AVLTreeList Documentation

Submitted by:

1.
Name: Oded Kesler
ID: 200973212
Username: odedkesler1

2.
Name: Nimrod Cohen
ID: 315427799
Username: nimrodc@mail.tau.ac.il

- Late submission authorized by Dr. Amir Rubinstein.

Introduction

The AVLTreeList object is a data structure that supports various list-like operations, with improved complexity. To the user, it may seem like an ordinary list, but in fact we implemented this data structure using the functionality of a ranked AVL tree, and thus the improved run time for these operations. In this document we will further elaborate on the implementation that we used and how it resulted in the desired complexity.

AVLNode

As in all trees or lists, the data structure is comprised of individual data units that hold the stored values, and support various features for the data structure's functionality.
In this implementation, the AVLNode is the data unit, and it holds several fields:
Size and height of its subtree, pointers to its parent and children, and the value itself.
The node class has standard getters and setters functions, and several other supportive methods: isRealNode, isLeaf, setLeaf, getBalanceFactor and setProperChild (The latter function replaces a right or left son of a parent node, with a new child. If needed, it also adjusts the root of the tree).

AVLTreeList

The tree itself has two fields: a pointer to its root node, and its length.
Other than several getter and setter functions, this class has several methods that use recursive and iterative calls. These functions are in charge of creating the desired runtime complexity and supporting the tree's feature that enables these operations. In the next segment we'll take a deep dive into these functions and show how they operate, under which constraints, and how it results in the desired worst case runtime complexity.

Functions – General Logic & Complexity

search(val):
The function returns the first index of node that contains this value, and if the value does not exist it returns -1. To support this operation in O(n), where n is the length of the tree object, we apply a recursive InOrder search of the tree, using the static function **inOrderSearch(node, value):**
1. Recursively call the left child until we get to a virtual node and return.
2. Check the value of this node, and return its index if it fits.
3. Recursively call the right child.

If a certain node holds the input value, we stop the search and return its index by applying the **TreeRank** function. This function walks down the tree, following the logic that we learned, to get the rank of a node. **Thus, total runtime complexity is: O(n) + O(log(n)) = O(n).**

listToArray()
Same principal here as we need to implement an InOrder tree walk and append each value to the returned list. The difference here, is that we had to implement it by iteration instead of recursion. The iteration uses a stack, that represents visited nodes, and works as follow:
1. Start at the root, if you are a real node append it to the stack and iterate the left child.
2a. When getting to a virtual node, pop the stack and add the current value to the returned list.
2b. Iterate the right child
3. If stack is empty, we visited all nodes, we can break the loop and return the list.

This implementation sums up to O(1) operations on each node in the tree, and thus the total runtime complexity in this function is O(n).

Retrieve(i):
This method returns the value of a node at a certain index.
Since the inner implementation of this list is a ranked AVL tree, it does so by calling the treeSelect function, that finds a node based on its rank in the tree.
TreeSelect walks down the tree, starting at the root, applying the logic learned in class for finding the desired rank. Since this operation is done by a single walk down the height of the tree, the total runtime complexity, in the worst case, for the retrieve function is O(log(n)).

Insert(i, s):

In general, the method deals with three cases:

1. For an empty tree object it inserts a node as the root.
2. For adding an element to the end of the tree object, it finds the max node in the tree, and adds the inserted node as its right child
3. The last case it finds the current node in index i and either inserts it's left child (3.1), or the right child of i's predecessor (3.2).

For cases 2 & 3.1 we find the relevant node using getMaxInsert() or treeSelectInsert(), both functions require a single walk down the height of the tree. Case 3.2 require one more additional walk in the height of the tree, but in total all cases require O(log(n)) runtime.
We implemented getMaxInsert(), treeSelectInsert() and getPrevInsert(), as variations to the classic ranked AVL functions, to adjust heights and sizes of nodes according to our needs. These adjustments take O(1) runtime and do not damage the desired runtime complexity (this is true for all other variations of classic ranked AVL tree functions that we implemented for this class).

After inserting and adjusting the values of the tree and its nodes we are set to start rebalancing the tree, starting at the parent of the inserted node. We do so by calling BalanceTreeFrom(), which in turn calls the rotate() function. BalanceTreeFrom() requires a walk up the height of the tree and the rotation takes O(1) runtime (since several other merthods use these function, we will further elaborate on them at the bottom of this document).

In total, both the insertion and the rebalancing operations take O(log(n)), thus insert() as a whole works in the desired worst case runtime complexity of O(log(n)).

First() & Last()

These methods are more straight forward and takes a path, starting at the root, down the height of the tree to either the most left node, or the rightest node, respectively.
Both methods require O(log(n)) runtime complexity.

Delete(i):
First, the function finds the node to be deleted based on the input index, by applying treeSelectDel(), which is a variation of treeSelect that works in O(log(n) runtime.
Second it operates to properly delete this node according to several possible scenarios:
1. Deleted node is a leaf.
2. Deleted node has no left child.
3. Deleted node has no right child.
4. Deleted node has two children.

For cases 1,2 & 3 the required adjustments take O(1), and in the worst case the function handles with the 4'th case which requires to find the deleted node's successor, by applying getMinDel() (a variation of getMin()) which takes in the worst case a walk down the height of the tree, O(log(n)).

After the deletion and the proper adjustments we are again ready to rebalance the tree, staring at the structural change point, using BalanceTreeFrom() & rotate(), which in total takes another O(log(n))
Thus, deleting and rebalancing takes in total O(log(n)) runtime complexity.

Concat(lst):
The method joins an AVLTreeList object to the end of the self AVLTreeList.
It does so by creating a pointer to the max node of self, which takes O(log(n)), and then deleting it from the tree which is another O(log(n).
These are the preliminary preparations needed before joining the two trees using the deleted node as the join root. The actual join is done in the join function (elaboration on join below) and take another O(log(n) runtime. At the end of the function, we return the height differential of the joined trees.

Split(i):
We first indicate the split point by applying the treeSelect function to find the node at index i.
We implemented a variation of treeSelect, named treeSelectSplit(), where we find the relevant node while preparing the data needed for the upcoming joins.

We took advantage of the fact that finding node i requires going thru all the nodes that would be relevant for the upcoming join operations. Therefore, treeSelectSplit(), while walking down to node i, associate these nodes, and one of their subtrees, to one of two relevant lists:
1. List of nodes and their respective subtree, needed for the join to the larger than i tree.
2. List of nodes and their respective subtree, needed for the join to the smaller then i tree.
This first stage only includes walking down the tree which is upper bounded by O(log(n)).

The second stage of the function iteratively joins the left subtree of node i to its relevant subtrees from its respective list (that was created and returned by treeSelectSplit()), and then the same for the right subtree of node i.
Total number of iterations done for both left & right subtrees, is equivalent to the height of the tree, which again is O(log(n)), while each iteration applies the join function which requires O(log(n)) as well (**join()** logic & complexity explained next).
A naive assumption will access the worst runtime complexity as O(log(n)*log(n)).
However, we know that the tree was previously balanced so it should average on O(log(n)).

join(tree_lst, new_root):
This method also operates on the tree itself, as it joins it to an input tree, thru connecting both trees to the input node, which we'll call the join root (not always the root of the output tree).
Since this operation adds the input tree as the last elements of self,
it is attached as the right child of the join root.
The only adjustments need to be done depends on which tree is higher:

1. If self is higher, walk down the rightest branch of self until we encounter the first node in of tree_lst.height or tree_lst.height -1, this node will be the left child of the join root.
2. If tree_lst is higher, do the same on the most left branch of tree_lst.

For both cases this operation does not take more than the height of h1 or h2, and thus O(log(n)).
After that, all is left is to perform several adjustments to pointers, heights and sizes (all are done in O(1)), to prepare the tree for rebalancing.
Once again, this rebalancing is done by using BalanceTreeFrom() & rotate(), that take additional O(log(n)) runtime. In total, join operates in O(log(n) runtime complexity.

Balancetreefrom() & rotate()

We designed this class in a way that rebalancing is done via external services.
In this way, we keep a modular design, to keep the subtle balancing operation neat and bug free, while maintaining the desired complexity.

Each method that creates a structural change in the tree must rebalance it. Since we have several use cases that require it, all use cases call the rebalancing functions **after** they adjust all the attributes of the nodes and the tree itself. These use cases input the relevant nodes and balance factors to start rebalancing from.

From here on it's quite simple, since Balancetreefrom() goes up to the root and calls the rotate() function if needed. Thus, Balancetreefrom() takes at worst case O(log(n)) steps.
In turn, rotate() uses its input values to determine and execute one of four possible rotations.
Each rotation is limited to a unique set of input values, so the rotate() function itself does not care who called its service, it just perform pointers exchange in O(1) runtime.

In total, these functions operate together in O(log(n)) runtime complexity.

Theoretical Analysis

Question 1

i	insert	delete	both
1	5828	2970	2353
2	11699	6086	4442
3	23778	12031	8812
4	47208	23609	17522
5	94335	47844	35446
6	187892	95726	70182
7	377125	191610	140239
8	755508	384098	281103
9	1509186	768006	564045
10	3020143	1534800	1125102

As seen in the table above, the number of height adjustments and rotations doubles for each increase of i in every one of the experiments, indicating that the average number of such rotations for each number is O(1), meaning it is equal regardless of the size of the tree. Thus the complexity for the total number of rotations in each experiment is O(n), meaning it is directly correlated with the number of such operations being made.

Question 2

i	Average Random	Max Random	Average Last Left	Max Last Left
1	1.363636364	3	1.888888889	12
2	1.875	5	1.727272727	13
3	1.583333333	4	1.9	14
4	0.7692307692	2	1.923076923	16
5	1.846153846	3	2	17
6	1.933333333	5	1.533333333	18
7	1.75	5	1.6875	19
8	1.666666667	4	1.473684211	21
9	1.823529412	7	1.611111111	21
10	1.578947368	4	1.476190476	23

As was shown in class the AVL Tree is balanced before every action. As a result the determining factor in the height difference between two subtrees in a join function during a split operation is simply the number of nodes between the root of the first tree in the join and the root of the second tree in the join. We can see that this question is equivalent to the number of nodes since the path between the root of the tree being split and the lower node in the join last "changed direction". This means the average height difference between two subtrees in a join is equal to the average length of every sub-route of the route between the root of the tree being split and the node being split from where the the sub-route never "changed direction" between left and right.

Let h be the depth of the node being split from. When said node is the last node on the left subtree of the root, it is easy to see that for the left side of the tree there are h-1 joins of height 1, and for the right side there is one join operation of h height. In total, we get the average length of a join to be:
$\frac{(h-1)\cdot 1+h}{h} = 2 - \frac{1}{h}$
We can see that for any h this number is between 2 and 1, and thus the complexity of the join operations is O(1) regardless of the height of the tree or the node being split.

When the node being split from is in a random position, we get the same average complexity. The average length of these same direction routes is equal to the total number of nodes divided by the average number of such sections. Since each pair of such sections is divided by an instance of the route "changing direction", the average number of sections is equal to the average number of direction changes in the route plus 1. Since the tree is balanced, the odds of the route changing direction are equal to the odds of the route keeping the current direction, making the probability distribution for the number of direction changes be equal to Bin(0.5,h). This makes the average number of direction changes 0.5*h, and from this information the average complexity of the join can be calculated:

$$\frac{h}{0.5h+1} = \frac{2\cdot 0.5\cdot h+1-1}{0.5h+1} = 2 - \frac{1}{0.5h+1}$$

As above, we can see that for any h this number is between 2 and 1, and thus the complexity of the join operations is again O(1).

The maximum difference in height between two trees in a join in the case of the maximal node of the left tree of the root is during the single join required to create the right tree. The difference in sized of this join is either h of h-1 where h is the height of the tree.

The results match our expectations exactly. The average height differences are all between 1 and 2 as predicted, and the max height difference for the experiment involving the given node matching the heights of the respective trees.

Question 3

Average number of fixes:

i	AVL Insert First	Unbalanced Insert First	AVL Insert Balanced	Unbalanced Insert Balanced	AVL Insert Random	Unbalanced Insert Random
1	3.964	499.5	0.994	0.994	2.931	2.561
2	3.9805	999.5	0.997	0.997	2.9325	2.4345
3	3.985666667	1499.5	0.9976666667	0.9976666667	2.946333333	2.504333333
4	3.9895	1999.5	0.9985	0.9985	2.946	2.53225
5	3.9912	2499.5	0.999	0.999	2.9304	2.4386
6	3.992333333	2999.5	0.9988333333	0.9988333333	2.897833333	2.480166667
7	3.993428571	3499.5	0.999	0.999	2.940714286	2.502
8	3.994375	3999.5	0.99925	0.99925	2.963625	2.4625
9	3.994777778	4499.5	0.9994444444	0.9994444444	2.961888889	2.421333333
10	3.9953	4999.5	0.9995	0.9995	2.9214	2.496

The above numbers are exactly as expected. The number of fixes needed for an AVL tree insert remains equal regardless of the number of previous inserts, with only the shape of the tree effecting it. As expected, the insert-first order requires the most changes on the AVL, as almost each insertion requires at least one rotation. The balanced tree insertion process requires the least fixes, as no rotations are required.

For the unbalanced tree, it is expected that the average number of fixes is equal to the amount of nodes inserted to the tree divided by 2 when inserted always in the first rank, as each insertion requires increasing the height of all the nodes that were added before it. The amount of fixes for the unbalanced tree populated in the balanced order is also identical to the amount of fixes for the AVLtree populated in the balanced order as they are identical trees.

Average inserted node depth:

i	AVL Insert First	Unbalanced Insert First	AVL Insert Balanced	Unbalanced Insert Balanced	AVL Insert Random	Unbalanced Insert Random

1	7.987	499.5	7.987	7.987	9.019	17.918
2	8.982	999.5	8.982	8.982	10.1505	21.4205
3	9.639	1499.5	9.639	9.639	11.21866667	21.19333333
4	9.97925	1999.5	9.97925	9.97925	11.5485	25.8245
5	10.3644	2499.5	10.3644	10.3644	12.0786	22.3618
6	10.637	2999.5	10.637	10.637	12.2115	26.097
7	10.83171429	3499.5	10.83171429	10.83171429	12.14142857	23.72942857
8	10.97775	3999.5	10.97775	10.97775	12.6155	29.054875
9	11.18122222	4499.5	11.18122222	11.18122222	12.87733333	25.47744444
10	11.3631	4999.5	11.3631	11.3631	12.9163	26.9288

The above numbers are exactly as expected. The average depth of the added nodes for an AVL tree insert remains equal regardless of the order in which they were inserted, with only the number of previous inserts effecting it. As expected, average depth is relative to log(n), with n being the total number of inserts.

For the unbalanced tree, it is expected that the average depth of the node is equal to the amount of nodes inserted to the tree divided by 2 when inserted always in the first rank, as each insertion requires adding a depth layer to the tree. The average depth of the node for the unbalanced tree populated in the balanced order is also identical to the average depth of the node for the AVLtree populated in the balanced order as they are identical trees.