Advances Data Structures (COP 5536)

AVL TREE

Index

1. **Project Description**

   AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. The data is given in the form (key) with no duplicates, and the requirement is to implement an AVL Tree to store the key values.

   The following operations are required:

   - Initialize (): create a new AVL tree
   - Insert (key)
   - Delete (key)
   - Search (key): returns the key if present in the tree else NULL
   - Search (key1, key2): returns keys that are in the range key1 ≤ key ≤ key2

2. **Running Instructions**

   The program has been made using C++ and g++ compiler. It has been tested on remote server thunder.cise.ufl.edu using ssh

   To compile the program, use the following command:
   $ make

   To run the program, use the following command:
   $ ./avltree <filename>

   To clean binary files and output_file.txt, use the following command:
   $ make clean

3. **Structure of the Program**

   The program is structured around two classes AVLTree and AVLNode along with utilities.cpp for utility functions.

   class AVLNode:

   The AVLNode class consists of the following data members:

   - key: The integer value of node
   - height: The integer value of the height of the tree rooted at this node
   - balance_factor: The integer value of the of difference between the height of the left subtree and the height of the right subtree of the node.
   - left: pointer to the left child
   - right: pointer to the right child

   The AVLNode class consists of the following member functions:

   - AVLNode(int key)
     The constructor for the AVLNode class

class AVLTree:

The AVLTree class consists of the following data members:

- root: the pointer to root node of the tree
- nodes: map of integer node value to pointer of the node. (for deleting nodes at the end of the program to prevent memory leak)

The AVLTree class consists of the following functions:

void insert(int data)

Parameters:

- data: integer value for node to be inserted

This is the public method for inserting a node into the AVL tree. It first creates a new AVLNode with data as the key. It also adds the node to the nodes map. Then, to do the insert it calls the other recursive insert method for insertion.

void deleteNode(int data)

Parameters:

- data: integer value for node to be deleted

This is the public method for deleting a node from the AVL tree. First step is searching for the node to confirm that the node with data as key exists in the tree. If it doesn't exist it returns else it calls the recursive deleteNode method to delete the node from the tree.

void search(int data)

Parameters:

- data: integer value for node to be search

This is the public method for searching a node from the AVL tree. If the node exists it calls the writeFile method to write the data to the output file else, it writes NULL to the output file. To do the search, it calls the recursive search method.

void search(int low, int high)

Parameters:

- low: integer value for lower limit of range
- high: integer value for upper limit of range

This is the public method for searching nodes within a range from the AVL tree. It calls the recursive search method which returns a vector of integer nodes values within the defined range. This method then converts the vector into a string which is written to output file through writeFile. If the vector is empty, NULL is written to the output file.

void clearTree()

This method clears all remaining nodes in the tree as C++ requires manual deletion of memory allocated in heap memory to prevent memory leak.

AVLNode *insert(AVLNode *parent, AVLNode *child)

Parameters:

- parent: pointer to node where comparison is done
- child: pointer to node being inserted into the tree

Returns: pointer to root node

Recursive method which compares the key of parent node and key of child node, and then traverses through a path. The child node then gets inserted at a leaf of the tree. After insertion, on the way up the path during the recursive calls, it calls the update and rebalance method for updating height and balance factor as well as rebalancing the subtree rooted at parent node.

bool search(AVLNode *node, const int &data)

Parameters:

- node: pointer to node where comparison is done
- data: reference to integer value being searched

Returns: Boolean True if node with data value exists in tree else False

Recursive method which compares the key of node and data value. Depending on the comparison, it can return true if the value has been found else it calls the recursive search on left subtree and right subtree.

std::vector<int> search(AVLNode *node, const int &low, const int &high)

Parameters:

- node: pointer to node where comparison is done
- low: integer value for lower limit of range
- high: integer value for upper limit of range

Returns: Vector of Integers (empty if no node is found)

Uses a stack to potentially traverse through all nodes for comparison and adds node to result vector if the node key is within range.

AVLNode *deleteNode(AVLNode *parent, const int &data)

Parameters:

- parent: pointer to parent node where comparison is done
- data: reference to integer value being searched

Returns: pointer to root node

Recursive method to find node to be deleted and then remove the pointer to child and have a pointer from grandparent to child. When the node is found, depending on number of children, pointer to child is chosen to be redirected to grandparent node. When there are two children, the max node value of left subtree is replaced with deleted node.

int leftSubtreeMax(AVLNode *node)

Parameters:

- node: pointer to node whose left subtree is traversed

Returns: integer value of max value node in left subtree

Traverses through rightmost path in left subtree to find last node in the path.

void updateHeightAndBF(AVLNode *node)

Parameters:

- node: pointer to node update is done

This method updates the height of the node by adding 1 to the max of left subtree height and right subtree height. The balance factor is also updated with difference between left subtree height and right subtree height.

AVLNode *rebalance(AVLNode *node)

Parameters:

- node: pointer to node where rebalance is initiated

Returns: pointer to root node of subtree

If the balance factor of node is +2 or -2, rebalance is initiated. Depending on balance factor of children one of the four cases is chosen for rebalance.

AVLNode *llCase(AVLNode *node)

Parameters:

- node: pointer to node where rebalance is initiated

Returns: pointer to root node of subtree after rebalance

The Left-Left rotation case.

AVLNode *lrCase(AVLNode *node)

Parameters:

- node: pointer to node where rebalance is initiated

Returns: pointer to root node of subtree after rebalance

The Left-Right rotation case.

AVLNode *rrCase(AVLNode *node)

Parameters:

- node: pointer to node where rebalance is initiated

Returns: pointer to root node of subtree after rebalance

The Right-Right rotation case.

AVLNode *rlCase(AVLNode *node)

Parameters:

- node: pointer to node where rebalance is initiated

Returns: pointer to root node of subtree after rebalance

The Right -Left rotation case.

namespace Utilities

Utilities file consists of some utility function for I/O operations and parsing of input.

void clearFile()

Clears data from output file before new read

void readFile(const std::string &file_name, AVLTree &tree)

Reads the input file line by line and calls the parseInput() function

void writeFile(const std::string &output)

Writes or Appends result of search operation to output_file.

std::vector<std::string> parseInput(const std::string &input)

Parses the string input for line read by readFile()

void parseCommand(const std::vector<std::string> &parsed_input, AVLTree &tree, bool initialized)

Calls the correct operation method depending on input

4. Conclusion and References

Conclusion:

- AVL tree named after inventors Adelson, Velskii, and Landisis is one of the first balanced binary search tree data structures invented. Due to rebalancing, it takes advantage of the O(logN) time complexity for search and other operations. AVL trees are mostly used in database applications where insertions and deletions may be infrequent but lookups for data is quite frequent.

References:

- Adelson-Velsky, Georgy; Landis, Evgenii (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences (in Russian). 146: 263–266