Question 1

In this question, you must use the insertion and deletion algorithms as described in the Balanced Search Trees: AVL trees handout posted on the course web site. (10 marks);

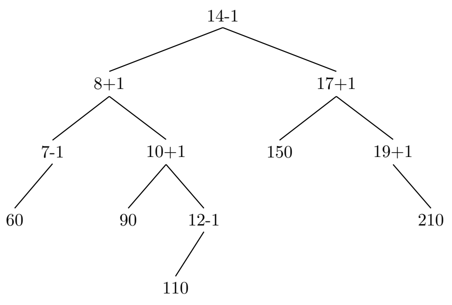
a. Insert keys 17, 7, 8, 14, 19, 6, 10, 21, 15, 12, 9, 11 (in this order) into an initially empty AVL tree, and show the resulting AVL tree T, including the balance factor of each node. (5 marks)

b. Delete key 17 from the above AVL tree T, and show the resulting AVL tree, including the balance factor of each node. (5 marks)

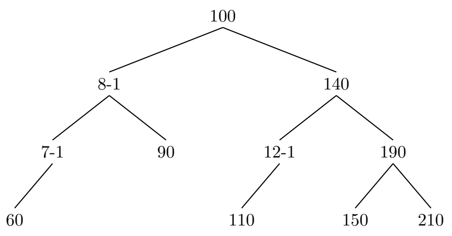
In each of the above questions, only the final tree should be shown: intermediate trees will be disregarded, and not given partial credit.

Solution.

a. We define our final tree as follows:



b. Once we delete 17, the resulting tree is doubly left heavy. To rebalance the tree, we require two rotations based on pivots centered around nodes 14 (the current root), 8 (left child of 14), and 10 (right child of 8). We first perform a left rotation on this cluster, and then a right rotation on the cluster. The result is the following tree:



□

Question 2

In this question, you must use the insertion and deletion algorithms described in the Balanced Search Trees: AVL trees handout posted on the course web site. (10 marks)

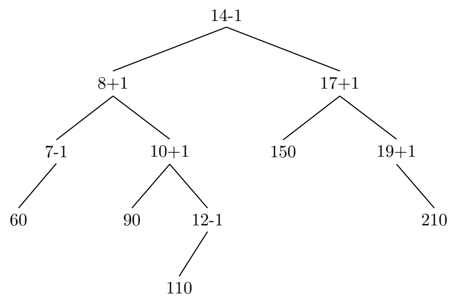
a. Prove or disprove: For any AVL tree T and any key x in T, if we let $T\prime$ = DELETE(T, x) and $T\prime\prime$ = INSERT($T\prime$, x), Then $T\prime\prime = T$." State clearly whether you are attempting to prove or disprove the statement. If proving, give a clear general argument; if disproving, give a concrete example where you show clearly each tree T, $T\prime$, $T\prime\prime$ with the balance factors indicated beside each node. (5 marks)

Solution.

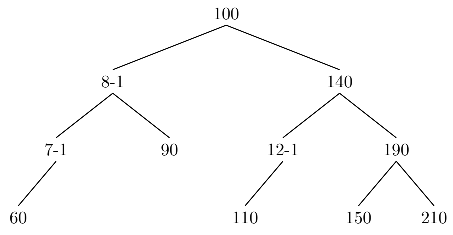
We will disprove this statement. Thus, we must show that in an arbitrary tree T, there exists a key k such that deleting k and then re-inserting k will result in a tree T'' such that $T \neq T''$.

Define equality here as, if $T = T'$ then all edges of T' connect to the same pattern of nodes as in T, and all keys and balance factors of T' are identical to those of T.

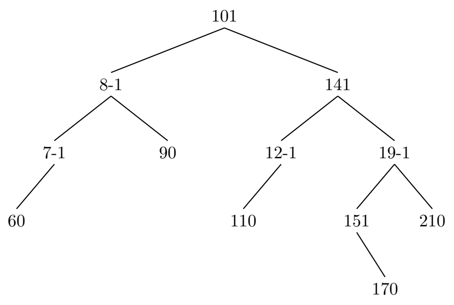
We can actually use the example from Question 1a. Let k be node 17 and T be our initial tree of Question 1a:



We then delete 17 resulting in the tree from 1b, which we can call T'. Thus:



Finally, we reinsert $k = 17$. This results in the tree T'' which is as follows:

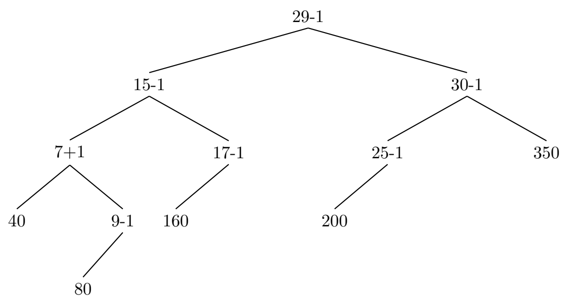


And we can clearly see that, by our definition of equality of trees, $T'' \neq T$, thus disproving the conjecture. □

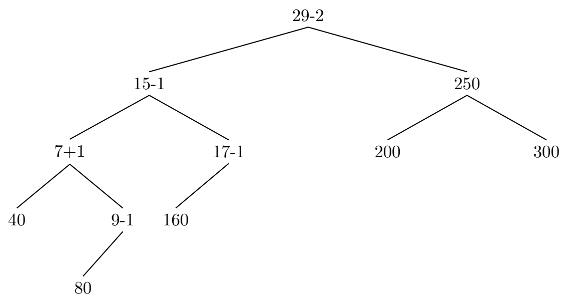
b. Find an AVL tree T and a key x in T such that calling DELETE(T, x) causes two rebalancing operations to take place. Your tree T should be as small as possible, in terms of the number of nodes, so that this takes place (you do not have to prove that it is indeed the smallest). Show your original tree T and the key x, then show the result of each rebalancing operation. Make sure to clearly indicate the balance factor next to each node. (5 marks)

Solution.

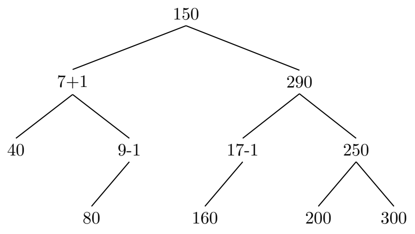
The AVL tree has a minimum size of 12 nodes shown below, we will delete 35 from the AVL tree.



After deleting 35, we rebalance the tree at node N_1 where N_1 is the node with key 30.



The previous rebalancing operation reduces the subtree rooted at N_1 by one. The change propagates upward and render the entire tree to be imbalanced. Here we introduce an introduce an additional rebalancing operation at the root of the AVL tree. The rebalancing procedure terminates with the following AVL tree.



Which is a valid AVL tree. □

Question 3

Give a simple, linear-time algorithm that determines if a Binary Search Tree (BST) satisfies the AVL balancing condition. The algorithms input is a pointer to the root of a BST T where each node u has the following fields: an integer *key*, an *lchild* and *rchild* which are pointers to the left and right children of u in T (if u has no left or right child, then *u.lchild* = NIL or *u.rchild* = NIL, respectively). **There is no balance factor or height information already stored in any node.** The algorithms output is TRUE if T satisfies the AVL balancing condition, and FALSE otherwise. (15 marks)
The worst-case running time of your algorithm ***must be*** $\Theta(n)$ where n is the number of nodes in T.
Describe your algorithm by giving its pseudo-code, and explain why its ***worst-case*** running

time is $\Theta(n)$. (15 marks).

Algorithm 1:

```
Function Main (root)
    Input: root is a pointer to the binary tree T given
    Output: True if T satisfies AVL balancing condition, False otherwise
    Assign-Height (root)
    return Check-AVL-Condition (root)
```

Algorithm 2:

```
Function Assign-Height (u)
    Input: u is a node in a binary tree T
    Output: None; Assigns height attribute to each node in subtree rooted at u
    if u is NIL then
        u.height = −1
    else
        Assign-Height (u.lchild)
        Assign-Height (u.rchild)
        u.height = Max(u.lchild, u.rchild) + 1
```

Algorithm 3:

```
Function Check-AVL-Condition (u)
    Input: u is a node in a binary tree T where every node in the tree has an
           attribute height
    Output: True if subtree rooted at u satisfies AVL balancing condition, False
            otherwise
    if u is NIL then
        return True
    else
        left = Check-AVL-Condition (u.lchild)
        right = Check-AVL-Condition (u.rchild)
        self = True
        heightDiff = u.lchild.height − u.rchild.height
        if $heightDiff \geq 2$ or $heightDiff \leq -2$ then /* node u not balanced */
            self = False
        if any one of left, right, or self is False then
            return False
        return True
```

Solution. MAIN calls ASSIGN-HEIGHT(*root*) and CHECK-AVL-CONDITION(*root*). Here we show that both both subroutines are $\Theta(n)$.

1. ASSIGN-HEIGHT(*root*) Let $T(n)$ be time taken by the algorithm when given the the root of a binary tree (subtree) T of size n. Since for any T of any given size,

ASSIGN-HEIGHT(*root*) visits every single node in the tree, spending a constant amount of operation. Then the running time $T(n)$ where n is size of binary tree is in $\Omega(n)$. Now we prove $T(n) = O(n)$. Let $c > 0$ such that $T(0) = c$ (i.e. When evaluating the node that is NIL) When $n > 0$, as ASSIGN-HEIGHT(u) is called on an arbitrary node u whose left subtree has k node and right subtree has $n - k - 1$ node, we characterize $T(n)$ by recurrence relation,

$$T(n) = T(k) + T(n-k-1) + d$$

for some $d > 0$, denoting the upper bound on the operation performed in node u, exclusive of the time spent in recurrence calls. Define predicate $P(n) : T(n) \leq (c+d)n + c$. When $n = 0$, $T(n) = (c+d)\cdot 0 + c \leq c$, hence $P(0)$ holds. When $n > 0$,

$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &= (c+d)k + (c+d)(n-k-1) + d && \text{(inductive hypothesis)} \\ &= (c+d)n + c \end{aligned}$$

Therefore $P(n)$ holds by induction. Here We proved that $T(n) = O(n)$.
In summary $T(n) = \Theta(n)$

2. CHECK-AVL-CONDITION(*root*) Following the exact same logic / procedure as shown above.

In conclusion, the worst time running time for MAIN(*root*) $= \Theta(n)$ □

Question 4

We want an efficient algorithm for the following problem. The algorithm is given an integer $m \geq 2$, and then a (possibly infinite) sequence of distinct keys are input to the algorithm, ***one at a time.*** A *query* operation can occur at any point between any two key inputs in the sequence. When a *query* occurs, the algorithm must return, ***in sorted order***, the m smallest keys among all the keys that were input before the *query.* Assume that at least m keys are input before the first *query* occurs.
For example, suppose $m = 3$, and the key inputs and query operations occur in the following order:

20, 15, 31, 6, 13, 24, *query*, 10, 17, *query*, 9, 16, 5, 11, *query*, 14, . . .

Then the first *query* should return 6, 13, 15; the second *query* should return 6, 10, 13; the third *query* should return 5, 6, 9.
Describe a simple algorithm that, for every $m \geq 2$, solves the above problem with the following worst-case time complexity:

- $O(\log\, m)$ to process each input key, and
- $O(m)$ to perform each *query* operation.

Your algorithm must use a data structure that we learned in class without any modification to the data structure.
To answer this question, you must:

1. State which data structure you are using, and describe the items that it contains.

   We will be using an AVL tree of size m. The AVL tree contains at most m input keys with proper balancing factors assigned. The tree is maintained in a way such that the m smallest keys are within the AVL tree.

2. Explain your algorithm ***clearly*** and ***concisely***, in English.

   Note we are not going in detail to the algorithm taught in class but instead focus on the algorithm in question.
   The algorithm roughly consists of two subroutines,

   (a) **Process-Key**(T, x)
   Input key x is processed where T is an AVL tree. In particular, if T is empty, we create a new AVL tree with root x with **AVL-Create**. Otherwise, x is inserted into T the same way that a node is inserted into an AVL tree using **AVL-Insert**(T, x), which includes possible rotation and rebalancing procedues. Now we check the size of AVL tree. If $T.size$ exceeds the limit m, then we find the largest element in AVL tree using **Maximum**(T), which traverses through the right child until it finds node j with the maximum key. We then delete this node with **AVL-Delete**(T, j) to preserve the size limit of the tree while still maintaining the balancing condition. In this step, the m smallest key in the sequence up to x is preserved in the AVL tree. The AVL tree's size (i.e. cannot exeed m) is invariant.

   (b) **Query**(T)
   The algorithm traverses the binary tree in order of keys, from the smallest to largest, similar to BST inorder traversal. We allocate memory for an array A of size m. During the traversal, we copy key of every node we visit to the the array A. The array A is returned as output. We see that A is a sorted array of the smallest m element for all keys inputed before **Query**(T) as the AVL tree maintains such property.

3. Give the algorithms ***pseudo-code***, including the code to process an input key k, and

the code for *query*.

```
Algorithm 4:
Function Process-Key ($T, k$)
    Input: $T$ is an AVL tree of at most size $m$, $k$ is input key to be processed
    Output: None; AVL tree's size and balancing condition maintained, and
            contains at most $m$ smallest element for the sequence of key up to $k$
    if $T$ is empty then
        $T$ = AVL-Create ()
    AVL-Insert ($T, k$)
    if $T.size > m$ then
        $maxNode$ = Maximum ($T$)
        AVL-Delete ($T, maxNode$)
```

```
Algorithm 5:
Function Query ($T$)
    Input: $T$ is an AVL tree of size $m$
    Output: Returns the $m$ keys stored in $T$ in sorted order
    $L = LinkedList$
    $S = Stack$
    $x = T.root$
    $S$.Push($x$)
    while $S$ is not empty or $x$ is not NIL do
        /* push til smallest value reached                          */
        if $x.left$ is not NIL then
            $S$.Push($x.left$)
            $x \leftarrow x.left$
        /* Now we pop and consider right child                      */
        if $S$ is not empty then
            $x \leftarrow S$.Pop()
            $L.next \leftarrow x$
            $x \leftarrow x.right$
    return $L$
```

4. Explain why your algorithm achieves the required worst-case time complexity described above.

   Solution.

   (a) **Process-Key(T, k)** achieves $\Theta(\log m)$ worst-case time complexity to process each input key because **AVL-Create()** takes constant time, and **AVL-Insert(T, k)**,

Maximum(T**)**, **AVL-Delete(**$T, maxNode$**)** all take $\Theta(\log m)$ time, assuming the AVL tree has at most m nodes. This condition is maintained as explained previously.

(b) Since a constant amount of time is taken during visiting each node, **Query(**T**)** is essentially the iterative algorithm for binary search which we have learnt in class to be $\Theta(m)$ where m is the size of the tree.

□