Project 1 Documentation

AVL Node

Properties Added

- *size* - this attribute will hold the number of nodes in an AVLNode's subtree (including the AVLNode itself).
  Given an AVLNode, its size == left son's size + right son's size + 1.
  If an AVLNode is virtual its size is 0.
  Initialization value: 0.
  Time complexity: O(1)

Methods

- *isRealNode(self)* - return self.height != -1
  If so, this is a virtual node, so we'll return False.
  Otherwise, it is a real node so we'll return True.
  Time complexity: O(1)

```
"""returns whether self is not a virtual node

@rtype: bool
@returns: False if self is a virtual node, True otherwise.
@Time complexity: O(1)
"""
def isRealNode(self):
    return self.height != -1
```

- *getBalanceFactor(self)* - returns the balance factor of the current node.
  Given an AVLNode, its BF (balance factor) = left son's height - right son's height.
  Since this function can be called on any AVLNode, even if it is virtual, we've decided that the default value that won't harm other implementations is if an AVLNode is virtual his BF is 0.
  Time complexity: O(1) - arithmetic operation between 2 integers.

```
"""returns the balance factor of a given node

@rtype: int
@returns: height of left child of self - height of right child of self, 0 if virtual node
"""

def getBalanceFactor(self):
    if self.isRealNode():
        return self.getLeft().height - self.getRight().height
    return 0
```

- *recomputeSize(self)* - recomputes the size of an AVLNode (see `size` property to understand what size is).

```
"""recomputes the size of a Node inplace

@returns: None
"""
def recomputeSize(self):
    if self.isRealNode():
        self.size = self.left.size + 1 + self.right.size
```

  Time complexity: O(1) - arithmetic operation between 2 integers.

- *recomputeHeight(self)* - recomputes the height of an AVLNode. AVLNode's height = max(self.left.height, self.right.height) + 1, if virtual node then -1.
  Time complexity: O(1) - arithmetic operation between 2 integers.

```
"""recomputes the height of a Node inplace

@returns: None
"""

def recomputeHeight(self):
    if self.isRealNode():
        self.height = max(self.left.height, self.right.height) + 1
```

- *getLeft(self)* - return a pointer to the left son of the current node. Returns None if there is no left child (happens when the current node is virtual).
  Time complexity: O(1)

```
"""returns the left child
@rtype: AVLNode
@returns: the left child of self, None if there is no left child
@Time complexity: O(1)
"""
def getLeft(self):
    return self.left
```

- *getRight(self)* - return a pointer to the right son of the current node. Returns None if there is no right child (happens when the current node is virtual).
  Time complexity: O(1)

```
"""returns the right child

@rtype: AVLNode
@returns: the right child of self, None if there is no right child
@Time complexity: O(1)
"""
def getRight(self):
    return self.right
```

- *getParent(self)* - return a pointer to the parent of the current node.
  If there is no parent (this node is the root of the AVL Tree), we'll return None.
  Time complexity: O(1)

```
"""returns the parent

@rtype: AVLNode
@returns: the parent of self, None if there is no parent
@Time complexity: O(1)
"""
def getParent(self):
    return self.parent
```

- *getValue(self)* - return the value of the current node.
  If this node is virtual, we'll return None.
  Otherwise, return self.value (the value of a non virtual node can still be None).
  Time complexity: O(1)

```
"""return the value

@rtype: str
@returns: the value of self, None if the node is virtual
@Time complexity: O(1)
"""
def getValue(self):
    if self.isRealNode():
        return self.value
    return None
```

- *getHeight(self)* - return the height of the current node.
  If this node is virtual, we'll return -1.
  Otherwise, return self.height.
  Time complexity: O(1)

```
"""returns the height

@rtype: int
@returns: the height of self, -1 if the node is virtual
@Time complexity: O(1)
"""
def getHeight(self):
    if self.isRealNode():
        return self.height
    return -1
```

- *getSize(self)* - return the size of the current node.
  (size of virtual node is 0)
  Time complexity: O(1)

```
"""returns the size

@rtype: int
@returns: the size of self, virtual node size is 0
"""

def getSize(self):
    return self.size
```

- *setLeft(self, node)* - set the current node property (self.left) to a given pointer of a node.
  Time complexity: O(1)

```
"""sets left child

@type node: AVLNode
@param node: a node
@Time complexity: O(1)
"""
def setLeft(self, node):
    self.left = node
```

- *setRight(self, node)* - set the current node property (self.right) to a given pointer of a node.
  Time complexity: O(1)

```
"""sets right child

@type node: AVLNode
@param node: a node
@Time complexity: O(1)
"""
def setRight(self, node):
    self.right = node
```

- *setParent(self, node)* - set the current node property (self.parent) to a given pointer (AVLNode).
  Time complexity: O(1)

```
"""sets parent

@type node: AVLNode
@param node: a node
@Time complexity: O(1)
"""
def setParent(self, node):
    self.parent = node
```

- *setValue(self, value)* - set the current node property (self.value) to a given (str) value.
  Time complexity: O(1)

```
"""sets value

@type value: str
@param value: data
@Time complexity: O(1)
"""
def setValue(self, value):
    self.value = value
```

- *setHeight(self, h)* - set the current node property (self.height) to a given integer (h).
  Time complexity: O(1)

```
"""sets the height of the node

@type h: int
@param h: the height
@Time complexity: O(1)
"""
def setHeight(self, h):
    self.height = h
```

- *setSize(self, s)* - set the current node property (self.size) to a given integer (s).
  Time complexity: O(1)

```
"""sets the size of the node

@type s: int
@param s: the size
"""

def setSize(self, s):
    self.size = s
```

- *isLeaf(self)* - return self.height == 0.
  If so, our node is a leaf (both sons are virtual nodes).
  If not, one of the sons isn't a virtual node, or the given node is a virtual node, which is not a leaf.

```
""" returns true whether self is a leaf

    @rtype: bool
    @returns: False if self is not a leaf (has a right/left son such that they are not virtual nodes),
    True otherwise
    @Time complexity: O(1)
    """

def isLeaf(self):
    return self.getHeight() == 0
```

AVL Tree List

Properties Added

- *first_node* - pointer to the first node of the tree.
  If the tree is empty, it is None.
  Initialization value: None.
  Time complexity: O(1)

- *last_node* - pointer to the last node of the tree.
  If the tree is empty, it is None.
  Initialization value: None.
  Time complexity: O(1)

Methods

- *empty(self)* - return self.root is None
  If self.root is None, the tree is empty, and the method will return True.
  Otherwise, the tree has a root, so the method will return False.
  Time complexity: O(1)

```
"""returns whether the list is empty

@rtype: bool
@returns: True if the list is empty, False otherwise
@Time complexity: O(1)
"""
def empty(self):
    return self.root is None
```

- *retrieve(self, i)* - retrieves the value of the i'th item in the list. retrieve works as Tree-Select works as we saw in the lecture (implemented recursively):

```
retrieve(self, i):
        root = self.root
        r = root.left.size + 1
        if (i == r):
                return root.value
        else:
                if i < r, search for the i^th smallest item in the left subtree
                otherwise (i > r), search for the (i - r)^th smallest item in the right subtree
```

Time complexity: as we saw in the lecture O(h) = O(logn)

```
"""retrieves the value of the i'th item in the list

@type i: int
@pre: 0 <= i < self.length()
@param i: index in the list
@rtype: str
@returns: the the value of the i'th item in the list

Time Complexity:
Recursion that in worst case goes every call goes one son left / one son right until the deepest leaf
Meaning that the maximum calls is the height of tree
In every recursion call there is O(1) work, and the height of the tree is O(logn)
That is why, in total, as we saw in the lecture, the time complexity is O(logn) in the worst case
"""
def retrieve(self, i):

    root = self.getRoot()

    def retrieveRec(node, j):
        loc = node.left.size + 1

        if loc == j:
            return node.getValue()
        elif j < loc:
            return retrieveRec(node.left, j)
        else:
            return retrieveRec(node.right, j - loc)

    return retrieveRec(root, i + 1)
```

- *retrieveNode(self, i)* - retrieves the AVLNode that is the i'th item in the list.
  retrieve works as Tree-Select works as we saw in the lecture:

```
retrieve(self, i):
        root = self.root
        r = root.left.size + 1
        if (i == r):
                return root
        else:
                if i < r, search for the i^th smallest item in the left subtree
                otherwise (i > r), search for the (i - r)^th smallest item in the right subtree
```

  Time complexity: as we saw in the lecture O(h) = O(logn)

```
"""retrieves the AVLNode that is the i'th item in the list

@type i: int
@pre: 0 <= i < self.length()
@param i: index in the list
@rtype: AVLNode
@returns: the AVLNode that is the i'th item in the list

Time Complexity:
Recursion that in worst case goes every call goes one son left / one son right until the deepest leaf
Meaning that the maximum calls is the height of tree
In every recursion call there is O(1) work, and the height of the tree is O(logn)
That is why, in total, as we saw in the lecture, the time complexity is O(logn) in the worst case
"""

def retrieveNode(self, i):

    root = self.getRoot()

    def retrieveRec(node, j):
        loc = node.left.size + 1

        if loc == j:
            return node
        elif j < loc:
            return retrieveRec(node.left, j)
        else:
            return retrieveRec(node.right, j - loc)

    return retrieveRec(root, i + 1)
```

- *swapNodes(self, node1, node2)* - swapping 2 nodes (changing pointers)
  To understand the function it is best to think of 2 nodes that their values are changed.
  Time complexity: constant amount of pointer changes - O(1)

  (The implementation is too long for a screenshot :( )

- *reBalance(self, nodeToCheckBF, treeOp)* - Re balancing the tree.
  This function is responsible for rebalancing the tree after tree operations for example insert, delete.
  If the nodeToCheckBF or any node from him to the root needs a re-balance operation (rotation / recomputing height) this function will take care of it.
  In addition, it maintains the size field of the AVLNodes in that route.
  Time complexity:

  Worst case - maximum route from a node to the root is O(h) = O(logn).
  In every node in that route, O(1) work is executed in the worst case (rotation + arithmetic operation)
  Total: O(logn) + O(1) = O(logn)

  (The implementation is too long for a screenshot :( )

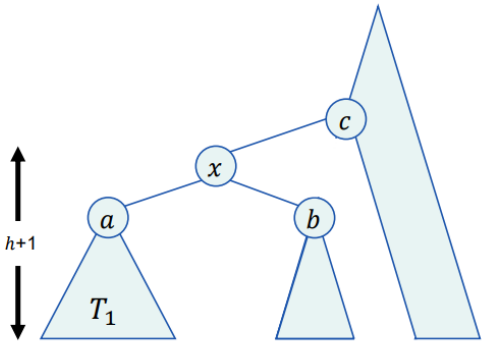
- *rotate(self, BFcriminal, balanceFactor)* - Applies the correct rotation to the tree, during the rebalance process.
  According to the BFcriminal's BF (balanceFactor), it applies the rotation that is needed (according the logic we saw in the lecture)
  Time complexity:
  Two rotations are executed in the worst case - O(1)

```
""" Applies the correct rotation to the tree, during rebalance process

@type BFcriminal: AVLNode
@pre: BFcriminal is not None
@param BFcriminal: a node that its BalanceFactor violating the AVLTreeList balance rules (+2/-2)

@type balanceFactor: int
@param balanceFactor: the balance factor of BFcriminal

@rtype: int
@return: number of balancing operations that took place
"""

def rotate(self, BFcriminal, balanceFactor):
    balanceOps = 0
    if balanceFactor == 2:
        if BFcriminal.getLeft().getBalanceFactor() in [0, 1]:
            self.rightRotation(BFcriminal)
            balanceOps += 1
        elif BFcriminal.getLeft().getBalanceFactor() == -1:
            self.leftThenRightRotation(BFcriminal)
            balanceOps += 2

    elif balanceFactor == -2:
        if BFcriminal.getRight().getBalanceFactor() in [-1, 0]:
            self.leftRotation(BFcriminal)
            balanceOps += 1
        elif BFcriminal.getRight().getBalanceFactor() == 1:
            self.rightThenLeftRotation(BFcriminal)
            balanceOps += 2
    return balanceOps
```

- *join(T1, x, T2)* - Static method for joining 2 AVL trees and a node x
  Same as it was introduced in class, lecture 3 slides 100-104

Join(T_1,x,T_2) when "$T_1 < x < T_2$"

Assume $height(T_1) \leq height\ (T_2)$



Attach x to b's former parent (denoted c)

Do rebalancing from x upwards, if needed (when?)

Using 2 side methods (at most one of them is called):
find_right_subtree_heightH(h), find_left_subtree_heightH(h)
Time complexity:
In the worst case,
Find subtree with the asked height: O(abs(h2-h1))
Rotating up using rebalance: O(abs(h2-h1))
So, O(abs(h2-h1))
Returns a tuple containing on index 0 the joined tree, and in index 1 the number of rebalance operations, which is used for the theoretical part.

(The implementation is too long for a screenshot :( )

- *find_right_subtree_heightH(self, h)* - sub method for joining.
  Finding the subtree of the root whose height is h/h-1, as we saw in the lecture.
  Go right until found.
  Time complexity: O(self.getRoot().getHeight()-h)

```
"""
find right subtree with height h or height h-1
@pre - h>=0
@rtype: AVLNode
Time complexity: O(self.getRoot().getHeight()-h)
"""
def find_right_subtree_heightH(self, h):
    if self.getRoot().getHeight() == h:
        return self
    node = self.getRoot()
    help = node
    while h<help.getHeight():
        if help.getRight().isRealNode():
            help = help.getRight()
        else:
            return help.getRight()
    return help
```

- *find_left_subtree_heightH(self, h)* - sub method for joining.
  Finding the subtree of the root whose height is h/h-1, as we saw in the lecture.
  Go left until found.
  Time complexity: O(self.getRoot().getHeight()-h)

```
"""
find left subtree with height h or height h-1
@pre - h>=0
@rtype: AVLNode
Time complexity: O(self.getRoot().getHeight()-h)
"""
def find_left_subtree_heightH(self, h):
    if self.getRoot().getHeight() == h:
        return self.getRoot()
    node = self.getRoot()
    help = node
    while h<help.getHeight():
        if help.getLeft().isRealNode():
            help = help.getLeft()
        else:
            return help.getLeft()
    return help
```

- *concat(self,lst)* - concatenates lst to self
  In order to simply understand in terms of List ADT what's happening in the function, think of 2 lists in python that you concat (self + lst). In terms of AVLTreeList, we delete the maximum node of self calling self.join(self, deletedNode, lst).
  Time complexity:
  In the worst case, we call delete once, join once, and other constant O(1) operations. Therefore, in total it is O(logn).

```
"""concatenates lst to self

@type lst: AVLTreeList
@param lst: a list to be concatenated after self
@rtype: int
@returns: the absolute value of the difference between the height of the AVL trees joined

@Time complexity:
Worst case:
delete - called once - O(logn)
insert - called once - O(logn)
join - called once - O(logn)
Total: O(logn)
"""
def concat(self, lst):
    selfHeight = -1 if self.getRoot() is None else self.getRoot().getHeight()
    lstHeight = -1 if lst.getRoot() is None else lst.getRoot().getHeight()
    absHeightDiff = abs(lstHeight - selfHeight)

    if self.empty():
        self.root = lst.getRoot()
        self.set_First(lst.get_First())
        self.set_Last(lst.get_Last())
        return absHeightDiff

    if lst.empty():
        return absHeightDiff

    x = self.get_Last()

    self.delete(self.getRoot().getSize() - 1)
    joinedTree = AVLTreeList.join(self, x, lst)[0]

    self.root = joinedTree.getRoot()
    self.set_First(joinedTree.get_First())
    self.set_Last(joinedTree.get_Last())

    return absHeightDiff
```

- *split(self,i)* - we start by finding the i'th item in our AVL tree(O(logn)).
  After that, we loop through the tree and save a pointer to the parent of the current node, and execute a join with the matching subtrees (changing the order of trees joined).
  After that, we maintain the first and last items of each tree using min, max.
  From lecture 3 (slide 107) the time complexity of split is O(logn), fitting our implementation because each join is O(abs(height(T1)-height(T2))+1), which is the work the slide counts on that takes each join. So total O(logn), retrieve node at index i is also O(logn), and minimum and maximum are called twice totally, and each one of them is also O(logn).
  To conclude, time complexity is O(logn).

  (The implementation is too long for a screenshot :( )

- *create_tree_from_node(self, node)* - return a tree whose root is a given node. We'll use this method only for splitting so we don't set the first and last nodes of the tree.
  Time complexity: O(1)

```
"""
Create a tree that his root is a given node
@param: AVLnode
@pre - node.isRealNode() == True
@returns: a tree t which t.getRoot() == node
Time complexity: O(1)
"""


def create_tree_from_node(self, node):
    node.setParent(None)
    t = AVLTreeList()
    t.root = node
    return t
```

- *insert(self, i, val)* - insert a node with value val into the tree at index i. Retrieving the node at index i. If he doesn't have a real left child, then insert a new AVLNode with value val as his left child. If he has a real left child, then we are adding AVLNode with value val as a right child to his predecessor (max of left subtree).
  Time complexity:
  In the worst case, functions that are being called:
  empty - O(1)
  length - O(1)
  retrieve(i) - O(logn)
  isRealNode() - O(1)
  predecessor() - O(logn)
  rebalance() - O(logn)
  Therefore, total time is O(logn)
  (The implementation is too long for a screenshot :( )

- *delete(self, i)* - deletes the i'th item in the list
  Retrieving the node at index i.
  If this is a leaf, we delete it and do rebalance operations from its parent to the root.
  If this is a `one childed` node then we do a bypass (connecting his parent to his one child and vice versa)
  If this is a two-childed node then we swap him and his successor. The new position of the node (the successor) position applies now one of the cases above (it is either a leaf or has only left child - one childed)
  After that, rebalancing the tree from the new position's parent to the route.
  Time complexity:
  In the worst case, functions that are being called
  successor - called once - O(logn)
  swapNodes - called once - O(1)
  deleteLeaf/deleteOneChildedNode - called once - O(logn)
  Other O(1) constant operations inside the func
  Therefore, total time is O(logn)

  (The implementation is too long for a screenshot :( )

- *deleteOneChildedNode(self, nodeToDelete, childSide)* -
  deletes a node that has one child (also termed 'bypass' in lecture's slides - see explanation in delete).
  Time complexity:
  reBalance - called once - O(logn)
  successor / predecessor - called once - O(logn)
  Other O(1) constant operations inside the func
  Therefore, total time is O(logn)

  (The implementation is too long for a screenshot :( )

- *deleteLeaf(self, nodeToDelete, case)* - deletes a node that is a leaf (unconnecting a node and his parent)
  Time complexity:
  reBalance - called once - O(logn)
  predecessor & successor - called maximum once each - O(logn)
  Other O(1) constant operations inside the func
  Therefore, total time is O(logn)

  (The implementation is too long for a screenshot :( )

- *getRoot(self)* - return the root of the tree.
  If the tree is empty, the method returns None.
  Otherwise, the method returns a pointer to the root of the tree.
  Time complexity: O(1)

```
"""returns the root of the tree representing the list

@rtype: AVLNode
@returns: the root, None if the list is empty
@Time complexity: O(1)
"""
def getRoot(self):
    if self.root is not None:
        return self.root
    return None
```

- *first(self)* - returns the value of the first item of the tree
  If the tree is empty, the method returns None.
  Otherwise, the method returns self.first_node.getValue()
  (self.first_node.getValue() can also be None)
  Time complexity: O(1)

```
"""returns the value of the first item in the list

@rtype: str
@returns: the value of the first item, None if the list is empty
@Time complexity: O(1)
"""
def first(self):
    if self.first_node is not None:
        return self.first_node.getValue()
    return None
```

- *last(self)* - returns the value of the last item of the tree.
  If the tree is empty, the method returns None.
  Otherwise, the method returns self.last_node.getValue()
  (self.last_node.getValue() can also be None)
  Time complexity: O(1)

```
"""returns the value of the last item in the list

@rtype: str
@returns: the value of the last item, None if the list is empty
@Time complexity: O(1)
"""
def last(self):
    if self.last_node is not None:
        return self.last_node.getValue()
    return None
```

- *listToArray(self)* - returns an array representing list
  Starting from the minimal element and doing n-1 successor operations, until the successor of the last element returns None:

```
1. listToArray(self):
2.         lst = [] #empty list that has append method
3.         node = self.get_First()
4.         while node is not None:
5.                 lst.append(node.getValue())
6.                 node = self.successor(node)
7.         return lst
```

Time complexity:
Rows 2 & 3 takes O(1) time - initializing an empty list and returning an attribute (of the AVLNode) from an object.
Naively, at first, it seems like n-1 successor operations where every successor takes O(logn) would be O(nlogn) work in total.
But, as we saw in recitation 3 ex 3, starting at the minimal element and doing n-1 successor operations takes O(n) time.
Reminder: we go through every edge in the tree at most 2 times.
Number of edges in an n nodes tree is n-1 and therefore the total Time complexity is O(n).

```
"""returns an array representing list

@rtype: list
@returns: a list of strings representing the data structure

Time Complexity:
Initiliazing an empty list - O(1)
get_First() - returns an attribute of self, without any calculations - O(1)

Conclusion from recitation 3 ex 3 - starting at the minimal element of a tree and calling
n-1 times to successor
is O(n) work since we go through every edge (there are n-1 edges) at most 2 times.
Plus,
    node is not None - O(1), n times -> O(n)
    lst.append(node.getValue()) - O(1), n times -> O(n)

Therefore, the entire while loop takes O(n) time in the worst case.
"""
def listToArray(self):
    lst = []
    node = self.get_First()
    while node is not None:
        lst.append(node.getValue())
        node = self.successor(node)
    return lst
```

- *length(self)* - returns the size of the list
  If self.empty(), returns 0, else returns self.root.getSize()
  Time complexity: O(1)

```
"""returns the size of the list

@rtype: int
@returns: the size of the list
"""
def length(self):
    if not self.empty():
        return self.root.getSize()
    # returns 0 if list is empty
    return 0
```

- *successor(self, node)* - returns the successor of a given node.
  If node == self.get_Last() - returns None.
  If node.right is not a virtual node, then the successor of node is the minimum element in node.right subtree.
  If node.right is a virtual node then the successor of node is the lowest ancestor y of x such that x is in its left subtree.
  Successor works as we saw in the lecture:

```
successor(self, node):
        if node == self.get_Last():
                return None
        x = node
        if x.right is not None:
                return minimum(x.right)
        y = x.parent
        while y is not None and x = y.right:
                x = y
                y = x.parent
        return y
```

  Time complexity: as we saw in the lecture the time complexity analysis is in the worst case O(h) = O(logn).
  In case node == self.get_Last() then O(1).

```python
"""returns the successor of a given node

@pre: node.isRealNode() == True
@type node: AVLNode
@param node: the node of which we will return its successor
@rtype: AVLNode
@returns: The successor of node, None if node is the last element in the list

Time complexity:
As we saw in the lecture the time complexity analysis is in the worst case O(h) = O(logn)
In case that node == self.get_Last() then O(1)
"""
def successor(self, node):
    if node == self.get_Last():
        return None

    x = node
    if x.getRight().isRealNode():
        return self.minimum(x.getRight())
    y = x.getParent()
    while y is not None and x == y.right:
        x = y
        y = x.parent
    return y
```

- *predecessor(self,node)* - return the predecessor of a given node.
  If node.left is not a virtual node, the predecessor of node is the maximum node on the left subtree.
  If node.left is a virtual node, then we go up to the lowest ancestor of node such that node is in its right subtree.
  Time complexity: as we saw in the lecture the time complexity analysis is in the worst case O(h) = O(logn).
  In case node == self.get_First() then O(1).

```
""" find the predecessor of a given node
@param - AVLNode
@return - AVLNode, the predecessor of the node. if it's the first node, return null
@Time complexity: O(logn) worst case, go through the height of the tree(logn),
each move O(1) work(pointers switch)
if our node is the first node of the tree, O(1) time complexity
"""
def predecessor(self, node):
    if not node.isRealNode():
        return None
    if self.get_First() == node:
        return None
    if node.getLeft() is not None and node.getLeft().isRealNode():
        help = node.getLeft()
        while help.getRight() is not None and help.getRight().isRealNode():
            help = help.getRight()
        return help

    help = node.getParent()
    while help is not None and help.isRealNode():
        if help.getRight() == node:
            return help
        node = help
        help = help.getParent()
    return None
```

- *minimum(self, node)* - returns the minimum of a given sub tree that node is its root.

```
1. minimum(self, node):
2.         minimum = node
3.         while minimum.left is not None:
4.                 minimum = minimum.left
5.         return minimum
```

Time complexity:
(2) minimum = node is O(1) work
(3) minimum.left is not None is O(1) work
(4) minimum = minimum.left is O(1) work
Rows 3 & 4 are executed at most as many times as the height of the tree.
Therefore, the total time complexity is O(h) = O(logn).

```
"""returns the minimum of a given sub tree that node is its root
i.e. the deepest node that is on the `/` branch that starts from node

@pre: node.isRealNode() == True
@type node: AVLNode
@param node: the node of which we will return the minimum of his subtree
@rtype: AVLNode
@returns: The minimum of node's subtree, if it is a leaf, returns itself

Time complexity:
minimum = node - O(1)
(*) minimum.getLeft() is not None && minimum = minimum.getLeft() are O(1) each
(*) is executed at most as many times as the height of the tree.
Therefore, the total time complexity is O(h) = O(logn).
"""
def minimum(self, node):
    minNode = node
    while minNode.getLeft().isRealNode():
        minNode = minNode.getLeft()
    return minNode
```

- *maximum(self, node)* - returns the maximum of a given sub tree that node is its root.

```
1. maximum(self, node):
2.         maximum = node
3.         while maximum.right is not None:
4.                 maximum = maximum.right
5.         return maximum
```

Time complexity:
(2) maximum = node is O(1) work
(3) maximum.right is not None is O(1) work
(4) maximum = maximum.left is O(1) work
Rows 3 & 4 are executed at most as many times as the height of the tree.
Therefore, the total time complexity is O(h) = O(logn).

```
"""returns the maximum of a given sub tree that node is its root
i.e. the deepest node that is on the `\` branch that starts from node

@pre: node.isRealNode() == True
@type node: AVLNode
@param node: the node of which we will return the maximum of his subtree
@rtype: AVLNode
@returns: The maximum of node's subtree, if it is a leaf, returns itself

Time complexity:
maximum = node - O(1)
(*) maximum.getRight() is not None && maximum = maximum.getRight() are O(1) each
(*) is executed at most as many times as the height of the tree.
Therefore, the total time complexity is O(h) = O(logn).
"""
def maximum(self, node):
    maxNode = node
    while maxNode.getRight().isRealNode():
        maxNode = maxNode.getRight()
    return maxNode
```

- *set_First(self, node)* - sets first_node property of the tree (self) to a given node.
  Time complexity: O(1)

```
"""sets the first item of the list to a given node

@param node: a pointer to a AVLNode
@Time complexity: O(1)
"""
def set_First(self, node):
    self.first_node = node
```

- *get_First(self)* - return a pointer to self.first_node property.
  Time complexity: O(1)

```
"""returns a pointer to the first node
@rtype: AVLNode
@Time complexity: O(1)
"""
def get_First(self):
    return self.first_node
```

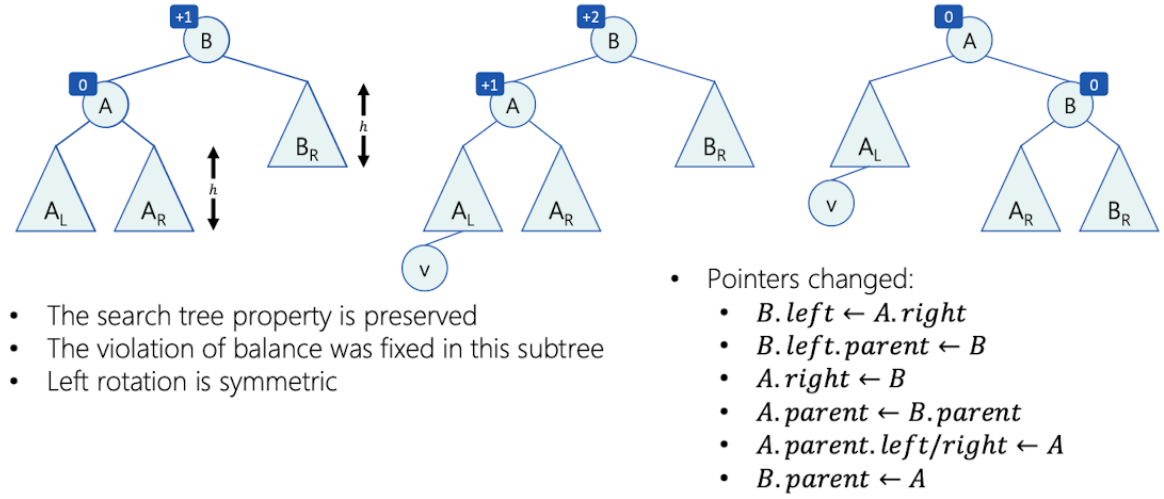
- *set_Last(self, node)* - sets last_node property of the tree (self) to a given node.
  Time complexity: O(1)

```
"""sets the last item of the list to a given node

@param node: a pointer to a node
@Time complexity: O(1)
"""
def set_Last(self, node):
    self.last_node = node
```

- *get_Last(self)* - returns a pointer to self.last_node property.
  Time complexity: O(1)

```
"""return a pointer to the last item in the list
@param node: a pointer to AVLNode
@Time complexity: O(1)
"""
def get_Last(self):
    return self.last_node
```

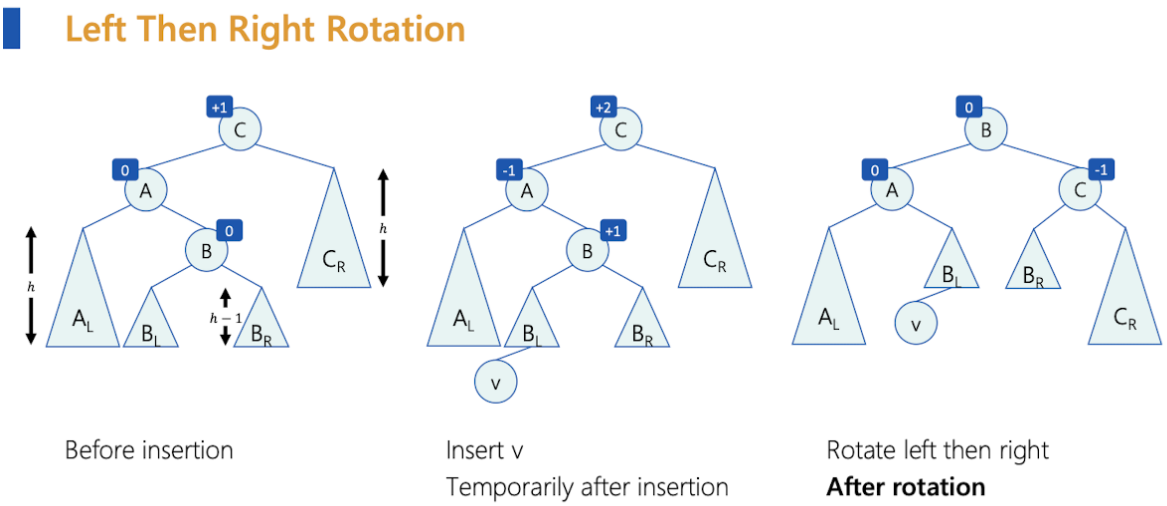
- *rightRotation(self, BFcriminal)* - performs a right rotation inplace.
  Visualization & pseudocode (taken from lecture's slides):



  Time complexity: changing constant amount of pointers + recalculating size & height of B & A (constant arithmetic operations) - O(1)
  (The implementation is too long for a screenshot :( )

- *leftRotation(self, BFcriminal)* - performs a left rotation inplace.
  Visualization & pseudocode are symmetrical for rightRotation from above.
  Time complexity: changing constant amount of pointers + recalculating size & height of B & A (constant arithmetic operations) - O(1)
  (The implementation is too long for a screenshot :( )

- *leftThenRightRotation(self, BFcriminal)* - performs self.leftRotation(BFcriminal.getLeft()) and then self.rightRotation(BFcriminal).
  See leftRotation & rightRotation for more details.
  Visualization (taken from lecture's slides):



Time complexity: leftRotation + rightRotation - O(1)

```
"""performs a left then right rotation inplace

@pre: called from reBalance function (due to a tree operation)

@type BFcriminal: AVLNode
@pre: BFcriminal's BF is +2 (i.e. it has a real left son) and left son BF is -1
(i.e. it has a real right son)
@param BFcriminal: node that violates the balance rules of an AVL tree
@post BFcriminal: node won't violate the balance rules of an AVL tree anymore

@rtype: None
"""

def leftThenRightRotation(self, BFcriminal):
    self.leftRotation(BFcriminal.getLeft())
    self.rightRotation(BFcriminal)
```

- *rightThenLeftRotations(self, BFcriminal)* - performs self.rightRotation(BFcriminal.getRight()) and then self.leftRtoation(BFcriminal). Symmetrical to leftThenRightRotation. See rightRotation & leftRotation for more details.
  Time complexity: rightRotation + leftRotation - O(1)

```
"""performs a right then left rotation inplace

@pre: called from reBalance function (due to a tree operation)

@type BFcriminal: AVLNode
@pre: BFcriminal's BF is -2 (i.e. it has a real right son) and right son BF is +1
(i.e. it has a real left son)
@param BFcriminal: node that violates the balance rules of an AVL tree
@post BFcriminal: node won't violate the balance rules of an AVL tree anymore

@rtype: None
"""

def rightThenLeftRotation(self, BFcriminal):
    self.rightRotation(BFcriminal.getRight())
    self.leftRotation(BFcriminal)
```

- *Search(self, val)* - given a value val, we return the first index of a node such that node.getValue() == val.

Return -1 if there is no such a node.
First of all, if the tree is empty, we will return -1.
If the first node of the list has the value, we will return 0.
If our value is not one of the above, we will convert the tree to a list using listToArray() method (O(n) time complexity), and iterate through it from the first value to the last value (O(n) time complexity).
If some value == val then we will return the index of the first appearance.
If not found, return -1.
Time complexity: O(n)

```
    """searches for a *value* in the list

    @type val: str
    @param val: a value to be searched
    @rtype: int
    @returns: the first index that contains val, -1 if not found.
    @Time complexity: O(n) worst case (list to array(O(n), iterate through the array: O(n))
    """
    def search(self, val):
        if self.empty():
            return -1
        if self.first() == val:
            return 0
        lst = self.listToArray() #O(n), see listToArray time complexity
        i = 0
        for x in lst:
            if x == val:
                return i
            i += 1
        return -1
```

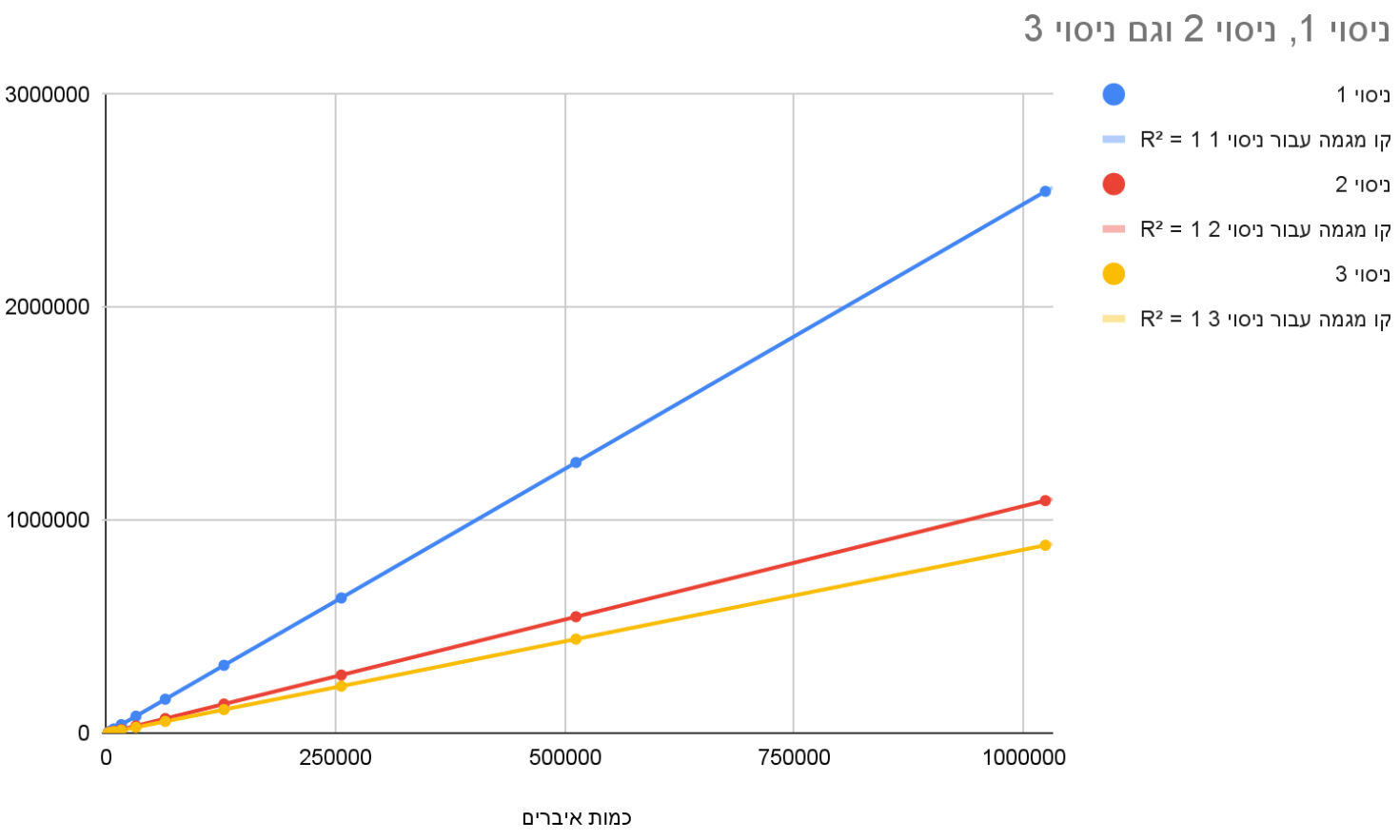
חלק ניסויי

שאלה 1

.1

מספר סידורי i	ניסוי 1 - הכנסות	ניסוי 2 - מחיקות	ניסוי 3 - הכנסות ומחיקות לסירוגין
1	4952	2087	1687
2	9902	4257	3469
3	19915	8484	6941
4	39720	17018	14005
5	79414	33934	27292
6	159172	68419	55055
7	318501	136470	110595
8	634680	272782	220460
9	1271009	546552	441540
10	2544418	1091849	882189

.2

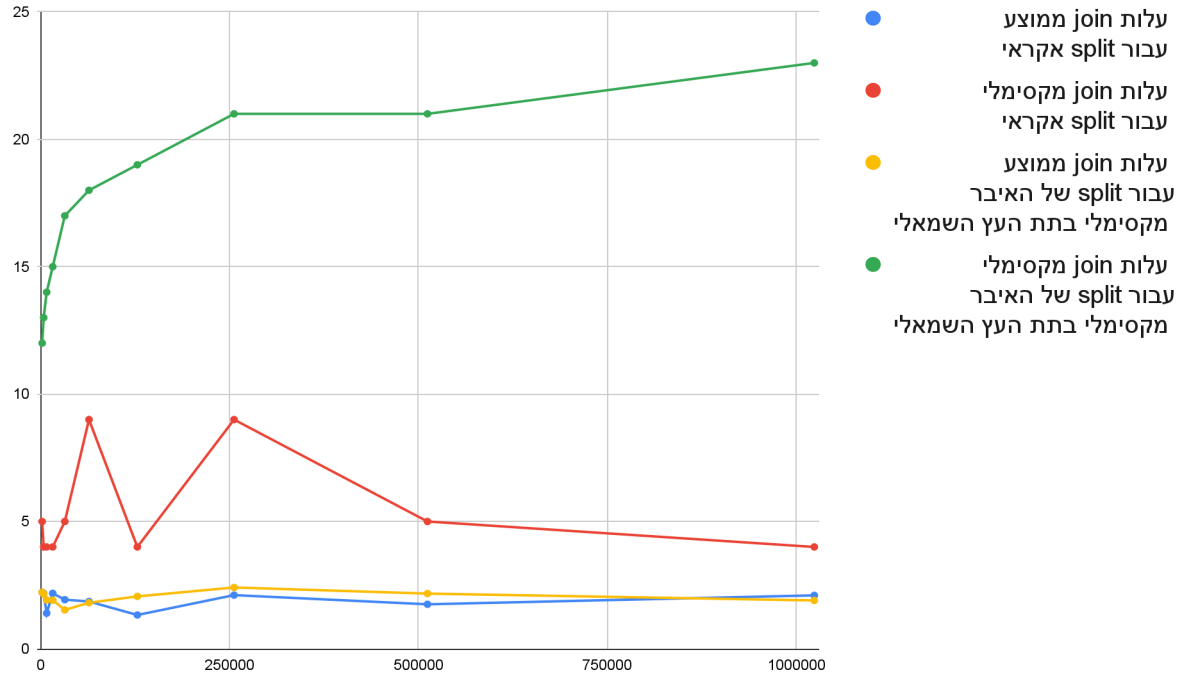


בנוסף, בכל העלאה של i ב 1, אנחנו מגדילים את כמות האיברים פי 2, וכפי שניתן לראות מהנתונים, כמות פעולות האיזון גדלה בקירוב פי 2 לכל אחד מהניסויים, מה שתואם את כך שאנחנו מקבלים עבור R^2 את הערך 1.

שאלה 2

1.

מספר סידורי i	עלות join **ממוצע** עבור split **אקראי**	עלות join **מקסימלי** עבור split **אקראי**	עלות join **ממוצע** עבור split של **האיבר** מקסימלי בתת העץ השמאלי	עלות join **מקסימלי** עבור split של **האיבר** מקסימלי בתת העץ השמאלי
1	2.22	5	2.22	12
2	2.18	4	2.16	13
3	1.4	4	1.92	14
4	2.18	4	1.91	15
5	1.93	5	1.53	17
6	1.86	9	1.81	18
7	1.33	4	2.06	19
8	2.11	9	2.41	21
9	1.75	5	2.17	21
10	2.1	4	1.9	23



2. כפי שראינו בהרצאה העלות של join היא O (הפרש הגבהים +1).

כפי שניתן לראות מהנתונים עבור 2 התרחישים, הפרש הגבהים הממוצע חסום ע"י קבוע כלשהו(בסדרת הניסוים האקראית שלנו, ע"י 3), אז ניתן להסיק שהעלות של join ממוצע היא O(3+1) ששקול ל- O(1).

התוצאות אכן מתיישבות עם ניתוח הסיבוכיות התיאורטי של split (שראינו בהרצאה שהוא O(logn)) וזאת מכיוון שכמות ה joinים בפעולת split היא לכל היותר גובה העץ (ששקול ל O(logn) בעצי AVL) ולכן עבור כמות O(logn) פעולות join, הסיבוכיות הכללית תצא O(logn) במידה והעלות הממוצעת של כל פעולה תהיה O(1).

כעת נוכיח פורמלית:

יהי T, עץ AVL כלשהו.
נוכיח שלכל $v \in T$ שנתחיל ממנו פעולת split, העלות הממוצעת של join תהא $O(1)$.

- ההוכחה תקפה ל2 התרחישים, גם עבור split אקראי וגם למקרה של split מהצומת המקסימלי.

יהי $v \in T$ נסמן h להיות הגובה של T, וh_v להיות הגובה של v. נסמן בjOp את מספר פעולות הjoin במהלך פעולת הsplit, ונסמן בt את העלות המשותפת של כל פעולות הjoin שבוצעו במהלך הsplit.
לכן, העלות הממוצעת תהא $\frac{t}{jOp}$.
נראה כי $\frac{t}{jOp} = O(1)$.
נגדיר את rt להיות העלות המשותפת של פעולות הjoin אשר התבצעו במהלך פעולת הsplit עבור תת העץ הימני שנוצר מפעולת הsplit (כל מי שגדול מ v) באופן סימטרי נגדיר את lt.
מתקיים: $t = rt + lt$.
נסמן את $T_1, T_2, \ldots, T_k$ העצים שביצענו עליהם join בתהליך יצירת העץ הימני.
מתקיים, מהשקופית (הרצאה 3, שקופית 107) של ניתוח הסיבוכיות של split אשר ראינו בהרצאה:

$$rt = \sum_{i=2}^{k} |h(T_i) - h(join(T_1, \ldots, T_{i-1}))| =^{(c>0,\ lemma\ 2\ from\ slide)} \sum_{i=2}^{k} |h(T_i) - (h(T_{i-1}) + c)| \leq^{(Lemma\ 1 + c>0)}$$

$$\sum_{i=2}^{k} |h(T_i) - h(T_{i-1}) + c| = \sum_{i=2}^{k} h(T_i) - h(T_{i-1}) + c =^{(Telescopic\ sum)} h(T_k) - h(T_1) + c(k-1) \leq^{(k-1 \leq jOp)}$$

$$h - h(T_1) + c * jOp \leq^{(h(T_1) \geq h_v - 2)} h - (h_v - 2) + c * jOp = h - h_v + 2 + c * jOp =^{(h - h_v = O(jOp))} O(jOp)$$

באותו האופן עבור תת העץ השמאלי, כלומר $lt = O(jOp)$.
לכן נקבל בסך הכל:
$\frac{t}{jOp} = \frac{lt}{jOp} + \frac{rt}{jOp} = \frac{O(jOp)}{jOp} + \frac{O(jOp)}{jOp} = O(1)$
כנדרש.

3. ננתח את העלות של join מקסימלי בתרחיש.

ראשית ניזכר שעלות join היא $O(abs(h(T_1) - h(T_2)) + 1)$ לכל 2 תתי עצים של T, עץ AVL.

לכן הפרש הגבהים המקסימלי יתקבל כאשר $h(T_1)$ מינימלי (בניסוי אחד, מאחר ואנו בוחרים צומת אקראי, אזי הגובה המינימלי הוא -1 אם נבחר לעשות split מצומת שהוא עלה ואז 2 בניו יהיו עצים ריקים שכאמור גובהם הוא -1) ו$h(T_2)$ מקסימלי (שזה קורה כאשר ה$h(T_2)$ שווה לגובה העץ) שכידוע בעצי AVL הוא $O(logn)$. באופן דומה אם $h(T_1)$ מקסימלי, ו$h(T_2)$ מינימלי.

כלומר,

$$abs(h(T_1) - h(T_2)) \leq abs(O(logn) + 1) = O(logn)$$

ולכן העלות המקסימלית של פעולת join היא

$$O(abs(h(T_1) - h(T_2)) + 1) \leq O(O(logn) + 1) = O(logn).$$

עובדה זו מתיישבת עם ניתוח הסיבוכיות התיאורטי שעשינו עבור join, שכן ראינו בהרצאה כי היא $O(logn)$. ובנוסף זה מתיישב עם התוצאות האמפיריות שקיבלנו, שכן הכפלה פי 2 של כמות האיברים מגדילה בקבוע קטן יחסית את עלות הjoin המקסימלי.

שאלה 3

מספר פעולות האיזון בממוצע	עץ AVL סדרה חשבונית	עץ ללא מנגנון איזון סדרה חשבונית	עץ AVL סדרה מאוזנת	עץ ללא מנגנון איזון סדרה מאוזנת	עץ AVL סדרה אקראית	עץ ללא מנגון איזון סדרה אקראית
מספר סידורי i						
1	2.974	499.5	0.994	0.994	2.47	2.58
2	2.986	999.5	0.997	0.997	2.51	2.42
3	2.989	1499.5	0.998	0.998	2.44	2.40
4	2.992	1999.5	0.9985	0.9985	2.516	2.452
5	2.9938	2499.5	0.999	0.999	2.485	2.479
6	2.9945	2999.5	0.998	0.998	2.476	2.439
7	2.995	3499.5	0.999	0.999	2.474	2.422
8	2.996	3999.5	0.9992	0.9992	2.464	2.447
9	2.996	4499.5	0.9994	0.9994	2.481	2.41
10	2.996	4999.5	0.9995	0.9995	2.491	2.495

עבור הסדרה החשבונית, בעץ לא מאוזן אנחנו בעצם יוצרים שרוך, ולכן מספר פעולות האיזון בכל הכנסה (בכל הכנסה העומק של הצומת המוכנס עולה ב1) בעומק i היא i, ולכן כמות פעולות האיזון עבור הסדרה היא סדרה חשבונית עם הפרש 1, ועל כן הממוצע יהיה סכום הסדרה חלקי כמות ההכנסות, מסתדר עם תוצאות הניסוי (לינארי בעומק העץ).
לגבי המקרה הזהה עבור עץ AVL, זה תואם את הציפייה ע"פ המסקנות משאלה 1, כי ראינו שעבור n הכנסות במיקום 0 לעץ, אנחנו מקבלים O(n) פעולות איזון, ולכן בממוצע O(1) פעולות איזון.

לגבי הכנסה מאוזנת, מסתדר מאוד עם האינטואיציה שכמות פעולות האיזון של עץ AVL ועץ לא מאוזן יהיו זהות, כי מטרת ההכנסה היא שלא יהיה צורך בגלגולים, ובעבור הכנסות אלו אין הבדל כלל בין עץ AVL לעץ לא מאוזן. בעת הכנסות אלו, העץ יהיה עץ בינארי כמעט מושלם, שכידוע עומקו הוא $floor(log(n))$ ולכן נצפה שהעומק הממוצע יהיה נמוך וזהה, בדומה לתוצאות הניסוי.

לגבי הכנסת סדרה אקראית, הציפייה מבחינת עץ AVL היא שהעץ יהיה מאוזן ולכן עומק ההכנסה והשינויים בגובה יהיו נמוכים מכמות פעולות האיזון בעץ רגיל, אך במקרים מסוימים בהם בעץ AVL נצטרך לבצע גלגול, בעץ רגיל אין צורך בכך, ועל כן יש tradeoff מסויים בין עומק ההכנסה לבין הגלגולים. זה factor שאין לנו דרך טובה להעריך בגללו את תוצאות הניסוי, משום שההכנסה היא אקראית.
על פי תוצאות הניסוי, אנו רואים שכאשר מכניסים איברים בצורה אקראית, אין תוצאה חד משמעית בדומה להערכתינו, הממוצע נשאר דומה בין שני סוגי העצים, והוא בין 2.4-2.6 פעולות איזון ממוצעות להכנסה אחת, ומכך שניתן לשער שבממוצע, בניית עץ אקראית יוצרת עץ יחסית מאוזן, כלומר הגובה לא גדל באופן משמעותי בעץ ללא איזונים בהכנסה אקראית, ולכן מספר פעולות האיזון יחסית קרוב בין שני הסוגים.
בסה"כ ניתן לראות שבכל ההכנסות ללא תלות בסדרה, ממוצע פעולות האיזון ב- AVL הוא O(1). הממוצע עולה כאשר מכניסים רק באינדקס 0, אך נשמר סביב 3 פעולות בממוצע, והכנסה ששומרת על איזון אופטימלי היא בעלת מספר פעולות האיזון הוא הנמוך ביותר, פחות מ-1 בממוצע להכנסה.

עץ ללא מנגנון איזון סדרה אקראית	עץ AVL סדרה אקראית	עץ ללא מנגנון איזון סדרה מאוזנת	עץ AVL סדרה מאוזנת	עץ ללא מנגנון איזון סדרה חשבונית	עץ AVL סדרה חשבונית	עומק הצומת המוכנס בממוצע
						מספר סידורי i
10.998	8.727	7.987	7.987	499.5	8.977	1
12.77	9.74	8.982	8.982	999.5	9.9765	2
12.17	10.29	9.64	9.64	1499.5	10.635	3
13.42	10.79	9.98	9.98	1999.5	10.976	4
13.9	11.14	10.36	10.36	2499.5	11.36	5
13.91	11.32	10.64	10.64	2999.5	11.63	6
14.61	11.6	10.83	10.83	3499.5	11.83	7
15.56	11.75	10.977	10.977	3999.5	11.97	8
14.77	11.95	11.18	11.18	4499.5	12.18	9
15.52	12.11	11.36	11.36	4999.5	12.36	10

לגבי הכנסת סדרה חשבונית, בעץ לא מאוזן זה בדומה לכמות פעולות האיזון, כל פעם נכניס צומת ברמה חדשה i, ועל כן החישוב זהה לחישוב פעולות האיזון הממוצעות.

באותו מקרה עבור עץ AVL, בכל רמה i אנחנו נכניס פעם אחת עלה בעומק הזה, ולאחר מכן $2^i - 1$ עלים בעומק i, וברמה האחרונה, נכניס פעם אחת עלה בעומק שלה, ובנוסף את יתר האיברים שנותרו בעץ בעומק הרמה האחרונה ועוד 1. מה שיוצר לנו סכום יחסית נוח לחישוב ותואם בדיוק את העומק הממוצע. למשל עבור i=1:

$$\frac{\sum_{i=0}^{8}(i+(2^i-1)*(i+1))+9+488*10}{1000} = 8.977$$

עבור הכנסות מאוזנות, העצים זהים מבחינת עומק ההכנסה, כי לא נצטרך פעולות גלגול בהכנסות אלו, מה שמסתדר עם הנתונים שקיבלנו. למשל עבור i=1:

$$\frac{(\sum_{i=0}^{8}2^i*i)+489*9}{1000} = 7.987$$

לגבי הכנסה אקראית, ראשית מסתדר עם ההיגיון כי העומק הממוצע של עצי AVL יהיה נמוך מהעומק הממוצע של הכנסות עבור עץ לא מאוזן, כי עץ AVL הוא מאוזן ולכן גובהו יהיה קטן שווה לעומק של העץ הלא מאוזן, ולכן בממוצע נכניס בעומק קטן יותר מזה של העץ הלא מאוזן.

בנוסף, מסתדר גם כי העומק הממוצע הינו בטווח בין סדרת ההכנסות הכי גרועה (כל פעם במקום האפס) לבין סדרת ההכנסות הכי טובה (הכנסה מאוזנת).