קובץ תיעוד – פרויקט מעשי 1 מבני נתונים

שם משתמש 1: lb3

שם משתמש 2: royfainaru

תעודת זהות 1: 206913519

תעודת זהות 2 : 207002403

שם מלא 1: ליאור בן אברהם

שם מלא 2: רועי פיינרו

**חלק מעשי**

תיאור המחלקות שמומשו ותיעוד הסיבוכיות:

**class AVLNode:**

\* calculateSize(self):

Time complexity: O(1)

This is because the method has a precondition that the size attributes of the left and right children

are already updated.

\* calculateHeight(self):

Time complexity: O(1)

Same as calculateSize()

\* getValueByRank(rank):

Time complexity: O(log(n))

This is because the method must travel downwards in the tree in order to find the node with the given value.

\* insert(self, rank, node):

Time complexity: O(logn)

This is because the method must perform a binary search to the rank,

which takes O(logn).

The method also performs a constant number of rotations, which takes O(1).

\* insertLast(self, node):

Time complexity: O(logn)

This is because the method must travel downwards in the tree to reach the last node.

\* delete(self, rank):

Time complexity: O(logn)

This is because the method must perform a logarithmic number of rotations in the worst case.

\* rebalance(self):

Time complexity: O(1), since the method performs a constant number of operations.

\* min(self)/max(self):

Time complexity: O(logn)

This is because the method must travel downwards in the tree to reach the first/last node.

\* successor(self)/predecessor(self)/findSuccessorValue(self):

Time complexity: O(logn)

In the worst case, this method will traverse the height of the tree in order to find the successor.

\* listToArrayHelper(self):

Time complexity: O(n).

This method will traverse every node in the tree in order to add its value to the list.

**class AVLTreeList:**

\* retrieve(self, index):

Time complexity: O(log(n)).

In the worst case, this method will traverse the logarithm of the number

of nodes in the tree in order to find the value at the given index.

\* insert(self, index, value)/append(self, value):

Time complexity: O(log(n)).

In the worst case, this method will traverse

the logarithm of the number of nodes in the tree in order to

find the correct position for the new value.

A constant number of rotations are performed which takes O(1).

\* delete(self, index):

Time complexity: O(log(n))

In the worst case, this method will traverse

the logarithm of the number of nodes in the tree in order to find the correct node to delete,

and in the worst case, it also performs rotations to maintain the balance of the AVL tree.

The number of rotations performed depends on the balance factor of the nodes and can be at most O(log n).

\* first(self)/last(self):

Time complexity: O(log(n)), because it performs a search in the tree to find the first/last element, which is the node with rank 1/'size' in the tree.

\* listToArray(self):

Time complexity: O(n).

Because it must traverse through all the nodes of the tree in order to create the resulting list object.

\* regular\_insert(self, node):

Time complexity: O(log(n))

Because it excerts a method of insertion which makes a binary search

to the correct position to insert the given node.

\* \_sort\_rec(self, sorted\_tree):

The time complexity of this method is O(n log(n))

This is because the method iterates through every node in the AVL tree once in order to add it to the sorted tree list, and each insertion has a time complexity of O(log(n)). Therefore, the overall time complexity is O(n \* log(n)).

\* sort(self):

The time complexity of this method is O(n log(n))

This is because the method calls the `\_sort\_rec` method, which has a time complexity of O(n log(n)). The rest of the method has a o(n log(n)) time complexity.

\* permutation(self):

The time complexity of this method is O(n log(n))

This is because the method calls the `retrieve` and `delete` methods, which both have a time complexity of O(log(n)).

\* concat(self, lst):

The time complexity of this method is O(k log(n)), where k is the number of nodes in the tree list 'lst'. This is because the method calls the `append` method, which has a time complexity of O(log(n)), for every inesrtion done from 'lst'.

\* search(self, val):

The time complexity of this method is O(n).

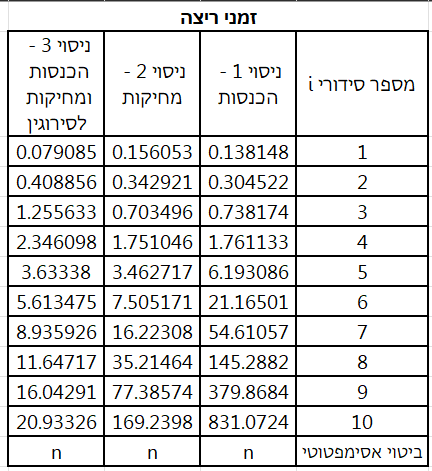
This is because the method iterates through every node in the AVL tree once in order to find the node with the target value.

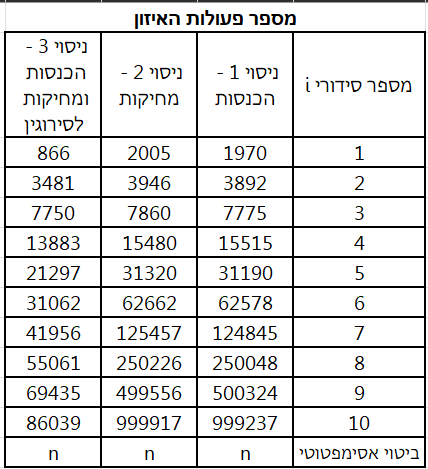
\* \_\_repr\_\_(self)/\_\_iter\_\_(self):

Time complexity is O(n), since it relies on a list converstion to return.

**חלק ניסויי / תיאורטי**

**שאלה 1**

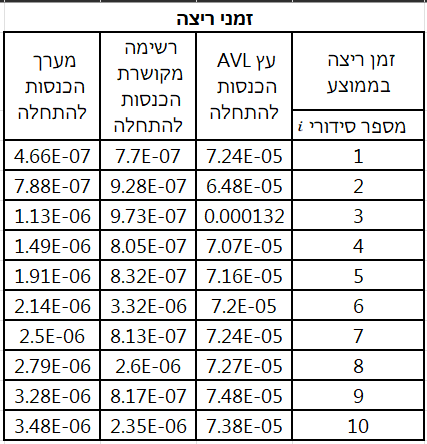




\*סעיף 2 נמצא בתחתית הטבלה

**שאלה 2**

1. הכנסות להתחלה

****

הציפייה שלנו מהתוצאות -

המבנה הכי מהיר לסדרת פעולות זו: רשימה מקושרת

הסבר

המבנה הכי איטי לסדרת פעולות זו: AVLTree

הסבר

לפרט:

* רשימה מקושרת מכניסה מההתחלה בo(1), רק סידור מצביעים
* בשביל עץ AVL זה אחד מהמקרים הגרועים בו נוצרים כמה שיותר מצבים של חוסר איזון ולכן יש צורך במספר מקסימלי של רוטציות במהלך הסדרה
* תואם את הציפיות

1. סדרה אקראית

**A screenshot of a computer

Description automatically generated with low confidence**

הציפייה שלנו מהתוצאות -

המבנה הכי מהיר לסדרת פעולות זו: AVLTree

הסבר

המבנה הכי איטי לסדרת פעולות זו: רשימה מקושרת

הסבר

לפרט:

* רשימה מקושרת מכניסה מההתחלה בo(1), רק סידור מצביעים
* בשביל עץ AVL זהו מקרה די אופטימלי של הכנסות לכל תת עץ בסבירות שווה ולכן האיזון ייפגע כמה שפחות ולפי תנאי AVL, צפוי מספר מינימלי של רוטציות ולכן פחות זמן לרוץ
* לא תואם את הציפיות המערך היה מהיר יותר - צריך להסביר למה

1. הכנסות לסוף

**Table

Description automatically generated**

הציפייה שלנו מהתוצאות -

המבנה הכי מהיר לסדרת פעולות זו: מערך

הסבר

המבנה הכי איטי לסדרת פעולות זו: רשימה מקושרת

הסבר

לפרט:

* רשימה מקושרת צריכה לרוץ על כולם (n)
* ועץ רץ ומגלגל חזרה לאיזון בlogn לכן הוא עדיין צפוי להיות יותר מהיר מרשימה מקושרת לסדרה זו
* תואם את הציפיות