Name: Georges Hatem

CS 590 Homework 3: Binary Search Trees Application Exercises

Due Date: February 13, 2022

Problem 4.7.42:

Describe an efficient scheme for designing an automated way to add and remove dogs from this website (as they are respectively put up for adoption and placed in loving homes):

An efficient scheme for designing an automated way to add and remove dogs from this website is using AVL Tree.

For adding dogs, we will do that by using the normal way of adding to a Binary Search Tree (BST). And then, we will rebalance the tree to make it an AVL Tree. We will structure the Tree based on dog's age like stated in the exercise statement.

For removing dogs, we will do that by using the Algorithm for removing a specific node from an AVL Tree and then rebalancing the tree to keep it as an AVL Tree.

The pseudocode for adding dogs is as follows:

```
Algorithm insert(k, e):
    Input: An element e with its search key k to
    Add
    Output: Addition of element e with key k to
    the AVL Tree.

    Let w ← TreeSearch(k, T.root())
    While w is an internal node do
        Let w ← TreeSearch (k, T.leftChild(w))
    Expand w into an internal node with two
    external-node children
    Store(k, e) at w
    rebalanceAVL(w, T)
```

```
Algorithm TreeSearch(k, v):
    Input: A search key k, and a node v of
    a binary search tree T
    Output: A node w of the subtree T(v) rooted
    at v, such that either w is an internal node
    storing key k or w is the external node
where
    an item with key k would belong if it existed

    if v is an external node then
        return v
    if k = key(v) then
        return v
    else if k < key(v) then
        return TreeSearch(k, T.leftChild(v))
    else
        return TreeSearch(k, T.rightChild(v))
```

```
Algorithm rebalanceAVL(v, T):
    Input: A node v, where an imbalance may
    Have occurred in an AVL tree, T
    Output: An update of T to now be balanced

    v.height ← 1 + max{v.leftChild().height, v.rightChild().height}
    while v is not the root of T do
        v ← v.parent()
        if |v.leftChild().height – v.rightChild().height| >1 then
            Let y be the tallest child of v and
            let x be the tallest child of y
            v ← restructure(x)
        v.height ← 1 + max{v.leftChild().height, v.rightChild.height}
```

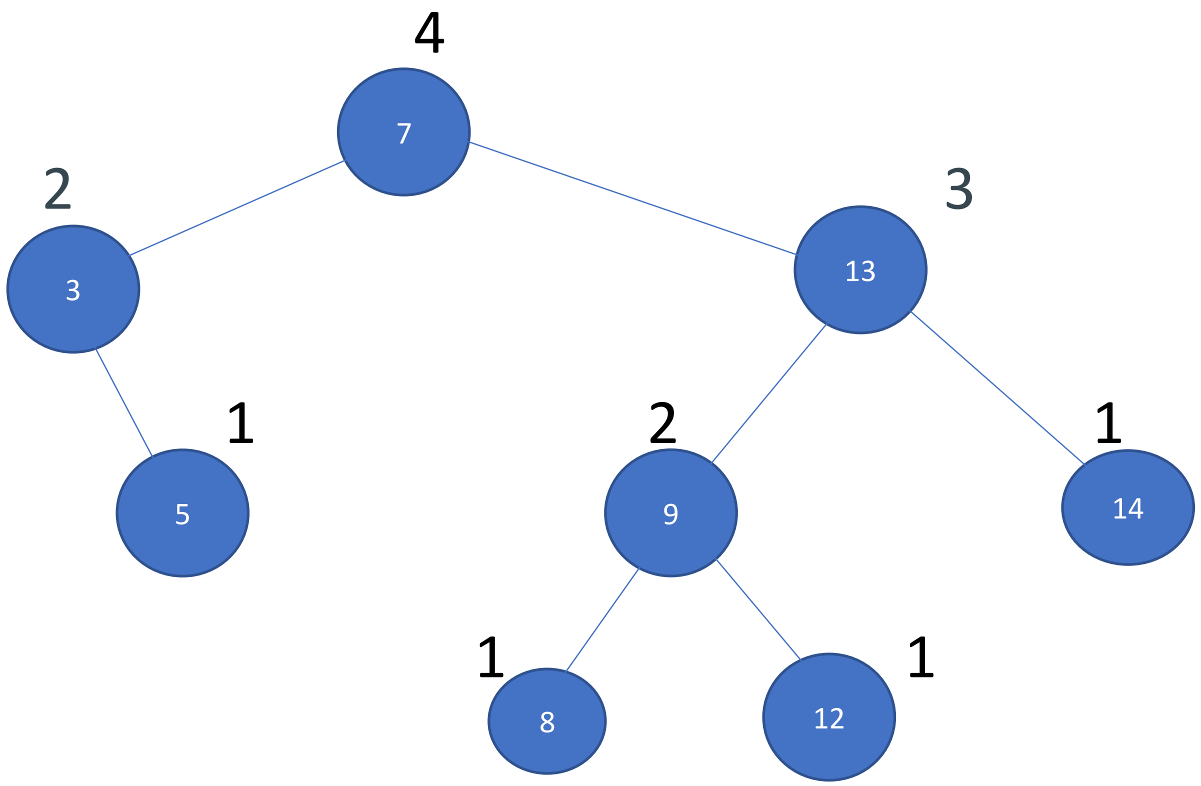
The pseudocode for removing dogs is as follows:

```
Algorithm removeAVL(k, T):
    Input: A key k and an AVL Tree T
    Output: An update of T to now have an
    Item (k, e) removed
    V ← IterativeTreeSearch(k, T)
    If v is an external node then
        Return "There is no item with key K in T"
    if v has no external-node child then
        Let u be the node in T with key nearest
        to k
        Move u's key-value pair to v
        v ← u
    Let w be v's smallest-height child
    Remove w and v from T, replacing v with
    w's sibling, z
    rebalanceAVL(z, T)

Algorithm IterativeTreeSearch(k, v):
    Input: A search key k, and a node v of
    a binary search tree T
    Output: A node w of the subtree T(v) rooted
    at v, such that either w is an internal node
    storing key k or w is the external node
    where an item with key k would belong
    if it existed
    if v is an external node then
        return v
    if k = key(v) then
        return v
    else if k < key(v) then
        return IterativeTreeSearch(k, T.leftChild(v))
    else
        return IterativeTreeSearch(k, T.rightChild(v))
```

```
Algorithm rebalanceAVL(v, T):
    Input: A node v, where an imbalance may
    Have occurred in an AVL tree, T
    Output: An update of T to now be balanced

    v.height ← 1 + max{v.leftChild().height, v.rightChild().height}
    while v is not the root of T do
        v ← v.parent()
        if |v.leftChild().height – v.rightChild().height| >1 then
            Let y be the tallest child of v and
            let x be the tallest child of y
            v ← restructure(x)
        v.height ← 1 + max{v.leftChild().height, v.rightChild.height}
```

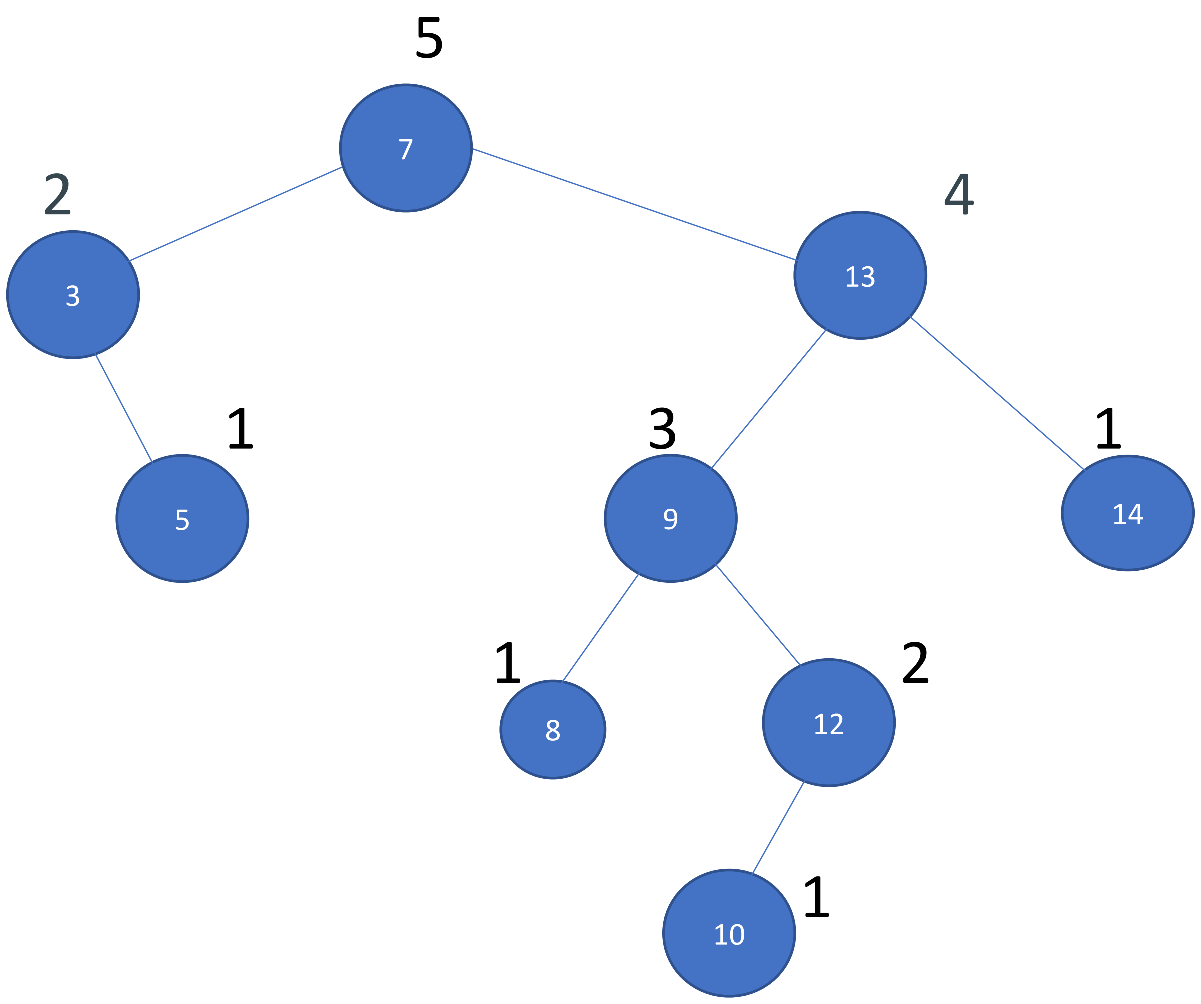
Sketch your methods for adding and removing dogs, and characterize the running times of these methods in terms of n, the number of dogs currently displayed on the website:

For sketching our method, let's suppose we have the following AVL Tree for dog's age:

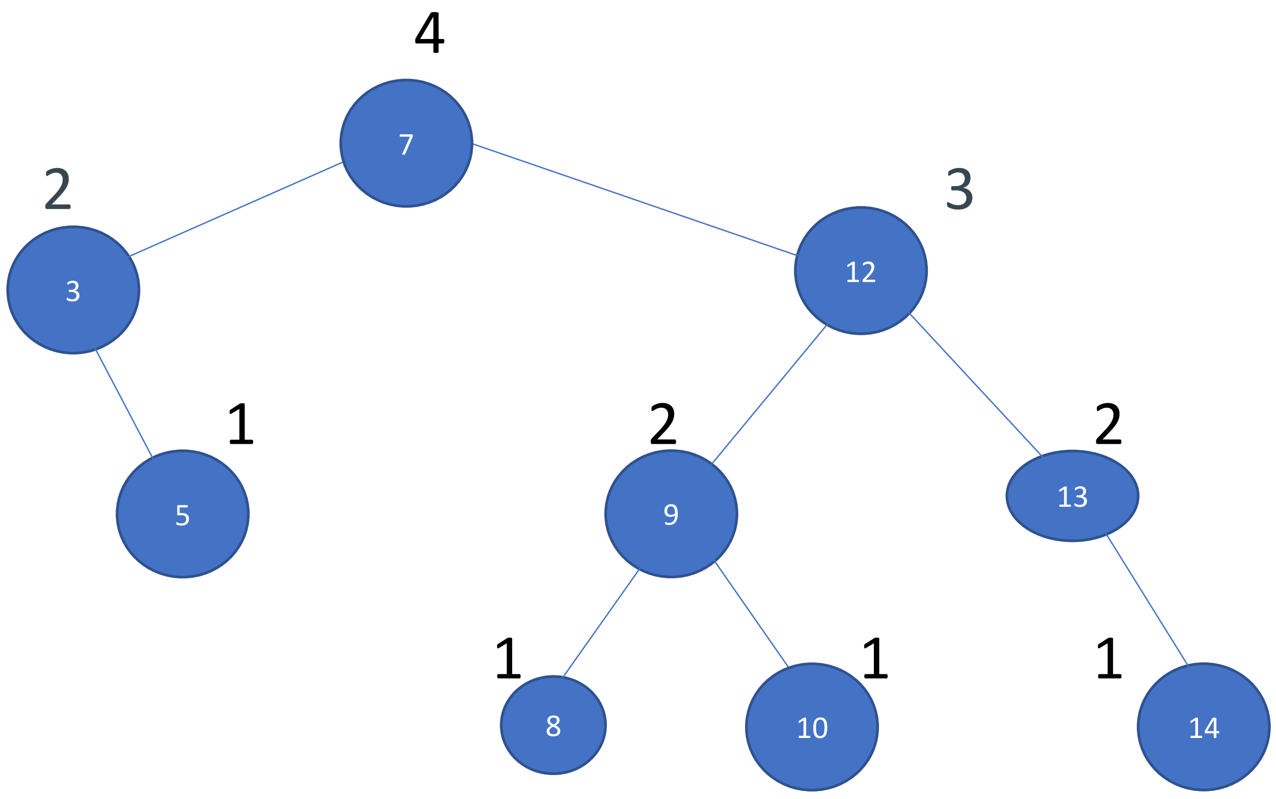


For Insertion:

Now, let's add a dog of age 10, we will get the following:

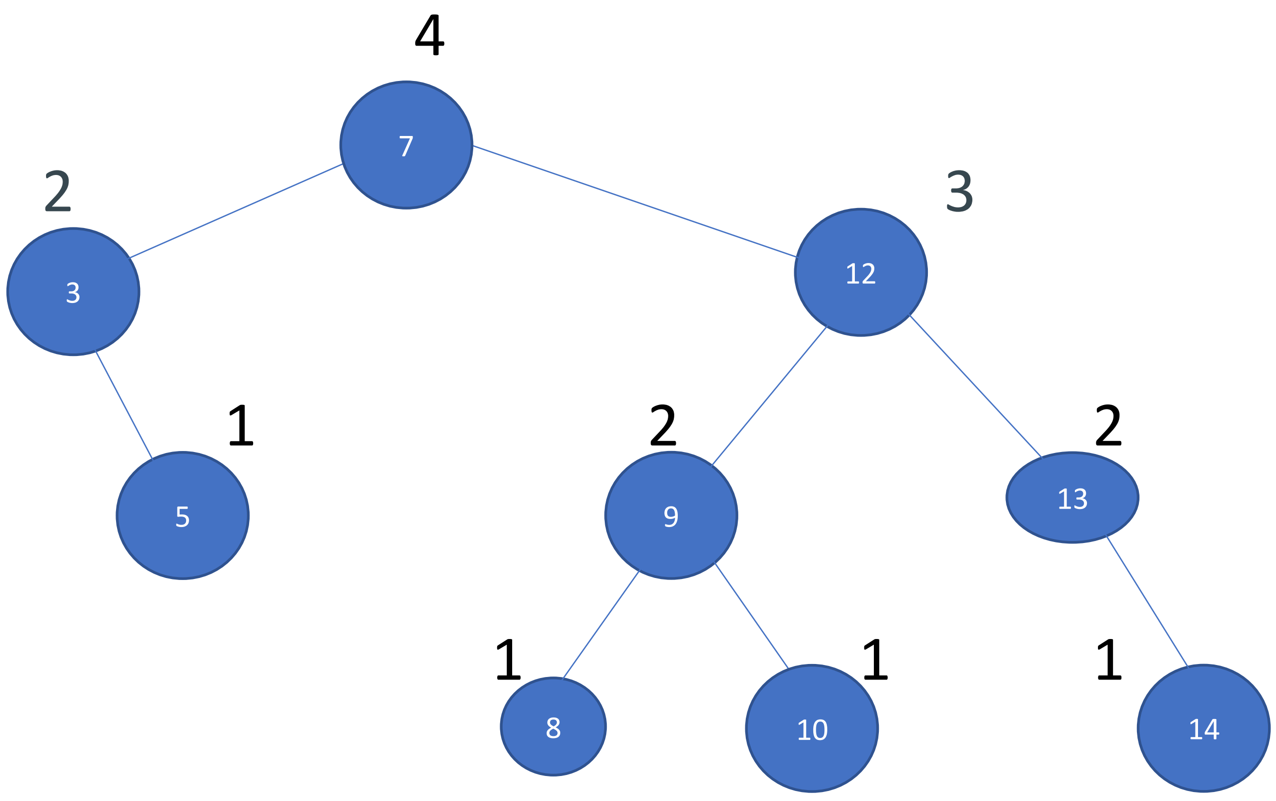


The Tree is not balanced, so let's balance it to make it back an AVL Tree:

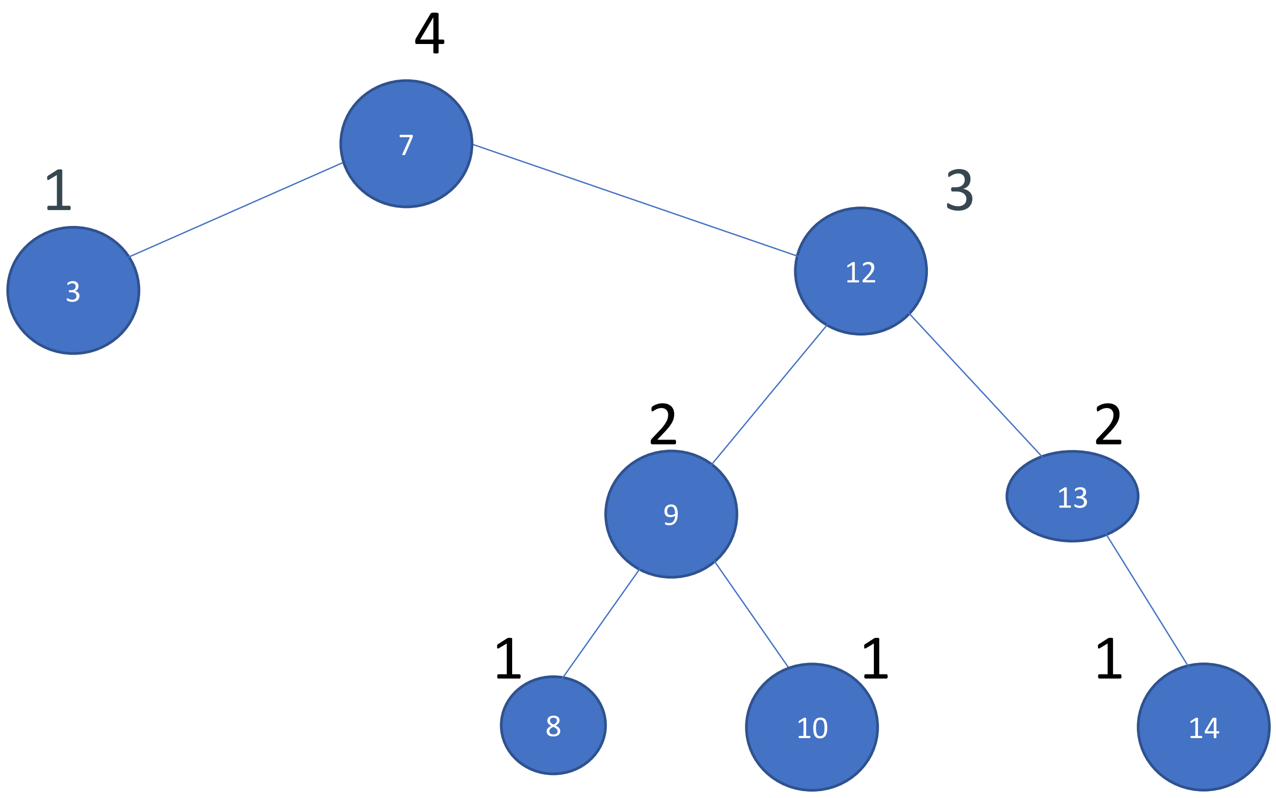


For Removal:

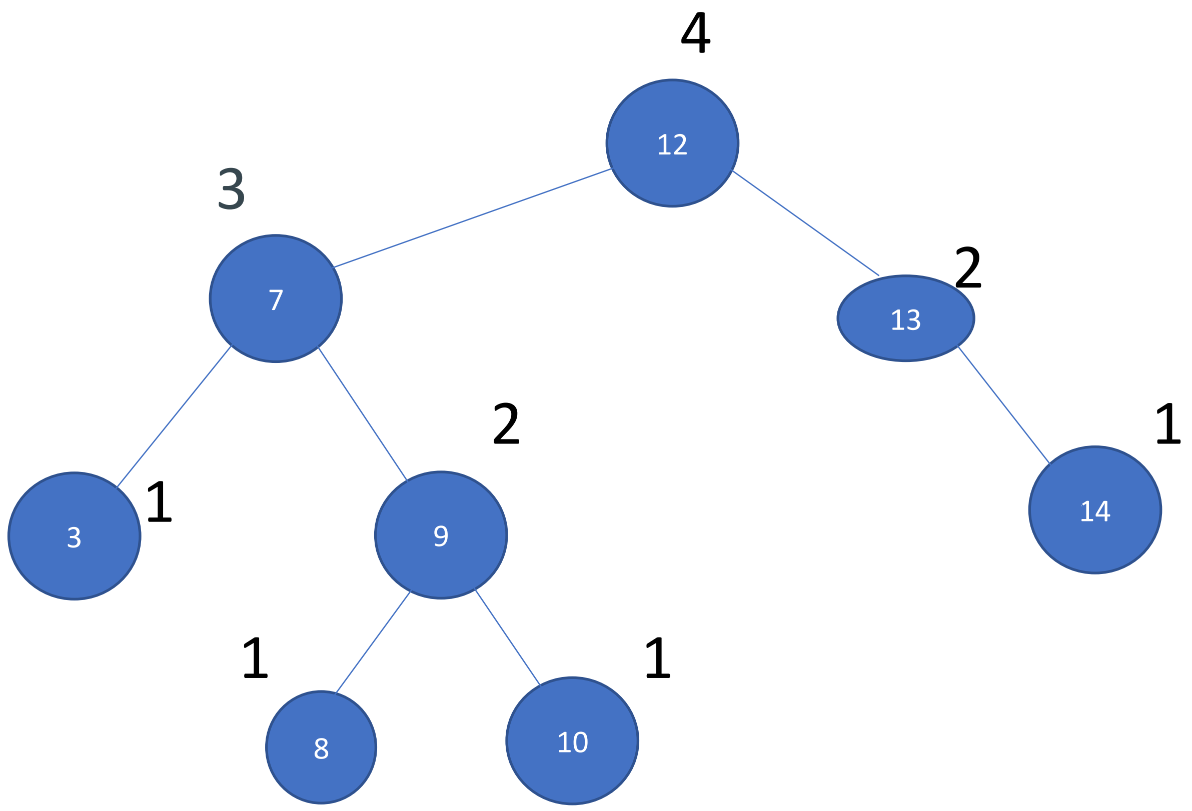
Let’s remove node 5 from this below:



Removal Process:



Now, the tree is not balanced. So, let's balance it to make it back as an AVL Tree:


4
12
3
7
2
13
3
1
2
9
1
14
1
8
1
10

The running time of the insert Algorithm method is O(log(n)), and this is indicated in Table 4.3.1

The running time of the remove Algorithm method is O(log(n)), and this is indicated in Table 4.3.1