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:

|  |  |
| --- | --- |
|  | **Time Complexity** |
| **empty()** | O(1) |
| **retrieve(i)** | O(log(n)) |
| **insert(I, s)** | O(log(n)) |
| **delete(i)** | O(log(n)) |
| **first()** | O(log(n)) |
| **last()** | O(log(n)) |
| **listToArray()** | O(n) |
| **length()** | O(1) |
| **permutation()** | O(n\*log(n)) |
| **sort()** | O(n\*log(n)) |
| **concat(lst)** | O(k\*log(n))\*\* |
| **search(val)** | O(n) |

1. List ADT implemented with AVL Tree:

\*\* k = lst.size

Full time complexity description of the methods incl. private methods:

1. class AVLNode:
   1. 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.

* 1. calculateHeight(self):

Time complexity: O(1)

Same as calculateSize()

* 1. 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.

* 1. 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).

* 1. insertLast(self, node):

Time complexity: O(logn)

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

* 1. delete(self, rank):

Time complexity: O(logn)

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

* 1. rebalance(self):

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

* 1. 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.

* 1. 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.

* 1. listToArrayHelper(self):

Time complexity: O(n).

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

1. class AVLTreeList:
   1. 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.

* 1. 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).

* 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).

* 1. 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.

* 1. 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.

* 1. 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.

* 1. \_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)).

* 1. 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.

* 1. 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)).

* 1. 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'.

* 1. search(self, val):

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

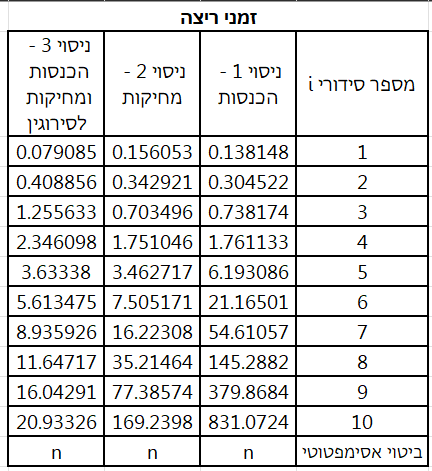
This is because the method iterates through every node in the AVL tree once in order to find the node with the target value.

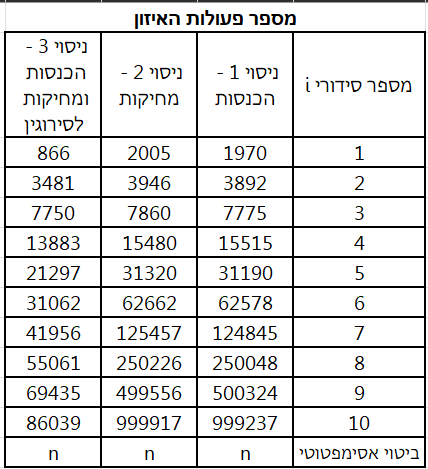
* 1. \_\_repr\_\_(self)/\_\_iter\_\_(self):

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

**חלק ניסויי / תיאורטי**

**שאלה 1**

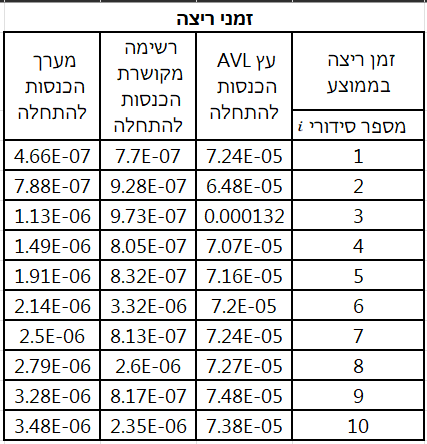




\*סעיף 2 נמצא בתחתית הטבלה

**שאלה 2**

1. הכנסות להתחלה

****

תוצאות משוערות:

|  |  |  |
| --- | --- | --- |
| AVL | Linked List | Python list |
| סיבוכיות הזמן הדרושה לביצוע פעולה זו היא **O(log(n)**, כמו כל הכנסה. זה כתוצאה מכך שהכנסות לעץ מתבצעות ע"י "תלייה" על עלי העץ, מה שדורש טיול מהשורש עד לעלה הרלוונטי. עבור עץ בינארי פשוט, מדובר בO(n) במקרה הגרוע, אך בעץ AVL מובטח לנו שגובה העץ הוא O(log(n)). | הכנסה בראש הרשימה דורשת מספר פעולות קבוע עבור רשימה מקושרת, ללא תלות בגודל הרשימה – **O(1)**. לפיכך, נצפה לביצועים הטובים ביותר ברשימה המקושרת בסעיף זה. | במערך פשוט הכנסה בראש הרשימה היא המקרה הגרוע ביותר של הכנסה כללית. נדרשות **O(n)** פעולות בעת ביצוע ההכנסה הזאת בכדי לעדכן את מיקום כל שאר איברי הרשימה. |

הציפייה שלנו מהתוצאות -

1. המבנה הכי מהיר לסדרת פעולות זו: רשימה מקושרת, בשל סיבוכיות זמן קבועה למול סיבוכיות זמן התלויה בגודל הקלט.
2. המבנה הכי איטי לסדרת פעולות זו: AVL!

בכדי לדבר על הביצוע האיטי ביותר עלינו לקחת בחשבון שסיבוכיות זמן אופטימלית איננה מניבה בהכרח ביצועים אופטימליים עבור קלטים בגדלים קטנים מספיק. לפיכך, אנחנו משערים שהביצועים של AVL יהיו גרועים יותר מהביצועים של list, **על אף שמדובר במבנה נתונים בסיבוכיות זמן טובה יותר יותר, בשל גודל הקלט**. מבנה הנתונים הדיפולטי של פייתון הוא מאוד משופצר בביצועיו במובני המקדמים של פונקציית הסיבוכיות. **עבור קלטים גדולים מספיק, המבנה האיטי ביותר יהיה list של פייתון.**

על פי הטבלה בראש העמוד, ניתן לראות כי אכן, גם עבור הקלט הגדול ביותר, הרשימה הממומשת בפייתון מהירה יותר מעץ AVL. בכדי לאשש את השערתינו, הרצנו השוואה ראש בראש בין list לבין AVL עבור הקלטים משאלה 1, ואכן ראינו פריצה בביצועים של AVL עבור גודל קלט .

1. סדרה אקראית

**A screenshot of a computer

Description automatically generated with low confidence**

תוצאות משוערות:

|  |  |  |
| --- | --- | --- |
| AVL | Linked List | Python list |
| סיבוכיות הזמן הדרושה לביצוע פעולה זו היא **O(log(n)**, כמו כל הכנסה. זה כתוצאה מכך שהכנסות לעץ מתבצעות ע"י "תלייה" על עלי העץ, מה שדורש טיול מהשורש עד לעלה הרלוונטי. עבור עץ בינארי פשוט, מדובר בO(n) במקרה הגרוע, אך בעץ AVL מובטח לנו שגובה העץ הוא O(log(n)). | הכנסה במיקום אקראי ברשימה תדרוש **O(n)** פעולות במקרה הממוצע, וO(n) פעולות במקרה הגרוע. זה מכיוון שאנחנו נדרשים לטייל עד הצומת הרלוונטית להכנסה, ובמקרה הממוצע מדובר בn/2 = O(n) פעולות. לפיכך, ניתן לראות כי מבנה AVL יתעלה על LL במקרה הממוצע ובמקרה הגרוע. | במערך פשוט הכנסה אקראית תדרוש **O(n)** פעולות במקרה הממוצע, וO(n) פעולות במקרה הגרוע (הכנסה בראש הרשימה). זה מכיוון שבמקרה הממוצע נידרש לעדכן את האינדקס של n/2 = O(n) איברים אחרי האיבר המוכנס. לפיכך, עץ AVL גובר גם על המערך במקרה הממוצע ובמקרה הגרוע. (במונחים אימפטוטיים). |

הציפייה שלנו מהתוצאות-

1. המבנה הכי מהיר לסדרת פעולות זו: list של פייתון

בדומה להסבר מהסעיף הקודם, אנחנו משערים שהקלטים הספציפיים שנבחרו לבעיה הם קטנים מספיק בכדי שlist יגבר על ביצועיה של רשימת AVL, **על אף שהסיבוכיות של AVL טובה יותר עבור הכנסות אקראיות**. אם בהכנסות בראש הרשימה, מה שמהווה את התרחיש הגרוע מבחינת list, שיערנו שlist יגבר על AVL, אז על אחת כמה וכמה כאשר מדובר בפעולה דומה בתרחיש טוב יותר עבור list ובתרחיש זהה עבור AVL.

1. המבנה הכי איטי לסדרת פעולות זו: סיבוכיות הזמן של LinkedList ושל list של פייתון היא אותה הסיבכויות, אך מאותן הסיבות של אופטימיזציה של הקוד, אנחנו מאמינים שLinkedList פחות אופטימלית ולכן תנפיק ביצועים גרועים יותר משל list.

תוצאות בפועל:

על פי הטבלה בראש העמוד, ניתן לראות כי אכן, גם עבור הקלט הגדול ביותר, הרשימה הממומשת בפייתון מהירה יותר מעץ AVL. בכדי לאשש את השערתינו, הרצנו השוואה ראש בראש בין list לבין AVL עבור הקלטים משאלה 1, ואכן ראינו כי עבור קלטים עץ AVL עוקף את list בביצועים (אפע"פ שזה לא היה קל כמו בתרחיש מס' 1).

1. הכנסות לסוף

**Table

Description automatically generated**

|  |  |  |
| --- | --- | --- |
| AVL | Linked List | Python list |
| סיבוכיות הזמן הדרושה לביצוע פעולה זו היא **O(log(n)**, כמו כל הכנסה. זה כתוצאה מכך שהכנסות לעץ מתבצעות ע"י "תלייה" על עלי העץ, מה שדורש טיול מהשורש עד לעלה הרלוונטי. עבור עץ בינארי פשוט, מדובר בO(n) במקרה הגרוע, אך בעץ AVL מובטח לנו שגובה העץ הוא O(log(n)). | הכנסה במיקום האחרון מהווה את המקרה הגרוע ברשימה מקושרת. המקרה הגרוע הוא המקרה הממוצע, ולכן יידרשו **O(n)** פעולות להשלמת הפרוצדורה. | במערך פשוט הכנסה בסוף הרשימה היא המקרה הטוב, ודורשת מספר פעולות קבוע – **O(1)**. |

הציפייה שלנו מהתוצאות -

1. המבנה הכי מהיר לסדרת פעולות זו: מערך.

סיבוכיות זמן O(1) תהיה המהירה ביותר מבין מבני הנתונים, כאשר יתר מבני הנתונים תלויים בגודל הרשימה לסיבוכיות.

1. המבנה הכי איטי לסדרת פעולות זו: רשימה מקושרת.

כאן ניתן להזניח הפרשים משמעותיים באופטימיזציה של הקוד ביחס לגודל הקלט ולהניח בביטחון שסיבוכיות הזמן האופטימלית תניב את זמן הריצה האופטימלי.