CSC263 Fall 2017 Assignment 2

**Name:** Ruijie Sun, Guanchun Zhao
**SN:** 1003326046, 1002601847

**Q1.** SOLUTION

(a). Pseudo Code:

```
def make_weight_balanced_tree(nodes):
    if len(nodes) == 0:
        return NIL
    else:
        median = len(nodes) // 2
        t = TreeNode(nodes[median])
        t.left = make_weight_balanced_tree(nodes[:median])
        t.right = make_weight_balanced_tree(nodes[median+1:])
```

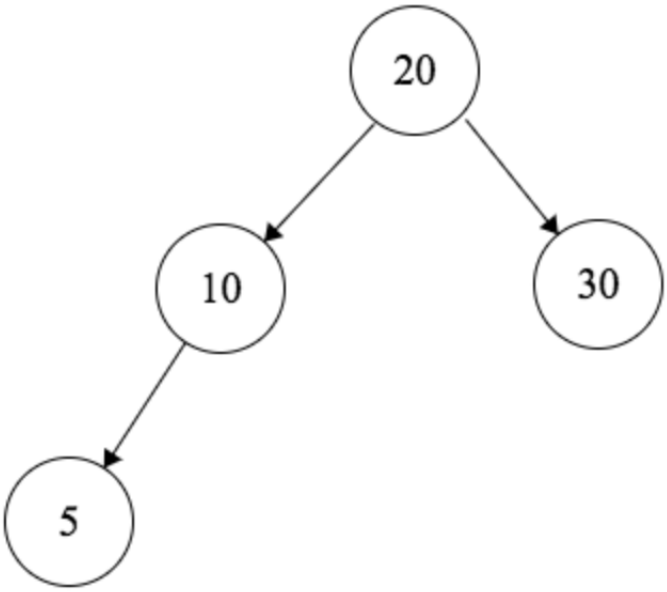
$$\mathrm{T(n)} = \begin{cases} 1 & \text{if } n = 0 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 0 \end{cases}$$

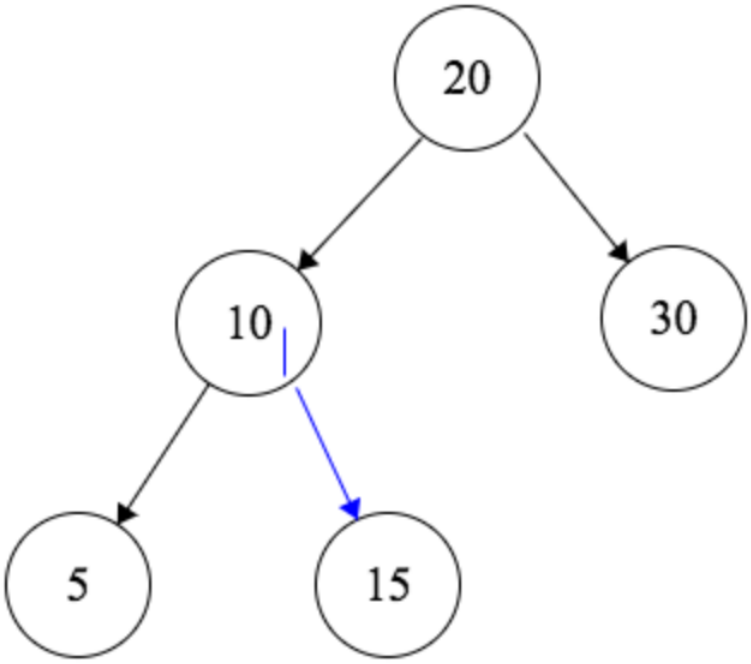
By using Mater Theorem, a = 2, b = 2, d = 0, thus $T(n) \in \Theta(n)$

(b). Prove that the height of a weight balanced BST is O(logn)

Given a arbitrary weight balanced BST, since for every node, the number of nodes in its two subtrees differ by at most 1, it means the height of the two subtrees must differ by at most one as well. So this weight balanced BST is a AVL tree. As we have learnt in class, the height of AVL Tree $\in O(log(n))$. Therefore, the height of a weight balanced BST is $O(log(n))$.

(c). No. For the weight balanced BST below, when we insert 15 in it, it needs reblance operation since number of nodes in left subtree is bigger than number of nodes in right subtree by 2. However, if it is a AVL tree, it doesn't need reblance operation in this case.


20
10
30
5


20
10
30
5
15

**Q2.** SOLUTION

(a). Pseudo Code:

```
1  #t is the given BST, and total_price is m
2  def find_two_items(t, m):
3      list_ = []
4      max = TreeMaximum(t)
5      node = TreeMinimum(t)
6      list_.append(node)
7      # Start inorder traversal
8      while node.value != max.value:
9          node = Successor(node)
10         list_.append(node)
11         # According to the property of Binary Search Tree,
12         # We know that list_ is sorted.
13     i = 0;
14     j = len(list_)-1;
15     while i != j:
16         if list_[i].price+list_j[j].price > m:
17             i += 1
18         elif list_[i].price+list_j[j].price < m:;
19             j -= 1
20         else:
21             return list_[i], list_[j]
22     return "There is no such two items"
23
```

(b). Want to Prove that $T(n) \in \Theta(n)$, where n is the size of t

(1). Want to prove that $T(n) \in O(n)$

In line-8 while loop, since it describes an inorder traversal of a binary search tree, its running time is n according to Tutorial-Week4.

In line-15 while loop, since the case when i == j would happen in the worst case , and there is an increment of $a_j$ by 1 or a decrement of $b_j$ by 1 in each iteration, it takes at most n times of basic operations.

Therefore, The upper bound running time $T(n) = n + n = 2n$

Now, prove that $T(n) \in O(n)$

Prove that $T(n) \in O(n^2)$

Want to show that $\exists\ b \in N$ and $c \in R^+$ such that $n > b \implies T(n) \leq c * n$.

Let b = 0, c = 5

By assumption, Let n $\in N$ and n> 0

$5n - 2n$
$= 3n$ (Because n $> 0$)
$\geq 0$
Thus, $5n \geq 2n$.
Therefore, $T(n) \in O(n)$.

(2). Want to find an input family which has $T(n) \in \Theta(n)$
Set input family I as all binary search trees whose each internal node only contains left subtree.
For input family I, we could easily get the running time of line-8 while loop is n since each node has no right subtree. Thus, we could get sorted list list_.
Thus, let m be the sum of list_[0] + list_[1]. (m is an input)

In line-15 while loop, since j would decreases to 1, it only $\Theta(n)$ time.
Thus, its running time is 2n $\in \Theta(n)$ .

Therefore, $T(n) \in \Theta(n)$.

**Q3.** SOLUTION

(a). Each node t stores three attributes: Closest-Pair(t), Max-in-Left(t), Min-in-Right(t).
Note: Max-in-Left(t) returns the node with the maximum key in its left subtree, while Closest-to-Right(t) returns the node with the minimum key in its right subtree.
In addition, the values of the three attributes of each leaf is infinity.

(b). Since each node has attribute Closest-Pair(t), we could directly look at the attribute of the root, which takes running time $\Theta(1)$

(c). (1). For Max-in-Left(t),

(i). For AVL-Insert(root, x)
The general idea is that Update-Left(t, x) updates Max-in-Left(t) by choosing the bigger one from (the original Max-in-Left.key, x) if root.key $<$ x. Notice that it takes $\Theta(1)$.

Thus, recursively, when x $<$ root.key, we would insert x to the left subtree, which would affect Max-in-Left(root).
If there is no rebalance need, we could just call Update-Left(root, x).
If there is a rebalance need, we should go to the AVL-REBALANCE-TO-THE-RIGHT(root).
Thus, we need look at AVL-ROTATE-TO-THE-LEFT and AVL-ROTATE-TO-THE-RIGHT.
We only need to update root.left in AVL-ROTATE-TO-THE-LEFT, and root in AVL-ROTATE-TO-THE-RIGHT, which still takes a constant time.
Thus, the running time of Insert is still $\Theta(log(n))$

(ii). For AVL-DELETE(root, x),
The general idea is that Update-Left(t, x) updates Max-in-Left(t) with t.parent if root.key $<$ x and the original Max-in-Left.key $==$ x. Notice that it takes $\Theta(1)$.

(The same reason with the above one)
Thus, recursively, when x $<$ root.key, we would remove x from the left subtree, which would affect Max-in-Left(root).
If there is no rebalance need, we could just call Update-Left(root, x).
If there is a rebalance need, we should go to the AVL-REBALANCE-TO-THE-RIGHT(root).
Thus, we need look at AVL-ROTATE-TO-THE-LEFT and AVL-ROTATE-TO-THE-RIGHT.
We only need to update root.left in AVL-ROTATE-TO-THE-LEFT, and root in AVL-

ROTATE-TO-THE-RIGHT, which still takes a constant time.
Thus, the running time of Delete is still $\Theta(log(n))$

(2). For Min-in-Right(t),

(i). For AVL-Insert(root, x)
The general idea is that Update-Right(t, x) updates Min-in-Right(t) by choosing the smaller one from (the original Min-in-Right.key, x) if root.key> x. Notice that it takes $\Theta(1)$.

Thus, recursively, when x > root.key, we would insert x to the right subtree, which would affect Min-in-Right(root).
If there is no rebalance need, we could just call Update-Right(root, x).
If there is a rebalance need, we should go to the AVL-REBALANCE-TO-THE-LEFT(root).
Thus, we need look at AVL-ROTATE-TO-THE-LEFT and AVL-ROTATE-TO-THE-RIGHT.
We only need to update root in AVL-ROTATE-TO-THE-LEFT, and root.right in AVL-ROTATE-TO-THE-RIGHT, which still takes a constant time.
Thus, the running time of Insert is still $\Theta(log(n))$

(ii). For AVL-DELETE(root, x),
The general idea is that Update-Right(t, x) updates Min-in-Right(t) with t.parent if root.key > x and the original Min-in-Right.key == x. Notice that it takes $\Theta(1)$.

(The same reason with the above one)
Thus, recursively, when x > root.key, we would insert x to the right subtree, which would affect Min-in-Right(root).
If there is no rebalance need, we could just call Update-Right(root, x).
If there is a rebalance need, we should go to the AVL-REBALANCE-TO-THE-LEFT(root).
Thus, we need look at AVL-ROTATE-TO-THE-LEFT and AVL-ROTATE-TO-THE-RIGHT.
We only need to update root in AVL-ROTATE-TO-THE-LEFT, and root.right in AVL-ROTATE-TO-THE-RIGHT, which still takes a constant time.
Thus, the running time of Delete is still $\Theta(log(n))$

(3). For Closest-Pair(t),
Because Closest-Pair(t) = minClosest-Pair(t.left), Closest-Pair(t.right), Max-in-Left(t), Min-in-Right(t) only depends on t itself and the children of t. Thus, by the theorem we learned in Tutorial-Week4, it does not affect the running time of the operations of AVL-Tree.

**Q4.** SOLUTION

(a). For Algorithm 1: 394
For Algorithm 2: 6385036879

(b). For Algorithm 1,
Property 1: For a fixed input, the output is fixed. Thus it has this property.
Property 2: The runtime of this algorithm is O(n). Thus it is quick to compute the hash value for any given message.
Property 3: It doesn't have this property. For example, if we have a hash value 100, in order to get the corresponding message, by ignoring signs whose ASCII value is bigger than 100, we don't need to try all combinations.
Property 4: This algorithm doesn't have this property. If we change "abcd" into "abce", the new hash value will be 395.Thus the small change doesn't result extensively change.
Property 5: For message "22" and "d", their hash values are both 100. Thus it doesn't have this property.

(c). Compared to Algorithm 1, Algorithm 2 also has the same good properties. Furtherly, it has some properties that Algorithm lacks. For example, for property 4, a small change of input will lead extensive change of output in Algorithm 2. If we change "abcd" into "ebcd" in algorhitm 2, the hash value will be 6385180627. Compared to the old version hash value 6385036879, the new one has huge change. Thus it is better than algorithm 1.

(d). MD5: e2fc714c4727ee9395f324cd2e7f331f
SH256: 88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589