CS-202 Data Structures

Assignment 2

Trees

Due: 11pm on Friday, March 6, 2020

Late Submission Policy

10% per day deduction for up to 3 days after the due date

NOTE: In this assignment, height of a tree is the number of nodes in the path from root to its deepest descendant. For example, the tree with only two nodes (root and a child) will have height 2 and so on.

Part 1: Trees

In this part you will be implementing a general tree data structure where each node of a tree can have any number of children nodes. Please refer to **tree.hpp** which provides all the required definitions. Don't modify the header file of this part. You will be implementing the following functions:

Member Functions:

- Tree(node<T,S> *root)
  - Constructor
- ~Tree ()
  - Free the memory occupied by the pointers
- node<T,S>* findKey(T key)
  - Finds node with the given key and returns pointer to that node. NULL is returned if key doesn't exist.
- node<T,S>* findKeyHelper(node<T,S> *currNode, T key)
  - Helper function to be used in findkey(T) function
- bool insertChild(node<T,S> *newNode, T key)
  - Inserts the given node as the child of the given key. Returns true if insertion is successful, false if key doesn't exist. Insertion should also fail if another node with the same key as the new node already exists.
- std::vector<node<T,S>*> getAllChildren(T key)
  - Returns all the children of the node with given key. Should return an empty vector in case the node has no child or key doesn't exist.
- int findHeight();
  - Returns the height of the tree
- int findHeightHelper(node<T,S> *currNode)
  - Helper function to be used in the findHeight() function
- void deleteTree(node<T,S> *currNode)
  - Helper function to delete tree.
- bool deleteLeaf(T key)
  - Delete node with given key if and only if it is a leaf node i.e. have no child. Doesn't delete the root node even if it is the only node in the tree. Returns true on success, false on failure.
- node<T,S>* deleteLeafHelper(node<T,S> *currNode, T key)

- Helper function to delete leaf node.

You can test your functions using test1.cpp.

Part 2: BSTs and AVL Trees

In this part, you will be implementing a single class which will server for both BST and AVL tree. Refer to the **AVL.hpp** file for the class definitions. You will be implementing the following functions:

Member functions:

- AVL(bool isAVL)
  - Constructor initializes the isAVL flag. If isAVL flag is set, insertions and deletions will follow AVL property. Otherwise, it will be a simple BST.
- ~AVL()
  - Free the memory occupied by every node
- void insertNode(node<T,S>*)
  - Insert the given node into the tree such that the AVL (or BST) property holds
- void deleteNode(T k)
  - Delete the node with the given key such that the AVL (or BST) property holds
- node<T,S>* searchNode(T k)
  - Returns the pointer to the node with the given key. NULL is returned if key doesn't exist.
- node<T,S>* getRoot()
  - Returns the pointer to the root
- int height (node<T,S>* p)
  - Returns the height of the tree

In this task, you are free to declare any private function as per your need but the already declared functions should not be modified. **While deleting a node with 2 children, new root should be selected from right sub-tree.** test2.cpp is for BST testing while test3.cpp is for the AVL.

Part 3: Zambeel Enrollment System

In this part, you will be implementing very simple Zambeel enrollment system. Refer to the **Zambeel.hpp** file for the class definitions. Suppose that Zambeel only maintains course id and number of students currently enrolled in that course. In our scenario, course can represent key of a node whereas number of students can be stored as value. You will be implementing the following functions:

Member functions:

- Zambeel(bool isAVL)
  - If isAVL flag is set, this class will use AVL tree as underlying data structure. Otherwise, it will be a simple BST.
- ~Zambeel()
  - Release memory occupied by the underlying data structure
- bool istAddCourse(int course_id)
  - Function used by IST to add new course on Zambeel. Number of students in that course will be initialized to zero.
- bool istDropCourse(NodeT k)
  - Function used by IST to remove course from Zambeel.
- void istCleanEnrollment()
  - Function used by IST to delete all enrollments i.e. number of students in each course is reset to zero.
- bool stuAddCourse(int course_id)
  - Function used by students to enroll a particular course. It will simply increase the enrolled student count by one for that course. If course does not exist, false is returned.
- bool stuDropCourse(int course_id)
  - Function used by students to drop from a particular course. It will simply decrement the enrolled student count by one for that course. If the count is already zero or the course doesn't exist, false will be returned.
- bool stuCleanEnrollment(int course1_id, int course2_id)
  - Funciton used by students to swap two courses. Enrollment count for course1 will increase by one and decrease by one for course2. If count for course2 is already zero or any of the course do not exist, false will be returned.

Inside **Zambeel.cpp**, write a main function to add 10 distinct courses (by IST). Now generate 100 random requests of add, drop and swap (by students). Compare the time taken in completing 100 requests when underlying tree is BST vs AVL. Repeat the experiment for following 3 scenarios: Courses added by IST have course IDs in

1. the ascending order e.g. 100, 200, 202, 225, ...

2. order such that BST is always filled from left to right e.g. 200, 150, 250, 144, 177, 202, 252, …
3. random order

Fill the following table with the running time in milliseconds. You can use a smaller unit if values are too small.

	Case I	**Case II**	**Case III**
BST			
AVL			

Submit your table in a PDF file along with your source files zipped as PA2_rollnumber.zip.