מסמך תיעוד – AVLTree

שדות המחלקה:

mockAVLNode (צומת מדומה) – צומת אשר משמש כצומת וירטואלי אליו מצביעים העלים בעץ עבור בן שמאלי/ימני (טיפוס מסוג IAVLNode)

root (שורש העץ) – צומת מטיפוס מסוג IAVLNode

min (צומת מינימום) - צומת בעל המפתח המינימלי (טיפוס מסוג IAVLNode)

max (צומת מקסימום) – צומת בעל המפתח המקסימלי (טיפוס מסוג IAVLNode)

בנאי המחלקה:

AVLTree()

AVLTree(IAVLNode root)

קיים בנאי המגדיר עץ ריק ובנאי נוסף המגדיר עץ כאשר מקבל את השורש שלו.

כל עץ AVL מחזיק מצביע לשורש – root, מצביע לאיבר בעל המפתח המינימלי – min ומצביע לאיבר בעל הערך המקסימלי – max.

בנוסף לכל עץ קיים mockAVLNode – צומת מדומה יחיד אליו מצביעים כל העלים בעץ.

מתודות המחלקה:

`empty()` :

מתודה המחזירה האם העץ ריק.

סיבוכיות זמן ריצה: O(1)

למעשה מחזירה האם השורש של העץ הוא null – אם כך העץ ריק.

`search(int k)`:

מתודה המחפשת את הערך הנדרש בעץ.

סיבוכיות זמן ריצה: O(log n)

המתודה מחזירה null אם האיבר אינו קיים בעץ, אחרת מחזירה את הצומת המתאים.

מתודה זו קוראת למתודת עזר בשם serchRec.

`searchRec (int k, IAVLNode x)`:

מתודת עזר למתודה search. זוהי מתודה רקורסיבית.

סיבוכיות זמן ריצה: O(log n)

המתודה עוברת במסלול על העץ החל מהצומת המקבל x ועד הצומת המדומה הראשון ומחזירה את הצומת האמיתי האחרון בו עברה כאשר בכל פעם קוראת לעצמה באמצעות אחד הבנים של הצומת הנוכחי.

`insert(int k, String i)`:

מתודה שמטרתה להכניס צומת חדש לעץ. נעזרת **במתודת עזר**:

`insert(IAVLNode newLeaf)` - אשר מכניסה בפועל את הצומת לעץ.

המתודה בונה צומת AVL חדש עם הערכים שמתקבלים בקריאה למתודה ומכניסה אותו לעץ. היא משתמשת במתודה serchRec כדי למצוא את המקום אליו אמור להיכנס הערך (כאשר אם חוזר הערך הדרוש היא לא מכניסה את הצומת כי הוא כבר <u>קיים</u> בעץ).

מעדכנת את הגודל (size) של כל הצמתים שעבורם ערך זה השתנה.

מעדכנת מינימום ומקסימום בהתאם.

בנוסף, לאחר הכנסת הערך המתודה בודקת האם העץ מאוזן. במידה והעץ אינו מאוזן בעקבות ההכנסה המתודה בודקת באיזה מצב מאילו שנלמדו בכיתה אנו נמצאים ומטפלת בהתאם כך שבסוף המתודה העץ מאוזן. לשם כך משתמשת ב- RotateLeft , RotateRight.

ערך ההחזרה הינו <u>מספר פעולות האיזון שנדרשו</u>.

סיבוכיות זמן ריצה: O(log n)

`RotateRight(IAVLNode head)`:

מתודת עזר לגלגולים כפי שנלמדו בכיתה. המתודה משנה מצביעים בהתאם לגלגול. לכן -

סיבוכיות זמן ריצה: O(1)

בנוסף המתודה מסדרת את הגבהים של כל הצמתים שהושפעו בהתאם. מתודה זו מבצעת **גלגול ימינה**.

`RotateLeft(IAVLNode h)`:

מתודת עזר לגלגולים כפי שנלמדו בכיתה. המתודה משנה מצביעים בהתאם לגלגול. ולכן -
סיבוכיות זמן ריצה: O(1)

בנוסף המתודה מסדרת את הגבהים של כל הצמתים שהושפעו בהתאם. מתודה זו מבצעת גלגול שמאלה.

`rebalanceAfterInsertion(IAVLNode place)`:

המתודה מבצעת את פעולות האיזון הנדרשות לאחר הכנסה בהתאם למצבים שנלמדו בכיתה על מנת לשמור על עץ AVL **<u>תקין</u>**.

`delete(int k)`:

סיבוכיות זמן ריצה: O(log n)

מתודת מחיקת צומת מהעץ. אם הצומת הדרוש אינו קיים בעץ המתודה מחזירה 1-.

מתודה זו משתמשת במתודת העזר searchRec כדי למצוא את הצומת אותו נרצה למחוק.

כמו כן, הביצוע בפועל של המחיקה מתבצע במתודת עזר בשם deleteNode.

בנוסף ישנה חלוקה למקרים:

1. הצומת הדרוש הינו השורש של העץ. המתודה מתייחסת לכל האפשרויות הקיימות מבחינת כמות הבנים ה"אמיתיים" של השורש.
2. הצומת הנדרש אינו השורש וגם שם נבדקים כל המקרים האפשריים מבחינת כמות הבנים שלו. במקרה בו לצומת שני בנים (שאינם וירטואליים) נשתמש במתודה successor כדי למצוא את העוקב שלו ולהחליף ערכים כפי שנלמד בכיתה.

המתודה מעדכנת מינימום ומקסימום בהתאם בעזרת המתודה updateMinMax.

המתודה מחזירה את כמות פעולות האיזון שנדרשו לאחר המחיקה ולשם כך משתמשת במתודת העזר rebalancingAfterDeletion.

`deleteNode(IAVLNode item)`:

המתודה אשר בפועל מבצעת את מחיקת הצומת מהעץ לאור המקרים שתוארו לעיל במתודה delete. המתודות פוצלו על מנת לאפשר מימוש של מחיקת צומת בעץ גם במחלקת TreeList.

סיבוכיות זמן ריצה: O(log n)

`updateMinMax(int k)`:

עדכון מינימום/מקסימום בעץ במידה והצומת שנמחק היה המינימום/מקסימום.

ישנו צורך להגיע אל הצומת המינימלי/מקסימלי החדש (ולעדכנו בהתאם) ולכן לרדת משורש העץ שמאלה/ימינה.

סיבוכיות זמן ריצה: O(log n)

`rebalanceAfterDeletion(IAVLNode x)`:

מחשבת את מספר פעולות האיזון הנדרשות לאחר מחיקה של צומת בעץ. למעשה עוברת על מסלול מהמיקום בו נמחק הצומת ועד השורש ובודקת האם צריך לאזן את העץ. מסדרת גם את הגבהים בהתאם.

`successor(IAVLNode x)`:

מציאת העוקב של צומת בעץ.

סיבוכיות זמן ריצה: O(log n)

מקבלת צומת ומחזירה את הצומת העוקב לצומת שנשלח על-פי חישוב דרגה ולא על בסיס ערך המפתח של אותה צומת. האידיאל מאחורי מימוש זה, הוא לאפשר מימוש למתודה זו גם במחלקת TreeList.

`downSize (IAVLNode x)`:

מתודת עזר שתפקידה לעדכן את הגודל של תתי העצים שהשתנו בעקבות מחיקה.

סיבוכיות זמן ריצה: O(log n)

`min()`:

מחזירה את הצומת בעל המפתח המינימלי.

סיבוכיות זמן ריצה: O(1)

הסיבה לכך היא שמתוחזק מצביע למינימום.

`max()`:

מחזירה את הצומת בעל המפתח המקסימלי.

סיבוכיות זמן ריצה: O(1)

הסיבה לכך היא שמתוחזק מצביע למקסימום.

`getMock()`:

מחזירה את הצומת המדומה אליו קיים מצביע ולכן –

סיבוכיות זמן ריצה: O(1)

`keysToArray()`:

סיבוכיות זמן ריצה: O(n logn)

קוראת למתודה ()empty על מנת לבדוק אם העץ ריק – אם כן מחזירה מערך ריק. אחרת, עוברת על כל איברי העץ החל מהמינימום באמצעות מתודת successor ובונה מערך של המפתחות בעץ.

`infoToArray()`:

סיבוכיות זמן ריצה: O(n logn)

קוראת למתודה ()empty על מנת לבדוק אם העץ ריק – אם כן מחזירה מערך ריק. אחרת, עוברת על כל איברי העץ החל מהמינימום באמצעות מתודת successor ובונה מערך של ערכי value השייכים למפתחות כאשר הסדר נקבע על ידי המפתחות.

`size()`

סיבוכיות זמן ריצה: O(1)

למעשה מחזירה את גודל תת העץ של השורש (כולל) - זהו גודל העץ.

`getRoot()`:

מחזירה את שורש העץ אליו קיים מצביע ולכן –

סיבוכיות זמן ריצה: O(1)

`IAVLNode treeSelect(IAVLNode node, int i)`:

פונקציה אשר מחזירה את הצומת בעל דרגה i בעץ $(1 \leq i \leq n)$.

הפונקציה פועלת בצורה איטרטיבית ומבצעת מקסימום log n איטרציות (ירידה משורש העץ עד לאחד העלים).

סיבוכיות זמן ריצה: O(log n)

`int treeRank(IAVLNode node)`

פונקציה אשר מחזירה את הדרגה של צומת בעץ.

עלייה מצומת בעץ אל עבר השורש (log n צעדים במקרה והצומת הוא עלה).

סיבוכיות זמן ריצה: O(log n)

תיעוד AVLNode

<u>שדות המחלקה:</u>

key – מפתח מסוג מספר שלם (int).

value – ערך מסוג מחרוזת.

parent , **rightSon** , **leftSon** - בן ימני, בן שמאלי ואב מסוג IAVLNode

height – גובה הצומת (ביחס לעץ)

size – גודל תת העץ של הצומת (כולל הצומת עצמה)

כמו כן, ישנו מימוש של הממשק **IAVLNode**.

<u>בנאי המחלקה:</u>

```
AVLNode(int key, String value)
```

בנאי המקבל מפתח וערך מסוג מחרוזת ויוצר צומת כאשר הגובה מוגדר להיות 0 וגודלו של תת העץ מוגדר להיות 1.

מתודות המחלקה:

לכל שדה קיימות פונקציות Get ו- Set:

```
getKey()
setKey(int x)
getValue()
setValue(String x)
setLeft(IAVLNode node)
getLeft()
setRight(IAVLNode node)
getRight()
setParent(IAVLNode node)
getParent()
setHeight(int height)
getHeight()
setSize(int x)
getSize()
```

בנוסף, מתודה שמחזירה האם הצומת הינו "אמיתי" או מדומה: `isRealNode()`

מסמך תיעוד – TreeList

<u>שדות המחלקה:</u>

tree (עץ AVL) - טיפוס מסוג AVLTree

mockAVLNode (צומת מדומה) – טיפוס מסוג AVLNode

<u>בנאי המחלקה:</u>

TreeList():

בנאי אשר מאתחל את העץ ואת הצומת המדומה לפי האתחול הקיים במחלקת AVLTree.

<u>מתודות המחלקה:</u>

Item retrieve(int i):

מחזיר איבר ברשימה העצית באינדקס i, אם קיים. לשם כך, ניעזר בפונקציה treeSelect שמחזירה את האיבר בעל הדרגה i+1 בעץ.

סיבוכיות זמן ריצה: O(log n)

int insert(int i, int k, String s):

ביצוע של הכנסה לרשימה העצית באינדקס i על-פי חלוקה לשני מקרים:

1) האיבר אותו רוצים להכניס נכנס **לסוף הרשימה** – קריאה לפונקציה insertLast
2) האיבר מוכנס לאינדקס $0 \leq i < n$ – במקרה זה יש חלוקה לשתי אופציות:
   a. במידה ולא קיים בן שמאלי לאיבר באינדקס i – נכניס את האיבר החדש כבן <u>שמאלי</u> שלו.
   b. במידה וקיים בן שמאלי לאיבר באינדקס i – נמצא ראשית את האיבר הקודם לו (איבר באינדקס i-1) ולאחר מכן, נכניס את האיבר החדש כבן <u>הימני</u> של האיבר הקודם.

ישנו שימוש במתודה treeSelect על מנת לגלות בעזרתה בתחילה את האיבר הנוכחי באינדקס i ולאחר מכן, את האיבר הקודם לאיבר זה.

לבסוף, ישנה קריאה למתודה rebalancrAfterDeletion כדי לאזן את העץ ולשמור על תקינותו.

סיבוכיות זמן ריצה: O(log n)

void insertLast (int i, int k, String s):

ראשית, אם העץ ריק או שהשורש הינו איבר מדומה – ניעזר במתודה insert של המחלקה AVLTree.

במידה ולא, המתודה מחפשת את המיקום של האיבר בעל הדרגה הכי גדולה בעץ.

לאחר מכן, מכניסה את האיבר החדש בתור בן **<u>ימני</u>** שלו. ישנו בנוסף עדכון גובה ואיזון העץ לאחר ההכנסה.

סיבוכיות זמן ריצה: O(log n)

int delete(int i):

ראשית, הפונקציה מחפשת את האיבר אותו יש למחוק מהרשימה (במידה וקיים) בעזרת הפונקציה treeSelect (עבור הדרגה i+1 = אינדקס i ברשימה העצית).

לאחר מכן, נעזרת במתודה deleteNode של מחלקת AVLTree.

סיבוכיות זמן ריצה: O(log n)

מסמך תיעוד – CircularList

<u>שדות המחלקה:</u>

array - מערך של אובייקטים מסוג Item

maxLen – גודל המערך המקסימלי (מטיפוס מספר שלם int)

start – מיקום האיבר הראשון ברשימה המעגלית (מטיפוס מספר שלם int)

length – אורך הרשימה המעגלית (מטיפוס מספר שלם int)

<u>בנאי המחלקה:</u>

CircularList (int maxLen):

מאתחל את הרשימה להיות בגודל מקסימלי maxLen.

ומאתחל את האורך ואת מיקום האיבר הראשון בתור 0.

<u>מתודות המחלקה:</u>

Item retrieve(int i):

הפונקציה מחזירה את האיבר באינדקס i ברשימה.

סיבוכיות זמן ריצה: O(1)

int insert(int i, int k, String s):

הפונקציה מכניסה איבר לרשימה לפי חלוקה למקרים:

1. האיבר במיקום האחרון ברשימה המעגלית – ניעזר בפונקציה insertLast
2. האיבר במיקום הראשון ברשימה המעגלית – ניעזר בפונקציה insertFirst
3. $0 < i < n$ - נחלק לשני תתי-מקרים:
   a. $\boldsymbol{i < n - i}$ – הזזת i איברים מיקום אחד **<u>שמאלה</u>** ברשימה והכנסת האיבר החדש
   b. $\boldsymbol{i \geq n - i}$ – הזזת i איברים מיקום אחד **<u>ימינה</u>** ברשימה והכנסת האיבר החדש

לפיכך, ניתוח סיבוכיות הזמן תלוי באינדקס ההכנסה i ולכן –

סיבוכיות זמן ריצה: O(min{i+1, n–i+1})

void insertLast (int k, String s):

הכנסת האיבר במיקום האחרון ברשימה. ניתן לגשת לאיבר זה במספר פעולות קבוע על-ידי המצביעים length ו- start.

סיבוכיות זמן ריצה: O(1)

void insertFirst (int k, String s):

הכנסת האיבר במיקום הראשון ברשימה. מתוחזק מצביע למיקום זה ולכן –

סיבוכיות זמן ריצה: O(1)

int delete(int i):

הפונקציה מוחקת איבר ברשימה לפי חלוקה למקרים:

1. מחיקת האיבר במיקום האחרון ברשימה המעגלית – ניעזר בפונקציה deleteLast
2. מחיקת האיבר במיקום הראשון ברשימה המעגלית – ניעזר בפונקציה deleteFirst
3. $0 < i < n$ - נחלק לשני תתי-מקרים:
   a. $\boldsymbol{i < n - i}$ – מחיקת האיבר באינדקס i על-ידי הזזת i איברים מיקום אחד **<u>ימינה</u>** ברשימה ועדכון מצביעים.
   b. $\boldsymbol{i \geq n - i}$ – מחיקת האיבר באינדקס i על-ידי הזזת i איברים מיקום אחד **<u>שמאלה</u>** ברשימה ועדכון מצביעים.

לפיכך, ניתוח סיבוכיות הזמן תלוי באינדקס המחיקה i ולכן –

סיבוכיות זמן ריצה: O(min{i+1, n–i+1})

void deleteLast():

מחיקת האיבר במיקום האחרון ברשימה. ניתן לגשת לאיבר זה במספר פעולות קבוע על-ידי המצביעים length ו- start.

סיבוכיות זמן ריצה: O(1)

void deleteFirst():

מחיקת האיבר במיקום הראשון ברשימה. מתוחזק מצביע למיקום זה ולכן –

סיבוכיות זמן ריצה: O(1)

מסמך תיעוד – Item

שדות המחלקה:

key (מפתח) - מטיפוס **מספר שלם** (int)

info (מידע) – מטיפוס **מחרוזת** (String)

בנאי המחלקה:

```
Item (int key, String info)
```

בנאי אשר מאתחל את ערכי המפתח והמידע של האובייקט.

מתודות המחלקה:

במחלקה זו יש בסך הכל 2 מתודות Get:

```
int getKey()
```:

מחזירה את ערך המפתח.

```
String getInfo()
```:

מחזירה את ערך המידע.

מדידות

ניסוי 1: יתרון של רשימה מעגלית על פני רשימה עצית

מספר סידורי	מספר פעולות	זמן הכנסה ממוצע עבור רשימה מעגלית **(ננו-שניות)**	זמן הכנסה ממוצע עבור רשימה עצית **(ננו-שניות)**	כמות גלגולים **ימינה** ממוצעת עבור רשימה עצית	כמות גלגולים **שמאלה** ממוצעת עבור רשימה עצית
1	10,000	1.9E-8	1.76E-7	0.9986	0.0
2	20,000	7.5E-9	1.435E-7	0.99925	0.0
3	30,000	6.333E-9	1.3966E-7	0.9995	0.0
4	40,000	4.75E-9	1.4325E-7	0.9996	0.0
5	50,000	7.0E-9	1.424E-7	0.99968	0.0
6	60,000	5.6667E-9	1.6116E-7	0.9997333	0.0
7	70,000	6.0E-9	1.52E-7	0.999757	0.0
8	80,000	4.875E-9	1.48875E-7	0.9997875	0.0
9	90,000	6.0E-9	1.4844E-7	0.999811	0.0
10	100,000	5.3E-9	1.49599E-7	0.99983	0.0

הסבר התוצאות:

בניסוי זה בחרנו את מיקום ההכנסה בתור **האינדקס הראשון ברשימה** (המקרה עבור האינדקס האחרון סימטרי) – תקף לכל האיברים המוכנסים.

בחרנו את מיקום זה מכיוון שלרשימה המעגלית ישנה **עדיפות** מבחינת סיבוכיות הזמן עבור הכנסה למקום הראשון/אחרון ברשימה על פני הרשימה העצית.

סיבוכיות של **O(1) (רשימה מעגלית)** אל מול סיבוכיות של **O(log n) (רשימה עצית)** עבור פעולת הכנסה בודדת.

ברשימה המעגלית, משמעות ההכנסה בתחילת הרשימה היא שלא נצטרך להזיז שום איבר ברשימה. מכאן, זמן ההכנסה הממוצע אמור להישאר **קבוע**, כי כל הכנסה לרשימה מתבצעת בסיבוכיות של O(1).

ברשימה העצית, בכל הכנסה נצטרך לרדת במורד העץ עד הצומת השמאלי ביותר. כלומר, כל הכנסה מתבצעת בסיבוכיות זמן של O(log n) כאשר: n = מספר הצמתים בעץ.

מספר הגלגולים:

נצפה שכמעט בכל הכנסה, ברשימה העצית יהיה **גלגול אחד ימינה** (פרט לשתי ההכנסות הראשונות בכל סדרת הכנסות). זאת כיוון שכל איבר שנכנס לעץ, ייכנס לעלה השמאלי ביותר.

מכאן נצפה שמספר הגלגולים שמאלה יהיה **אפס** ומספר הגלגולים ימינה יישאף ל-**1** ככל שנגדיל את מספר ההכנסות לרשימה העצית.

לפיכך, נצפה שזמן ההכנסה עבור רשימה עצית יהיה בפועל **ארוך** יותר.

ניסוי 2: יתרון של רשימה עצית על פני רשימה מעגלית

מספר סידורי	מספר פעולות	זמן הכנסה ממוצע עבור רשימה מעגלית **(ננו-שניות)**	זמן הכנסה ממוצע עבור רשימה עצית **(ננו-שניות)**	כמות גלגולים **ימינה** ממוצעת עבור רשימה עצית	כמות גלגולים **שמאלה** ממוצעת עבור רשימה עצית
1	10,000	1.67E-5	1.9E-6	0.8103	0.8118
2	20,000	4.02E-5	9.0E-7	0.8115	0.81195
3	30,000	4.5466E-5	3.333E-7	0.811633	0.812266
4	40,000	6.0075E-5	3.0E-7	0.8117	0.8125
5	50,000	8.704E-5	3.2E-7	0.81202	0.81232
6	60,000	9.38667E-5	2.8333E-7	0.81208	0.812333
7	70,000	1.1287E-4	2.857E-7	0.812128	0.8123857
8	80,000	1.399E-4	3.625E-7	0.8121875	0.8123875
9	90,000	1.58688E-4	3.1111E-7	0.8121777	0.8124222
10	100,000	1.821E-4	3.5E-7	0.81221	0.81241

הסבר התוצאות:

בניסוי זה בחרנו את מיקום ההכנסה בתור **האינדקס האמצעי ברשימה** – תקף לכל האיברים המוכנסים (מתעדכן תוך כדי הריצה).

בחרנו את מיקום זה מכיוון שלרשימה העצית ישנה **עדיפות** מבחינת סיבוכיות הזמן עבור הכנסה למקום האמצעי ברשימה על פני הרשימה המעגלית.

סיבוכיות של **O(log n) (רשימה עצית)** אל מול סיבוכיות של **O(n) (רשימה מעגלית)** עבור פעולת הכנסה בודדת.

ברשימה המעגלית, משמעות ההכנסה באמצע הרשימה היא שנצטרך להזיז בכל הכנסה כמחצית מאיברי הרשימה ($\frac{n}{2}$ איברים). זוהי למעשה פעולת הכנסה בסיבוכיות של O(n).

ברשימה העצית, הסיבוכיות אינה תלויה במיקום ההכנסה (ישנו שימוש בפונקציית treeSelect אשר מחפשת את הצומת במיקום בו נרצה להכניס את הצומת החדש) ובממוצע הינה O(log n). כך גם למעשה במצב זה בו האיבר מוכנס למיקום האמצעי ברשימה. כלומר, כל הכנסה מתבצעת בסיבוכיות זמן של O(log n) כאשר: n = מספר הצמתים בעץ.

מספר הגלגולים:

לאחר ביצוע מדידות ובדיקת כמות הגלגולים הכוללת עבור כל מדידה, ניתן היה להבחין כי ישנם 1 או 2 גלגולים.

לכן, **בממוצע מספר הגלגולים** עבור סדרת ניסויים כלשהי הינה **1.5 גלגולים**.

יתר על כן, משום שאנו נדרשים לבדוק בנפרד את הגלגולים ימינה ואת הגלגולים שמאלה, ובניסוי זה המקרים הינם **סימטריים** – נצפה לקבל כי כמות הגלגולים ימינה/שמאלה הינה מחצית ממוצע הגלגולים הכולל.

מסקנה: כ-0.8 גלגולים עבור כל סוג גלגול.

ניסוי 3: מיקום הכנסה בהתפלגות אחידה

מספר סידורי	מספר פעולות	זמן הכנסה ממוצע עבור רשימה מעגלית **(ננו-שניות)**	זמן הכנסה ממוצע עבור רשימה עצית **(ננו-שניות)**	כמות גלגולים **ימינה** ממוצעת עבור רשימה עצית	כמות גלגולים **שמאלה** ממוצעת עבור רשימה עצית
1	10,000	9.58E-6	4.9E-7	0.35009	0.34923
2	20,000	1.9065E-5	4.15E-7	0.34929	0.35015
3	30,000	2.901666E-5	4.4E-7	0.34907666	0.3498533
4	40,000	4.67025E-5	4.0E-7	0.349735	0.3500625
5	50,000	6.278E-5	4.38E-7	0.349774	0.34948
6	60,000	7.901666E-5	4.316667E-7	0.34726166	0.3479833
7	70,000	2.9405E-4	5.02857E-7	0.3496157	0.34949
8	80,000	9.88125E-5	4.9E-7	0.34891	0.35014125
9	90,000	1.1803555E-4	5.2444E-7	0.34993111	0.3488422
10	100,000	1.13156E-4	4.82E-7	0.349389	0.349285

הסבר התוצאות:

בניסוי זה בחרנו את מיקום ההכנסה בתצורה של **התפלגות אחידה**.

כלומר, אם יש ברשימה N איברים, מיקום ההכנסה של האיבר ה-N+1 יוגרל בצורה אחידה בטווח [0,N].

על סמך התיאוריה לסיבוכיות פעולות ההכנסה ברשימה עצית ורשימה מעגלית, אנו מצפים כי זמני הריצה עבור רשימה עצית יהיו קצרים יותר מאשר זמני הריצה עבור רשימה מעגלית.

ההסבר לכך:

ברשימה מעגלית, כל עוד ההכנסה אינה לקצוות הרשימה, סיבוכיות הזמן של מתודת ההכנסה היא בממוצע גדולה יותר מאשר O(log n). ניתן אף לומר כי היא שואפת לסיבוכיות זמן של O(n).

זאת להבדיל מסיבוכיות הזמן של פעולת הכנסה ברשימה עצית אשר תמיד בעלת סיבוכיות זמן של O(log n).

ולסיכום:

סיבוכיות של **O(log n) (רשימה עצית)** אל מול סיבוכיות של **O(n) (רשימה מעגלית)** **בממוצע** עבור פעולת הכנסה בודדת.

ניתן לראות על-פי נתוני הטבלה כי הציפיות שלנו התממשו - זמני הריצה הממוצעים של ההכנסות בסדרות הניסויים השונות אכן עונים על התיאוריה.

כלומר, זמני הריצה של הכנסות ברשימה עצית **משמעותית קצרים יותר** מאשר זמני הריצה של אותן ההכנסות ברשימה מעגלית.