חלק תיאורטי- תומר רודניצקי (20662787820) ועידו רוזינר(209617000)

להלן תיעוד AVLNode:

חישוב סיבוכיות זמן	תיאור	פונקציה
$O(1)$	הפונקציה מחזירה את הבן השמאלי של הצומת. מוגדר להיות None במקרה והצומת וירטואלית.	getLeft()
$O(1)$	הפונקציה מחזירה את הבן הימני של הצומת. מוגדר להיות None במקרה והצומת וירטואלית.	getRight()
$O(1)$	הפונקציה מחזירה את ההורה של הצומת. מוגדר להיות None במקרה והצומת וירטואלית	getParent()
$O(1)$	הפונקציה מחזירה את הערך של הצומת. מוגדר להיות None במקרה והצומת וירטואלית.	getValue()
$O(1)$	הפונקציה מחזירה את הגובה של הצומת. מוגדר להיות 1- במקרה והצומת וירטואלית.	getHeight()
$O(1)$	הפונקציה מחזירה את הגודל של הצומת. מוגדר להיות 0 במקרה והצומת וירטואלית.	getSize()
$O(1)$	הפונקציה מחזירה את ה-balance factor של הצומת.	getBF()
$O(1)$	הפונקציה מקבעת את הבן השמאלי של הצומת להיות node.	setLeft(node)
$O(1)$	הפונקציה מקבעת את הבן הימני של הצומת להיות node.	setRight(node)
$O(1)$	הפונקציה מקבעת את ההורה של הצומת להיות node.	setParent(node)
$O(1)$	הפונקציה מקבעת את הערך של הצומת להיות value.	setValue(value)
$O(1)$	הפונקציה מקבעת את הגובה של הצומת להיות h.	setHeight(h)
$O(1)$	הפונקציה מקבעת את הגודל של הצומת להיות s.	setSize(s)
$O(1)$	הפונקציה מקבעת את ה-balance factor של הצומת להיות bf.	setBF(bf)
$O(1)$	הפונקציה מחזירה True אם הצומת היא לא צומת וירטואלית (הגובה שונה מ-(1-)) ואחרת False.	isRealNode()
$O(1)$	הפונקציה מחשבת ומחזירה את ה-balance factor של הצומת. כלומר, מחשבת ומחזירה $1 + height(leftSubtree) - (1 + height(rightSubtree))$	computeBF()
$O(1)$	הפונקציה מחשבת ומחזירה את הגובה של הצומת. כלומר, מחשבת ומחזירה $1 + \max\{height(leftSubtree), height(rightSubtree)\}$	computeHeight()
$O(1)$	הפונקציה מחשבת ומחזירה את הגודל של הצומת. כלומר, מחשבת ומחזירה את מספר הצמתים שקטנים שווים לה – $1 + size(leftSubtree) + size(rightSubtree)$	computeSize()

להלן תיעוד AVLTreeList:

חישוב סיבוכיות זמן	תיאור	פונקציה
$O(1)$	הפונקציה מחזירה את השורש של העץ. מוחזר AVLNode(None) במקרה והעץ וירטואלי.	getRoot()
$O(1)$	הפונקציה מחזירה את הצומת המקסימלית בעץ. מוחזר None במקרה והעץ וירטואלי	getMax()
$O(1)$	הפונקציה מחזירה את הצומת המינימלית בעץ. מוחזר None במקרה והעץ וירטואלי.	getMin()
$O(1)$	הפונקציה מחזירה את גודל העץ - מספר הצמתים הלא וירטואליים שנמצאים בעץ. מוחזר 0 במקרה והעץ וירטואלי.	getSize()
$O(1)$	הפונקציה מקבעת את השורש של העץ להיות node.	setRoot(node)
$O(1)$	הפונקציה מקבעת את הצומת המקסימלית בעץ להיות node.	setMax(node)
$O(1)$	הפונקציה מקבעת את הצומת המינימלית בעץ להיות node.	setMin(node)
$O(1)$	הפונקציה מקבעת את הגודל של העץ להיות s.	setSize(s)
$O(1)$	הפונקציה מחזירה True אם הרשימה (העץ) ריקה (עץ וירטואלי) ואחרת False.	empty()
הפונקציה קוראת ל - treeSelect(self, i+1), שרצה ב- $O(\log i)$, וכל שאר הפעולות קורות ב-$O(1)$. לכן הפונקציה רצה ב-$O(\log i)$.	הפונקציה מחזירה את ערך האיבר במקום ה-i אם קיים, אחרת היא מחזירה None.	retrieve(i)
תחילה הפונקציה קוראת ל- treeSelect(i) שרצה בסיבוכיות $O(\log i)$. לאחר מכן הפונקציה קוראת ל- rebalanceTree(node) שרצה בסיבוכיות $O(\log n)$. מכיוון שכל שאר הפעולות קורות ב-$O(1)$ הפונקציה רצה בסיבוכיות $O(\log n)$.	הפונקציה מכניסה איבר בעל ערך val לרשימה במקום ה-i במידה וקיימים לפחות i איברים ברשימה. הפונקציה מחזירה את מספר פעולות האיזון שנדרשו בשלב תיקון העץ על מנת לשמר את תכונת האיזון.	insert(i, val)

תחילה הפונקציה קוראת ל-treeSelect(i) שרצה בסיבוכיות $O(\log i)$ או לפונקציה successor(x) שרצה בסיבוכיות $O(\log n)$. לאחר מכן הפונקציה קוראת ל-delete_node(node_to_del) ול- rebalanceTree(node), שרצות בסיבוכיות $O(\log n)$, אחד אחרי השני. מכיוון שכל שאר הפעולות קורות ב-$O(1)$, הפונקציה רצה בסיבוכיות $O(\log n)$.	הפונקציה מוחקת את האיבר במקום ה-i ברשימה, אם הוא קיים. הפונקציה מחזירה את מספר פעולות האיזון שנדרשו בשלב תיקון העץ על מנת לשמר את תכונת האיזון. אם לא קיימים מספיק איברים ברשימה הפונקציה מחזירה 1-.	delete(i)
אם node_to_del בעל שני ילדים אמיתיים הפונקציה קוראת ל-successor(node), שרצה בסיבוכיות $O(\log n)$, לכל היותר $O(1)$ פעמים. מכיוון שכל שאר הפעולות בפונקציה קורות ב-$O(1)$ הפונקציה רצה בסיבוכיות $O(\log n)$.	הפונקציה מוחקת את הצומת node_to_del מהעץ ומחזירה את הצומת שממנה נתחיל את פעולת האיזון של העץ.	delete_node(node_to_del)
$O(1)$	הפונקציה מחזירה את ערך האיבר הראשון ברשימה, או None ברשימה ריקה.	first()
$O(1)$	הפונקציה מחזירה את ערך האיבר האחרון ברשימה, או None ברשימה ריקה.	last()
הפונקציה קוראת לפונקציה inorder(node, L) שרצה בסיבוכיות $O(n)$. מכיוון שכל שאר הפעולות קורות ב-$O(1)$, הפונקציה רצה בסיבוכיות $O(n)$.	הפונקציה מחזירה מערך המכיל את הערכים של איברי הרשימה (העץ) לפי סדר האינדקסים (in-order), או מערך ריק אם הרשימה ריקה.	listToArray()
ההכנסה מבוצעת כמו באלגוריתם in-order שנלמד בשיעור (במקום הדפסה נוסיף את הערך של הצומת למערך L) ולכן הפונקציה רצה בסיבוכיות $O(n)$.	הפונקציה מכניסה לרשימה L את הערכים של איברי הרשימה (העץ) לפי סדר האינדקסים. הפונקציה עושה זאת על ידי ריצה in-order על איברי העץ.	inorder(node, L)
$O(1)$	הפונקציה מחזירה את מספר האיברים ברשימה.	length()
תחילה הפונקציה קוראת ל-listToArray(self), שרצה בסיבוכיות $O(n)$. לאחר מכן הפונקציה מפרידה את כל האיברים שהם None	הפונקציה מסדרת את איברי הרשימה בסדר עולה ומחזירה רשימה (עץ) חדשה.	sort()

מהאיברים שהם לא None בסיבוכיות $O(n)$. לאחר מכן הפונקציה קוראת ל-merge_sort(lst) שרצה בסיבוכיות $O(n \cdot \log n)$. אחר כך הפונקציה קוראת ל-buildTreeFromList(lst, first, last) שרצה בסיבוכיות $O(n)$ ושומרת את הצומת שהתקבלה כשורש של AVLTreeList חדש. לבסוף הפונקציה מעדכנת את השדות של ה-AVLTreeList החדש. לשם כך הפונקציה קוראת ל- max_node(node) ו-min_node(node) אשר רצות בסיבוכיות $O(\log n)$. סה"כ נקבל כי הפונקציה sort(self) רצה בסיבוכיות $O(n \cdot \log n)$.		
תחילה הפונקציה קוראת ל-listToArray(self) שרצה בסיבוכיות $O(n)$. לאחר מכן הפונקציה קוראת לפונקציה shuffle(lst), שרצה בסיבוכיות $O(n)$. אחר כך הפונקציה קוראת ל-buildTreeFromList(lst, first, last) שרצה בסיבוכיות $O(n)$ ושומרת את הצומת שהתקבלה כשורש של AVLTreeList חדש. לבסוף הפונקציה מעדכנת את השדות של ה-AVLTreeList החדש. לשם כך הפונקציה קוראת ל- max_node(node) ו-min_node(node) אשר רצות בסיבוכיות $O(\log n)$. סה"כ נקבל כי הפונקציה permutation(self) רצה בסיבוכיות $O(n)$.	הפונקציה מחזירה רשימה (עץ) חדשה המכילה את אותם האיברים של הרשימה (העץ) הנוכחי (self) בסדר אקראי.	permutation()
במקרה שבו lst או self ריקים הפונקציה מחברת את העצים (אם יש מה לחבר) ומסיימת לרוץ – פעולה זו קוראת בסיבוכיות $O(1)$. אם האורך של אחד העצים הוא 1 הפונקציה תבצע insert לאיבר של העץ בעל האיבר היחיד אל תוך העץ השני ותסיים את הריצה –סיבוכיות $O(\log n)$. אחרת, הפונקציה שומרת את האיבר המקסימלי של self	הפונקציה מקבלת רשימה ומשרשר אותה אל סוף הרשימה הנוכחית. הפונקציה מחזירה את הערך המוחלט של הפרש הגבהים של עצי ה-AVL שמוזגו.	concat(lst)

		וקוראת ל-delete(self.length()-1) – סיבוכיות $O(\log n)$. כעת, הפונקציה משווה בין הגובה של העצים (לאחר המחיקה) ומחברת אותם בהתאם: אם הגובה של self שווה לגובה של lst הפונקציה קוראת ל-joinSelfSame(self, x, lst) שרצה בסיבוכיות $O(1)$. אם הגובה של self גדול מהגובה של lst הפונקציה קוראת ל-joinSelfBigger(self, x, lst) שרצה בסיבוכיות $O(\log n)$. אם הגובה של self קטן מהגובה של lst הפונקציה קוראת ל-joinSelfSmaller(self, x, lst) שרצה בסיבוכיות $O(\log n)$. לבסוף הפונקציה תעדכן את השדות של העץ (סיבוכיות $O(1)$) ותקרא ל-rebalanceTree(node) כאשר node הוא הצומת המקסימלית שמחקנו ושמרנו מקודם – סיבוכיות $O(\log n)$. סה"כ הפונקציה רצה בסיבוכיות $O(\log n)$.
joinSelfBigger(x, lst)	הפונקציה מחברת את העץ ל-self דרך הצומת x	הפונקציה מניחה כי גובה העץ self גדול מגובה העץ lst ומחברת את העצים דרך הצומת x כנלמד בשיעור (הפונקציה Join מהשיעור) – סיבוכיות $O(\log n)$
joinSelfSmaller(x, lst)	הפונקציה מחברת את העץ ל-self דרך הצומת x	הפונקציה מניחה כי גובה העץ self קטן מגובה העץ lst ומחברת את העצים דרך הצומת x כנלמד בשיעור (הפונקציה Join מהשיעור) – סיבוכיות $O(\log n)$
joinSelfSame(x, lst)	הפונקציה מחברת את העץ ל-self דרך הצומת x	הפונקציה מניחה כי גובה העץ self שווה לגובה העץ lst ומחברת את העצים דרך הצומת x כנלמד בשיעור (הפונקציה Join מהשיעור). נשים לב כי הפעם אין צורך לרדת במורד אף עץ או לאזן יותר מצומת אחת בעץ, ולכן הפונקציה רצה בסיבוכיות $O(1)$
search(val)	הפונקציה מחזירה את האינדקס הראשון ברשימה בו מופיע הערך val או 1- אם לא קיים אינדקס כזה.	תחילה הפונקציה קוראת ל-listToArray(self) – סיבוכיות $O(n)$. ולאחר מכן הפונקציה רצה בלולאת for על איברי הרשימה – סיבוכיות $O(n)$

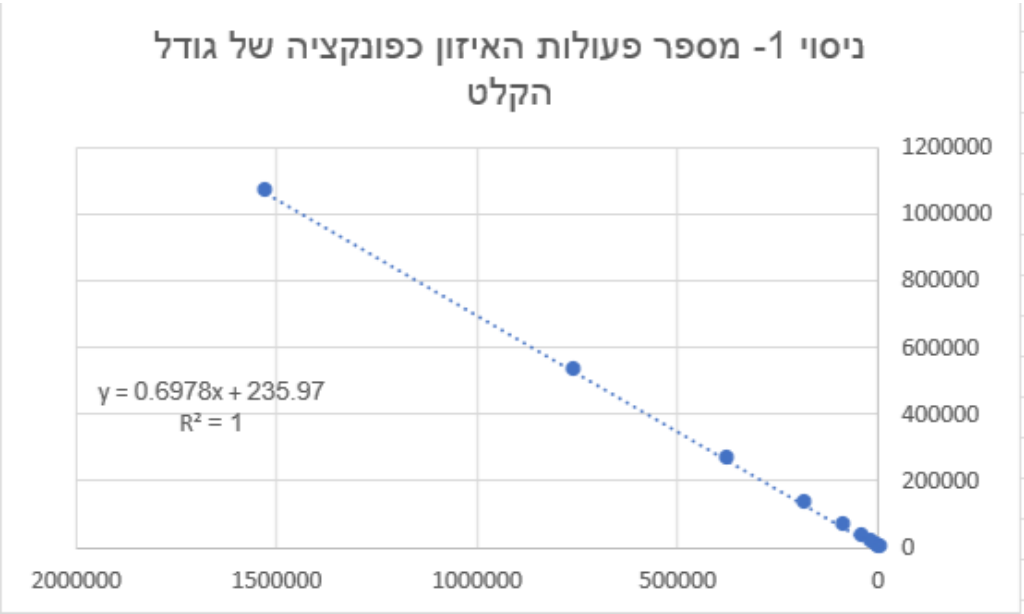
סה"כ הפונקציה רצה בסיבוכיות $O(n)$.		
הפונקציה פועלת כמעט כמו שנלמד בהרצאה (לאחר המודיפיקציה של הוספת השדה size ו-height לצמתים). ההבדל היחיד הוא הבדיקה של האם node הוא השורש של העץ. אם node הוא השורש של העץ נעדכן את השורש להיות הבן השמאלי של node – סיבוכיות $O(1)$. סה"כ הפונקציה רצה בסיבוכיות $O(1)$.	הפונקציה מבצעת סיבוב ימינה על node	rotateRight(node)
כמעט כמו שנלמד בהרצאה (לאחר המודיפיקציה של הוספת השדה size ו-height לצמתים). ההבדל היחיד הוא הבדיקה של האם node הוא השורש של העץ. אם node הוא השורש של העץ נעדכן את השורש להיות הבן הימני של node – סיבוכיות $O(1)$. סה"כ הפונקציה רצה בסיבוכיות $O(1)$.	הפונקציה מבצעת סיבוב שמאלה על node	rotateLeft(node)
הפונקציה זהה לחלק של האיזון של העץ של הפונקציה Delete(D,x) שנלמדה בכיתה (לאחר המודיפיקציה שלכל צומת קיים שדה size) - סיבוכיות $O(\log n)$.	הפונקציה מאזנת את העץ ומעדכנת את השדות כמו שנלמד בהרצאה	rebalanceTree(node)
הפונקציות זהות לפונקציית Select(D, k) שנלמדה בכיתה (לאחר המודיפיקציה שמתחילים את החיפוש בעץ מהצומת המינימלית) - סיבוכיות $O(\log i)$	הפונקציות מחזירות את האיבר ה-i הכי קטן בעץ כמו שנלמד בכיתה (לאחר המודיפיקציה של התחלת החיפוש מהאיבר המינימלי)	treeSelect(i) treeSelectHelper(node, i)
הפונקציה יורדת מהשורש של התת עץ אל הצומת המקסימלית שלו במסלול ישיר ולכן הסיבוכיות היא $O(\log n)$	הפונקציה מחזירה את האיבר המקסימלי בתת-עץ שבו node הוא השורש.	max_node(node)
הפונקציה יורדת מהשורש של התת עץ אל הצומת המינימלית שלו במסלול ישיר ולכן הסיבוכיות היא $O(\log n)$	הפונקציה מחזירה את האיבר המינימלי בתת-עץ שבו node הוא השורש.	min_node(node)
הפונקציה זהה לפונקציית Predecessor(D, x) שנלמדה בשיעור - סיבוכיות $O(\log n)$.	הפונקציה מחזירה את האיבר הקודם ברשימה. כלומר, אם x הוא האיבר במקום ה-i הפונקציה תחזיר את האיבר ה-i-1.	predecessor(x)

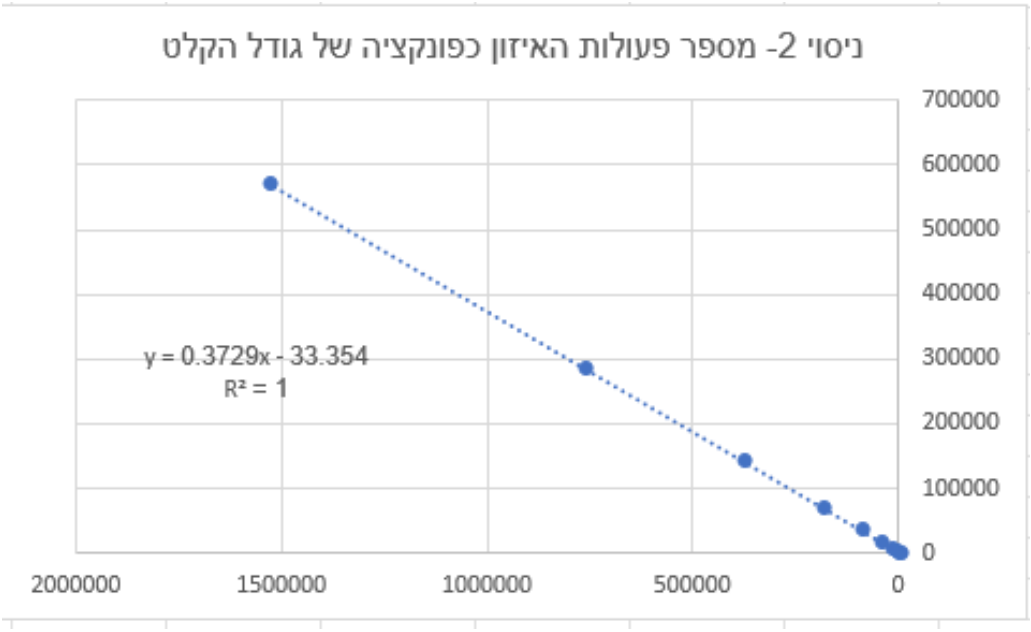
successor(x)	הפונקציה מחזירה את האיבר הבא ברשימה. כלומר, אם x הוא האיבר במקום ה-i הפונקציה תחזיר את האיבר ה-i+1.	הפונקציה זהה לפונקציית Successor(D, x) שנלמדה בשיעור - סיבוכיות $O(\log n)$.
buildTreeFromList(lst, left, right)	הפונקציה מקבלת רשימה של ערכים ומחזירה צומת שמהווה שורש לעץ AVLTreeList הנוצר מהרשימה.	כל פעם הפונקציה מבצעת שתי קריאות רקורסיביות על שני חצאים שונים של הרשימה (אנו עושים זאת באמצעות מצביעים כך שלא תיווסף סיבוכיות) ולכן עומק עץ הרקורסיה יהיה $\log n$ כאשר בכל קריאה מתבצעת $O(1)$ עבודה. על כן, סיבוכיות הפונקציה היא $2^{\log n} = O(n)$
merge_sort(lst) merge(lst1, lst2)	הפונקציות מחזירות רשימה מסודרת בסדר עולה.	הקוד זהה לקוד של mergeSort שלמדנו בקורס מבוא מורחב למדעי המחשב – $O(n \cdot \log n)$
Shuffle(lst)	הפונקציה "מבלגנת" את הרשימה באופן אקראי in-place.	הפונקציה עוברת על הרשימה בלולאות for וכל פעם מחליפה בין שני איברי הרשימה - $O(n)$

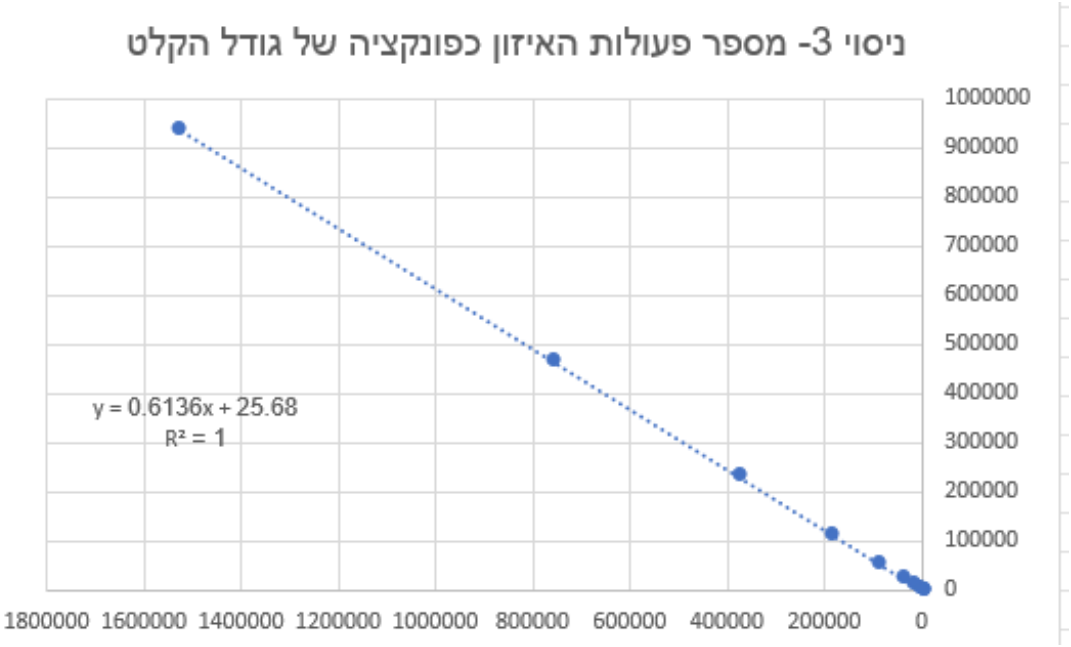
1.1. להלן טבלה המתארת את מספר פעולות האיזון שנדרשו כדי לתקן את העץ בכל ניסוי:

i	ניסוי 1- הכנסות	ניסוי 2- מחיקות	ניסוי 3- הכנסות ומחיקות לסירוגין
1	2146	1146	1830
2	4130	2143	3699
3	8332	4478	7367
4	16817	8933	14625
5	33467	17812	29600
6	67365	35793	58789
7	134828	71788	117405
8	268134	142998	236183
9	537312	286055	471642
10	1071498	572806	942261

2.1. להלן תרשימים המתארים את תוצאות הניסוי בצורה גראפית. הוספנו קו מגמה, מדד R^2 ומשוואה המתארת בקירוב את הקו. מכל אלה המסקנה היא כי הביטוי האסימפטומטי הוא o(n) בשלושת הניסויים:




ניסוי 2- מספר פעולות האיזון כפונקציה של גודל הקלט
y = 0.3729x - 33.354
R² = 1
700000
600000
500000
400000
300000
200000
100000
0
2000000
1500000
1000000
500000
0


ניסוי 3- מספר פעולות האיזון כפונקציה של גודל הקלט
y = 0.6136x + 25.68
R² = 1
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0

2. להלן שלושת הטבלאות המתארות את זמן הריצה בממוצע עבור עץ AVl, רשימה מקושרת ועבור מערך:

i	זמן ריצה- עץ AVL הכנסות להתחלה	זמן ריצה- רשימה מקושרת הכנסות להתחלה	זמן ריצה- מערך הכנסות להתחלה
1	3.664356e-05	5.654738e-07	3.295837e-07
2	2.923434e-05	6.545347e-07	5.545469e-07
3	3.192972e-05	6.878794e-07	7.467864e-07
4	3.123943e-05	6.564767e-07	9.565846e-07
5	3.225311e-05	6.747463e-07	1.113233e-06
6	3.487298e-05	6.657457e-07	1.515454e-06
7	3.123936e-05	6.754744e-07	1.724939e-06
8	3.286947e-05	6.336785e-07	1.835562e-06
9	3.112393e-05	6.112436e-07	1.964126e-06
10	3.133183e-05	6.868746e-07	2.021410e-06

i	זמן ריצה- עץ AVL הכנסות אקראיות	זמן ריצה- רשימה מקושרת הכנסות אקראיות	זמן ריצה- מערך הכנסות אקראיות
1	4.129876e-05	1.775654e-05	8.469857e-07
2	3.700732e-05	3.748905e-05	8.965450e-07
3	3.774336e-05	7.969767e-05	1.193485e-06
4	4.012432e-05	9.021049e-04	1.327493e-06
5	4.498348e-05	1.156784e-04	1.344458e-06
6	4.744344e-05	1.331895e-04	1.230485e-06
7	4.425390e-05	1.567349e-04	1.446758e-06
8	5.345783e-05	1.936695e-04	1.546281e-06
9	4.745946e-05	2.188605e-04	1.748480e-06
10	5.449583e-05	2.324536e-04	1.645678e-06

מספר סידורי – i	זמן ריצה- עץ AVL הכנסות בסוף	זמן ריצה- רשימה מקושרת הכנסות בסוף	זמן ריצה- מערך הכנסות בסוף
1	2.802437e-05	5.812048e-07	2.241384e-07
2	2.923845e-05	6.458395e-07	1.934423e-07
3	3.001652e-05	6.835368e-07	1.748259e-07
4	3.168567e-05	6.412367e-07	1.539534e-07
5	3.432853e-05	6.503696e-07	1.934423e-07
6	3.023855e-05	6.436063e-07	1.493735e-07
7	3.359357e-05	6.130633e-07	1.495953e-07
8	3.004939e-05	6.543698e-07	1.495560e-07
9	3.583897e-05	6.663460e-07	1.535794e-07
10	3.668567e-05	6.346602e-07	1.569639e-07

לפני תחילת הניסוי, היינו מצפים כי התוצאות האמיתיות יהיו דומות לאלה שיצאו בסופו של דבר. ברשימה מקושרת, הכנסות לתחילתה ולסופה מתבצעות ב(1)o (זוהי רשימה עם מצביע לסוף) ולכן התקבלו תוצאות מהירות יותר מאשר בהכנסות לסוף ולתחילת עץ AVL. מנגד, בהכנסות אקראיות ברשימה מקושרת "נאלץ" לעבור על חלק גדול מהרשימה בכל הכנסה((n)o במקרה הגרוע) ולכן זמן הריצה היה איטי יותר. נשאלת השאלה מדוע במערך התקבלו התוצאות הטובות ביותר בכמעט כל המקרים? ובכן, המימוש של פייתון למערך משודרג ויעיל, ככה שקשה להתחרות בו.

הערה: במידה והיינו ממשים רשימה מקושרת ללא מצביע לסוף, אנו סבורים כי זמן הריצה היה האיטי ביותר בהכנסות לסוף, שכן במצב כזה בכל הכנסה היינו נאלצים לעבור על כל איבר ואיבר ברשימה((n)o ולא במקרה הגרוע, אלא תמיד!).