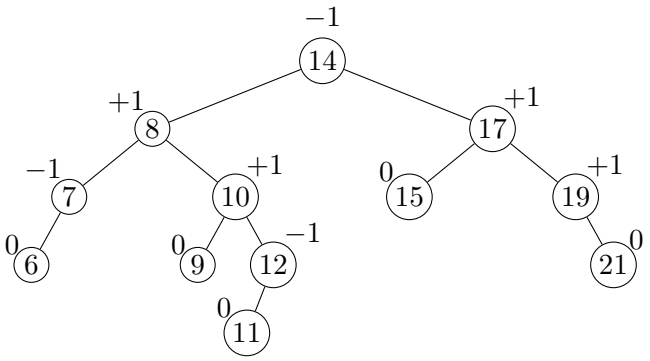
CSC263: Assignment 2

Junjie Cheng, Jiayun Liu, Zi Hao Lin

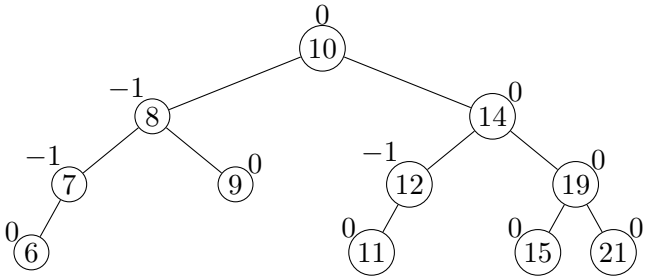
February 9th, 2017

1. **Solution:** Written by Zi Hao Lin, revised by Junjie Cheng and Jiayun Liu

(a)

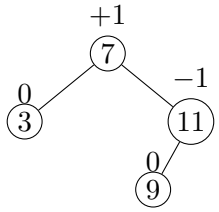


(b)

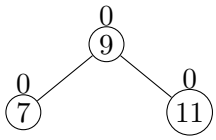


2. **Solution:** Written by Jiayun Liu, revised by Junjie Cheng and Zi Hao Lin

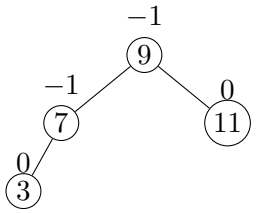
(a) Disprove:
Let T be:



Then $T' = DELETE(T, 3)$ will be:

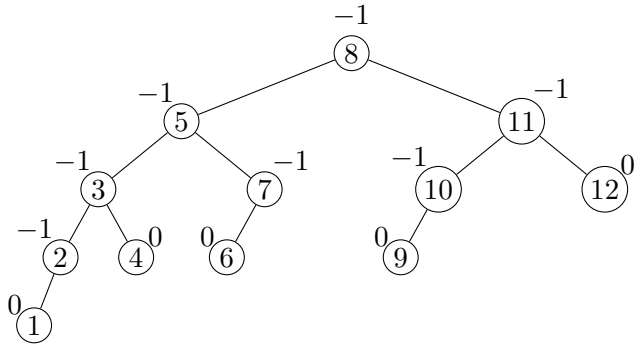


$T'' = INSERT(T', 3)$ will be:

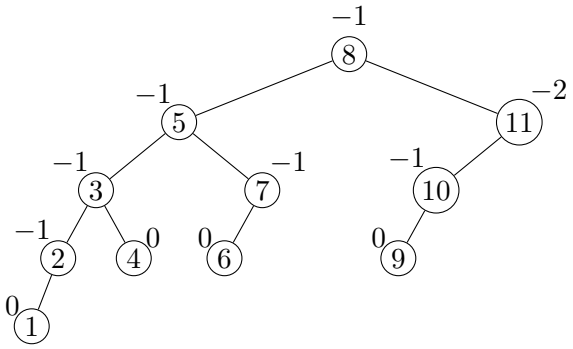


$T'' \neq T$. Hence statement disproved.

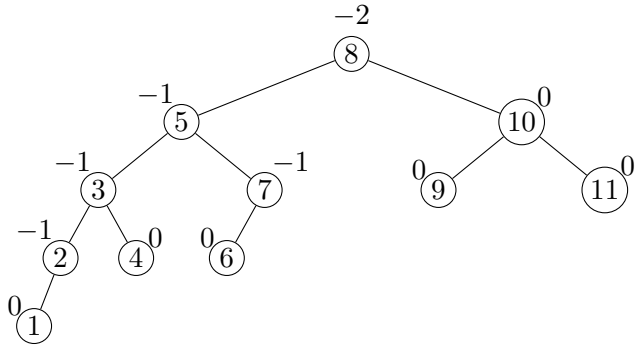
(b) Suppose T is a tree with 12 nodes:



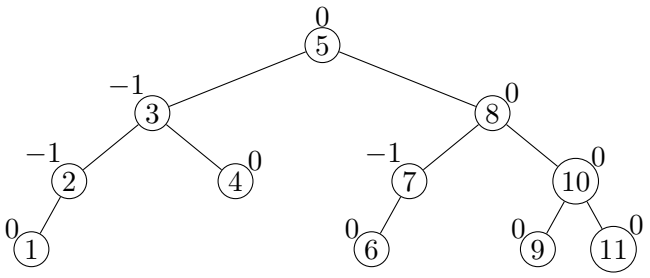
Then we invoke $DELETE(T, 12)$.
Immediately after the node is removed, we get



Node 11 is unbalanced. After the first rebalancing we have:



Now Node 8 is unbalanced, so we need a second rebalancing:



3. **Solution:** Written by Junjie Cheng, revised by Jiayun Liu and Zi Hao Lin

   The condition for a binary search tree to be balanced is that for each node, the height of the left sub-tree and the height of the right sub-tree differ by at most one.

   The following is the algorithm.

```
AVL_Balance_Check(T)
1  if (BF_Check(T) == −1)
2       return FALSE
3  else
4       return TRUE
```

```
BF_Check(T)
 1  if (T == Nil)
 2       return 0 //Base case
 3  else
 4       left = BF_Check(T.lchild)
 5       right = BF_Check(T.rchild)
 6       if (left == −1 or right == −1) // Already unbalanced
 7            return −1
 8       elseif (right − left == 1) // BF = 1
 9            return right + 1
10       elseif (right − left == 0) // BF = 0
11            return right + 1
12       elseif (right − left == −1) // BF = −1
13            return left + 1
14       else // Unbalanced
15            return −1
```

   The helper function $BF\_Check(T)$ checks whether the binary search tree T is an AVL tree. It returns -1 if the tree T is unbalanced, otherwise, it returns the height of tree $T + 1$.

   First it check T's sub-trees (we are treating Nil as an "empty tree", which is vacuously an AVL tree). If either of T's sub-tree is not an AVL tree, T cannot be an AVL tree. -1 is returned.

   Also, to satisfy the AVL balancing condition, the difference between the height of T's left sub-tree (i.e. the return value of $BF\_Check(T.left) - 1$, if it is balanced) and the height of T's

right sub-tree (i.e. the return value of $BF\_Check(T.right) - 1$, if it is balanced) must be at most 1. In these cases, the height of tree $T + 1$ is returned.

If the difference between the heights of the sub-trees of T is greater than 1, then T does not satisfy the AVL balancing condition, and -1 is returned.

The function $AVL\_Balance\_Check(T)$ translate the result of $BF\_Check(T)$ into $TRUE$ or $FALSE$ according to its return value.

- Worst-case running time

The helper function $BF\_Check(T)$ has a worst case running time of $\Theta(n)$, where n is the number of the nodes in the tree T.

First, by line 4 and line 5, both left sub-tree and right sub-tree are called (for all non-empty trees). Because of the recursion, every node in tree T is called exactly once, and all the (imaginary) empty "sub-trees" of leaves and one-child nodes are called exactly once.

Also, there will be a total of $n + 1$ such empty "sub-trees" [1]. For each call on the "empty node", function $BF\_Check()$ executes only one comparison before returning. A total of $n + 1$ comparisons are made for all "empty nodes".

For each of the nodes in T, from line 1 to 5, 1 comparison and 2 assignments must be executed. In the worst case, there can be at most 5 more comparisons, executed on line 6, 6, 8, 10, 12 successively. So, in the worst case, 2 assignments and 6 comparisons are made for each node in tree T, resulting a total of $6n$ comparisons and $2n$ assignments.

Using c to denote the time for each comparison and a to denote the time for each assignment, the time complexity of $BF\_Check(T)$ is

$$(n + 1)c + 6nc + 2na == 7nc + c + 2na$$

. Note that $AVL\_Balance\_Check(T)$ performs one $BF\_Check(T)$ function all and one comparison, the time complexity of $AVL\_Balance\_Check(T)$ is

$$T(n) = 7nc + 2c + 2na$$

. Note that there exist positive constant$c_1 = 2a + 7c$, $c_2 = 2a + 9c$, $n_0 = 2$ such that

$$\forall n \geq n_0.\ 0 \leq c_1 n \leq T(n) \leq c_2 n$$

So, by the definition of big-Theta,

$$T(n) \in \Theta(n)$$

, i.e. the time complexity of the algorithm is in $\Theta(n)$.

Note(s):

1. The proof is easy. Let's create a new tree, T^*, by making all "empty nodes" in tree T "real nodes". Then, all of the "empty nodes" become the leaves in $T*$, and all nodes originally in T becomes internal nodes in T^*. Note that T^* is a full (or proper/plane) binary tree (that is, every node in the tree has either 0 or 2 children). So, the number of "empty nodes" in T, should be the total number of the leaves in T^*, which, according to the property of full binary tree, should equal the number of internal nodes in the tree T^* +1, i.e. $n + 1$.

4. **Solution:** Written by Zi Hao Lin, revised by Junjie Cheng and Jiayun Liu

   1. The data structure we are using is AVL tree.
   The tree will store m (or the number of inputs so far, whichever is smaller) smallest values of all the inputs so far.

   2. The algorithm is, insert the first m input values into the AVL tree directly, so that the tree gets a size of m. Then, for each of following inputs, say, *input*, compare *input* with the largest element[1], say, l, in the AVL tree. If *input* is less than l, then delete l from the tree and then add *input* to the AVL tree; else, do nothing[2]. When a query is requested, do an inorder tree traversal[3] and print each value.

      Notes:

      1. The largest element is obtained by traversing along the rightmost path down the tree.

      2. This ensures that the keys in the AVL tree are the smallest m values of all the inputs.

      3. According to the property of AVL tree, the inorder traversal will visit the nodes in the order that the data of the nodes are increasing.

   3. The following is the pseudo-code. Initial values for global variables:

```
1  (int) count=0
2  (AVL_Tree) T=Nil
```

```
Input(input, m)
 1  if (input == query)
    // The question assumes there are already enough inputs when query is called.
 2      In_Order_Print(T)
 3  else
 4      if (count < m) // Less than m inputs given
 5          AVL_Insert(T, input)
 6          count++
 7      else // At least m inputs given; T is not empty
 8          l = Max_Value(T)
 9          if (input < x)
10              AVL_Delete(T, l)
11              AVL_Insert(T, input)
```

   Helper functions:

```
Max_Value(T)
1  while (T.right) // Note that T cannot be an empty tree, so T has the attribute right.
2      T = T.right
3  return T.key
```

```
In_Order_Print(T)
1  if (T.left)
2      In_Order_Print(T.left)
3  print (T.key)
4  if (T.right)
5      In_Order_Print(T.right)
```

Pseudo-code for $AVL\_Insert()$ and $AVL\_Delete()$ is in the handout.

4.
- Time complexity for processing input key:

  When processing a key, the condition in line 1 is always evaluated as false (evaluation taking $O(1)$ time), so the else block from line 3 to 11 is executed.

  (Special case) Initiation of global variables takes $O(1)$ time, we count this as part of the operation to process the first input key. In fact, processing the first input takes only $O(1)$ time. Line 4 condition must be true, and what line 5 does is actually creating an AVL Node with $key = input$ and pointing T to that node, which takes constant time. Addition in line 6 also takes constant time. So, first key process takes $O(1)$ time.

  For the following inputs,

  (Case i) if the condition in line 4 is true, then $O(\log(sizof(T))$ time is taken on line 5 and constant time is taken on line 6. Note that $sizeof(T) = count < m$, so the total time taken is bounded above by $O(1) + O(\log m) + O(1) = O(\log m)$.

  (Case ii) If the condition in line 4 is false, then, T is already of m. In the worst case, the condition in line 9 is true and line 8, 10, 11 are executed. The function $Max\_Value(T)$ in line 8, traverses along the path to the right-most leaf. Note that the height of an AVL tree with m nodes, in the worst case, is $1.44\log_2(m+2) \in O(\log m)$, so at most $O(\log m + 1)$ nodes are visited. Also note that on each node, the code runs a constant time, line 8 runs $O(\log m + 1)) * O(1) = O(\log m)$ time. $AVL\_Insert()$ and $AVL\_Delete()$, according to the handout, can be done in $O(\log m)$ time. So, the worst case running time for (Case ii) is $O(1) + O(\log m) + O(\log m) = O(\log m)$ time, same as (Case i).

  Thus, in the worst case, processing a input key takes $O(\log m)$ time.

- Time complexity for processing query: When query is the input, only line 1 (constant time comparison) and line 2 are executed. Note that at least m keys are input, so the AVL tree T must has a size of m. For each node, the function $In\_Order\_Print(T)$ will check the existence of left and right child and visit every existing child. So every node in the tree is visited once. On each visit, the function makes 2 comparisons and 1 print, which takes constant time. Thus, the function takes $O(m) * O(1) = O(m)$ time.