CSC263, Assignment #3

Brendan Neal[1], Filip Tomin[2]

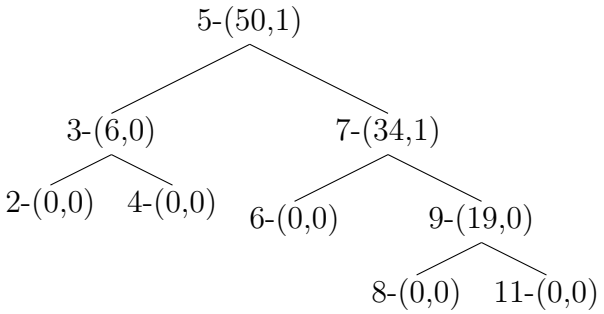
[1]Section L0201 Student number: 1001160236

[2]Section L0201 Student number: 1001329984

1. (a) **Data Structure Description**

Since the set S is comprised of integers, we can structure our AVL tree similarly to a binary search tree, such that for every node n in the tree, all nodes to the left of n have values less than n and all nodes to the right of n have values greater than n. Each node will also store an integer value representing the sum of the node's children and the balance factor of the node, as well as pointers to its two children and its parent. So for each node, the key is the value it obtains from S and the auxiliary information would contain pointers to the parent node, left child, and right child, as well as the sum of the node's children, and the balance factor.

As an example, here is a visualization of the set $S = \{3, 4, 5, 8, 11, 2, 7, 9, 6\}$ after inserting each value one at a time. Nodes will be represented in the form KEY-(SUM,BALANCE).

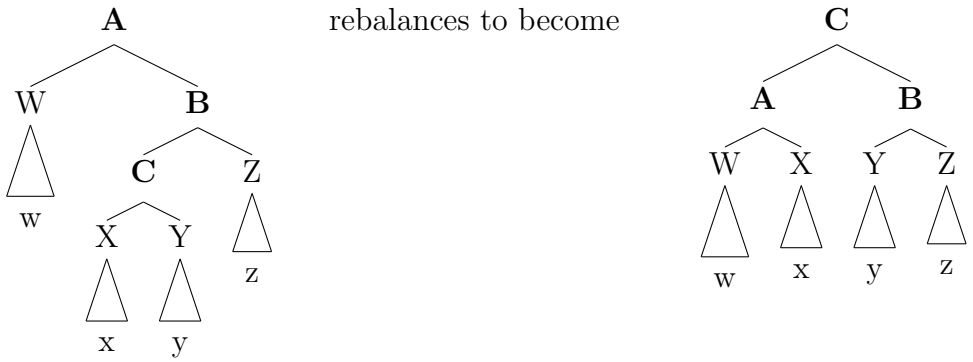


(b) **Algorithm Description**

Add(i):

Firstly, if the integer i already exists in S, nothing changes, so the first thing the algorithm would do is search for the value i in S. Conveniently, since our tree is structured like a BST, we can use binary search to find i, which is $O(logn)$. If i is not found, we can maintain our position on the tree and continue iterating through it as if we wanted to insert it, since searching for it would lead us to

where its supposed position would be had it existed and gets added as it would if it were a regular BST.
After the new node is added, we iterate up the tree adding the new key to the sums of each parent until we hit the root node, as well as updating the balance factor if necessary. Since we go through the whole height of the tree, this step takes $O(logn)$ steps.
After that, we need to make sure the tree is balanced. Doing a regular rotation takes $O(1)$ steps, including updating the pointers to children and parents as well as balance factors, but updating the sums is a bit trickier. In the following example, we'll show a generic right left rotation which we'll use to show that changing the sums for the nodes involved in a rotation is also within $O(1)$ time.



We will be referring to the sum values as the lowercase versions of the letters specifically, such that **a** is the sum value of **A**. It is also important to note that the sum the nodes store does not include their own value, so when taking the sum of a child node we also need to add the child's value to the child's sum. In the shown rotation, only **a**, **b** and **c** would have their values changed.

Initially, **a** would equal $w + W + \mathbf{b} + \mathbf{B}$, **b** would equal $\mathbf{c} + \mathbf{C} + Z + z$ and **c** would equal $x + X + y + Y$.
After the rotation, **a** can be written as $\mathbf{a} = w + W + x + X$, **b** can be written as $\mathbf{b} = y + Y + z + Z$ and **c** can be written as $\mathbf{c} = \mathbf{a} + \mathbf{A} + \mathbf{b} + \mathbf{B}$

Thus, we can set all the sum values within constant time, specifically only operations per rotation. Also, since every other type of rotation is similar to the shown type of rotation, we can claim this is true for all rotations without loss of generality. Then all rotations take an additional $O(1)$ steps to fix the sum values, which leaves rotations to still only take $O(1)$ steps in general.

In total, checking i is within S takes $O(logn)$, inserting and adding sum values takes $O(logn)$, and then rebalancing will take at most $logn$ rotations, at $O(1)$ steps each, so an additional $O(logn)$. Adding everything makes the algorithm run at $O(3logn)$ which is still equivalent to $O(logn)$ time.

Sum(t):
Pseudocode
Let:
ROOT be the root node of our tree.
NODE.value refer to the key of the node NODE.
NODE.left and **NODE.right** refer to the left or right child of node NODE respectively.
NODE.sum be the sum value of the node.

```
Sum(t):
    x = ROOT
    curr_sum = 0
    while t < x.value do
        if x.sum != 0 then
            x = x.left
        else
            return 0
    curr_sum = x.value + x.sum
    while t != x.value OR x.sum != 0 do
        if t > x.value then
            x = x.right
        else if t < x.value then
            curr_sum = curr_sum - x.value - x.right.value - x.right.sum
            x = x.left
    if x.value = 0 then
        return curr_sum
    else if t = x.value then
        return curr_sum - x.right.value - x.right.sum
```

Our $Sum(t)$ can basically be summarized as checking each left child until the current node's value is less than t, saving the current node's total sum (its sum including its value) and then continuing to iterate to either the left or right child based on the value of the nodes and if the node's value is greater than t, removing its value and its right child's value and sum from the current sum. Eventually, whether we find a node that has a value equal to t or we reach the bottom of the tree, we return the resulting sum.
Since $Sum(t)$ always moves the current node to the left or right child, in the worst case, it will end up at one of the leaves of the tree, and since the tree is an AVL tree by definition (it is balanced), the loops are limited to the height of the tree and can iterate about $logn$ times. Since everything within the while loops is done in constant time, this means the algorithm as a whole is in $O(logn)$.

2. (a) **Algorithm Description**

Let A be some array of integer arrays with $m \in \mathbb{N}$ ($m \in O(n)$) sub-arrays with indices $\{1, 2, 3, \ldots, m\}$, where $A[k]$ is the k-th sub-array. Let h be some hash function that maps its input to some $k \in \{1, 2, 3, \ldots, m\}$.

We will iterate over each value $x_i \in X$ and apply h to it to get $h(x_i)$. We will then append x_i to $A[h(x_i)]$. After we have inserted every $x_i \in X$ into some sub-array in A, we will iterate through each value $y_j \in Y$. We will apply h to $z - y_j$ to get $h(z - y_j)$. We will then go to sub-array $A[h(z - y_j)]$ and iterate through each value in the sub-array to see if there is some $a \in A[h(z - y_j)]$ such that $x = a$. If there is, we will stop the program and return true. Otherwise, we will repeat the same step as before with y_{j+1} in place of y_j.

Pseudo-code

Let:
$X[i] = x_i$
$Y[j] = y_j$

```
0 check_if_z_exists(z):
1     for i in {1,2,3, ..., n}:
2        subarray = A[h(X[i])]
3        subarray.append(X[i])
4     for j in {1,2,3, ..., n}:
5         subarray = A[h(z - Y[j])]
6         for k in {1,2,3,...,length(subarray)}:
7             if subarray[k] == (z - Y[j]):
8                 return True
9     return False
```

(b) **Expected Runtime of Algorithm:**
Assumptions:
Simple Uniform Hashing Assumption (SUHA) - assume that h's mapping of its elements to indices of A obeys a uniform probability distribution (independent of any values hashed before). In other words, the probability that a value will be hashed to any value is $\frac{1}{m}$. Moreover, if n values have been hashed into A, assume that $\forall k \in \{1, 2, 3, ..., m\}$, the number of elements at $A[k]$ is $\frac{n}{m} = \alpha$. Since $m \in O(n)$, $O(\alpha) = O(n/m) = O(1)$.

Hash Function Time Complexity - assume that h runs in $O(1)$ time

Analysis:
Under SUHA, the length of each sub-array is α. Therefore, using linear search to find $z - y_j$ in the sub-array (lines 6 to 8) will take $O(\alpha)$ time. However, we have verified above that $O(\alpha) = O(1)$ because $m \in O(n)$. Thus, searching through each sub-array will take constant time.
Since we are looping through 1 through n (inclusive), and using constant time search/hashing algorithms to find $z - y_j$ in each associated sub-array ($A[h(z - y_j)]$, the time it takes to run linear search on every $z - y_j$ is in $O(n)$. Therefore, the algorithm runs in $O(n)$ time.

(c) In the worst-case, the hash function used would put every x value into the same subarray, such that for all i, $h(x_i) = k$ where k is any index available on the hash table. Not only that, but there would not be any values such that $z = x_i + y_j$, so the algorithm would not find an answer.
In this worst-case, when trying to find compatible x_i and y_j values, we would need to iterate through every possible y value, and for each y we'd need to iterate through every x value. Neither iteration will end early since the answer cannot be found, and each iteration takes n steps since both sets have n distinct values. Since we have a nested loop here, with both loops taking n steps, this algorithm would be in $O(n^2)$.
To show this algorithm is also within $\Omega(n^2)$, we need to show an example input where the algorithm takes at least n^2 steps. We take the set X to be the set of even numbers, and Y to be the set of odd numbers, with both sets having the same number of values, (length = n). Then we try to search for some z where z is an arbitrary even number. Since it is impossible to have an even $x \in X$ and an odd $y \in Y$ be equal to another even number, let alone z. We will iterate through every y and internally iterate through every x. Since there are n distinct x and y values, we will iterate a total of n^2 times. Thus, showing that the algorithm is in $\Omega(n^2)$.

Since the algorithm is in both $O(n^2)$ and $\Omega(n^2)$, the algorithm is, by definition,

in $\Theta(n^2)$.