AVL Tree Dictionary and Approximate String Matching

1. Summary

The code implements a dictionary lookup system using an AVL Tree handelling insertion, and search operations efficiently even in the worst cases.

The program performs two primary functions:

1. Efficient Storage: It reads a list of strings (words) from an external file and stores them in memory using the AVL tree structure.
2. Approximate Search: It provides an interactive console allowing a user to input any string. If an exact match is found, it's returned. If not, the program finds and returns the closest matching word currently in the dictionary, based on lexicographical distance.

2. Data Structure Architecture

The core of the program is the Node struct, which defines the components of the tree. The node consits of two pointers pointing to the left and right children and a string containing the key and an integer containing the height. The height is stored to enable the calculation of a node's balance. The program then uses the single node creating a tree handelling insertion and search.

3. Component Analysis

3.1. Helping functions

These are the fundamental helper functions used to maintain the AVL tree's properties.

- height
    - Purpose: A null-safe getter for a node's height. An empty subtree is defined to have a height of 0.
    - Complexity: O(1)
- getBalance
    - Purpose: Calculates the balance factor of a node, which is the difference between the height of its left subtree and its right subtree by subtracting the hight of the right side from the hight of the left side.
    - The returned integer will either be -2 meaning the tree is right heavy and needs balancing or 2 meaning the tree is left heavy and needs balancing or between -1 and 1which are accepted cases that don't require balancing.
    - The caching of the height variable in each node causes the function to be O(1).
- updateHeight
    - Purpose: Recalculates a node's height after its children have been modified after an insertion or rotation.
    - The method of calculating the new height is by finding the max height from the childeren nodes causing it to be O(1)

3.2. Balancing functions

When an imbalance is detected, these functions are called to perform the necessary rotations to balance the tree again.

- leftRotate
    - Purpose: Corrects a right-heavy imbalance. It pivots the nodes counter-clockwise, making the right child (y) the new root of this subtree.
    - Complexity: O(1)
- rightRotate
    - Purpose: Corrects a left-heavy imbalance. It pivots the nodes clockwise, making the left child (x) the new root.
    - Complexity: O(1)

3.3. Insertion Algorithm

This function recursively inserts a new key and then rebalances the tree on its way back up the call stack.

1. Standard BST Insert: It first traverses the tree as a normal BST to find the correct nullptr location to create the new node.
2. Height Update: After insertion, it calls updateHeight on the current ancestor node.
3. Get Balance: It calculates the balance factor for this ancestor.
4. Rebalance : If the balance is more than 1 or less than -1, it triggers one of four rebalancing cases:
    - Left-Left (LL) Case: balance is more than 1 and key is smaller than the key in the left child solved with one call to the rightRotate function.
    - Right-Right (RR) Case: balance is smaller than -1 and key is bigger than the key in the right solved with one call to leftRotate function.
    - Left-Right (LR) Case: balance is more than one and key is bigger than the key in the left child solved by leftRotate for the left child followed by rightRotate for the node.
    - Right-Left (RL) Case: balance is less than -1 and key is smaller than the key in right child solved by rightRotate to the right child followed by leftRotate node.

3.4. Approximate Search Logic

The searching mechanism of the program.

- findPredSucc
    - Purpose: Finds the in-order predecessor (the largest key in the tree that is smaller than the target key) and the in-order successor (the smallest key larger than the key).
    - Logic: It performs a modified binary search. If the target key is smaller than the current node's key, the current node is a potential successor, so we store it as succ and move left. If the target key is larger, the current node is a potential predecessor, so we store it as pred and move right.
    - If an exact match is found, it finds the predecessor by locating the maximum value in the left subtree and the successor by finding the minimum value in the right subtree.
- findLexicographical
    - Purpose: Defines a custom "distance" metric between two strings.
    - Logic:
        1. It iterates through both strings, comparing them character by character. The distance is the absolute difference of the ASCII values of the first characters that do not match.
        2. If one string is a prefix of the other, the distance is the difference in their lengths.

- findClosest
    - Purpose: The main searching function.
    - Logic:
        1. First, it checks for an exact match.
        2. If no exact match exists, it calls findPredSucc to find the two neighbor words that bracket the target.
        3. It handles edge cases if the target is smaller than all words, pred will be nullptr and if the targetis bigger than all words, the succ will be a nullptr.
        4. It uses findLexicographical to calculate the distance from the target to the pred and from the target to the succ.
        5. It returns the node either predecessor or successor that has the smaller lexicographical distance.

4. Algorithmic Complexity Analysis

4.1. Time Complexity

The main purpose of using an AVL tree is to ensure that the search process will always remain O(log(n)) and in the worst case O(1.44 log(n)). Due to the balancing mechanisms and the efficient insertion and sorting the code complexity is O(log(n))

4.2. Space Complexity

The space complexity is completely dependent on the length of the dictionary making it O(n).

5. Main Function and Execution Flow

1. Initialization: Node* root is initialized to nullptr.
2. File Read: The program opens dictionary.txt. It reads words one by one in a while loop.
3. Tree Build: Each word read from the file is passed to the insert function. The root pointer is updated to the new root of the tree after each insertion in case a rotation at the root occurred.
4. The program then loops allowing the user to enter any number of strings and -1 to exit the loop.
5. Search: The user's input string is passed to findClosest.
6. Output: The program prints the key word from the node returned by findClosest.

6. Github