FULL SAIL UNIVERSITY

Data Structures and Algorithms

Lab 7: Binary Search Tree

Overview

The tree data structure is another non-contiguous way to store information (like the linked list). Unlike the linked list, which stores data sequentially, the tree stores its data hierarchically. This means that there is not a way to go straight from one end of the tree to the other. Even without that feature, trees are widely used to solve a number of complex problems, such as pathfinding, collision detection, compression algorithms, file structures, and many others.

This lab will introduce the tree data structure as a binary search tree (BST). This means that each node will have two child nodes (and a parent node, which is not always necessary). The main rule of a BST is that lower values will be to the left of any given node, and higher values to the right. Our implementation will not contain any duplicates, but that could easily be achieved by simply deciding where they should be placed (left or right of equal values).

Things to Review

- Linked lists (nodes and linking)
- Recursion

New Topics

- Hierarchical traversal algorithms

Data Members

Node

data	Value being stored
left	Pointer to the left child node
right	Pointer to the right child node
parent	Pointer to the parent node

BST

mRoot	Pointer to the root (top) node

Methods

Node Constructor

- Set the data and parent to the passed-in values
- *Nodes always start out as leaf nodes*

BST Constructor

- Set all data members to reflect that no nodes are currently allocated

Push

- Adds a value to the tree
- Dynamically allocate a Node and place it in the correct position in the tree
    - Added nodes will **always** be leaf nodes
- *If there is already at least one node in the tree, you will need a temporary pointer to traverse down the tree to the node you want to insert below*

Clear

- Free up the memory for all dynamically allocated nodes
- Use a post-order traversal and delete one node at a time
- Set the root back to its default state

Destructor

- Free up the memory for all dynamically allocated nodes *(There's a method that does this)*

Contains

- Checks to see if a value is present in the tree and returns true if found
- *Create a temporary pointer to traverse down the tree*

FindNode (Optional)

- Checks to see if a value is present in the tree and returns the address if found
- *Create a temporary pointer to traverse down the tree*

RemoveCase0

- Removes a node from the tree that has no children
    - Can assume the node passed in is a leaf node
- Three sub-cases
    - Root node
    - Is a left child
    - Is a right child

RemoveCase1

- Removes a node from the tree that has one child
    - Can assume the node passed in has exactly one child
- Six sub-cases
    - Root node with left child
    - Root node with right child
    - Left child that has a left child
    - Left child that has a right child
    - Right child that has a left child
    - Right child that has a right child

RemoveCase2

- Removes a node from the tree that has both children
    - Can assume the node passed in has both children
- *This will ultimately lead to a Case0 or Case1 removal*

Remove

- Removes a node from the tree by calling the appropriate RemoveCase method
- Find the address of the node to be removed *(There's potentially a method for this)*
- Check to see how many children that node has, and call the appropriate RemoveCase method
- Return true, if something was removed

InOrder

- Creates a space-delimited string that contains the values of the tree in ascending order
- Start with the root and use the recursive InOrder method to build out the string one value at a time
- *Use std::tostring to convert the int into its string equivalent*

Assignment Operator

- Assigns all values to match those of the object passed in
- Clean up existing memory before the deep copy *(There's a method that does this)*
- Deep copy the entire tree
    - Use the recursive **Copy** method to duplicate the tree with a pre-order traversal
        - Each recursive call to copy should create a single node by **Push**ing it

Copy Constructor

- Assigns all values to match those of the object passed in
- Deep copy the entire tree
    - Use the recursive **Copy** method to duplicate the tree with a pre-order traversal
        - Each recursive call to copy should create a single node by **Push**ing it
- *Remember that data members are not automatically initialized in C++*