Fundamental of Data Structures and Algorithms
Assignment 3

Do Huynh

May 4, 2024

1 Algorithm Analysis

According to the ADT ”SmartDictionary” description and requirements, a data structure like the AVL tree can be a good candidate to implement this ”Smarat-Dictionary”. These trees maintain balance, ensuring that the tree’s height remains logarithmic relative to the number of nodes. Words will be stored based on their alphabetical order.

1.1 Operations

1.1.1 LookUp(S,w)

You first need to compare w with the root node. If w is equal to the root node then return the value of the root node, if w is bigger, then go to the right child of the root node, or else go to the left child if w is smaller than the root node. Repeat this process of comparing w to the new node until you find the word. If you compare w with a non-existing node such as *null*, the algorithm will return null or empty because w doesn’t exist in the dictionary.
Pseudo-code:
Assume that S is the root node of the dictionary

```
def LookUp(S, w):
    if S = null:
        return null
    elif w = S.key.word:
        return S.key.meanings
    elif w < S.key.word:
        return LookUp(S.left , w)
    else:
        return LookUp(S.right , w)
```

This algorithm has the worst-case running time of $O(log(n))$. Let’s say that the word you are looking for is at the end of the tree thus, you will have to traverse the tree’s height to find that word. Since the height of the AVL is $O(log(n))$ therefore, this algorithm will be $O(log(n))$

1.1.2 AddMeaning(S, w, m)

First, you will need to check for the existence of the word (this will be done using the *LookUp*() method), if it's null then create a word w with that m as the first meaning of the meaning list and append the word to the dictionary. Since you're using the AVL tree, you must consider re-balancing if needed after inserting the new word to maintain the tree's property. If it returns the list of meanings, append the new meaning to the list, traverse the tree to that word, and set the meaning list to the new one.
Pseudo-code:
Assume that S is the root node of the dictionary, w is a given word string, and m is the meaning (string type)

```
def AddMeaning(S, w, m):
    # Find the node with the specified word
    word = FindWord(S, w)
    if word is not null and m is not in word.meanings:
        word.meanings.append(m)
    else if word is null:
        new_word = Word(w)
        new_word.meanings = [m]

        # Assume this method is available and it handles
        # rebalancing internally
        insertWord(S, new_word)
```

Finding the word will take $O(log(n))$ times ($FindWord()$ method in the pseudo-code above) because the height of the tree is $O(log(n))$. If the word is in the dictionary, append the new meaning to the end of the word's meaning list and this takes constant time to complete. If the word is not in the dictionary, create a new word object with that meaning (this takes constant time) and insert the word into the dictionary (both tree-insert and re-balancing take $O(log(n))$)

$$O(log(n)) + O(log(n)) = O(2log(n))$$

Since the constant factor does not matter, the worst-case running time of this operation is $O(log(n))$

1.1.3 DeleteWord(S, w)

Look for where the word is in the dictionary. If the word node is a leaf, then remove it. If the word node has 1 child, you connect the child node to the word's parent node. If the word node has 2 children, you look for its successor and replace it with the successor. After all this, you have to check for the balance factor of this node, its parent and grandparents.
Pseudo-code:
Assume that S is the root node of the dictionary, w is a given word string

```
def DeleteNode(root, word):
    if root is null:
        return root
    elif word < root.key.word:
        root.left = DeleteNode(root.left, word)
    elif word > root.key.word:
        root.right = DeleteNode(root.right, word)
    else:
        if root.left is null and root.right is null:
            free(root)
        elif root.left is null:
            temp = root.right
            free(root)
            return temp
        elif root.right is null:
            temp = root.left
            free(root)
            return temp

        # if the node has 2 children, find its successor
        # in the right subtree
        temp = FindSuccessor(root.right)

        # copy the successor content to this node
        root.key.word = temp.word
        root.key.meanings = temp.meanings

        # delete the successor node
        root.right = DeleteNode(root.right, temp.word)

    return root

def DeleteWord(S, w):
    node = FindWord(S, w)
    if node is not null:
        S = DeleteNode(S, w)
        rebalancing(S)
```

In the $DeleteWord()$ function, you need to check for the existence of the node of that word first and that takes $O(log(n))$. You define a helper function called $DeleteNode(root, word)$ for recursively deleting the node. If the *word* is equal to the current *root* and the root node is a leaf then remove that root node or if the root node has 1 child then return the child node.

If the root node has 2 children, find its successor (the min value of the right subtree), replace the content of the root node with its child's content, and remove the successor.

After recursively removing the word, you have to re-balance the tree if needed and that takes $O(log(n))$

1.1.4 HasBetween(S, w1, w2)

Traverse through the tree to look for the node of w1 and the node of w2. Check for the left and right child of w1's node to see if it equals w2. If that is true then return False, if not then check the left and right child of w2's node and if there is a node equal to w1, return false. Otherwise, return True.
Pseudo-code:
Assume that S is the root node of the dictionary, w is the given word string

```
def HasBetween(S, w1, w2):
    # Find the node with the specified word
    node_1 = FindWord(S, w1)
    node_2 = FindWord(S, w2)
    if node_1 is null or node_2 is null:
        return False
    if node_1.left.key.word = node_2.key.word or
    node_1.right.key.word = node_2.key.word:
        return True
    elif node_2.left.key.word = node_1.key.word or
    node_2.right.key.word = node_1.key.word:
        return True
    return False
```

For each time you look for the node of the word, it will be $O(log(n))$, and comparing the lexicographic order will be a constant time hence, the worst-case running time for this algorithm is going to be $O(log(n))$

2 Programming Question

2.1 High-level Description

The algorithm works by computing all the sums of the equation

$$Ax_1 + Bx_2^2$$

where $-50 \leq x_i \leq 50$ using nested for loops and storing those sums as keys and value is the occurrences of the sums into a hash table. Next, compute the following:

$$m = S - (Cx_3^3 + Dx_4^4 + Ex_5^5)$$

and go to the hash table to look for m. If m is in the hash table, increase the return variable by the value of m; if it is not, add 0 to the return variable.

2.2 Algorithm Runtime Analysis

The process of computing $Ax_1 + Bx_2^2$ using 2 nested 'for' loops where $-50 \leq x \leq 50$ is going to take 100^2 operations in the worst-case scenario. Storing the result in the hash table will take $O(1)$ Therefore, the worst-case running time for this computation is $O(n^2)$

The next process is to calculate $m = S - (Cx_3^3 + Dx_4^4 + Ex_5^5)$ using 3 nested 'for'loops and it takes about 100^3 operations. In between every computation, the algorithm checks for the existing key in the hash table and increases the return variable if the key exists. It takes $O(n^3)$ loop through every permutation of x value and $O(1)$ to check and increase the return variable using the hash table. Hence, this process will take $O(n^3)$ in the worst-case scenario.

This algorithm is much faster than the naive approach because as mentioned in the description, that approach will take 100^5 operations which is $O(n^5)$. Using my algorithm, you can increase the performance by lowering the running time to $O(n^3)$ which is a significant optimization.

2.3 HashTable class implementation

I use the sum of each different $Ax_1 + Bx_2^2$ as the key and its occurrences to be the value and store them in a node.

Besides the $HashTable$ class, I created another class called $Node$ which has key, $value$, and $next$ properties. In my $HashTable$ class, my hash table is an array of $Node$ because I use chaining to handle collisions. More specifically, when collisions happen I will make the $next$ property of the node that I am considering to be the new head of the spot in the hash table. This operation only takes $O(1)$

My hash function utilizes the division method where the key will mod the length of the table and I can ensure that my key will never be out of range of the hash table's indices.

My hash table's size is 1037. The first reason I choose this size is that calculating $Ax_1 + Bx_2^2$ will give me at most $10^2 = 10000$ results (since every x_1 will be paired with 100 x_2 and there are 100 x_1) in the worst-case that the result is different every time. 1037 is a prime number and choosing a prime number will help my algorithm to distribute the keys evenly and uniformly to every possible bucket/slot in the hash table.