Project 03 - Trees, Trees, and more Trees

Sunday, June 25 at 11:59pm

Introduction

You are required to implement three tree data structures: a general tree, a heap tree, and an AVL tree. No tree should contain duplicate values. 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 1 page report with your UML diagrams that explains the design of your data structures. Please include what each teammate did and approximate hours spent on the project.
- 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. Your TreeNode class can be used for all trees (not all trees will use all data members). If you would like to include further data members you may although it is not necessary. Some of the methods are the same but have different names, you may feel free to rename them as you see fit as long as they are descriptive. 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 instantiate a *TreeNode* object for each of the nodes in the tree. Each node has a **key**(int) - for Heap tree only, **value** (Type), **balanceFactor** (short int: -1 for right high, 0 for balanced, 1 for left high) - for AVL tree only, and a **parent**, **left** and **right** child (pointers for General and AVL tree, unused for Heap tree because it is an array implementation).

1.3 Member Functions

Constructors

defines constructor.

Destructor

defines destructor.

Accessors

getRoot() returns the root of the tree.

getSize() returns number of elements in the tree.

getHeight() returns the height of the three.

getHeight(node) returns the height of the node in the argument (from the root).

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

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

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

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

preorder() performs preorder traversal.

postorder() performs postorder traversal.

inorder() performs in order traversal.

Mutators

clear() removes all the elements in the tree

insert(data) inserts data in the tree.

del(data) removes data from the tree.

Friends

no friends for this class.

2 Heap

In this part of the project, you need to implement an additional *MaxHeapTree Class*; create the UML diagram. 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 order defined on the keys. The main operations that a priority key allows to do are:

- Identify or remove the entry that has the smallest (largest) 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:

- Identify or remove the entry that has the smallest (or largest) key. It is the only one that can be removed quickly.
- Insert anything you want at a more specific location in the tree.

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 instantiate a *TreeNode* object 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

defines constructor.

Destructor

defines destructor.

Accessors

getMax() returns the root of the tree.

getSize() returns number of elements in the tree.

getHeight() returns the height of the tree.

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

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

print() prints the heap.

Mutators

clear() removes all the elements in the tree

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

delMax() 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 an additional *AVL Class*; create the UML diagram. 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 in a tree. 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 instantiate a *TreeNode* object for each of the nodes in the tree. Each node has a value (Type), and access to its parent, left and right child.

3.3 Member Functions

Constructors

defines constructor.

Destructor

defines destructor.

Accessors

getRoot() returns the root of the tree.

getSize() returns number of elements in the tree.

getHeight() returns the height of the tree.

getHeight(node) returns the height of the node in the argument (from the root).

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

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

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

find(data) returns a pointer to a node that holds the data in the argument.

preorder() performs preorder traversal.

postorder() performs postorder traversal.

inorder() performs inorder traversal.

inorder() performs inorder traversal.

Mutators

clear() removes all the elements in the tree

insert(data) inserts data in the tree. Tree must be kept balanced after the insertion.

del(data) removes data from the tree. Tree must be kept balanced after the deletion.

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 (name your executable proj3). Please choose the *string* data type for the values (or 'data') of your entries. The TA will choose one group to demo the project.

Format for the Menu Program

First, take a character, 'g', 'h', or 'a' (lowercase) that specifies whether we will be working with a general tree, a heap, or an AVL tree, respectively, and creates an instance of the tree. Next, give the options for each specific tree (please have them in this **EXACT** order for grading purposes):

General Tree or AVL Tree

1. Return root
2. Return size
3. Return height
4. Return height (node)
5. Is tree empty?
6. Return number of leaves
7. Return number of siblings (node)
8. Find node (data)
9. Print preorder
10. Print postorder
11. Print inorder
12. Clear tree
13. Insert (data)
14. Delete (data)
15. Exit

Max Heap

1. Return root
2. Return size
3. Return height
4. Is tree empty?
5. Return number of leaves
6. Print
7. Clear tree
8. Insert (key, data)
9. Delete
10. Exit

Submit the following files to Canvas (named **EXACTLY** as shown below **without zipping** for grading purposes: (you may also include implementation files with the header files if you would like)

1. menu.cpp
2. treeNode.h
3. linkedTree.h
4. maxHeapTree.h
5. avlTree.h
6. makefile (name your executable 'proj3')
7. project3.pdf

The rubric is as follows:

1. A program that does not compile will result in a zero
2. Runtime error and compilation warning 5%
3. Commenting and style 15%
4. Report 10%
5. Functionality 70% (functions were declared and implemented as required)