Yoav Ben-Guigui-206570707
Gal Shani-322292210

Runtime analysis, Classes and function documentation:

Class: AVLNode

Description:

Represents a node in an AVL tree. Each node has the following fields:

key - sorted type, allows to determine the location of the node.

value - the data we will keep in this node.

left - the root of the left subtree - a node with a smaller key. If the node is a virtual node we initialize its left son to be None. Else, a virtual node.

right - the root of the right subtree - a node with a bigger key. If the node is a virtual node we initialize its right son to be None. Else, a virtual node.

parent - If the node is a virtual node we initialize its parent to be None. Else, a virtual node.

height - the maximum size of the node's subtrees- represents the longest
path from the node to one of the leaves in its subtree. If the node is a virtual node we initialize its height to be -1. Else, 0.

** A virtual node is a node with key and value None and height -1.

__init__() | $O(1)$

Is_real_node() | $O(1)$ returns true if the node is real (not virtual), false otherwise.

<u>Class: AVLTree</u>

Description:

Represents an AVL tree. Each tree has the following fields: root, size.

__init__() | $O(1)$ Set the root of the new tree to virtual node, and the size to 0.

search(k) :

Searches for a node in the AVL tree starting from the root. It takes a key as input and traverses the tree to locate the node with the matching key. If the node is found, it returns the node and the number of edges traversed. If the key is not found, it returns None along with the number of edges traversed.

Time Complexity: O(logn)
In an AVL tree, the height is maintained as O(logn).
Hence, searching for a node involves traversing at most O(logn) edges.

finger_search(k) :
 Searches for a node in the binary search tree, starting from the maximum node instead of the root. It first traverses upward to find a suitable starting point, then descends toward the target key. If the key is found, it returns the node and the number of edges traversed; otherwise, it returns None and the traversal count.
Time Complexity: Worst Case: O(h)— Proportional to the height of the tree O(logn)

rotate_right(node) :
The function performs a right rotation around a given node in the AVL tree. This operation is used to rebalance the tree when a left-heavy imbalance occurs (Balance Factor > 1). The left child of the node becomes its parent, and the node becomes the right child of its former left child.
Time Complexity: O(1)
The rotation involves updating a constant number of pointers, so it takes constant time

rotate_left(node) :
The function performs a left rotation around a given node in the AVL tree. This operation is used to rebalance the tree when a right-heavy imbalance occurs (Balance Factor < -1). The right child of the node becomes its parent, and the node becomes the left child of its former right child.
Time Complexity: O(1)
The rotation involves updating a constant number of pointers, so it takes constant time.

get_balance(node) :
The function calculates the balance factor of a given node in the AVL tree. The balance factor is defined as the difference between the height of the left subtree and the height of the right subtree. This value helps determine whether the node (and by extension, the tree) is balanced or if rebalancing is required.
Time Complexity: O(1)
The function directly accesses the heights of the left and right child nodes, which are stored as part of the node's attributes. This operation is constant in time.

update_height(node) :
Updates the height of a given node based on the heights of its left and right children. The new height is set to 1 + max(left_height, right_height).
Time Complexity: O(1)
The function performs a constant amount of work by accessing the heights of the left and right children and calculating the new height.

Rebalance(node) :
The function restores the balance of the AVL tree starting from a specific node and propagates upward towards the root. It calculates the balance factor of each node and performs rotations (rotate_left or rotate_right) as necessary to maintain the AVL tree's balance. It also updates the height of nodes during the process.
Time Complexity: O(logn)
In the worst case, the function traverses from the insertion/deletion point to the root, which involves at most O(logn) nodes (the height of the AVL tree).
Each node is rebalanced in O(1), so the overall complexity remains O(logn).

Insert(k,v) :
Adds a new node with a specified key (k) and value (v) to the AVL tree. It first locates the appropriate position for the new node (maintaining the binary search tree property) and then performs the insertion. After the insertion, it calls
the rebalance function to ensure the AVL tree remains balanced. It returns a tuple containing the newly added node, the number of edges traversed, and the number of "promote" operations performed during rebalancing.
Time Complexity: O(logn)
Finding the position for insertion takes O(logn) due to the height of the AVL tree.
The rebalance function, called after insertion, also takes O(logn).

finger_insert(k,v) :

Adds a new node with a given key (k) and value (v) into the AVL tree, starting the search from the maximum node instead of the root. This can be more efficient when the key to be inserted is close to the maximum. After insertion, the tree is rebalanced.

Time Complexity: (O(logn))

Best Case: O(1) — If the key is greater than the maximum key, the node is inserted as the right child of the maximum node.

Worst Case: O(logn) — If the key is not close to the maximum and a standard traversal is needed.

Rebalancing after insertion takes O(logn).

replace_node(old_node, new_node) :

The function replaces an existing node (old_node) in the AVL tree with another node (new_node). It updates the parent of the old_node to point to the new_node and sets the new_node's parent to the old_node's parent. This is typically used during deletion when a node needs to be replaced.

Time Complexity: O(1)

The function performs a constant number of pointer updates.

delete(x) :

The function removes a node x from the AVL tree while maintaining its balanced structure. Depending on the node to be deleted:

1. If it has no children (leaf), it is removed directly.
2. If it has one child, the child replaces the node.
3. If it has two children, the node's key and value are replaced by those of its in-order successor, and the successor is removed. After deletion, the tree is rebalanced using the _rebalance function.

Time Complexity: O(logn)
Finding the node to delete takes O(logn), as the AVL tree is height-balanced.
Rebalancing the tree after deletion also takes O(logn).

Join(t,k,v) :
The function merges two AVL trees (self and t) into a single AVL tree using a given key (k) and value (v) as the "bridge" between the two trees. The trees must satisfy the precondition that all keys in one tree are either smaller or larger than the bridge key. The function inserts the bridge node, connects the trees to the left and right of it, and rebalances the resulting tree.
Time Complexity: O(log (n)+log (m))
n is the size of the larger tree, and m is the size of the smaller tree.
The function traverses the height of the larger tree to find the joining position, which is O(logmax(n,m)).
Rebalancing involves at most O(log (n)+log (m)) work.

join_trees(key, value, left_tree, right_tree) :
The function combines two AVL trees (left_tree and right_tree) into a single AVL tree using a given key and value as the "bridge." The new node becomes the root, with left_tree as its left subtree and right_tree as its right subtree. The function rebalances the resulting tree to maintain AVL properties.
Time Complexity: O(log(n) + log(m))
Rebalancing the tree takes time proportional to the height of the larger tree, which is
O(log max (n\m))

split(x) :
The function divides the AVL tree into two separate AVL trees based on a given node x. All keys smaller than x.key are placed in one tree (t1), and all keys greater than x.key are placed in another tree (t2). This involves disconnecting subtrees, propagating upward from the split point, and rebalancing the resulting trees.
Time Complexity: O(logn)
Traversing from the split point (x) to the root involves at most O(logn) steps, as the tree height is O(logn).
The rebalancing at each level also takes O(1), so the total work is O(logn).

avl_to_array() :
The function converts the AVL tree into a sorted list of tuples, where each tuple contains a key and its corresponding value. It does this by traversing the tree in in-order (left subtree → current node → right subtree), ensuring the keys are added to the list in ascending order.
Time Complexity: O(n)
The function visits each node exactly once during the in-order traversal, where n is the total number of nodes in the tree.

max_node() :
The function finds and returns the node with the maximum key in the AVL tree. It starts from the root and traverses right until it reaches the rightmost node, which has the largest key.
*Time Complexity: O(logn)
In an AVL tree, the height is O(logn), so traversing down the rightmost path takes at most O(logn) steps.

Size() :

Returns the total number of nodes in the AVL tree. It directly accesses
the tree_size attribute, which is updated during insertions and deletions.
Time Complexity: O(1)
The function simply retrieves and returns the value of the tree_size attribute,
requiring constant time.

get_root() :

Returns the root node of the AVL tree. If the tree is empty, it returns None.
Time Complexity: O(1)
The function directly accesses and returns the root node, requiring constant time.

Successor(node) :

The function finds the in-order successor of a given node in the AVL tree. The
successor is the node with the smallest key greater than the key of the given node.

If the node has a right child, the successor is the leftmost node in the right subtree.

If the node doesn't have a right child, the function traverses up the tree to find the
first ancestor where the node is in the left subtree.

Time Complexity: O(logn)
In the worst case, the function may traverse the height of the tree (either
descending the right subtree or ascending to the root).

חלק 2

:1

מספר סידורי i	עלות איזון במערך ממוין	עלות איזון במערך ממוין- הפוך	עלות איזון במערך מסודר אקראי	עלות איזון במערך עם היפוכים סמוכים אקראיים
1	430	430	385.25	424.05
2	873	873	780.95	863.35
3	1,760	1,760	1,576.1	1,739.3
4	3,535	3,535	3,167.95	3,480.85
5	7,086	7,086	6,313.3	7,012.2
6	14,189	14,189	12,666.75	14,032.15
7	28,396	28,396	25,322.6	28,053.35
8	56,811	56,811	50,690.15	56,129.15
9	113,642	113,642	101,453.5	112,346.25
10	227,305	227,305	202,834.15	224,708.25

In the table above, we present the number of promotions for each type of list based on its size and structure. As demonstrated in class, during a sequence of insertions, the amortized cost of rebalancing is O(1). This is validated by our experiment, which shows that when n is doubled, the rebalancing cost also doubles. This confirms that the total cost of rebalancing is O(n) by definition.
We also observe that rotations tend to occur when the tree forms a one-sided curve. For instance, when keys are inserted in sorted order, each new key is larger than the previous one, making it the new maximum and placing it as the right child of the current max node.
As a result, the tree develops a rightward curve, requiring rotations after a few insertions.
Similarly, for a sequence of keys in reverse order, the tree develops a leftward curve and behaves symmetrically, requiring rotations to the left.
Consequently, ordered key sequences (ascending or descending) trigger more rotations than the other two types of sequences.
However, as shown in class, rotations have constant time complexity and do not asymptotically impact the overall time complexity.
The number of rotations is always less than or equal to the number of promotions, as each rebalancing involves at least one promotion. Since the number of rotations is bounded by the number of rebalancing operations, the total cost of rebalancing, including rotations, is O(n), aligning with the number of promotions.

:2

מספר סידורי i	מספר היפוכים במערך ממוין	מספר היפוכים במערך ממוין-הפוך	מספר היפוכים במערך מסודר אקראית	מספר היפוכים במערך עם היפוכים סמוכים אקראית
1	0	24,531	11,438.8	109.95
2	0	98,346	51,842	216.6
3	0	393,828	191,016.4	444.3
4	0	1,576,200	765,052.5	868.05
5	0	6,306,576	3,172,891.3	1,777.95

:3

מספר סידורי i	עלות חיפוש במערך ממוין	עלות חיפוש במערך ממוין-הפוך	עלות חיפוש במערך מסודר אקראית	עלות חיפוש במערך עם היפוכים סמוכים אקראיים
1	221	2,694	2,191.45	288.75
2	443	6,272	5,337.35	580.65
3	887	14,316	12,202.05	1,160.55
4	1,775	32,180	28,042.25	2,324.65
5	3,551	71,460	62,714.40	4,656.40
6	7,103	157,124	142,299.45	9,305.7
7	14,207	342,660	313,083.90	18,641.75
8	28,415	742,148	679,015.95	37,260.6
9	56,831	1,597,956	1,472,679.80	74,576.25
10	113,663	3,423,236	3,183,929.40	149,139.6

:4

(i)- We observe that $I = \sum d_i$, as each inversion is counted exactly once in the total sum of d_i,. Specifically, an inversion is accounted for when we encounter the smaller element in the inverted pair. For every d_i, we consider the elements preceding the current number that have larger values, thereby capturing all inversion pairs where the current element is the smaller one. Thus, each inversion pair is included exactly once, resulting in the total sum of d_i, being equal to I.

(ii)- As shown in class, finger-searching for a key in an AVL tree has a time complexity of O(h), where hh is the height of the subtree identified at the start of the algorithm. In the i^{th} search operation, this subtree contains at least d_i elements, as the search begins from the maximum key. Therefore, the height h of this subtree is at least log (d_i).
In the best case, the i^{th} key is larger than the current maximum key and is added as the right child in constant time. Thus, the time complexity can be expressed as O(log(max(d_i,1))) = O(log(d_i + 2).
Consequently, the total time complexity for all search operations is:
O($\sum$log(d_i + 2)) = O(log ($\Pi_{i=1}^{n}(d_i + 2)$)) as required.

(iii)- We aim to find an upper bound for the expression as a function of n and I. Using logarithmic rules and the inequality of arithmetic and geometric means, we can rewrite:

$\log(\Pi_{i=1}^{n}(d_i + 2)) \leq n * \log(\sum(d_i + 2)) / n) = n * \log((I + 2n)/n)$
This simplifies to:

$$n * \log\left(\left(\frac{I}{n}\right) + 2\right) = n * \max\left(1, log\left(\frac{I}{n}\right)\right)$$

This provides an asymptotic upper bound for the search cost. However, to consider the worst-case scenario, we assume all d_i values are distributed to maximize the cost:

1. For half the insertions, $d_i = 0$
2. For the other half, $d_i = \frac{I}{0.5n}$, spreading the search cost evenly.

For the second half of the sequence, the search cost becomes:

$$\frac{n}{2} * \log\left(\frac{2I}{n}\right)$$

By combining these results, the asymptotic worst-case cost simplifies to:

$$O\left(n * \max\left(1, \log\left(\frac{I}{n}\right)\right)\right) = O\left(n * \log\left(\frac{I}{n} + 2\right)\right)$$

Thus, the upper bound is proven.

(iv)- As calculated in the previous section, the overall cost of searching within an array (whether it is sorted, reversed, shuffled, or partially shuffled) can be expressed as: $n * \log\left(\left(\frac{I}{n}\right) + 2\right)$
when we represent n as: $n = 111 * 2^i$
We note that I represents the number of inversions, and as demonstrated in question 2, the inversions depend on the type of array.

Size(n)	sorted	Sorted(provided, ratio)	Reversed	Reversed (provided, ratio)	Shuffled	Shuffled (provided, ratio)	partially shuffled	partially shuffled (provided, ratio)
222	222	(221,0.99)	1512.65	(2694,1.78)	1274.76	(11,438.8, 1.71)	292.86	(288.75 ,0.98)
444	444	(443,0.99)	3465.03	(6,272,1.81)	3060.01	(5,337.35,1.744)	583.81	(580.65 ,0.99)
888	888	(887,0.99)	7813.76	(14,316,1.83)	6892	(12,202.05,1.77)	1174.04	(1,160.55,0.98)
1776	1776	(1775,0.99)	17399.20	(32,180,1.84)	15553.25	(28,042.25,1.8)	2336.20	(2,324.65,0.99)

3552	3552	(3551,0.99)	38346.09	(71,460,1.86)	34831.54	(62,714.40,1.8)	4696.61	(4,656.4 ,0.99)

From the analysis of the data presented in the table, we can draw important insights regarding the relationship between the theoretical cost of searching within an array and the experimental results obtained. For every pair of i and an array type (e.g., sorted, reversed, shuffled, or partially shuffled), two values were measured: the theoretically calculated cost (Upper-bounded total cost) and the experimental value, which reflects the actual cost observed during the experiment. In the table.

One of the key observations derived from the table is that the ratio between the theoretically calculated cost and the experimentally measured value remains constant across all combinations of i and array types. This result demonstrates a stable relationship between the theoretical model and the measured reality. The consistency of this ratio is not coincidental—it reinforces the assumption that the total cost of performing a search in an array, under various scenarios, aligns with the calculated time complexity presented earlier.

In essence, this finding validates the reliability of the theoretical model, as it accurately predicts the experimental results without significant deviations or mismatches. Consequently, we can conclude that the calculated time complexity serves as an accurate and relevant tool for understanding the cost of search operations in different types of arrays.