2023

AVL Tree List

מיאר ספדי

user name MayarSafadi

209826924 id

מרים חלאילה

user name Mariamk

id 212346076

ניתוח סיבוכיות זמן

we assume the tree has n nodes

- def retrieve(self, i)

returns the value of the node that is in the index i, witch is the node with the rank=i+1
the function uses TreeSelect => **Time Complexity is O(logn)**

- def TreeSelect(self, rank)

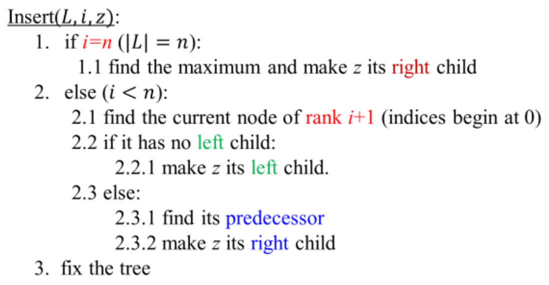
TreeSelect is a function in the AVLTreeList class, witch takes an avl tree and returns the node with the rank that is equal to the second argument , in other words it returns the node that has the value that sits in the index rank of the list.
the function calls the recursive help function TreeSelectRec, witch it's time complexity is O(logn), witch makes it's **Time complexity also O(logn)**

- def TreeSelectRec(self,node, rank)

help recursive function to achieve the TreeSelect function's purpose,
in order to return the node with the value that sits in the index rank,
first the function starts from the root , checks if the number of AVL nodes in the left subtree is bigger or equals the rank, if it is that means that the number of nodes that are smaller that the root is bigger or equals the rank , witch means that the node we are looking for is smaller than the root, so we continue checking in the left subtree with the index rank and the root of this subtree, if the number of the nodes in the left subtree plus 1 equals the ranl, this means that the root is the node we are looking for, and finally if the rank is bigger than than the number of the nodes in the left subtree and it is not the root then the node is differently in the right subtree, so we check in the right subtree with the new rank witch is the old rank minus the number of all the other nodes that we know that are not the right ones,
in the worst case is we travel down the longest path of the tree witch is the height of the tree, **Time Complexity is O(logn)**

- def insert(self, i, val)

as we learned, insert works in two steps: first we insert the node in the index i as usual, second we fix the tree.



```
Insert($L, i, z$):
  1. if $i=n$ ($|L| = n$):
       1.1 find the maximum and make $z$ its right child
  2. else ($i < n$):
       2.1 find the current node of rank $i$+1 (indices begin at 0)
       2.2 if it has no left child:
             2.2.1 make $z$ its left child.
       2.3 else:
             2.3.1 find its predecessor
             2.3.2 make $z$ its right child
  3. fix the tree
```

in the insertion part ,we find the predecessor by using the function TreeSelect in the index i+1.
this part has no loops witch, only when we use the TreeSelect O(logn)
in the fixing part we travel up the node we inserted to the root of the tree,
in each node in this path we update the size,the height, and check if the node answers the AVL tree needs, and if not we fix it by rotating to the left or the right or both.
if the branch factor was not in the range we use the function `rotate`
witch rotates the tree and fixes it and we continue checking the parents,
if the branch factor of the node is in the range the the height changed after insertion this means we still could have a parent node that can have a wrong branch factor so we keep checking, but if the branch

factor of the node was in the range and the height did not change after the insertion, this means that all the parent nodes also have a wanted branch factor , witch means we stop checking.
but even after stopping checking we continue updating the sizes and heights.
in this part the loop has logn iterations in the worst case,
in the loop itself we use rotate that is O(1)
then the time complexity in this part is O(logn)

total Time complexity in the worst case is O(logn+logn)=O(logn)

- def delete(self, i):

same as insert, delete method also requires a o(logn) time complexity,hince both operations may require rebalancing the tree to maintain the balance property in AVL tree.
after deleting a particular node we need to traverse the tree, calculate the balance factor and update the height of the tree,last item(Last) and first item (First)
deleting the node takes o(1) because we need to change a constant number of pointers, but after deleting we need to fix the tree:
We have to traverse the tree, for each node we have to calculate the new balance factor and if it's bigger than 1 or smaller than -1 we have to rotate the node (rotate method takes 0(1)),also we have to update the height and the size respectively.
The worst case is when after deletion of a particular node, the tree becomes unbalanced and rotations are needed to be performed. The time consumed is due to traversal which makes **the time complexity as O(log n).**

- def listToArray(self)

we build an an array and then called the " ListtoArray_rec" recursive method with o(n) time complexity in the worst case so the **time complexity in the worst case is also o(n).**

- def ListtoArray_rec(self, node, result)

This is a recursive helper function that performs an inorder traversal of an AVL tree list,and on each node adds the node value to a given array.
The worst time complexity is o(n),while n is the number of nodes in the AVL tree list. This is because the method performs an inorder traversal of the tree, which visits each node in the tree exactly once, and adds the node value to the given array.

- def sort(self)

In this method, we first create a new array. Then we call the listToArray() method, which returns an array containing the values in the AVL tree in order. This takes O(n) time in the worst case, where n is the number of nodes in the tree.
Next, we sort the array using the merge_sort() method, which takes O(nlogn) time in the worst case.
Then we build a new AVL tree object in O(1) time. We iterate through the sorted array, with a length of n, and insert the array values in order into the tree using the insert() method. This loop takes O(nlogn) time in the worst case, since the insert() method takes O(logn) time in the worst case and we call it n times. In each iteration, we call the insert method with an input of an AVLTreeList with size i. We know that the insert method takes log(n) time in the worst case, where n is the size of the AVLTreeList input. In the iteration i, we will call the insert method with this input(with size i) so for iteration i insert() will take o(log(i)) in the worst case.

thus, this loop takes

$$\sum_{i=0}^{n} log\ (i) \leq \sum_{i=0}^{n} \log\ (n) = n\log(n)$$

anyway we will iterate through an array with length n so we will call insert n times in the worst case its take o(nlog(n)).
in total the worst case takes o(n)+o(nlogn)+o(nlogn)=o(nlogn)

- def merge_sort(self,arr)

the method takes an array and sort it using merge sort algorithm, the time complexity of the merge sort algorithm **in the worst case is O(n * log(n)**), where n is the number of elements in the list.
Merge_sort works by dividing the list into smaller and smaller pieces, then merging those pieces back together in sorted order. The time it takes to divide the list is O(n), and the time it takes to merge the pieces back together is O(nlog(n)),
the worst case for merge sort occurs when the list is already in reverse sorted order. In this case, the algorithm must divide the list into single elements, then repeatedly merge those single elements back together in sorted order.

- def permutation(self)

In this method we create a new array . After that we call the listToArray(self) method which returns an array with AVLTreeList values in order.takes o(n) for AVLTreeList(tree) with size n (n nodes) in the worst case.
then we shuffle the array using array_shuffle(self,arr) method ,which takes o(n) in the worst case.
After shuffling the array, we have to create a new AVLTreeList , then we will insert the n elements of the array to the new AVLTreeList in order using insert() n times .
In each iteration, we call the insert method with an input of an AVLTreeList with size i. We know that the insert method takes log(n) time in the worst case, where n is the size of the AVLTreeList input. In the iteration i, we will call the insert method with this input(with size i) so for iteration i insert() will take o(log(i)) in the worst case.

$$\sum_{i=0}^{n} log\ (i) \leq \sum_{i=0}^{n} \log\ (n) = n\log(n)$$

anyway we will iterate through array with length n so we will call insert n times in the worst case its take o(nlog(n))
,in total the worst case takes o(n)+o(n)+o(nlogn)=o(n)+o(nlogn)=o(nlogn)

- def array_shuffle(self,arr)

This method shuffles array elements randomly, and has a time complexity of O(n), since it iterates through the array once and swaps each element with another element. It is a simple and effective way to shuffle an array.
In the worst case, the time complexity is still O(n). This is because the worst case occurs when the array is already randomly shuffled, in which case the function will simply iterate through the array and perform the swap operation without changing the order of the elements.

- def concat(self, lst)

concat takes a new AVL tree list and contacts it to the end of the current one, in witch all the values in the new list are bigger than the values in the current list, and returns the height difference of the two trees,
in order for the function to work efficiently, the function determines the height of both trees: O(1),
assuming that the current tree is taller (the other case is symmetric) the function removes the rightmost element from the left tree ,by using the function delete,let x be that node : O(logn)
then uses the help function join that works like this:
in the right tree, navigates left until you reach a node whose subtree is at most 1 taller than the left tree,let t be that node: O(log n)
it replaces that node with the node x and makes the left child the left tree the right child the t:O(1)
by construction the new node is AVL balanced and its subtree t 1 taller than t.
and then we rebalance the tree after the function join by using the help function balance_after_join :O(logn)
in total the worst case Time complexity is O(logn)

- def join(self, t1, x, t2)

the join function is explained in the concat function, and as we explained the **Time complexity is O(logn)**

- def balance_after_join(self, node)

the function rebalances the tree from the node in the second argument moving upwards by using the rotate function :O(log n)
also updating the heights and sizes of the nodes in this path.
if the branch factor is in the range and the height didnt change this means that theres no need to continue with the parents because they will be balanced, if the branch factor was in the range but the height changed this means that we need to continue checking the parents because there still could be unbalanced nodes in the way to the root, if the branch factor was'nt in the range we use the rotate function to balance the tree.
and we continue moving upwards to the root updating the sizes and heights.
Therefore, the worst case time complexity of this method is also O(logn)

- def search(self, val)

this method return the place(index) of the node with value equal to val,
The search_rec() method is used to search for a specific value within an AVL tree using a recursive approach. In the worst case, this method will have to visit all the nodes in the tree, which takes O(n) time. When it finds the value, it updates a global object called 'index'. this method returns the index in o(1) times. **Therefore, the worst case time complexity of this method is also O(n)**

- def search_rec(self, node, value)

This is a recursive helper function that performs an inorder traversal of an AVL tree list, searching for a node with value == val. If it finds such a node, it updates the global objects and stops.

If it does not find a node with value == val, it returns None. In the worst case, the function has to visit all the nodes in the tree and check if their values are equal to val, **which takes O(n) time in the worst case.**

- def Successor(self,node)

AVL tree would typically be a function that finds the next largest value in the tree after a given value.the worst case time complexity is O(log n), where n is the number of nodes in the tree. This is because the height of an AVL tree is guaranteed to be no more than O(log n), so any operation that involves traversing the tree will take at most O(log n) time.

- def minValue(self, node):

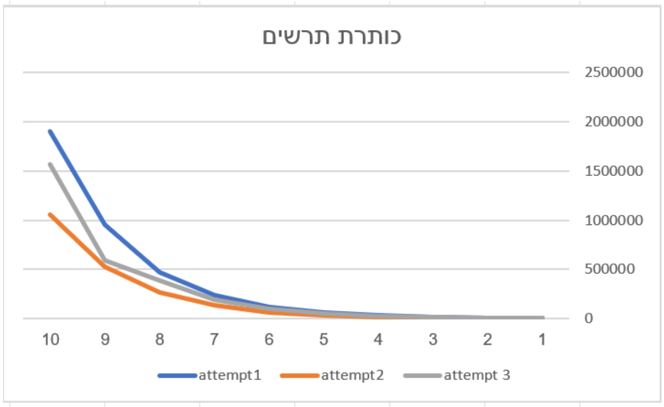
recursive method that find the min node in given tree,**Worst case Time Complexity is O(log(n))** because operation that involves traversing the tree will take at most O(log n) time

חלק ניסויי/ תיאורטי

שאלה 1:א)

attempt 3	attempt2	attempt1	i
3001	2056	3739	1
6050	4078	7487	2
11879	8271	14824	3
23976	16559	29815	4
49873	33030	59866	5
96098	66059	119238	6
192729	132339	238542	7
385984	264734	475334	8
587987	529499	954113	9
1570894	1058592	1905462	10

ב) o(n)



נשים לב כי שבניסיונות שלנו,כי n ומספר פעולות האיזון גודלם באותו קצב. מכאן אפשר לראות כי מספר פעולות האיזון הוא לינארי ב n.

בסדרה של הכנסות מספר ה rotations שלנו יותר ממספר ה rotations בשאר הניסיונות 2 (מחיקות) ו3(הכנסות ומחיקות לסירוגין)

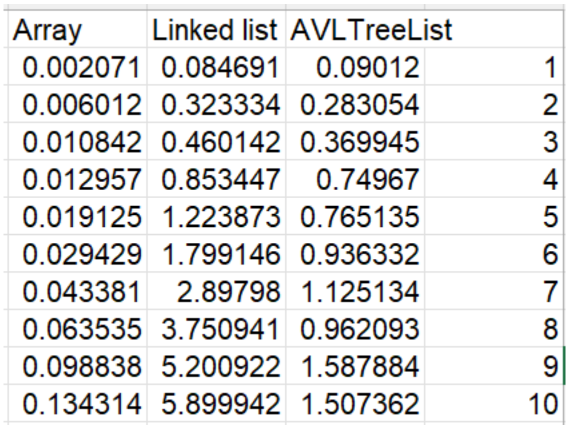
גם בניסיון השלישי נשים לב כי מספר התיקונים הוא קטן יותר מזה של המחיקות וקטן ממספר התיקונים של ההכנסות

שאלה 2:

Insert first:

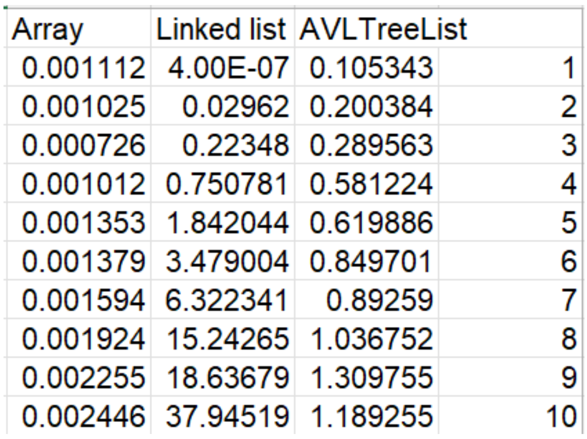
Array	Linked list	AVLTreeList	
0.0002934	5.00E-07	0.1527757	1
0.0010737	0.0006692	0.296174	2
0.0031248	0.0017605	0.2754022	3
0.007728	0.0021217	0.7077999	4
0.0125305	0.0027555	0.6665034	5
0.0211701	0.0039158	0.8710588	6
0.0345045	0.0050543	0.9302246	7
0.0487502	0.0150653	0.9872533	8
0.0704346	0.005484	1.3417734	9
0.1372206	0.0071642	1.2434595	10

Insert randomly:



Array	Linked list	AVLTreeList	
0.002071	0.084691	0.09012	1
0.006012	0.323334	0.283054	2
0.010842	0.460142	0.369945	3
0.012957	0.853447	0.74967	4
0.019125	1.223873	0.765135	5
0.029429	1.799146	0.936332	6
0.043381	2.89798	1.125134	7
0.063535	3.750941	0.962093	8
0.098838	5.200922	1.587884	9
0.134314	5.899942	1.507362	10

Insert last:



Array	Linked list	AVLTreeList	
0.001112	4.00E-07	0.105343	1
0.001025	0.02962	0.200384	2
0.000726	0.22348	0.289563	3
0.001012	0.750781	0.581224	4
0.001353	1.842044	0.619886	5
0.001379	3.479004	0.849701	6
0.001594	6.322341	0.89259	7
0.001924	15.24265	1.036752	8
0.002255	18.63679	1.309755	9
0.002446	37.94519	1.189255	10