Rashi Ghosh | 25086407 | rashighosh@ufl.edu

D A T A   S T R U C T U R E S

Assignment I

AVL Trees

This report provides an overview of program structure, function structure, and descriptions of each function. Please see code for more specific comments on implementation.

Table of Contents

GOAL

Implement a simple, small AVL tree that resides in main memory that can do 5 main things:

1. Initialize - Initialize()
2. Insert items - Insert(key)
3. Delete items - Delete(key)

4. Search for one specific item - Search(key)
5. Search for item(s) within a range - Search(key1, key2)

OVERVIEW

Node structure. Each node has 4 things:

1. data (int) - this is the data the node is holding
2. left (Node*) - pointer to node's left child
3. right (Node*) - pointer to node's right child
4. isRoot (bool) - keeps track of if the node is the root; is used for when rebalancing, if the root changes, to update which node in the AVL tree is the root

Global variables. These variables are referenced across functions, and are therefore declared outside of main:

1. outputFile (ofstream) - for outputting results from Search
2. AvlTree (Node*) - the AVL tree itself

FILE I/O AND PARSING

File is read in from command line arguments (in main), line by line. As each line is read in, command is parsed, and function is identified (extra work is done for Search, to determine if it is specific search or range search). Then, for each function, the arguments are parsed out and converted to integers using stringstream. Finally, the appropriate function is called, passing in the extracted argument.

The search functions output results directly to the output file (via the global outputFile variable).

5 MAIN FUNCTIONS

1. **void Initialize()**
   initializes AVL tree as an empty AVL tree.

2. **void Search(int key)**
   iteratively searches for key specified starting at the root of the AVL tree. If the key is found, prints that key. Otherwise, prints "NULL".

3. **void Search(int key1, int key2)**
   starts at root and searches for values in range of key1 and key2. Uses helper function ListItemsInRange (see Misc Helper Functions section for more details).

4. **void Insert(int key)**
   inserts specified key. First searches for where to insert key (if AVL tree is not empty), and uses a stack (trackStack) to keep track of all nodes visited. If the key already exists, it is not added.

Otherwise, the key is added, and a helper function called checkImbalances is called, passing the trackStack, for checking and resolving any imbalances (see (Re)balancing Helper Functions section for more details).

5. **void Delete(int key)**
   deletes specified key. First searches for where to insert key (if AVL tree is not empty), and uses a stack (trackStack) to keep track of all nodes visited. Once it is found, one of three cases is identified: either it is a (1) childless node, (2) a node with one child, or (3) a node with two children. (1) A childless node is trivially deleted. (2) A node with one child is deleted by having its parent point to its child, and then deleting that node. (3) A node with two children is deleted by replacing its value with the smallest item in its right subtree, and then deleting that smallest item in the right subtree (as it will not have two children).

   Once the node is deleted, a helper function called checkImbalances is called, passing the trackStack, for checking and resolving any imbalances (see (Re)balancing Helper Functions section for more details).

(RE)BALANCING HELPER FUNCTIONS

1. **void checkImbalances(stack<Node *> trackStack)**
   (Re)balancing helper function for Insert and Delete operations. Takes in the trackStack (which has all the nodes visited during Insert or Delete), backtraces through the nodes, checking for imbalances. The getHeight helper function is used to calculate balance factors (this function is detailed below).

   If the current node has a balance factor of 2, and its left child has a balance factor of 1, it is an LL imbalance, and the LL_imbalance helper function is called (this function is detailed later in this section).

   If the current node has a balance factor of 2, and its left child has a balance factor of -1, it is an LR imbalance, and the LR_imbalance helper function is called (this function is detailed later in this section).

   If the current node has a balance factor of -2, and its left child has a balance factor of -1, it is an RR imbalance, and the RR_imbalance helper function is called (this function is detailed later in this section).

   If the current node has a balance factor of -2, and its left child has a balance factor of 1, it is an RL imbalance, and the RL_imbalance helper function is called (this function is detailed later in this section).

2. **int getHeight(Node *n)**
   (Re)balancing helper function for checkImbalances. Recursive function that returns the height of a given node.

3. **void RR_imbalance(Node *n, Node *parent)**
   (Re)balancing helper function for resolving a Right-right imbalance via a left rotation. Pointers are rearranged between node with the imbalance, its parent, and its child/grandchildren. The way pointers are rearranged depends on if a parent exists, and if the node is a right or left child of its parent. Additional work is done if the node was originally the root, as it needs to be updated.

4. **void LL_imbalance(Node *n, Node *parent)**
   (Re)balancing helper function for resolving a Left-left imbalance via a right rotation. Same as RR_imbalance, but mirrored.

5. **void LR_imbalance(Node *n, Node *parent)**
   (Re)balancing helper function for resolving a Left-right imbalance via a left rotation followed by a right rotation. This is implemented by rearranging the pointers between the imbalance node, its parent, its child, and grandchild. The rearrangement of these pointers depends on if the node has a parent, and if it is a right or left child. Additional work is done if the node was originally the root, as it needs to be updated.

6. **void RL_imbalance(Node *n, Node *parent)**
   (Re)balancing helper function for resolving a Right-left imbalance via a right rotation followed by a left rotation. Implemented similarly to LR_imbalance, but mirrored.

MISC HELPER FUNCTIONS

1. **void ListItemsInRange(Node *t, int a, int b)**
   Helper function for range Search, or Search(key1, key2). Recursively traverses binary search tree inorder traversal, and checks if each node is between a and b.

7. **Node *createNewNode(int key)**
   Helper function for Insert. Creates a new node using the new operator (to allocate memory). Sets data element to the key that is passed in, left and right pointers to NULL, and isRoot to false.