Assignment 1 - Data Structures

Written by:

username1 - orerez
id1 - 318970951
name1 - Or Erez
username2 - noyagam
id2 - 314623372
name2 - Noy Agam

AVLTreeList - Documentation

Class 1 - AVLNode

Description **-** A class representing a single node in the list. Each node has a value, which can be of any object type, a parent node, a right son node, and a left son node. Several more class members are defined as well, as is described below:

Class members:

- **value**: The value of the AVLNode. Can be of any type.
- **left**: A reference to another AVLNode which is the left son of the current node from a binary tree point of view. Can be a virtual node.
- **right**: A reference to another AVLNode which is the right son of the current node from a binary tree point of view. Can be a virtual node.
- **parent**: A reference to another AVLNode which is the parent of the current node from a binary tree point of view. Can be None if current node is root.
- **height**: An integer number representing the length of the longest path down the tree from the current node to a leaf. Will be -1 if current node is a virtual node, 0 if is leaf.
- **size**: An integer number representing the amount of node in the current node's rooted tree (including itself).

The AVLNode also supports these methods:

Public Methods:

- **getLeft()**: A basic get function which uses the associated class member to return a reference to another AVLNode which is the left son of the current node. Works in $\Theta(1)$ time complexity.
- **getRight()**: A basic get function which uses the associated class member to return a reference to another AVLNode which is the right son of the current node. Works in $\Theta(1)$ time complexity.
- **getParent()**: A basic get function which uses the associated class member to return a reference to another AVLNode which is the parent of the current node. Works in $\Theta(1)$ time complexity.
- **getValue():** A basic get function which uses the associated class member to return a reference to the node's value object. Works in $\Theta(1)$ time complexity.
- **getHeight()**: A basic get function which uses the associated class member to return an integer representing the node's height. Works in $\Theta(1)$ time complexity.
- **setLeft(node)**: A basic set function which sets the associated class member "left" to another AVLNode which will be the new left son of the current node. Works in $\Theta(1)$ time complexity.
- **setRight(node)**: A basic set function which sets the associated class member "right" to another AVLNode which will be the new left son of the current node. Works in $\Theta(1)$ time complexity.
- **setParent(node):** A basic set function which sets the associated class member "parent" to another AVLNode which will be the new parent node of the current node. Works in $\Theta(1)$ time complexity.
- s**etValue(value):** A basic set function which sets the associated class member "value" to an object which will be the new value of the current node. Works in $\Theta(1)$ time complexity.
- **setHeight(h):** A basic set function which sets the associated class member height to an integer "h" which will be the new height of the current node. Works in $\Theta(1)$ time complexity.
- **isRealNode():** A function which returns true if current node is not virtual, otherwise returns False. Works in $\Theta(1)$ time complexity.

Private Methods:

- **initVirtualValues():** A function used to convert a node to a virtual one. Sets the members appropriately. Works in $\Theta(1)$ time complexity.

- **makeNodeLeaf**(): A function that creates 2 virtual nodes using **constructor** and **initVirtualValues**, then attaches them to current node as right and left sons. Works in $\Theta(1)$ time complexity.

- **isLeaf**(): A function which checks if node is a leaf. If so, returns True and otherwise false. Works in $\Theta(1)$ time complexity.

- **getSize**():A basic get function which uses the associated class member to return an integer representing the node's size. Works in $\Theta(1)$ time complexity.

- **setSize**(size):A basic set function which sets the associated class member height to an integer "size" which will be the new size of the current node. Works in $\Theta(1)$ time complexity.

- **computeHeight**(): A functions that computes the height of the node by using the heights of it's sons. Works in $\Theta(1)$ time complexity.

- **updateHeight**(): An encapsulating function which uses **computeHeight** to compute the new height of a node and **setHeight** to update the corresponding member to that value. Works in $\Theta(1)$ time complexity.

- **updateSize**(): A function which computes the new size of a node and uses **setSize** to update the corresponding member to that value. Works in $\Theta(1)$ time complexity.

- **updateMySizeHeight**(): A function which updates the size and height of the node using **updateHeight** and **updateSize**. Works in $\Theta(1)$ time complexity.
- **computeBF():** A functions that computes the Balance Factor of the node by using the heights of it's sons. Works in $\Theta(1)$ time complexity.

Class 2 – AVLTreeList

Description**:**
A class representing a AVL tree that represents a list. Each tree has size, root, first item and last item.

Class members:

- Size: An integer number representing the number of nodes in the tree.
- Root: a reference to the root of the tree.

- First item: a reference to the first item of the tree (list), from a binary tree point of view.
- Last item: a reference to the last item of the tree (list), from a binary tree point of view.

The AVLTreeList also supports these methods:

Public Methods:

- **Insert(i, val):** A function which gets an index “i” in the list and a value, and adds a node to the tree in this index. Returns the number of rotations which had to be done to rebalance the AVLTree.
  At first, the insert function creates the new node and then uses the **makeNodeLeaf** function ($O(1)$), which turns the node into a leaf (because when we insert a new node it will always be a leaf in the new tree).
  the function checks if the new node is the first node in the list. If it does, the insert function uses the **insertFirstNode** function ($O(1)$) which changes the references of the members. Nothing else should be done in this case, so the insert function returns 0.
  If it doesn’t, the insert function work like we saw in class. The function uses the **computeHeight** function ($O(1)$) to use later the **needBalance** function ($O(1)$). After the insertion the size of the tree increases by 1**.** we also use the **handleSizesHeights** function (O(log(n)). If the need balance function returns true, then we use the **balanceTree** function (O(log(n)) to retain the AVLTree form.
  Therefore, total worst-case complexity of insert is O(log(n).

- **delete(i)**: A function which gets an index “i” in the list and deletes the node pointed by it from the list. Returns the number of rotations which had to be done to rebalance the AVLTree in the process if delete was successful, otherwise returns -1. Uses function **retrieveNode** (O(log(n))) to get the correct node (if exists) in list, and the functions **successor** or **predecessor** (O(log(n))) to update class members if the node to delete is either first or last in list. Then, uses **isRealNode** ($\Theta(1)$) to determine whether node has 2 children or less, and calls **deleteNodeHasTwoChildren** (O(log(n))) or **deleteLessThenTwo** ($\Theta(1)$) accordingly. Also uses **handleSizesHeights** (O(log(n))) twice and **balanceTree** (O(log(n))) to retain the AVLTree form. All mentioned complexities are worst-case, and the calls are successive. All other code is in $\Theta(1)$ time. Therefore, total worst-case complexity of delete shall be: O(log(n)) + O(log(n)) + O(1) + O(log(n)) + O(1) + 2_O(log(n)) + O(log(n)) = 6 * O(log(n)) + 3 * O(1) = O(log(n).

- **listToArray()**: Returns a python list containing a string representation of the values in the nodes of AVLTreeList. Goes over every node and finds its **successor**, therefore requires n operations. Works in complexity O(n).

- **sort()**: A function which returns a new AVLTreeList, sorted by it's values. Uses a **listToArray** (O(n)**)** to create an array from AVLTreeList. Then uses **randQuicksort** to sort the array (O(nlog(n)) on average, O(n^2) in worst case). At last, makes n insertions to a new AVLTreeList in (O(nlog(n))) and returns it. Time complexity: O(nlog(n)) on average (expected value), O(n^2) in worst case.

- **permutation():** A function which returns a new AVLTreeList which is a permutation of the original. Uses a **listToArray** (O(n)**)** to create an array from AVLTreeList. Then uses n random **insertions** and **deletions** to fill a new array (O(nlog(n))). At last, reverses the original AVLTreeList to it's original state (O(nlog(n))) and returns the new AVLTreeList. Overall, O(nlog(n)) time complexity.

- **Empty():** A function which returns true if the list is empty, otherwise returns false .Works in ($\Theta(1)$).

- **Retrieve(i):** returns the value of the node in index i. If there is no index i in the list, the function returns None. At first the function checks if index i exists, if it does then retrieve function uses **retrieve_node** function (O(log(n)). Retrieve_node function works like the select function we saw in class.

- **First():** A function which returns the value of the first item of the list. Works in ($\Theta(1)$) because we have the first_item member.

- **Last():** A function which returns the value of the last item of the list. Works in ($\Theta(1)$) because we have the last_item member.

- **Length():** A function which returns the number of items in the list (the size of the tree. ($\Theta(1)$))
- **Search(val):** returns the first index of the list where the value appears. If not found then the function returns -1. The function goes over the nodes from the start (from the first item) and then uses the **successor** function to move on. Like we proved in the recitation in the worst case search function works in O(n) .

- **Concat(lst):** The function gets a list and concatenated our list to the given list. The function returns the absolute value of the difference between the height of the AVL trees joined. The function works like we saw in class. The function uses **insert**, **delete**, **balanceTree** and **handleSizesHeights** functions (all $O(\log(n))$) , and works in O(log(n).

Private Methods:

- **successor(node):** A function which returns the successor of the current node in the AVLTree. If node is last one in list, returns a virtual node. An implementation of the pseudo code we've seen in class, therefore works in O(log(n) time complexity (as have been proved).

- **predecessor(node):** A function which returns the predecessor of the current node in the AVLTree. If node is first one in list, returns a virtual node. An implementation of the pseudo code we've seen in class, therefore works in O(log(n)) time complexity (as have been proved).

- **getRoot():** A function that returns the root of the tree. If the tree is empty returns None. ($O(1)$)

- **deleteLessThanTwo**(**node**): A function which is called to delete a node if node is a leaf or has only one non-virtual son. Uses **isLeaf** ($\Theta(1)$) to check whether node is leaf, and if so uses **deleteNodeIsLeaf** ($\Theta(1)$) to delete it. Otherwise, uses **isRealNode** ($\Theta(1)$) to determine which of left and right sons is real, then handles the deletion of the the node given from the AVLTree accordingly, using a bounded by const number of basic get and set operations (c* $\Theta(1)$). Returns the node from which the balancing of the tree should start from. Does not contain any recursion or loops. All called methods are of O(1) time complexity. Therefore, works in $\Theta(1)$ time complexity.

- **deleteNodeHasTwoChildren**(**node**): A function which is called to delete a node if node has 2 non - virtual sons. implements deleting of 2 a node which has 2 sons like we were taught. Uses **successor** (O(log(n))) to find the successor of node. At that point successor has 1 or 0 children so it uses **deleteLessThenTwo** ($\Theta(1)$) to delete it, and bounded by const number of basic get and set operations (c* $\Theta(1)$) to rearrange the AVLTree and it's members correctly. Returns the node from which the balancing of the tree should start from. Contains no recursion or loops, all actions are done in $\Theta(1)$ except from finding successor (O(log(n))). Therefore, works in O(log(n)) time complexity.

- **balanceTree**(node, called_from): A function which balances the tree starting from a given node, based on whether it was called from a delete or insert function. If node is the root of AVLTree, handles balancing from the root by calling **balanceFromRoot** ($\Theta(1)$)**.** Otherwise, runs a while loop for as long as node is not the root, which updates the sizes and heights of the nodes in the route from first node to root using **updateMySizeHeight** ($\Theta(1)$), calculates the node's and the parent's Balance Factor using **computeBF** ($\Theta(1)$) and makes rotations using **makeRotation** ($\Theta(1)$) when necessary. Returns the number of rotations required to balance the tree. Going up from an arbitrary node to the root requires at most log(n) operations as we've proved in class. All functions which might be called in a single iteration are $\Theta(1)$, therefore together they cost c* $\Theta(1)$ = O(1) operations. So there are log(n) iterations in the while loop, each does O(1) operations, then in total the function works in O(log(n)) time complexity.

- **balanceFromRoot():** Balances the tree if the node to start balancing from is the root, using **makeRotation** ($\Theta(1)$). Therefore, works in O(log(n)) time complexity.

- **makeRotation(parent, BFparent, BFnode):** Checks which rotation should be made given the parameters parent, BFparent and BFnode and performs the rotation using the $\Theta(1)$ rotation functions: **rightRotation**, **leftRotation**, **leftThenRightRotation** or **rightThenLeftRotation**. Returns 1 if made a single rotation or 2 for a double rotation. Works in $\Theta(1)$ time complexity.

- **rightRotation(parent)**, **leftRotation(parent)**, **leftThenRightRotation(parent)** and **rightThenLeftRotation(parent):** An implementation of the rotation functions we've seen in class. Change a bounded amount of pointers. All work in $\Theta(1)$ time complexity.

- **handleSizeHeights(node):** uses **updateMySizeHeight** function from the given node upwards to the root. The number of actions is bounded by height of the tree, thus work in O(log(n)) time complexity.

- **updateMySizeHeight(node):** uses **updateSize** and **updateHeight** function to fix the size and height of the given node. Works in $\Theta(1)$**.**

- **randQuicksort(array, l, r):** A python implementation of the randQuicksort algorithm we saw in class, using **lomutoPartition.** As we have proved, works in O(n^2) in the worst case, but in O(nlogn) in the average case.

- **lomutoPartition(array ,l, r):** A python implementation of lomuto's partition algorithm. As we proved in class, works in O(n) time complexity.

- **replaceVals(array, k, m):** The naive algorithm for replacing the values in 2 indices in an array in place. Works in $\Theta(1)$.

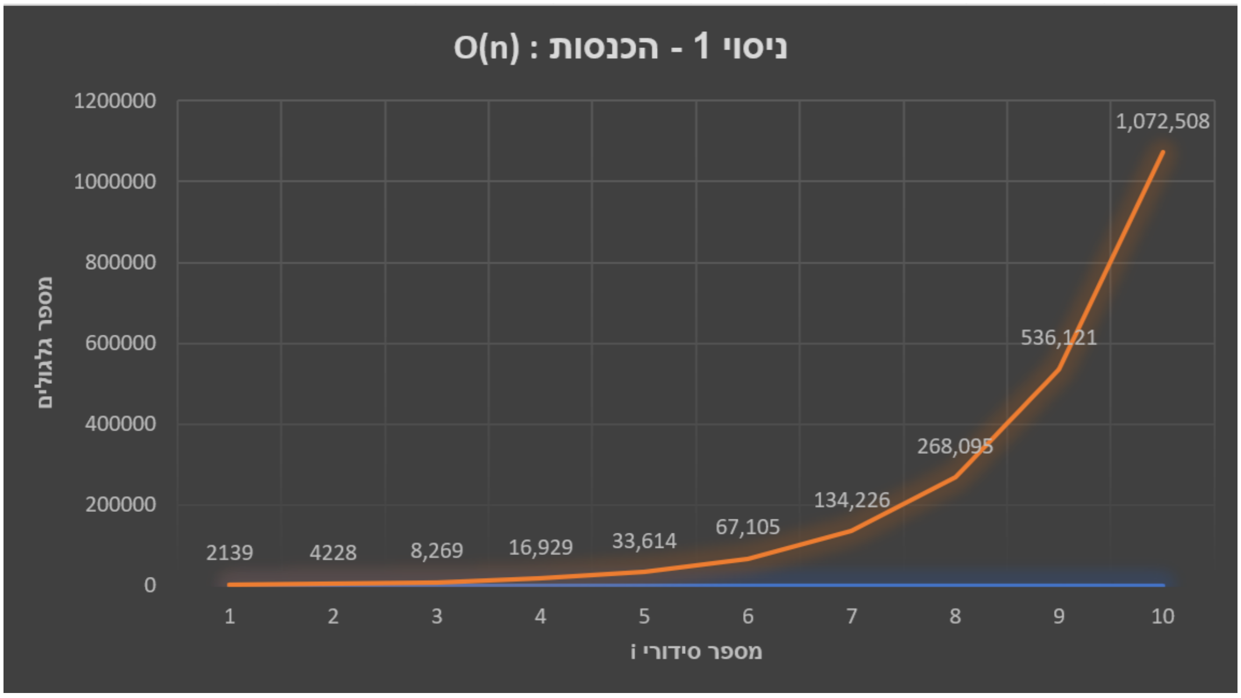
AVLTreeList - Theoretical Part

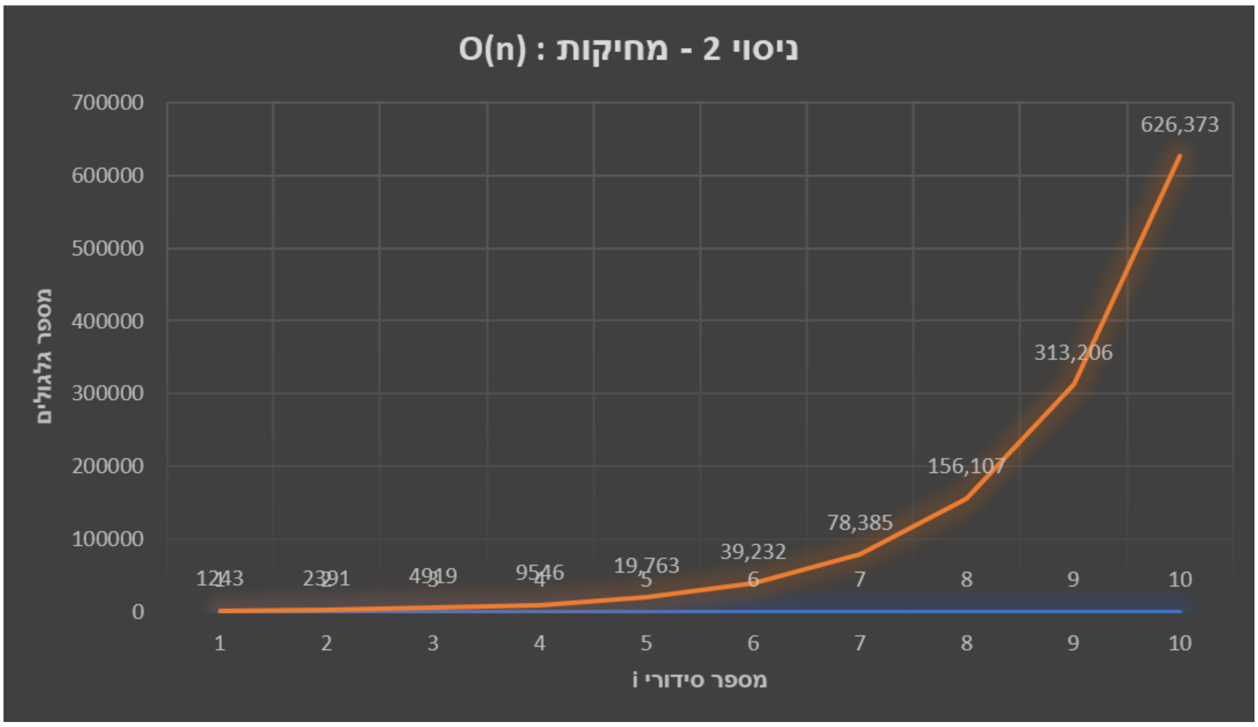
שאלה 1:

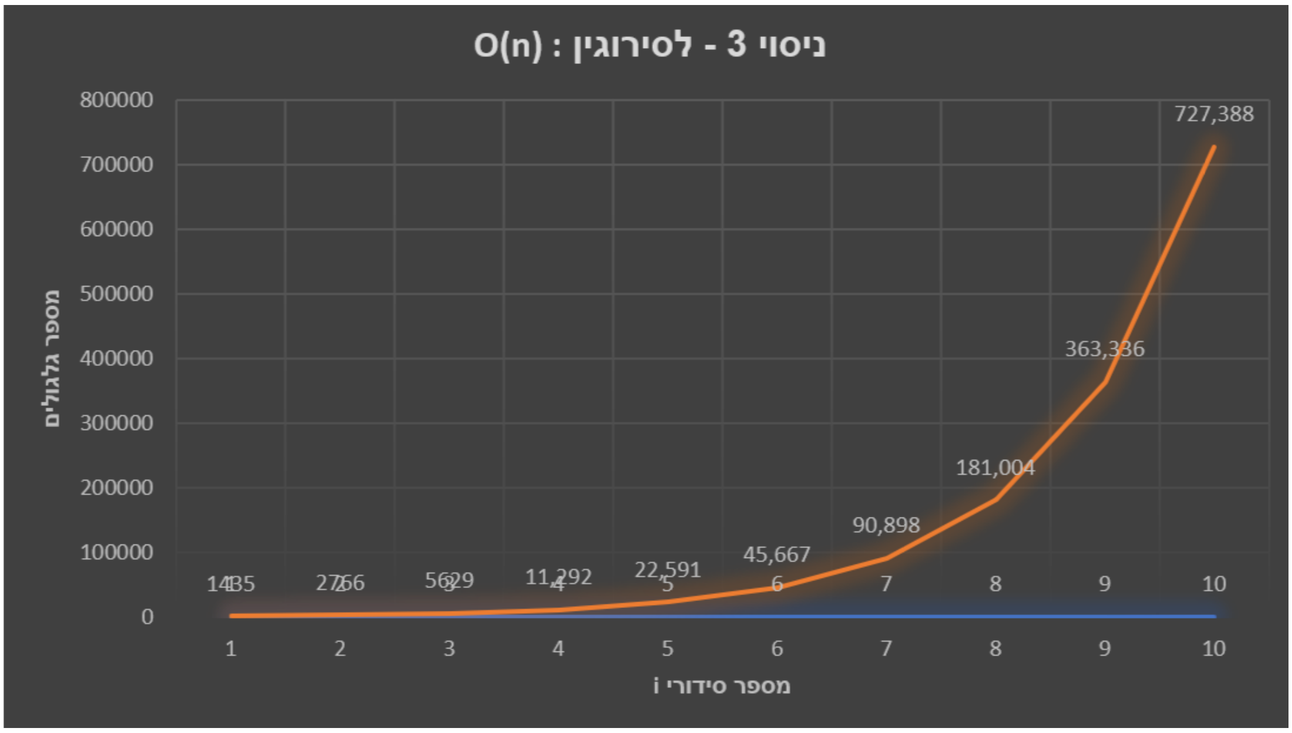
1.

מספר סידורי i	ניסוי 1 – הכנסות	ניסוי 2 – מחיקות	ניסוי 3 - לסירוגין
1	2139	1243	1435
2	4228	2391	2766
3	8,269	4919	5629
4	16,929	9546	11,292
5	33,614	19,763	22,591
6	67,105	39,232	45,667
7	134,226	78,385	90,898
8	268,095	156,107	181,004
9	536,121	313,206	363,336
10	1,072,508	626,373	727,388

.2


ניסוי 1 - הכנסות : O(n)
מספר גלגולים
מספר סידורי i
1200000
1000000
800000
600000
400000
200000
0
1 2 3 4 5 6 7 8 9 10
2139
4228
8,269
16,929
33,614
67,105
134,226
268,095
536,121
1,072,508


ניסוי 2 - מחיקות : O(n)
מספר גלגולים
מספר סידורי i
700000
600000
500000
400000
300000
200000
100000
0
1 2 3 4 5 6 7 8 9 10
1243
2391
4919
9546
19,763
39,232
78,385
156,107
313,206
626,373



שאלה 2:

זמן ריצה בממוצע מספר סידורי i	עץ AVL הכנסות להתחלה	רשימה מקושרת הכנסות להתחלה	מערך הכנסות להתחלה
1	4.183e^-5	6.472 e^-7	9.030 e^-5
2	4.704e^-5	7.033 e^-7	0.00033
3	5.024e^-5	5.983 e^-7	0.00075
4	5.242e^-5	6.268 e^-7	0.00132
5	5.550e^-5	5.794 e^-7	0.00210
6	5.495e^-5	6.344 e^-7	0.00304
7	5.936e^-5	1.019 e^-6	0.00413
8	5.768e^-5	6.104 e^-7	0.00536

זמן ריצה בממוצע מספר סידורי i	עץ AVL הכנסות להתחלה	רשימה מקושרת הכנסות להתחלה	מערך הכנסות להתחלה
9	6.177e^-5	6.029 e^-7	0.00679
10	6.044e^-5	1.239 e^-6	0.00838

זמן ריצה בממוצע מספר סידורי i	עץ AVL הכנסות אקראיות	רשימה מקושרת הכנסות אקראיות	מערך הכנסות אקראיות
1	5.412 e^-5	1.495 e^-6	8.122 e^-5
2	6.350 e^-5	1.612 e^-6	0.00033
3	7.031 e^-5	1.486 e^-6	0.00076
4	7.443 e^-5	1.518 e^-6	0.00137
5	7.850 e^-5	1.509 e^-6	0.00217
6	8.078 e^-5	1.552 e^-6	0.00313
7	8.673 e^-5	1.914 e^-6	0.00427
8	8.545 e^-5	1.531 e^-6	0.00556
9	9.297 e^-5	1.511 e^-6	0.00704
10	8.916 e^-5	2.086 e^-6	0.00876

זמן ריצה בממוצע מספר סידורי i	עץ AVL הכנסות בסוף	רשימה מקושרת הכנסות בסוף	מערך הכנסות בסוף
1	5.504 e^-5	6.713 e^-7	7.572 e^-7
2	6.219 e^-5	7.403 e^-7	6.650 e^-7
3	6.611 e^-5	6.460 e^-7	7.237 e^-7
4	6.926 e^-5	6.793 e^-7	7.290 e^-7
5	7.320 e^-5	6.006 e^-7	8.393 e^-7
6	7.307 e^-5	6.656 e^-7	6.048 e^-7
7	7.854 e^-5	1.055 e^-6	9.204 e^-7
8	7.642 e^-5	6.386 e^-7	6.052 e^-7
9	7.981 e^-5	6.445 e^-7	6.305 e^-7
10	8.028 e^-5	1.223 e^-6	1.040 e^-6

הציפייה היא שהכנסות לרשימה המקושרת בסוף ובהתחלה יהיו הכי מהירות (קבוע באורך הקלט), ובאופן אקראי הכי איטיות מבין כולם. (ליניארי בקלט)

לעומתם זמני הריצה של מערך צריכים להיות מהירים יחסית כשמדובר בהכנסה בסוף מפני שזמן הריצה אמורטייזד הוא כפי שהוכחנו קבוע באורך הקלט, אבל מאוד איטיים בהכנסה בסוף ובהכנסה אקראית כי יש צורך להעתיק את כל המערך.

העץ צריך לתת זמנים פחות טובים מהרשימה בהכנסות בהתחלה ובסוף בשל הצורך לאזן את העץ, קצת פחות טובים מהמערך עבור הכנסה בסוף, והכי טובים בהכנסה אקראית מבין כולם. (לוגריתמי באורך הקלט).

כפי שניתן לראות, תוצאות האמת מסתדרות עם התחזיות האלה, חוץ מבמקרה מפתיע אחד: מסתבר שרשימה מקושרת מצליחה להביס את העץ גם בהכנסות אקראיות, למרות הצורך לעבור על כל הרשימה עד לאינדקס ההכנסה בכל פעם. ייתכן שזה נובע מכך שעבור מספרים בסדר גודל כזה, הקבועים האסימפטוטיים בעץ יותר גדולים מהקבועים האסימפטוטיים עבור הרשימה המקושרת, והבדל משמעותי יתבטא רק החל מ - n מסוים גדול מאוד.