First Name: Zhicheng
Last Name: Yan
Student ID: 1002643142

First Name: Zhihong
Last Name: Wang
Student ID: 1002095207

First Name: Jialiang
Last Name: Yi
Student ID: 1002286929

We declare that this assignment is solely our own work, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters.

This submission has been prepared using LaTeX.

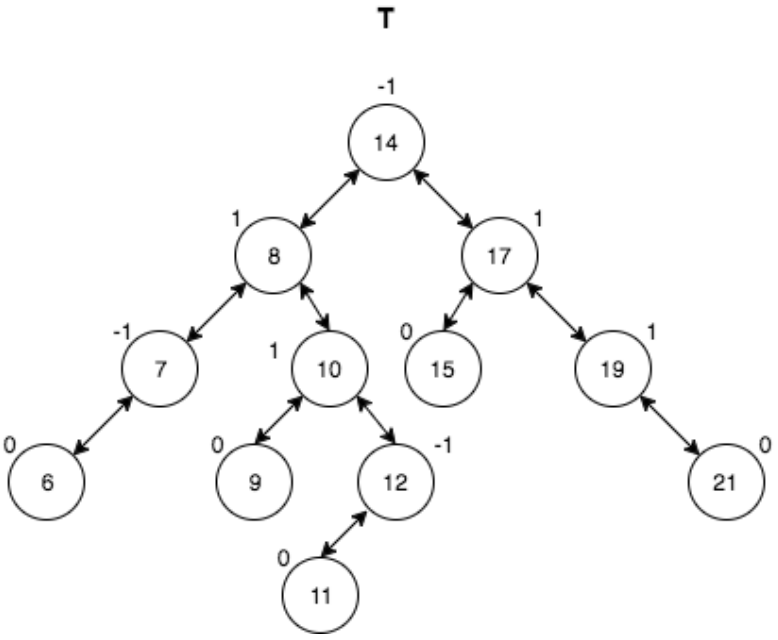
Writer: Jialiang Yi

Reader: Zhicheng Yan, Zhihong Wang

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

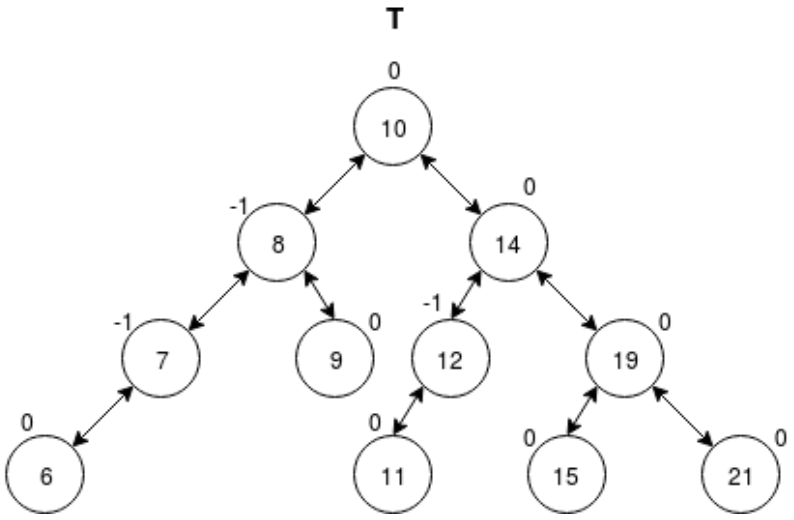
**a.** Insert keys 17, 7, 8, 14, 19, 6, 10, 21, 15, 12, 9, 11 (in this order) into an initially empty AVL tree, and show the resulting AVL tree T, including the balance factor of each node.

Solution:



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

Solution:



Writer: Jialiang Yi

Reader: Zhihong Wang, Zhicheng Yan

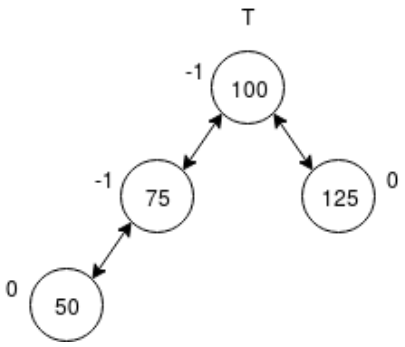
**Question 2**
In this question, you must use the insertion and deletion algorithms described in the "Balanced Search Trees: AVL trees" handout posted on the course web site.

**a.** Prove or disprove: "For any AVL tree $T$ and any key $x$ in $T$, if we let $T'$ = DELETE($T, x$) and $T''$ = INSERT($T', x$), then $T'' = T$." 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 $T, T', T''$ with the balance factors indicated beside each node.

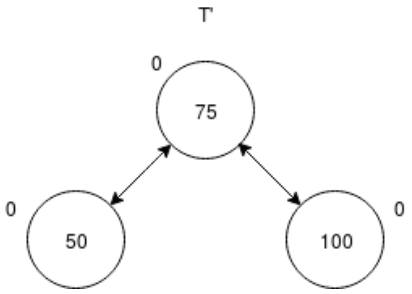
Solution:

Disproving the claim "For any AVL tree $T$ and any key $x$ in $T$, if we let $T'$ = DELETE($T, x$) and $T''$ = INSERT($T', x$), then $T'' = T$.".

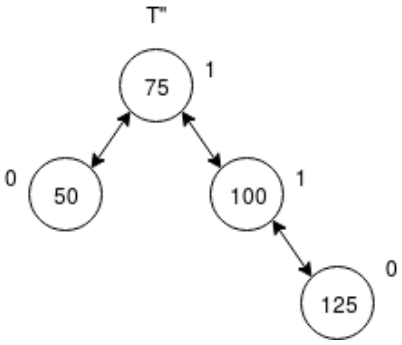
Counter Example:
Assume AVL tree $T$ as shown below



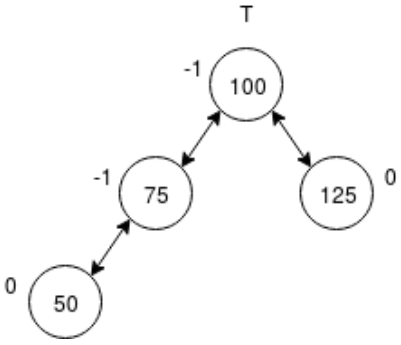
Let AVL tree $T' = DELETE(T, 125)$, as shown below

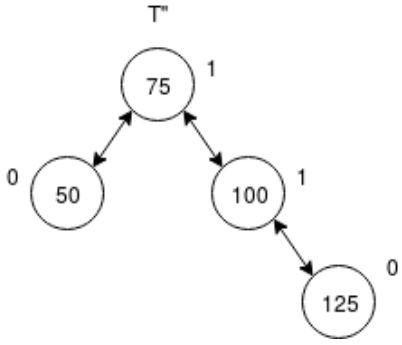



Let AVL tree $T'' = INSERT(T', 125)$, as shown below




Comparing tree $T$ and tree $T''$:

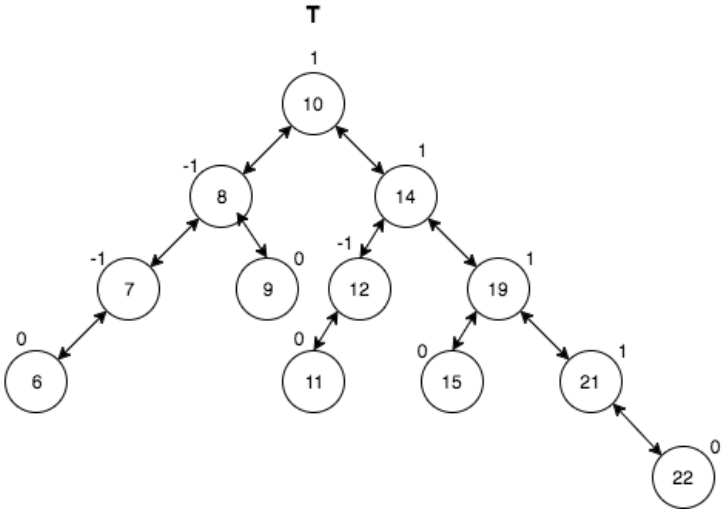


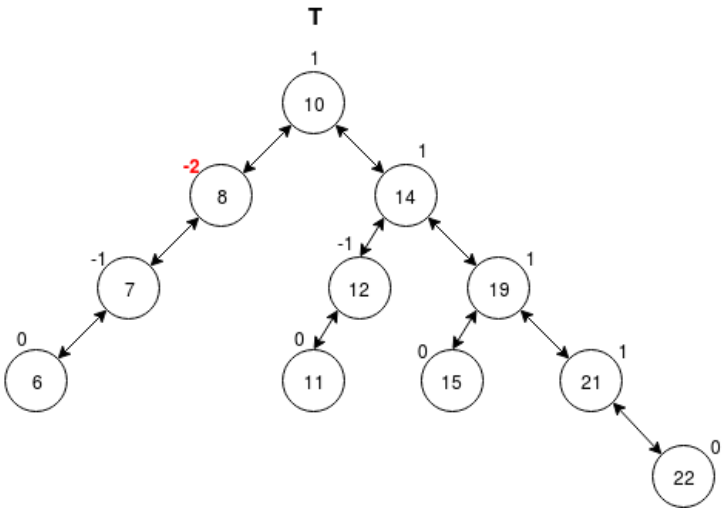

Clearly, AVL tree $T$ is not the same as AVL tree $T''$, which disproves the claim "For any AVL tree $T$ and any key $x$ in $T$, if we let $T'$ = DELETE($T, x$) and $T''$ = INSERT($T', x$), then $T'' = T$.".

**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.

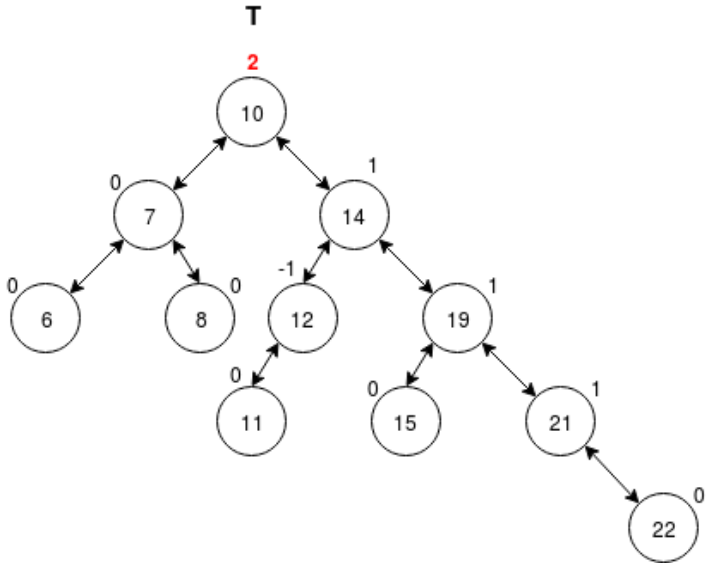
Solution:
AVL tree T as shown:



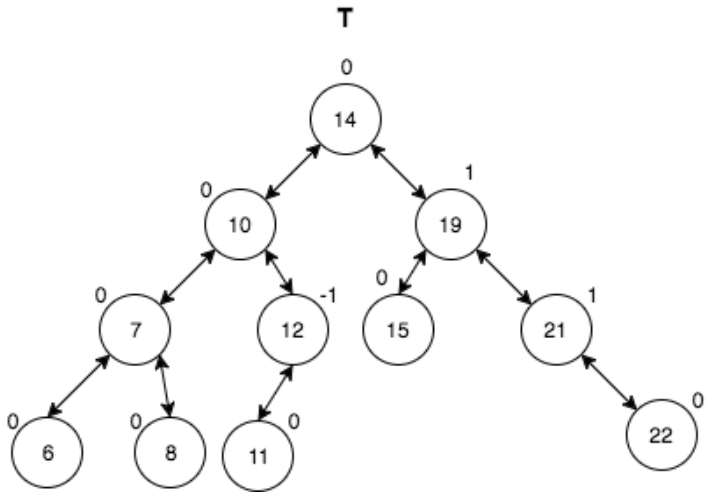

After calling $Delete(T, 9)$, T becomes unbalance:



Result of 1st rebalancing operation:




Result of 2nd rebalancing operation(tree T after calling $Delete(T, 9)$):



Writer: Zhihong Wang

Reader: Jialiang Yi, Zhicheng Yan

**Question 3**
Give a simple, linear-time algorithm that determines if a Binary Search Tree (BST) satisfies the AVL balancing condition. The algorithm's input is a pointer to the root of a BST $T$ where each node $u$ has the following fields: an integer *key*, and *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 algorithm's output is $TRUE$ if $T$ satisfies the AVL balancing condition, and $FALSE$ otherwise.
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 $\theta(n)$.

Solution:
Read me:
Assume the BST has $n$ number of nodes.
The height of only one node (no children) is 0; the height of empty node is -1.
For our algorithm, the input should be a node of BST (to check if the whole BST satisfies AVL, please input the root node of the BST). If the input is a root of AVL, return the height of this AVL; if not, return -2.
These variables below are only in pseudo-code, not in fields of node.
"left height" is an integer to represent height of left tree.
"right height" is an integer to represent height of right tree.
"balance factor" is an integer to represent the balance status of the current tree node.
"-2" represents the failed situation which shows the input is not a root of an AVL. We use -2 to show failed because the height can never be -2, and we already use -1 to represent the height of empty node.
i.e. It can be any negative number less than -1.

```
\\Precondition: The input is a BST node.
```

```
\\Postcondition: returns an integer, if the input is a root of AVL,
return the height of this AVL; if not, return −2.

int check_AVL(BST root node){
1.  if node is NULL,
2.    return −1;
3.  left_height = check_AVL(left child of the input node);
4.  right_height = check_AVL(right child of the input node);
5.  if left_height is −2 or right_height is −2,
6.    return −2;
7.  balance_factor = right_height − left_height;
8.  if balance_factor less than −1 or greater than 1,
9.    return −2;
10. if left_height greater than right_height,
11.   return left_height + 1;
12. else
13.   return right_height + 1;
}
```

Prove correctness:

Base case: $n = 1$. Since the algorithm checks if the input BST is a AVL, it is easy to see that a BST with only one node is also an AVL. Therefore, $n = 1$ holds.

Induction step: Assume left height and right height got integer values successfully, we want to show the current root is also a root of AVL or not.

Case 1: left subtree and right subtree are all AVL.
Because left subtree and right subtree are all AVL, we will get correct positive left height and right height, then line 5 to 6 will not execute. Line 7 to 13 will check the balance of current root and return height if current root is balanced (i.e. current root is a root of AVL) or return -2 if current root is not balanced (i.e. current root is not a root of AVL).

Case 2: more than one of left subtree and right subtree is not AVL. (i.e. left subtree is an AVL but right subtree is not; left subtree is not an AVL but right subtree is; left subtree and right subtree are both not AVL).

Because more than one of left subtree and right subtree is not AVL (i.e. the current root is not a root of AVL), so we will get at least one -2 in left height or right height, then line 5 will execute and return -2.

Therefore, all situations hold, our algorithm is functional.

Before Proofs:

Let the worst-case time complexity to be $T(n)$.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

i.e., line 1 to 2 and line 5 to 13 only require constant time.

Worst case: $T$ is a totally balanced full binary tree, because in binary trees of same height, only a totally balanced full binary tree has maximum numbers of nodes.

To prove the worst-case running time of my algorithm is $\theta(n)$ where $n$ is the number of nodes in $T$, we have to prove my algorithm is $O(n)$ and $\Omega(n)$.

i.e., $T(n)$ is $O(n)$ and $\Omega(n)$.

Proof of $O(n)$:

To prove $T(n)$ is $O(n)$, we need to show $T(n) \leq cn$ for constant $c > 0$.

By line 3 and 4, these recursions will take $n$ times to go over all nodes in the whole $T$ because:

The *check AVL* is a recursive tree traversal operation.

Claim: *check AVL* is $O(n)$ where n is the number of nodes.

Let $P(T)$ be the claim that *check AVL* on BST takes $O(n)$ time.

Let $c$ be the maximum cost of calling this function on a single node, excluding all recursive calls.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

Base Case: $P(T)$ where $T$ is a single node. It is constant time operation because in this case it don't have recursive calls. Also, we have $n = 1$. So $P(T)$ holds.

Inductive Step: Assume $P(T_L)$ and $P(T_R)$ hold, where $T_L$ and $T_R$ are independent binary tree. (Recalling the definition of a tree; the most basic tree is a single node. If $T_L$ and $T_R$ are independent binary tree, then we can construct a new tree by connecting all trees to a new parent node.) We want to show if a new binary tree $T'$ is constructed by a root and the left subtree $T_L$ and the right subtree $T_R$ such that $P(T')$ holds.

Let $n_L$ and $n_R$ be the number of nodes of sub-tree $T_L$ and sub-tree $T_R$ respectively. Then we obtain a new constructed binary tree $T'$ with $n'$ nodes where $n' = n_L + n_R + 1$. By assumption, we can get both left and right subtree takes $O(n)$ time to perform *check AVL* operation. So, together takes not greater than $c_1 n_L + c_2 n_R$ steps for recursive calls. With constant, it takes $c_1 n_L + c_2 n_R + c_3$ steps, and $c_1 n_L + c_2 n_R + c_3 \leq c + cn_L + cn_R \leq c(n_L + n_R + 1) = cn'$.

Therefore P(T') holds.

So, there exists $c > 0$ such that $T(n) \leq cn$.

Therefore, $T(n)$ is $O(n)$.

Proof of $\Omega(n)$:

To prove $T(n)$ is $\Omega(n)$, we need to show $T(n) \geq cn$ for constant $c > 0$.

Worst case lower bound means for all inputs, there exists an input x, T(x) is no smaller than cn, where c is a positive constant.

Worst case: $T$ is a totally balanced full binary tree, because in binary trees of same height, only a totally balanced full binary tree has maximum numbers of nodes.

So, by line 3 and 4, these recursions will take $n$ times to go over all nodes in the whole $T$ because:

The *check AVL* is a recursive tree traversal operation.

Claim: *check AVL* is $\Omega(n)$ where n is the number of nodes.

Let $P(T)$ be the claim that *check AVL* on BST takes $\Omega(n)$ time.

Let $c$ be the maximum cost of calling this function on a single node, excluding all recursive calls.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

Base Case: $P(T)$ where $T$ is a single node. It is constant time operation because in this case it don't have recursive calls. Also, we have $n = 1$. So $P(T)$ holds.

Inductive Step: Assume $P(T_L)$ and $P(T_R)$ hold, where $T_L$ and $T_R$ are independent binary tree. (Recalling the definition of a tree; the most basic tree is a single node. If $T_L$ and $T_R$ are independent binary tree, then we can construct a new tree by connecting all trees to a new parent node.) We want to show if a new binary tree $T'$ is constructed by a root and the left subtree $T_L$ and the right subtree $T_R$ such that $P(T')$ holds.

Let $n_L$ and $n_R$ be the number of nodes of sub-tree $T_L$ and sub-tree $T_R$ respectively. Then we obtain a new constructed binary tree $T'$ with $n'$ nodes where $n' = n_L + n_R + 1$. By assumption, we can get both left and right subtree takes $\Omega(n)$ time to perform *check AVL* operation. So, together takes not greater than $c_1 n_L + c_2 n_R$ steps for recursive calls. With constant, it takes $c_1 n_L + c_2 n_R + c_3$ steps, and $c_1 n_L + c_2 n_R + c_3 \geq c + cn_L + cn_R \geq c(n_L + n_R + 1) = cn'$.

Therefore P(T') holds.

So, there exists $c > 0$ such that $T(n) \geq cn$.

So, $\Omega(n)$ is the worst case lower bound.

Therefore, $T(n)$ is $\Omega(n)$.

Therefore, $T(n)$ is $O(n)$ and $\Omega(n)$.

Therefore, the worst-case running time of my algorithm is $\theta(n)$ where $n$ is the number of nodes in $T$.

Another way to prove (by master theorem):

Let the worst-case time complexity to be $T(n)$.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

i.e., line 1 to 2 and line 5 to 13 only require constant time, which is 1.

Worst case: $T$ is a totally balanced full binary tree, because in binary trees of same height, only a totally balanced full binary tree has maximum numbers of nodes.

By pseudo-code, line 3 and 4 will cost $T(\lfloor \frac{n}{2} \rfloor)$ and $T(\lceil \frac{n}{2} \rceil)$

So,

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

By Master Theorem:

$a_1 = 1, a_2 = 1, f(n) = 1, b = 2, d = 0.$

Then $a = a_1 + a_2 = 2, f(n) \in \theta(1^d) = \theta(1)$

Then $a = 2 > b^d = 1 \implies T(n) \in \theta(n)$

Therefore, the worst-case running time of my algorithm is $\theta(n)$ where $n$ is the number of nodes in $T$.

Writer: Zhicheng Yan

Reader: Jialiang Yi, Zhihong Wang

**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:
. $O(logm)$ 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
2. Explain your algorithm **clearly and concisely**, in English.
3. Give the algorithm's **pseudo-code**, including the code to process an input key k, and the code for *query*.
4. Explain why your algorithm achieves the required worst-case time complexity described above.

Solutions that are unclear, inefficient, or overly complex may not get any points.

Solution:

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

We will use AVL tree.

Every node of AVL has following fields: an integer *key*, *lchild*, *rchild* and *parent* which are pointers to the left and right children and parent of nodes, and an integer $balance\ factor$ to represent the balance status of the current tree node.

This AVL tree will contain nodes whose *key* are same as a part of distinct keys in input sequence.

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

Note: every input is one distinct key or a *query*. i.e. Input one key or *query* one time.

The algorithm can be develop base on an AVL tree.

$query\ op$ is an in-order traversal.

$sizeof AVL$ and $AVL$ are global variables.

Our goal to construct an AVL tree of size $m$ containing $m$ distinct keys before encountering $query\ op$. This AVL tree can be apply $query\ op$ and output $m$ number of sorted integers.

Perform *insert* method from AVL library on all distinct keys from the very first distinct key and to the key before *query* until the AVL tree reaches the maximum size. Once the AVL tree reaches the maximum size $m$, insert the next key first, then find the node holds max key and perform *delete* to remove it. These steps show how to maintain the $sizeof AVL$ small or equal to $m$.

When *query* occurs, perform *query op* on current AVL tree, algorithm will return an list of $m$ smallest keys in sorted order.

Step 3. Give the algorithm's **pseudo-code**, including the code to process an input key k, and the code for *query*.

pseudo-code:

```
global variable: integer sizeofAVL initially is 0.
global variable: an empty AVL.

\\Precondition: The input is a key or query
\\Postcondition: returns a list of sorted integers or nothing

  processing_key(a distinct integer key or query){
1.  if input is query{
2.      query_op(AVL);
3.  }else{
4.    insert key in AVL;
5.    sizeofAVL = sizeofAVL + 1;
6.    If sizeofAVL > m {
7.            max = find_max(AVL);
8.            Detete max in AVL;
9.            sizeofAVL = sizeofAVL - 1;
10.   }
11.   return;
    }
9.}

\\Precondition: The input is an AVL tree
\\Postcondition: returns a list of sorted integers
list_of_integers  query_op(AVL){
1.    define list_of_integers ListA and ListB initially empty lists
2.    if AVL's left child exists{
3.      ListA = copy of query_op(AVL's left child);
4.    }
5.    ListA append the value of the AVL's root
6.    if AVL's right child exists{
```

```
7.       ListB = copy of query_op(AVL's right child);
8.    }
9.    ListA concatenate with ListB;
10.   return ListA;
11.}

\\Helper Function
\\Precondition: AVL is an AVL tree
\\Postcondition: returns the maximum value of the all AVL Tree nodes
int find_max(AVL){
1.    if AVL's Right Child does not exist{
2.       return AVL's value;
3.    }
4.    else{
5.        max = find_max(AVL's Right Child);
6.        return max;
7.    }
8.}
```

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

Prove correctness of our algorithm:

1. For function *query op*:

Function *query op* performs an in-order traversal recursively

Base Case: $m = 1$; $P(T)$ where $T$ is only a single node. It is obvious that T has no left child and right child. Only line5 will be executed, which append T's root into List A and returns it when function ends. Therefore, $m = 1$ holds.

Inductive Step: Assume AVL tree $T$ is constructed by sub-tree $T_L$ and sub-tree $T_R$ where $P(T_L)$ and $P(T_R)$ holds. Want to show $P(T)$ holds.

By Line3 and Line7, List A contains all nodes in $T_L$ in sorted order and List B contains all nodes in $T_R$ in sorted order. According to the properties

of AVL tree, all nodes in List A are smaller than root of $T$, all nodes in List B are larger than root of $T$. By Line5, the root of $T$ will be appended into List A and concatenate with List B at Line9. Therefore the returning list, List A in our algorithm contains all nodes in tree $T$ in sorted order. Therefore, $P(T)$; our algorithm is functional.

2. For $find\ max$:

Function $find\ max$ returns the maximum value in an AVL tree.

According to the properties of Binary Search Tree and AVL Tree, the node with max value locates at the bottom right-most of the tree. By Line4-6, the function trace down to the bottom right-most leaf recursively. In the next recursive call, Line1-3 will be executed, and the bottom right-most node will be return as wanted. Therefore, our algorithm is functional.

3. For *processing key*:

Function *processing key* require a *query* input or a distinct integer key input.

This function will return a list of integer when the input is *query*, this list is the same as *query op*'s return.

This function will return a nothing when the input is a distinct integer key.

Case 1: when input is *query*.

By line 1 to 2, if the input is *query*, this function will call function *query op*. By 1 we already proved, *processing key* is correct.

Case 2: when input is a distinct integer key.

By line 3 to 9, *processing key* will modify the global value AVL. After all, by line 11, this function will return anyway.

Therefore, all situations hold, our algorithm is functional.

(a),(b),(c) will explain why your algorithm achieves the required worst-case time complexity described above:

(a) Show it takes $O(logm)$ time ($m$ is number of nodes) to find the max value of the AVL Tree.

Let the worst-case time complexity to be $T(m)$.

All algorithm based on AVL, whose height is $logm$ by AVL properties.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

To prove $T(m)$ is $O(logm)$, we need to show $T(m) \leq clogm$ for constant $c > 0$.

Worst case: $T$ is a totally balanced full binary tree (AVL is already a totally balanced full binary tree), because in binary trees of same height, only a totally balanced full binary tree has maximum numbers of nodes.

By help function $find\ max$ line 5, the recursion will take $logm$ (the height of AVL) times to find the node with max key value in the whole $T$ because by the property of AVL, the right child always holds greater key value than its parent node.

Then, the worst case complexity is always less than $2logm$.

So, there exists $c > 0$ such that $T(m) \leq clogm$.

Therefore, $T(m)$ is $O(logm)$.

(b) Show it takes $O(logm)$ to process each input key(the function *processing key*).

Let the worst-case time complexity to be $T(m)$.

All algorithm based on AVL, whose height is $logm$ by AVL properties.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

To prove $T(m)$ is $O(logm)$, we need to show $T(m) \leq clogm$ for constant $c > 0$.

By function *processing key*, when the input is a distinct integer key, line 1 to 2 will never execute. By line 4, *insert* method will cost $logm$ time in worst case because after rotation, ancestors needs to be updated (by CLRS chap.14). By line 7, calling help function to find max will cost $logm$ time (by (a) we already proved). By line 8, *delete* method will cost $logm$ time because in worst case, ancestors needs to be updated after rotation(by CLRS chap.14). So, when line 6 do not execute, the time complexity is $logm$; when line 6 do execute, the time complexity is $3logm$

Then, the worst case complexity is always less than $4logm$.

So, there exists $c > 0$ such that $T(n) \leq clogm$.

Therefore, $T(m)$ is $O(logm)$.

(c) Show it takes $O(m)$ to process to perform each *query op*.

The *query op* is a recursive tree traversal operation, specifically, an in-order traversal operation.

Claim: *query op* is $O(m)$ where m is the number of nodes.

Let $P(T)$ be the claim that *query* on $T$ takes $O(m)$ time.

Let $c$ be the maximum cost of calling this function on a single node, excluding all recursive calls.

By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.

Base Case: $P(T)$ where $T$ is a single node. It is constant time operation because in this case it don't have recursive calls. Also, we have $m = 1$. So $P(T)$ holds.

Inductive Step: Assume $P(T_L)$ and $P(T_R)$ hold, where $T_L$ and $T_R$ are independent binary tree. (Recalling the definition of a tree; the most basic tree is a single node. If $T_L$ and $T_R$ are independent binary tree, then we can construct a new tree by connecting all trees to a new parent node.) We want to show if a new binary tree $T'$ is constructed by a root and the left subtree $T_L$ and the right subtree $T_R$ such that $P(T')$ holds.

Let $m_L$ and $m_R$ be the number of nodes of sub-tree $T_L$ and sub-tree $T_R$ respectively. Then we obtain a new constructed binary tree $T'$ with $m'$ nodes where $m' = m_L + m_R + 1$. By assumption, we can get both left and right subtree takes $O(m)$ time to perform *query* operation. So, together takes not greater than $c_1 m_L + c_2 m_R$ steps for recursive calls. With constant, it takes $c_1 m_L + c_2 m_R + c_3$ steps, and $c_1 m_L + c_2 m_R + c_3 \leq c + c m_L + c m_R \leq c(m_L + m_R + 1) = cm'$.

Therefore P(T') holds.