Adaptive AVL Tree Algorithmic Research (Python)

Function	Description	Complexity
AVLNode		
__init__	Initializes a tree node	O(1)
Is_real_node()	Checks if the node is real or virtual; returns a boolean value	O(1)
set_virtual()	Sets a node as virtual by updating the key and height fields	O(1)
Update_max_locally()	Updates the maximum value of the node's subtree locally (within the node only), based on its children's values	O(1)
Update_height_locally()	Updates the node's height field based on the heights of its subtrees. Returns 1 if the height increased, 0 otherwise	O(1)
Update_fields_locally()	Updates the node's height and max fields. Returns 1 if the height increased, 0 otherwise	O(1)
Set_left(node, update height)	Updates the node's left child by setting the parent field and defining the left child. Also updates fields if a height update is needed. Returns 1 if the height increased, 0 otherwise	O(1)
Set_right(node, update_height)	Updates the node's right child by setting the parent field and defining the right child. Also updates fields if a height update is needed. Returns 1 if the height increased, 0 otherwise	O(1)
Successor()	Returns the next node in the tree according to in-order traversal	$O(\log n)$
_replace_child(current, replacement)	Replaces the current node's right or left child with the given replacement node	O(1)
Remove_child(node)	Defines a virtual child for self, on the right or left as requested	O(1)
_replace_with_succ_or_pre(node,typ)	Replaces the node with its successor or predecessor as needed	O(1)
Replace_with_successor()	Finds the node's successor and replaces the node with it	O(logn)
_roll_right()	Performs a right rotation	O(1)
_roll_left()	Performs a left rotation	O(1)
_double_roll_left()	Performs a right then left rotation (Double Rotation)	O(1)

_double_roll_right()	Performs a left then right rotation (Double Rotation)	O(1)
Rebalance()	Rebalances the tree and returns the number of balancing operations performed. Updates relevant affected fields during the process	O(log n)
Predecessor()	Returns a pointer to the node's predecessor	O(log n)
Replace_with_predecessor()	Replaces the item with its predecessor	O(log n)
In_order_keys()	Performs a recursive traversal of the tree and returns an array containing the tree keys in in-order configuration	O(n)
AVLTREE		
__init__	Class constructor	O(1)
_update_root	Updates the tree root	O(logn)
_find_left_join_point(height)	Returns a node in the tree for performing a left join with a tree of height height (where all values are smaller than self)	O(logn)
_find_right_join_point()	Returns a node in the tree for performing a right join with a tree of height height (where all values are larger than self)	O(logn)
_left_join(separator, smaller, bigger)	Performs a join where the separator node separates the trees (keys: smaller < separator < bigger), and bigger.height > smaller.height	O(logn)
_right_join(separator, smaller, bigger)	Performs a join where the separator node separates the trees (keys: smaller < separator < bigger), and bigger.height < smaller.height	O(logn)
_middle_join(separator, smaller, bigger)	Performs a join where the separator node separates the trees (keys: smaller < separator < bigger), and the tree heights are equal	O(1)
_search_path(key)	Searches the tree for a node with the key key. Returns a tuple containing the relevant node and the path length to it	O(logn)
search(k)	Searches for an item with key k. Returns a tuple (x, e), where x is the node pointer (or None) and e is the search path length (edges + 1)	O(logn)
finger_search(k)	Searches for an item with key k starting from the maximum node. Returns a tuple (x, e), where x is the node pointer (or None) and e is the search path length (edges + 1)	O(log n)

insert(k, v)	Inserts an item with value v and key k. Returns a tuple (x, e, h): x is the new node, e is the path length, and h is the number of promotions (height changes)	O(logn)
finger_insert(k, v)	Inserts an item with value v and key k starting from the maximum node. Returns a tuple (x, e, h): x is the new node, eis the path length, and h is the number of promotions.	O(log n)
delete(x)	Deletes node x from the tree given a pointer	O(logn)
join(t, k, v)	Receives another tree t (where all keys are smaller/larger than current) and a key k. Merges t and the new item (k, v)into the current tree. After the operation, tree t is unusable and cannot be accessed	O(logn)
split(x)	Receives a pointer to node x. Splits the tree into two trees (t1, t2): t1 has keys smaller than x, t2 has keys larger than x. After the operation, the current tree and x are unusable	O(logn)
avl_to_array()	Returns a sorted array (Python list sorted by keys) of the items, where each item is a tuple (key, value)	O(n)
max_node()	Returns a pointer to the node with the maximum key in the tree	O(1)
size()	Returns the number of items in the tree. The size field is maintained only for the tree, not per node. After split, the size may be invalid	O(1)
get_root()	Returns a pointer to the tree root	O(1)

N (Number of elements in array)	Balancing cost (Array with random adjacent inversions - Average)	Balancing cost (Random array - Average)	Balancing cost (Reverse sorted array)	Balancing cost (Sorted array)
600	1173	1055.2	1186	1186
1200	2354.9	2135.8	2385	2385
2400	4720.7	4265.1	4784	4784
4800	9467.6	8552.6	9583	9583
9600	18949.3	17089.8	19182	19182
19200	37933.3	34252.8	38381	38381
38400	75893.4	68484.9	76780	76780
76800	151798.6	137067.7	153579	153579
153600	303736.5	274236.2	307178	307178
307200	607234.8	548465.9	614377	614377

In AVL trees, the amortized balancing cost of a single insertion is O(1). For n operations, the total cost sums to O(n). This analysis indeed aligns with the table values.

The addition of rotations to the count does not change the asymptotic complexity, as the number of rotations is bounded by a constant. During an insertion, a rotation (single or double) is performed in O(n), and for n insertions, we obtain O(n).

$$Total\ Cost = (Promotions\ Cost) + (Rotations\ Cost) = O(n) + O(n) = O(n)$$

N (Number of elements in array)	Number of inversions in an array with random adjacent inversions (average)	Number of inversions in a random array (average)	Number of inversions in a reverse-sorted array	Number of inversions in a sorted array
600	303.2	89464.4	179700	0
1200	602.1	360051.8	719400	0
2400	1192.8	1438734.6	2878800	0
4800	2402.7	5765355.9	11517600	0
9600	4782.2	23031280.2	46075200	0

N (Number of elements in array)	Search cost in array with random adjacent inversions (average)	Search cost in random array (average)	Search cost in reverse-sorted array	Search cost in sorted array
600	1094	8221	9012	599
1200	2171.9	18829.1	20420	1199
2400	4331.9	42627.2	45636	2399
4800	8686	94932.6	100868	4799
9600	17371.2	208829.6	220932	9599
19200	34757.8	454312.6	480260	19199
38400	69576.9	991648.4	1037316	38399
76800	139165.5	2149643.5	2228228	76799
153600	278386.6	4602795.1	4763652	153599
307200	556684.9	9819423.3	10141700	307199

It holds that $\mathrm{I} = \sum \mathrm{d_i}$ because for every i, the number of existing inversions in the tree is $\mathrm{d_i}$, since, by definition, these are all the elements already found in the tree and are larger than the value of i (meaning they satisfy the condition of an inversion). Therefore, for the entire tree, the sum of inversions is the sum of $\mathrm{d_i}$ for all i.

Since we are implementing finger_insert, the traversal in the tree starts from the maximum and ends upon reaching a node with a key smaller than the key we wish to insert. It is known that there exist d_i elements in the tree larger than the i-th element, and therefore, we will ascend a total of $\log(d_i)$ nodes in the traversal. Overall, we climbed $\log(d_i)$ times in the traversal. Therefore, in order for this to be valid for every i , it holds: $O(\max(1, logd))$. We will convert this to a simple expression and get: $\log(d_i + 2)$. In total, it holds: $\sum_{i=1}^{n} \log(d_i + 2) = \log(\prod_{i=1}^{n} d_i + 2)$.
Total: $O(\log(\prod_{i=1}^{n} d_i + 2))$.

Using the Inequality of Arithmetic and Geometric Means (AM-GM), it holds: $\sqrt[n]{\prod(d_i + 2)} \leq \frac{\sum(d_i+2)}{n}$, therefore, $(\prod(d_i + 2))^n \leq \frac{\sum(d_i+2)}{n}$. It holds:
$\frac{\sum(d_i+2)}{n} = \frac{\sum d_i + \sum 2}{n} = \frac{I+2n}{n} = \frac{I}{n} + 2$.
Substituting back into the formula, we get: $\sqrt[n]{\prod(d_i + 2)} \leq \frac{I}{n} + 2$ and therefore. $\prod(d_i + 2) \leq \left(\frac{I}{n} + 2\right)^n$.
Substitute the relevant expression into the bound we received:

$$O(\log\left(\prod_{i=1}^{n} d_i + 2)\right) \leq O\left(\log\left(\frac{\mathrm{I}}{\mathrm{n}}+2\right)^{n}\right) = O(n \cdot log\left(\frac{I}{n}+2\right))$$

This is the dominant cost asymptotically because the rest of the operations in the insertion action are performed in O(1) and are therefore negligible.

Experimental Validation (Comparison with Metrics):

Sorted Array:

At each step, we insert an element that is larger than the previous maximum.
Since the search starts from the maximum and the new element is the new maximum, the distance between them is minimal and the number of inversions is I=0.
Substituting into the bound: $n \cdot \log(0+2) = nlog2 = O(n)$.
Experimental Results: In the table, it can be seen that the search cost grows **linearly** relative to n.
Conclusion: The results match the obtained bound.

Reverse Sorted Array:

The worst case in terms of inversions. **Every** element appearing earlier in the array is necessarily larger than all elements appearing after it.
Meaning $I = (n-1) + (n-2) + \cdots + 1 = \sum_{k=1}^{n-1} k$, and according to the arithmetic series sum: $I = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$.
Asymptotically $I \approx \frac{n^2}{2}$ meaning $\frac{I}{n} \approx \frac{n}{2}$.
Substituting into the bound: $n \cdot \log\left(\frac{n}{2}+2\right) \approx nlogn = O(nlog\, n)$.
Experimental Results: The growth is exponential/super linear and consistent with $nlog\, n$.
Conclusion: The results match the obtained bound.

Randomly Ordered Array:

The number of possible pairs of indices (i, j) such that $i < j$ is $\binom{n}{2} = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$. For every pair, there is a probability of $\frac{1}{2}$ to be an inversion, therefore $I \approx \frac{n^2}{4}$.
Here too, the ratio $\frac{I}{n}$ is linear in n, so we obtain $O(nlog\, n)$ similarly to the reverse sorted array.
Experimental Results: The growth is exponential/super linear and consistent with $nlog\, n$.
Conclusion: The results match the obtained bound.

Array with Random Adjacent Inversions:

Here it holds that $I \approx \frac{n}{2}$ clearly.

Substituting into the bound: $n \cdot \log\left(\frac{\frac{n}{2}}{n} + 2\right) = n \cdot \log(2.5) = O(n)$.

Experimental Results: In the table, it can be seen that the search cost grows **linearly** relative to n.

Conclusion: The results match the obtained bound.