating specific elements of a set. Depending on the structure choice (e.g. binary, ternary, n-ary) and the goal (e.g. search), the tree will adapt its insertion, deletion, and look-up operations accordingly.

Submission

In today's lab, you will implement the *Node* structure for a **Binary tree**. The starter code contains a node structure extracted from a doubly linked list implementation. All you need to do is modify the node so that instead of pointing to the previous and next nodes, it instead points to left and right nodes (children). You will need to add another node pointer for the parent.

More concretely, the node structure for our binary tree requires a reference (pointer) to the node's parent named *parent_*, and to its left (*left_*) and right (*right_*) children.

You will also implement a simple **insertion** algorithm and the **destructor** for a given Tree class.

The insertion algorithm should be implemented as follows. For a root-less tree, you should first try to insert the new node as its root. If the root exists, try to insert the new node first at its left child if no left child. If there is a left child but there's no right child, insert the new node as the right child. If both left and right children have been assigned, continue looking for a place to insert the new node in the left subtree following the same strategy (first try left, then right child). Hint: use recursion to make this last step easy!

The destructor should clear delete the nodes in the tree. You will find it easiest to use recursion in your destructor. For instance, you could recursively call the destructor on the left and right children before ultimately deleting the current node (their parent) after they've also been cleared.

Lastly, you will answer two questions concerning tree insertion and deletion.

Complete the "Lab 11. Introduction to Trees" activity on Mimir. You do not need to submit source files. Simply add complete code blocks for each question within mimir**.**

You will implement the Node structure for a Binary tree. You will also implement a simple insertion algorithm and the destructor for a given Tree class. Lastly, you will answer two questions concerning tree insertion and deletion. A battery of automated tests will be run to ensure that the errors have been fixed and the results can be properly computed.

A grade of "complete" on this lab work requires a score of 100%.