Project 03 - Trees, Trees, and more Trees

June 20, 2016

Introduction

You are required to implement three tree data structures: a general tree, a heap tree, and an AVL tree. You are also required to create UML diagrams for each of the classes that you implement. Finally, you have to provide the means to test your system by developing a menu program that allows the user to manipulate your structures.

It is strictly prohibited to use the structures and/or algorithms defined in the C++ standard library (STL). So, if your design requires a list, queue, stack, or a hash table, you can only use your own implementations.

Underflow exceptions might be generated in some of the functions you implement. Make sure exceptions are thrown and caught where appropriate.

Deliverables

- A report that explains the design of your data structures.
- An implementation of a general tree.
- An implementation of a heap tree.
- An implementation of an AVL tree.
- A menu program to test the implemented data structures.

1 General Tree

In this part of the project, you need to implement two classes, a *TreeNode Class* and a *LinkedTree Class*; create their respective UML diagrams. Calculate the running time of your functions and include them in your report.

1.1 Description

A tree is a structure that is formed by Nodes (vertices) and Edges (arcs). Edges connect nodes together. Any two nodes in the tree are connected by one and only one path.

1.2 Data Members

Specify all the data members your implementation needs. You are required to design and create a *TreeNode class* for each of the nodes in the tree. Each node has a key (integer) and a value (Type), a parent, and it stores its children in a linked list.

1.3 Member Functions

Constructors (1 point)

defines constructor.

Destructor (1 point)

defines destructor.

Accessors

getRoot()(1 point) returns the root of the tree.

getSize()(1 point) returns number of elements in the tree.

getHeight() (1 point) returns the height of the three.

getHeight(node) (1 point) returns the height of the node in the argument.

getDepth(node) (1 point) returns the depth of the node in the argument.

empty() (1 point) returns true if the tree is empty, false otherwise.

leaves() (1 point) returns the number of leaves in the tree.

siblings(node) (1 point) returns the number of siblings of the node in the argument.

findCommonAncestor(node1,node1) (2 point) finds the common ancestor of node1 and node2.

findNode(data) (2 point) returns a pointer to a node that holds the data in the argument.

preorder() (1 point) performs preorder traversal.

postorder() (1 point) performs postorder traversal.

levelorder() (1 point) performs level order traversal.

Mutators

void buildTree() (3 points) reads structure from a text file and builds a tree.

clear() (3 points) removes all the elements in the tree

insert(data,...) (3 points) inserts data in the tree. You define this operation. Feel free to use as many parameters as you need.

del(data) (3 points) removes data from the tree.

Friends

defines friends for this class.

2 Heap

In this part of the project, you need to implement two classes, a *TreeNode Class* and a *MaxHeapTree Class*; create their respective UML diagrams. Calculate the running time of your functions and include them in your report.

2.1 Description

A priority queue is like a dictionary in the sense that it stores entries (key,value). However, a dictionary is used when you want to look up a particular key. A priority queue is used when you want to prioritize entries. There is a total ordered defined on the keys. The main operations that a priority key allows to do are:

- Identify or remove the entry that has the smallest (biggest) key. It is the only one that could be removed quickly.
- Insert anything you want in any time.

A binary heap is a particular implementation of the priority queue abstract data type. '**A heap data structure should not be confused with the heap which is a common name for the pool of memory from which dynamically allocated memory is allocated**'. Its definition encompasses several things:

- A heap is a complete binary tree. A complete binary tree is a tree where each level is full except, possibly, the last level (leaves), which is filled from left to right.
- The entries in a binary tree satisfy a heap property: no child has a key less (or bigger) than its parent's key.

For this part of the project you will implement a max heap and all of its operations. Entries are stored in a dynamic array. If an entry is being inserted, and the array is already full, the capacity of the array is doubled. If, after removing an entry from the heap, and the number of entries is one-quarter (1/4) the capacity of the array, then the capacity of the array is halved. The capacity of the array may not be reduced below the initially specified (by you) capacity.

2.2 Data Members

Specify all the data members your implementation needs. You are required to design and create a *TreeNode class* for each of the nodes in the tree. Your nodes have a key (integer) and a value (Type), and they are stored in a dynamic array.

2.3 Member Functions

Constructors (1 point)

defines constructor.

Destructor (1 point)

defines destructor.

Accessors

getMax()(1 point) returns the root of the tree.

getSize()(1 point) returns number of elements in the tree.

getHeight() (1 point) returns the height of the three.

empty() (1 point) returns true if the tree is empty, false otherwise.

leaves() (1 point) returns the number of leaves in the tree.

Mutators

void buildTree() (3 points) reads structure from a text file and builds a max heap.

clear() (3 points) removes all the elements in the tree

insert(key,data) (3 points) inserts data in the tree. This operation must satisfied the heap property.

delMax() (3 points) removes the entry specified by maximum key in the tree. This operation must satisfied the heap property.

Friends

defines friends for this class.

3 AVL Tree

In this part of the project, you need to implement two classes, a *TreeNode Class* and an *AVLTree Class*; create their respective UML diagrams. Calculate the running time of your functions and include them in your report.

3.1 Description

A binary search tree (BST) is a powerful tool to quickly search values specified by a key. However, if the BST is not balanced its operations can degenerate to linear behavior, which makes them no faster than lists.

An AVL tree is a BST that includes complicated updating rules to keep the tree balanced. An AVL tree requires that the height of the left and right children of every node to differ by at most $\pm$ 1. Simple and double rotation are executed when inserting or deleting an entry to/from the tree.

3.2 Data Members

Specify all the data members your implementation needs. You are required to design and create a *TreeNode class* for each of the nodes in the tree. Each node has a key (integer) and a value (Type), and it has access to its parent, left and right child.

3.3 Member Functions

Constructors (1 point)

defines constructor.

Destructor (1 point)

defines destructor.

Accessors

getRoot()(1 point) returns the root of the tree.

getSize()(1 point) returns number of elements in the tree.

getHeight() (1 point) returns the height of the three.

getHeight(node) (1 point) returns the height of the node in the argument.

getDepth(node) (1 point) returns the depth of the node in the argument.

empty() (1 point) returns true if the tree is empty, false otherwise.

leaves() (1 point) returns the number of leaves in the tree.

siblings(node) (1 point) returns the number of siblings of the node in the argument.

find(key, data) (2 point) returns a pointer to a node that holds the data in the argument.

preorder() (1 point) performs preorder traversal.

postorder() (1 point) performs postorder traversal.

levelorder() (1 point) performs level order traversal.

inorder() (1 point) performs inorder traversal.

Mutators

void buildTree() (3 points) reads entries from a text file and builds an AVL tree.

clear() (3 points) removes all the elements in the tree

insert(key,data) (3 points) inserts data in the tree using key. Tree must be kept balanced after the insertion.

del(key) (3 points) removes data from the tree. Tree must be kept balanced after the insertion.

Friends

defines friends for this class.

4 The Menu Program

In order to test your program, you are required to implement a menu program that provides the means to run each of the functions in your classes. Please choose the *string* data type for the values of your entries. The TA will choose one member of your group to defend the demo, and the same grade will be assigned to all of the members of the group.

5 Rubric

This is how your project is going to be graded.

1. Report 40%.
2. Demo 60%. 20% for each structure.

6 The Project Report

You must include everything you consider relevant in your design, UML diagrams, and algorithm analysis.