AVLTree Function Documentation

AVLNode Class - *A class representing a node in an AVL tree.*

init(key, value, left=None, right=None, parent=None)

Initializes an AVL tree node with a given key, value, and optional references to its left child, right child, and parent.

Time Complexity: O(1), as it directly assigns values to the node attributes.
Returns: None – Initializes the node in place.
Helper Functions: None.

is_real_node()

Checks if the node is real (not External).

Time Complexity: O(1), as it directly evaluates the node's key.
Returns: bool – True if the node is real, False if it is a virtual node.
Helper Functions: None

EXTERNAL_LEAF is a **singleton** AVLNode representing a virtual (non-real) node with key, value, and height set to None and -1. It simplifies AVL tree operations by serving as a placeholder for missing children, ensuring structural consistency and avoiding null checks.

AVLTree Class - A class implementing an AVL tree.

init()

Initializes an empty AVL tree with a None root, size set to 0, and a max pointer to track the maximum node initialized to None.
Time Complexity: O(1), as it directly assigns initial values.
Returns: None – Initializes the tree in place.
Helper Functions: None

search(key)

Searches for a node with the given key starting at the root.

Time Complexity: $O(\log log\, n)$. Traverse from root to node, max length is max height of tree, as the tree is balanced so maximum log n.

Returns: (AVLNode, int) – the wanted node, and the path (number of edges) from root to node.

Helper Functions: _search_from(node,key)

finger_search(key)

Searches for a node starting at the maximum node.

Time Complexity: $O(log\, n)$ - First, it traverses upward until the key falls within the subtree of the current node (worst-case we get to the root), then traverses downward like a standard search, resulting in logarithmic complexity.

Returns: (AVLNode, int) The found node, or None if not found, and the path length (number of edges) from the starting node (maximum node) to the target node.

Helper Functions: _finger_track_up(key), _search_from(node, key)

insert(key, val)

Inserts a new node with the given key and value starting at the root.

Time Complexity: $O(log\ n\ )$ – The search for the insertion position takes logarithmic time due to the balanced structure of the AVL tree, and rebalancing operations (rotations and height updates) are also performed in $O(\log log\ n\ )$

Returns: (AVLNode, int, int) - The newly inserted node, the path length (number of edges) from the root to the inserted node before rebalancing, and the number of promote cases during AVL rebalancing.

Helper Functions: _search_from(node, key), _insert_to_parent(parent, key, val)

finger_insert(key, val)

Inserts a new node with the given key and value, starting at the maximum node for optimized insertion near the maximum.

Time Complexity: $O(log\ n\ )$ The traversal upwards to find the common ancestor and then downward to the insertion point both take logarithmic time. Rebalancing operations (rotations and height updates) also take $O(\log log\ n\ )$

Returns: (AVLNode, int, int) – The newly inserted node, the path length (number of edges) from the starting node (maximum node) to the inserted node before rebalancing, and the number of promote cases during AVL rebalancing.

Helper Functions: _finger_track_up(key), _search_from(node, key), _insert_to_parent(parent, key, val)

delete(node)

Deletes the given node from the AVL tree and rebalances it if necessary.

Time Complexity: $O(log\ n\ )$- Adjusting pointers and rebalancing operations (rotations and height updates) each take logarithmic time due to the balanced structure of the AVL tree.

Returns: None - The node is removed, and the tree is updated in place.

Helper Functions: _balance_factor(node), _predecessor(node), _find_max(node), _update_height(node), _rebalance(node)

join(tree2, key, val)

Joins the current AVL tree with another AVL tree (tree2) and a new node with the given key and value.

Time Complexity:$O(h_1 - h_2)$ where h1 and h2 are the heights of the taller and shorter trees, respectively. Since we traverse along the taller one until we get to height of shorter, and then rebalance from there up. (Height is bound by log n in each tree).

Returns: None – the self tree becomes the merged single AVL tree.

Helper Functions: _join_with_bigger_subtree(bigger_tree, smaller_tree, key, val, is_left_bigger)

split(node)

Splits the AVL tree into two separate trees at the given node, where one tree contains keys smaller than the node and the other contains keys larger than the node.

Notice this function violates the maintenance of tree size filed (for efficiency reasons).

Time Complexity: $O(log\, n\,)$ – we implement the split with consecutive join operation on the subtrees along the path from node to root, see the analysis for time comp of this algo in the lecture's AVL slide. Also, at last we find the new max node of the returned trees in traversal along the right extreme edges, also logarithmic complex.

Returns: (AVLTree, AVLTree)- Two AVL trees, where the first contains keys smaller than the given node and the second contains keys larger than the given node.

Helper Functions: join(tree2, key, val)

avl_to_array()

Converts the AVL tree into a sorted list of key-value pairs using an in-order traversal.

Time Complexity: $O(n)$ - where n is the number of nodes in the tree. Each node is visited exactly once during the traversal.

Returns: list - A sorted list of tuples (key, value) representing the elements in the AVL tree.

Helper Functions: None

max_node()

Returns the node with the maximum key.

Time Complexity: $O(1)$ - since the maximum node is tracked and updated during insertions and deletions.

Returns: AVLNode the node with the maximum key, or None if the tree is empty.

Helper Functions: None

size()

Returns the number of real nodes in the AVL tree. **Notice** after performing split operation this is no longer maintained.

Time Complexity: $O(1)$ – as the size is maintained and updated during insertions and deletions.

Returns: int - The number of real nodes in the tree.

Helper Functions: None

get_root()

Returns the root node of the tree.

Time Complexity: $O(1)$ - as it simply returns a reference to the root node that is always kept in the structure.

Returns: AVLNode the root node of the tree, or None if the tree is empty.

Helper Functions: None

Helper function inside AVLTree class

_finger_track_up(key)

Finds the first common ancestor of the maximum node and the given key by traversing upward in the tree from max.

Time Complexity:$O(h)$, where h is the height of the minimum subtree containing key (and max). In the worst case, it traverses to the root so max logarithmic complexity.
Returns: (AVLNode, int) – The ancestor node, and the number of edges traversed upward.
Used in: finger_search(key), finger_insert(key, val)
Helper Functions: None

_insert_to_parent(parent, key, val)
Inserts a new node with the given key and value as a child of the specified parent node and performs rebalancing if necessary.
Time Complexity: $O(logn)$ - Insertion takes constant time, while rebalancing (promotes and rotations) may traverse up to the root, which is logarithmic.
Returns: (AVLNode, int) – The newly inserted node and the number of promote cases during AVL rebalancing.
Used in: insert(key, val), finger_insert(key, val)
Helper Functions: _balance_factor(node), _update_height(node), _rebalance(node)

_join_with_bigger_subtree(bigger_tree, smaller_tree, key, val, is_left_bigger)
Joins two AVL trees when one tree is taller than the other, uses same logic as shown in lecture (see AVL slide).
$O(h_1 - h_2)$ where h1 and h2 are the heights of the taller and shorter trees, respectively. Since we traverse along the taller one until we get to height of shorter, and then rebalance from there up. (Height is bound by log n in each tree).

Returns: None – the self tree becomes the merged single AVL tree.

Helper Functions: _balance_factor(node), _update_height(node), _rebalance(node)

Independent Helper Functions

_find_max(node)

Finds and returns the node with the maximum key in the given subtree by traversing to the rightmost node.
Time Complexity: $O(h)$ where h is the height of the subtree. In a balanced AVL tree, this is $O(logn)$
Returns: AVLNode – The node with the maximum key, or None if the subtree is empty.
Used in: delete(node), split(node)
Helper Functions: None

_balance_factor(node)

Calculates and returns the balance factor of a given node, defined as the difference between the heights of its left and right subtrees.

Time Complexity: O(1) - as it simply accesses pre-computed heights of the child nodes.
Returns: int – The balance factor of the node.
Used in: delete(node), _insert_to_parent(parent, key, val), _rebalance(node)
Helper Functions: None

_update_height(node)

Updates the height of the given node based on the heights of its left and right children.
Time Complexity: O(1) - as it directly calculates the height using the child nodes' heights.
Returns: None – Updates the height of the node in place.
Used in: _insert_to_parent(parent, key, val), _rebalance(node), _join_with_bigger_subtree(bigger_tree, smaller_tree, key, val, is_left_bigger)
Helper Functions: None

_rebalance(node)

Restores the AVL tree balance at the given node by performing rotations if the node is unbalanced.
Time Complexity: O(1), as it performs a constant number of operations (rotations and height updates) based on the balance factor.
Returns: AVLNode – The new root of the subtree after rebalancing.
Used in: _insert_to_parent(parent, key, val), _join_with_bigger_subtree(bigger_tree, smaller_tree, key, val, is_left_bigger), delete(node)
Helper Functions: _balance_factor(node), _rotate_left(z), _rotate_right(z), _update_height(node)

_rotate_left(z)

Performs a left rotation on the given subtree rooted at node z to restore AVL balance.
Time Complexity: O(1), as it involves a fixed number of pointer updates and height recalculations.
Returns: AVLNode – The new root of the rotated subtree.
Used in: _rebalance(node)
Helper Functions: _update_height(node)

_rotate_right(z)

Performs a right rotation on the given subtree rooted at node z to restore AVL balance.
Time Complexity: O(1), as it involves a fixed number of pointer updates and height recalculations.
Returns: AVLNode – The new root of the rotated subtree.
Used in: _rebalance(node)
Helper Functions: _update_height(node)

_search_from(node, key)

Searches for a node with the given key starting at the specified node and returns the result along with traversal details.
Time Complexity: $O(h)$ where h is the height of the subtree. In a balanced AVL tree, this is $O(logn)$.
Returns: (AVLNode, int, AVLNode) – The found node (or None if not found), the number of edges traversed, and the parent node of the search position.
Used in: search(key), finger_search(key), insert(key, val), finger_insert(key, val)
Helper Functions: None

_predecessor(node)

Finds and returns the predecessor of the given node, which is the node with the largest key smaller than the given node's key.
Time Complexity:$O(logn)$ - It either traverses the left subtree to find the maximum or ascends the tree, both of which are logarithmic in height.
Returns: AVLNode – The predecessor node, or None if no predecessor exists.
Used in: delete(node)
Helper Functions: _find_max(node)

Question 1

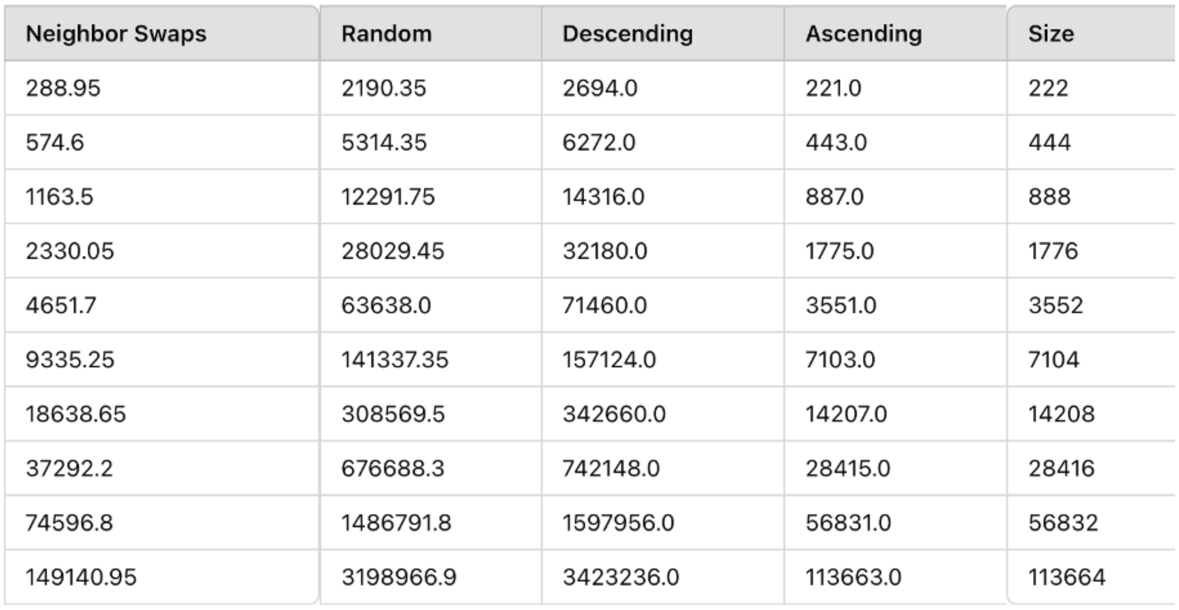
מספר סידורי i	עלות איזון במערך ממוין	עלות איזון במערך ממוין הפוך	עלות איזון במערך מסודר אקראית	עלות איזון במערך עם היפוכים סמוכים אקראיים
i=1 222	430	430	385.25	399.45
i=2 444	873	873	782.45	813.55
i=3 888	1760	1760	1572.45	1639.0
i=4 1776	3535	3535	3148.45	3294.25
i=5 3552	7086	7086	6304.05	6595.6
i=6 7104	14189	14189	12646.15	13196.15
i=7 14208	28396	28396	25326.35	26425.7
i=8 28416	56811	56811	50690.3	52845.9
i=9 56832	113642	113642	101462.0	105730.85
i=10 113664	227305	227305	202819.6	211486.75

כפי שהראינו בהרצאה, עלות האיזון בעץ AVL ללא גלגולים היא זמן קבוע באמורטייזד היא $O(1)$.
הוכחנו זאת בשיטת הפוטנציאל, תוך הגדרת הפוטנציאל עבוד כמות הצמתים המאוזנים "לחלוטין" (1,1). ראה בהרחבה במצגת ההרצאה עמוד 33.
נשים לב כי גם אם נחשב את העלות כולל הגלגולים, הזמן יישאר קבוע כפי שהוכחנו בהרצאה שמספר הגלגולים הוא 1 (בשונה ממחיקה ששם המספר הגלגולים יכול להיות כגובה העץ) וזה אכן תואם את הנתונים שקיבלנו. שעלות האיזון היא לינארית לגודל הקלט.

טבלה 2

מספר סידורי i	מספר היפוכים במערך ממוין	מספר היפוכים במערך ממוין הפוך	מספר היפוכים במערך מסודר אקראית	מספר היפוכים במערך עם היפוכים סמוכים אקראיים
i=1 222	0	24531	12368	108
i=2 444	0	98346	48906	222
i=3 888	0	393828	197805	445
i=4 1776	0	1576200	791773	888
i=5 3552	0	6306576	3149405	1776

טבלה 3



Neighbor Swaps	Random	Descending	Ascending	Size
288.95	2190.35	2694.0	221.0	222
574.6	5314.35	6272.0	443.0	444
1163.5	12291.75	14316.0	887.0	888
2330.05	28029.45	32180.0	1775.0	1776
4651.7	63638.0	71460.0	3551.0	3552
9335.25	141337.35	157124.0	7103.0	7104
18638.65	308569.5	342660.0	14207.0	14208
37292.2	676688.3	742148.0	28415.0	28416
74596.8	1486791.8	1597956.0	56831.0	56832
149140.95	3198966.9	3423236.0	113663.0	113664

4) I) (הוכחה) נשים לב כי לכל $1 \le i < j \le n$ $d_i \cap d_j = \emptyset$.

זה כיוון ש-d_i מייצג את ההשוואות של האיבר ה-i עם אינדקסים שקטנים ממנו
ולכן בוודאי שלא מייצג השוואה עם האיבר במקום ה-j שגדול ממנו.)

נשים לב כי כל איבר בסכום $\sum d_i$ מייצג קיום של איבר באינדקס k שקטן
מאיבר שנמצא באינדקס l ומתקיים $l > k$. וזו אכן ההגדרה של היפוך. ולכן, יש כיוון הכלה
זה כיווני. כיוון שאיבר ששייך לסכום מייצג איבר שנמצא באינדקס מסוים, שקטן מאיבר שנמצא
באינדקס קטן ממש, ולכן מייצג היפוך ולכן שייך ל-I.

וגם, איבר ששייך ל-I מייצג היפוך, כלומר, מייצג קיום של איברים באינדקס i, j עבורן
מתקיים $i < j \wedge A[i] > A[j]$ ולכן שייך ל-d_j. ולכן שייך לסכום.

לסיכום, נקבל כי הקבוצות שוות. כנדרש.

ראשית נבין, שנצטרך לעלות עד תת עץ מינימלי שמכיל את d_i האיברים הקטנים מהנוכחי מהעץ.

בהכרח תת עץ שאליו נגיע, מכיל d_i איברים, אם פחות, לא נוכל להגיע שמכילו נכון. יהיה בתת עץ ששייך לאיברים שגדולים מאחד מהאיברים ב d_i.

כמו כן תת עץ אליו נגיע, הוא המינימלי שמכיל d_i איברים, אחרת היינו עוצרים בתת עץ המינימלי ולא ממשיכים לעלות.

עפ"י תרגיל בית 3 שאלה 4 סעיף ג', תת עץ מינימלי המכיל לפחות d_i איברים מקסימליים מקיים שגובהו $h \le (\log_\phi d_i) + 2$

(כי תת עץ ימני שלו יש פחות מ d_i איברים אז גובהו הכי הרבה $\log_\phi d_i$, ועוד 2 בגלל תת עץ שמאלי שלו שיכול להיות יותר גדול ב 1. אז סה"כ $h \le 1 + \log_\phi d_i + 1$)

כלומר בחיפוש נעלה ונרד לאורך עץ בגובה $\log_\phi d_i + 2$

סה"כ כל פעולת הכנסה עלות החיפוש היא לכל היותר:

$$O\left(\max(\underset{\uparrow}{1}, \log d_i)\right) = O\left(\log_i (d_i+2)\right)$$

עבור $d_i = 1, 0$

ואם נסכום את כל ההכנסות מחוקי לוגים נקבל

$$O\left(\log\left(\prod_{i=1}^{n} (d_i+2)\right)\right)$$

$\boxed{III}$ מסעיף קודם סך עלויות החיפוש - $O(\log \prod_{i=1}^{n}(d_i+2))$.
מאי שוויון הממוצעים (לוג ⊛ פונ' מונוטונית עולה)

$$\prod_{i=1}^{n}(d_i+2) \le \left(\frac{(d_1+2)+ \dots +(d_n+2)}{n}\right)^n = \left(\frac{I+2n}{n}\right)^n$$

$$⊛ \Downarrow$$

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

כלומר, בסה"כ קבלנו כי עלות החיפוש היא $O(n\log(\frac{I}{n}+2))$.
וכן גם זוהי העלות הדומיננטית מבחינה אסימפטוטית. כיוון שהעלות של האלגוריתם הינה $O(n)$ כפי שהראינו בהרצאה. (כי כל הכנסה עלות האלגוריתם הינה $O(n)$).

ד.4
אכן ניתן לראות שעבור מערך ממוין שעבורו I=0 מקבלים זמן ליניארי לגודל הקלט.
ועבור מערך ממוין בסדר יורד מקבלים את הזמן הגבוה ביותר כי במקרה זה מספר ההיפוכים הוא מקסימלי ובכל מקרה לא גבוה יותר מהביטוי שאמרנו.