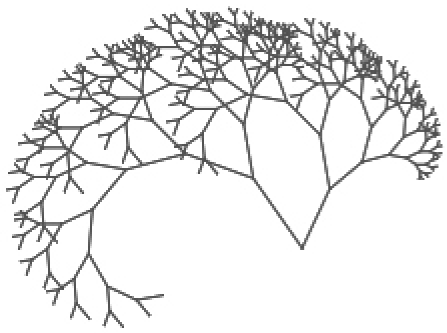
CS 319 Applied Programming

Fall 2018

Programming Assignment 4

Binary Search and AVL Trees



Instructor: Muhammad Saqib Ilyas

Assigned on: November 30, 2018

Due on: December 14, 2018 at 5 pm

Image source: https://yourbasic.org/algorithms/binary-search-tree/

Summary instructions

Binary Search Trees (BSTs) are a useful data structure that allow storage and retrieval of key-value pairs. However, BSTs don't provide efficient worst case performance for search, insert and delete. For this purposes, improved BSTs such as AVL tree were created.

In this assignment, you will implement a C++ class for an AVL tree. This class inherits from a C++ class for BST, which you will also implement. The BST class inherits from a class for Binary Tree, which you will also implement.

The summary instructions for this assignment are:

- Start reading this handout as soon as possible. The more you delay, the deeper you will be in trouble. This assignment will take some time.
- Read this handout carefully before starting implementation.
- Ample starter code has been provided.
- The code in the files bintreenode.h, bintree.h, bst.h and avl.h declares the classes for the node used in a binary tree, the binary tree itself, the BST and the AVL tree, respectively. The code in these three files may not be modified in any way.
- The implementation for the binary tree node should be done in bintreenode_impl.h, that for the binary tree class should be done in bintree_impl.h, that for the BST should be done in the bst_impl.h and that for AVL tree should be done in avl_impl.h.
- A skeletal main() function is provided to you in the file pa4.cpp. You are allowed to make changes to this main() function to test and debug your code. While grading your assignment, we may replace the pa4.cpp entirely, so don't implement any critical functionality here.
- Exercise caution with the use of pointer variables. For instance, be careful when calling functions or accessing data members using pointer variables. If the pointer is NULL or invalid, this may cause the program to crash.

Description of individual classes and/or files

Class BinTreeNode

This class defines a node in the binary tree data structure. It is templatized to be able to store generic keys. Ironically, the member to store the key is called val. Each binary tree node stores pointers to its parent, left child and right child. The parent child pointer should be NULL for the root node. In addition, the binary tree node includes two Boolean members to indicate if it is the root node or an external node.

An explanation of what each member function of this class is supposed to do, follows. The implementation of these functions must be provided in the file bintreenode_impl.h:

BinTreeNode<T>(): This is the default constructor. It should set all pointers to NULL and indicate that this object is an external node. Leave the val attribute unassigned.

BinTreeNode<T>(T val): This is a parameterized constructor that allows you to initialize the key for this node. All pointers should be assigned NULL. The node should neither be root nor external.

T Value(): This function returns the value stored in the key for this node.

BinTreeNode<T>* Parent(): This function returns the pointer to the parent node for this object.

BinTreeNode<T>* Left(): This function returns the pointer to the left child of this object.

BinTreeNode<T>* Right(): This function returns the pointer to the right child of this object.

int Depth(): This function returns the depth of this node.

int Height(): This function returns the height of this node.

bool isExternal(): This function returns true if this node is external, false otherwise.

bool isRoot(): This function returns true if this node is the root node, false otherwise.

void SetParent(BinTreeNode<T>* p): This function assigns the node pointed to by p as the parent of this node.

void SetLeftChild(BinTreeNode<T>* p): This function assigns the node pointed to by p as the left child of this node.

void SetRightChild(BinTreeNode<T>* p): This function assigns the node pointed to by p as the right child of this node.

void SetRoot(bool f): This function sets the root Boolean member to the value passed as argument.

void SetExternal(bool f): This function sets the external Boolean member to the value passed as argument.

void SetValue(T val): This function sets the key for this node equal to the value passed as argument.

void visit(): This function performs a visit operation on this node. Out of our great generosity. This function is already implemented for you.

Class BinaryTree

This class implements the binary tree data structure using the class BinTreeNode. This class maintains a pointer to the root node of the tree as the sole means of accessing the nodes in the tree. Some of the functions in this class have been declared private. The reason is that these functions are not needed in the public interface of the class, but help the public functions achieve their goals. A brief description of what each function in the class BinaryTree is expected to do follows:

void PreOrder(BinTreeNode<T>* p): This is a private function that should perform a pre-order traversal on the subtree rooted at node p.

void PostOrder(BinTreeNode<T>* p): This is a private function that should perform a post-order traversal on the subtree rooted at node p.

void InOrder(BinTreeNode<T>* p): This is a private function that should perform an in-order traversal on the subtree rooted at node p.

void Destroy(BinTreeNode<T>* p): This is a private function that removes the nodes from and frees the memory allocated for all the nodes in the subtree rooted at node p.

BinaryTree<T>(): This is the constructor for the class. Initialize an empty binary tree here.

~BinaryTree<T>(): This is the destructor for the class. If the tree is non-empty call the Destroy() function on the root node.

void PreOrder(), void PostOrder(), void InOrder(): These three functions are called by code outside the class to perform the pre-order, post-order and in-order traversal on the binary tree, respectively. These are already implemented for you.

int Size(): This function returns the size of the binary tree. It is already implemented.

int Height(): This function should return the height of the binary tree. Just call the Height() function of the root node.

bool isEmpty(): This function should return true if the tree is empty, false otherwise.

void AddRoot(T val): This function creates a new internal node with key equal to val. It creates two external nodes and sets them as the left and right children of the internal node. It then sets the newly create internal node as the root node of the binary tree. Ignore checking if the tree already has a root node.

void ExpandExternal(BinTreeNode<T>* e, T val): This function converts the external node e into an internal node by doing two things. First, it stores the key val at this node. Second, it creates two external nodes and assigns them as the left and right children of e.

void RemoveAboveExternal(BinTreeNode<T>* e): This function removes the node e and its parent from the binary tree, replacing the parent of e with the sibling of e. It also deallocates the memory reserved for e and its parent.

Class BST

This class inherits from BinaryTree and extends its functionality to realize a BST. All you need to do is to implement the following three functions:

BinTreeNode<T>* Insert(T val): This function implements the inserts a new node with key val in the BST, if it doesn't already exist and returns a pointer to the newly inserted node. If such a node already exists, it should return NULL.

BinTreeNode<T>* Delete(T val): This function deletes the node with key val from the BST and returns a pointer to its parent. If the key val doesn't exist in the BST, this function should return NULL.

BinTreeNode<T>* Search(T val): This function searches for a node storing the key val in the BST. If such a node is found, its pointer is returned. Otherwise, this function should return NULL.

Class AVL

Having implemented the BST, we can extend its functionality to realize an AVL tree. In this class, you need to implement the following functions:

void Rotate(BinTreeNode<T>* p): This is a private function that is called every time a node is inserted into or deleted from the AVL tree. In case of an insert, it is passed a pointer to the newly inserted node.

In case of a delete, it is passed a pointer to the parent of the deleted node. This function should check if a rotation is needed and perform it, as it was discussed in class.

bool isValid(BinTreeNode<T>* p): This is a private function that should check if the BST rooted at node p is a valid AVL tree by verifying the height balance property on ever node of the subtree rooted at p. Return true if the BST is an AVL tree, false otherwise.

bool isValid(): This function is already implemented for you. It calls the private isValid() function with the root node pointer to verify that the entire BST is valid. Return true if the BST is an AVL tree, false otherwise.

BinTreeNode<T>* Insert(T val): This function is called from outside the class to insert a new key into the AVL tree. It is already implemented for you.

BinTreeNode<T>* Delete(T val): This function is called from outside the class to delete a key from the AVL tree. It is already implemented for you.

The main() function

The main() function creates an AVL tree object t to store keys of type unsigned int. It then inserts integers from 0 to 14 into this tree. After inserting each integer, it performs a pre-order traversal and displays the current height of the tree to help you debug your code as you write it.

The main() function then performs various traversals on the final tree to help you verify the final output. It tries to delete a valid and an invalid key to make sure that the delete functionality is working correctly.

The main() function then deletes all the keys from the AVL tree to make it empty. It then inserts some randomly arranged numbers into the AVL tree to verify that the AVL tree works correctly for such input as well.

To demonstrate the difference between a BST and an AVL tree, both types of trees are then created. Values from 0 to 99 are inserted into this tree. The AVL tree keeps itself balanced maintaining a much smaller height compared to the BST.

Submission instructions

Once you are done, compress all the .cpp and .h files into one .zip file and upload to slate.nu.edu.pk. Do try to submit before the deadline to avoid any unnecessary hassle. Good luck!