מגישים: מתן בן טוב, שיר פרנקל

מבני נתונים – פרויקט 1

חלק א' – תיעוד הקוד

תיעוד מחלקה - Item

1. **תיאור כללי:** מחלקה שמייצרת מופעים המכילים שני רכיבי נתונים – key מסוג מספר שלם ו-info מסוג מחרוזת. איברי מחלקה זו יהיו האלמנטים במימושי הרשימות שנציג.
2. **שדות המחלקה:**
   א. **key:** מספר שלם המייצג את המפתח של ה-Item.
   ב. **info:** מחרוזת שמכילה את המידע של ה-Item.
3. **מתודות המחלקה:**
   שתי המתודות פועלות בסיבוכיות $O(1)$.
   א. **getKey:** מחזיר את שדה key.
   ב. **getInfo:** מחזיר את שדה info.

תיעוד מחלקה - CircularArray

1. **תיאור כללי:** מימוש ה-ADT רשימה באמצעות מערך מעגלי, כך שאיברי הרשימה יהיו מוכלים במערך בסידור מעגלי רציף.
   הרעיון – באמצעות המבנה המעגלי יהיה ניתן לבחור במקרים מסוימים מאיזה 'צד' לעבד את המערך ובכך לייעל את שינויי הרשימה.
   המימוש – נשמור את הרשימה במערך ונחזיק בכל עת מצביע לאיבר הראשון ברשימה, נשתמש בחישובים בשדה מודולו maxLen כדי להתייחס למערך כמעגלי.

2. **שדות המחלקה:**
   א. **maxLen:** מספר שלם המייצג את אורך המערך באמצעותו ממומשת הרשימה. מספר זה הינו כמות האיברים המקסימלית ברשימה ומוגדר ע"י המשתמש.
   ב. **array:** מערך מטיפוס []item באורך maxLen המכיל בתוכו את איברי הרשימה מסוג item.
   ג. **start:** שדה 'המצביע' (=אינדקס) לאיבר הראשון ברשימה במערך array.
   ד. **len:** אורך הרשימה. מתקיים תמיד $len \leq maxLen$.

3. **מתודות המחלקה:**
   א. <u>CircularList (int maxLen)</u>:
      1) **הסבר:** הבנאי (היחיד) של המחלקה, מאתחל את המערך array, מאפס את len ו-start ומבצע השמה ל-maxLen.
      2) **הנחות:** $0 \leq maxLen$.
      3) **סיבוכיות:** $O(1)$. [ייתכן והסיבכיות לינארית לגודל המערך, הדבר **תלוי** במימוש של ג'אווה לאיתחול המערך array].

ב. Retrieve (int i):

1) **הסבר:** גישה לאיבר במיקום ה-i ברשימה ע"י חישוב אריתמטי במודולו maxLen. אם i לא מקיים $0 \le i < len$ אין משמעות לקלט ויוחזר null.
2) **סיבוכיות:** $O(1)$. גישה למערך.

ג. Insert (int i):

1) **הסבר:** הכנסת איבר למיקום ה-i ברשימה. ההכנסה תיעשה ע"י חלוקה למקרים לפי כמות האיברים ברשימה לפני ואחרי האינדקס ה-i. ייבחר החלק הקטן יותר ברשימה ואיבריו ימוקמו מחדש בהזזה **מעגלית** כך שיתפנה מקום במערך לאיבר החדש. ההזזה המעגלית מתאפשרת בעזרת חישובים מעל מודולו maxLen. נעדכן את המצביע לתחילת הרשימה במידת הצורך (במקרה בו ההזזה המעגלית שינתה אותו) ונעדכן את אורך הרשימה. עם סיום הכנסה מוצלחת יוחזר הערך (0).
2) **מקרי קצה:** האינדקס הנתון אינו חוקי (לא בגבולות הרשימה), המערך arrays הינו מלא, במקרים אלה יוחזר (-1).
3) **סיבוכיות:** $O(min\{i+1, len-i+1\})$. כאשר הסיבוכיות נובעת מכמות ההזזות שנזדקק אליהן לשם ההכנסה, כמו-כן מכיוון שכל הזזה מתבצעת בזמן קבוע ובחרנו את האופציה המינימלית.

ד. Delete (int i):

1) **הסבר:** מחיקת האיבר ה-i ברשימה. המחיקה, כמו ההכנסה, תיעשה ע"י חלוקה למקרים לפי כמות האיברים ברשימה לפני ואחרי האינדקס ה-i. למשל, אם נבחר החלק שאחרי (מימין, או עם כיוון השעון במונח מעגלי) האינדקס ה-i, אז נזיז את כלל האיברים הללו שמאלה כך שהאיבר ה-i+1 ייהפך להיות האיבר ה-i. נבחין כי נוצר לנו איבר מיותר במערך ונבצע השמה ל-null במקום זה. נעדכן את אורך הרשימה ובמידת הצורך את המצביע לתחילתה.
2) **מקרי קצה:** האינדקס הנתון אינו חוקי (לא אינדקס של איבר ברשימה), במקרה זה יוחזר (-1).
3) **סיבוכיות:** $O(min\{i+1, len-i+1\})$. באותו אופן כמו בהכנסה, כאשר הסיבוכיות נובעת מכמות ההזזות שנזדקק אליהן לשם המחיקה.

ה. הערה: במקרים הפרטיים של מחיקה/הכנסה לתחילת הרשימה או סופה סיבוכיות הפונקציות תהיה $O(1)$.

תיעוד מחלקה – AVLNode

1. **תיאור כללי:** מחלקה שהיא חלק ממימוש עץ AVL. מופע של המחלקה מייצג איבר בעץ AVL, דהיינו צומת. המחלקה כוללת את כלל השדות שדרשנו מצומת במימוש עץ AVL ומספר פעולות שימושיות שניתן לבצע על צומת, בעיקר של חישובים בזמן קבוע.
מימוש המחלקה נעשה לאור הצרכים שעלו במימוש מחלקת AVLTree ועל כן גם הפונקציות הנוספות שמימשנו בה.
המחלקה מממשת את הממשק הנתון במטלה IAVLNode.

2. **שדות המחלקה:**

א. **node_item:** איבר מטיפוס Item ששומר את מפתח הצומת (key) וערכה (value).

ב. **parent, left, right:** מצביעים לצמתים (מאותו טיפוס) המייצגים את האב, הילד השמאלי והילד הימני (בהתאמה). במידה ואחד מהם לא קיים, ערך אותו מצביע יהיה null.

ג. **height:** מספר שלם המייצג את גובה הצומת כאיבר בתוך העץ, כך שגובה עלה מוגדר להיות 0 וגובה עץ ריק 1-.

ד. **size:** מספר שלם המייצג את כמות הצמתים בתת העץ שנוצר ע"י הצומת (=בו היא השורש), כך שגודל עלה מוגדר להיות 1 וגודל עץ ריק 0.

3. **מתודות המחלקה:**

ו. AVLNode (int k,string s):

1) **הסבר:** בנאי במחלקה המאתחל את הצומת ומאתחל את ערכי המפתח והערך שלה לפי הקלט. הגובה מאותחל ל-0 והגודל ל-1 (עץ שמורכב מצומת בודד
קיים בנאי נוסף שמקבל בנוסף את הצומת המוגדרת להיות האב.

2) **סיבוכיות:** $\boldsymbol{O(1)}$.

ז. getKey, getVal, getLeft, getRight ,getParent, getHeight, getSize, getItem

3) **הסבר:** מתודות "getters" המאפשרות לאחזר כל שדה של הצומת בהתאמה.
קיימים גם "setters" מקבילים.
בהפעלת פעולות בגורמות לשינוי בנים (למשל setLeft) יעודכנו שדות הגובה והגודל בהתאם לשינוי (ע"י הפונקציות שיפורטו בסעיף ד').

4) **סיבוכיות:** $\boldsymbol{O(1)}$.

ח. getBF():

5) **הסבר:** חישוב ה-balance-factor של הצומת.

6) **סיבוכיות:** $\boldsymbol{O(1)}$, גישה לשדות גובה של הילדים וחישוב אריתמטי בזמן קבוע.

ט. updateHeight(), updateSize():

7) **הסבר:** עדכון הגובה והגודל של הצומת (בהתאמה) כך ששדות אלה יהיו בהלימה לשדות הילדים של הצומת.

8) **סיבוכיות:** $\boldsymbol{O(1)}$, גישה לשדות המקבילים של הילדים בעזרת פונקציות העזר getChildrenHeights, getChildrenSizes וחישוב אריתמטי בזמן קבוע.

ב. פונקציות עזר במחלקה:

1) **הסבר:** ע"מ לאפשר את רציפות הקוד והימנעות מקוד חוזר מימשנו במהלך הכתיבה מספר פונקציות שאמנם מבצעות פעולות טריוויאליות על צומת אך שימושיות – למשל בדיקה האם הצומת הוא עלה, איחזור הילד היחיד של הצומת וכו'.

2) **רשימת הפונקציות:**

getChildrenHeights, getChildrenSizes, setSide, getSide, isLeaf, hasOneChild, getSideOf, getOnlyChild, getSubtreeRank

הערה: פונקציות שפועלות עם השדה הבוליאני side מתייחסות לצדדים בצורה הבאה: true = ימין, false = שמאל.

3) **סיבוכיות:** $\boldsymbol{O(1)}$. גישה לשדות הצומת / ילדיה וחישובים אריתמטיים.

תיעוד מחלקה – AVLTree

1. **תיאור כללי:** מימוש ADT מילון באמצעות עץ AVL. המחלקה המממשת את מבנה הנתונים עץ AVL, מסוג עץ דרגות, אשר איבריו (צמתיו) עצמים ממחלקה AVLNode.
נסמן בסעיף זה: $n \coloneqq number\ of\ nodes\ in\ an\ AVL\ tree$.

2. **שדות המחלקה:**
   א. **root:** מצביע לצומת העץ, אובייקט מטיפוס AVLNode.
   ב. **min, max:** מצביעים לצומת עם המפתח המינימלי והמקסימלי בעץ, בהתאמה. שדות אלו מאותחלים ל-null ומתוחזקים במהלך פעולות הכנסה ומחיקה.

3. **מתודות המחלקה:**

   א. AVLTree():
   1) **הסבר:** בנאי במחלקה המאתחל את כל השדות ל-null, כלומר יוצר עץ חדש ריק.
   2) **סיבוכיות:** $\boldsymbol{O(1)}$.

   ב. empty ():
   1) **הסבר:** מחזיר האם העץ ריק ע"י בדיקת שדה השורש (ערך בוליאני).
   2) **סיבוכיות:** $\boldsymbol{O(1)}$.

   ג. getSize ,getRoot ,getMin ,getMax:
   1) **הסבר:** מתודות "getters" המאפשרות לאחזר כל שדה (או שדה פנימי שלו כעצם במחלקה AVLNode בעץ) של הצומת בהתאמה. כאשר גודל העץ נתון בגודל שורש העץ.
   2) **סיבוכיות:** $\boldsymbol{O(1)}$.

   ד. search (int k):
   1) **הסבר:** חיפוש צומת עם המפתח ה-k ע"י קריאה לפונקציה findbykey (שתפורט בסעיף הבא) והחזרת **ערך** הצומת, במידה ונמצא בעץ.
   2) **פונקציית עזר – findbykey (int k)**: מימוש איטרטיבי של חיפוש בעץ-חיפוש-בינארי, סיור במורד העץ מהשורש (לפי ערך המפתח) ועצירה בצומת בעל מפתח זה או בעלה. מחזירה את הצומת עם המפתח ה-k אם קיים כזה, אחרת מחזירה **עלה** שיכול להיות אבא של צומת עם מפתח כזה.
   במקרה הגרוע נסייר מהשורש לעלה ולכן סיבוכיות הפונקציה $\boldsymbol{O(log\ n)}$.
   3) **סיבוכיות:** $\boldsymbol{O(log\ n)}$, לפי סיבוכיות פונקציית העזר.

   ה. insert(int k, String val):
   1) **הסבר:** ביצוע הכנסה של צומת חדשה.
   נעזרת בפונקציית העזר findbykey (המפורטת לעיל) ע"מ לוודא שהמפתח אינו קיים בעץ, ולבצע הכנסה לצומת המתאימה. במקרה של עץ ריק נגדיר צומת זו להיות השורש. במקרה והצומת החדשה מהווה צומת מקסימלית/מינימלית שדות אלו יעודכנו בהתאמה. מכיוון שזו הכנסה 'נאיבית' לעץ חיפוש בינארית, תבצע פונקציית העזר rebalanceUpwards את פעולות האיזון הדרושות מהצומת החדש לשורש, כמו גם את עדכוני השדות (גובה, גודל) בצמתים שהושפעו מההכנסה ונחזיר את כמות הרוטציות שבוצעו במסגרתה.

2) **סיבוכיות:** $O(log\ n)$. סיבוכיות שתי פונקציות העזר, כפי שטענו, היא $O(log\ n)$ במקרה הגרוע, לכן סיבוכיות ההכנסה תהיה הסיבוכיות המקסימלית.

ו. <u>rebalanceUpwards (AVLNode node)</u>:

1) **הסבר:** פונקציה זו תיקרא לאחר הכנסה / מחיקה של צומת, על מנת לשמור על איזון העץ, כמו כן כדי לתחזק את שדות הגובה והגודל שבצמתים (ע״מ להבטיח גישה אליהם בזמן קבוע).
בהינתן צומת, הפונקציה תבדוק את המסלול מהצומת עד לשורש העץ, כך שיבוצע:

- עדכון גובה וגודל הצומת (בהתאם לילדיו) בזמן קבוע.
- בדיקת גורם האיזון של הצומת, וביצוע רוטציות על AVL Criminals במידת הצורך. כל רוטציה שבוצעה תימנה ומנייה זאת תוחזר ע״י הפונקציה. בחירת הרוטציה המתאימה תיעשה משיקולים שראינו והוכחנו בכיתה.

הפונקציה תחזיר את כמות הרוטציות שבוצעו במהלכה.

2) **פונקציות עזר - roateRight ,rotateLeft:** מקבלות צומת ומבצעות עליו רוטציה בכיוון המתאים, דהיינו מעדכנות את כלל השדות הרלוונטים של הצמתים שהושפעו מהרוטציה.
הפונקציות יחזירו את הצומת שהחליפה את תפקידו של מושא הרוטציה (כלומר הצומת שנהפכה לשורש תת העץ הנתון כקלט).
סיבוכיות פונקציית העזר: $O(1)$. מכיוון שמדובר במספר קבוע של צמתים שהושפעו מהרוטציה, סיבוכיות הזמן של הפונקציה קבועה.

3) **סיבוכיות:** $O(log\ n)$. בכל מקרה תעלה הפונקציה מהצומת הנתון, במעלה העץ ועד לשורש. במקרה הגרוע הצומת הנתון הוא עלה.

ז. <u>delete (int k)</u>:

1) **הסבר:** מחיקת איבר בעל המפתח k בעץ, אם הוא קיים. הפונקציה מחזירה את מספר פעולות האיזון שנדרשו בסה״כ בשלב תיקון העץ על מנת להשלים את הפעולה. אם לא קיים איבר בעל המפתח k בעץ הפונקציה תחזיר 1-.
הפונקציה משתמשת בפונקציה findByKey ע״מ למצוא את הצומת המועד למחיקה. אם הצוומת לא קיים, מוחזר 1-. אם הצומת אכן קיים, נקראת הפונקציה deleteNode שמקבלת את הצומת שצריך למחוק ומבצעת את תהליך המחיקה לפי חלוקה למקרים ע״פ מספר הבנים של הצומת שמוחקים (כפי שראינו בהרצאה ע״י שינוי כמות קבועה של מצביעים בכל מקרה). לאחר המחיקה, שתבוצע בפונקציית עזר המתאימה למקרה, תיקרא פונקציית העזר rebalanceUpwards על הורה הצומת שנמחקה פיזית ותעדכן את שדות הצמתים שהושפעו מהמחיקה. פונקציית העזר תמנה את כמות הרוטציות שביצעה ו-deleteNode תחזיר ערך זה לdelete שגם תחזירו. בנוסף על כך, במידת הצורך לאור המחיקה, יעודכנו השדות min, max באמצעות פונקציות העזרgetSuccessor, getPredecessor.

2) **פונקציות עזר:**

- **deleteNode** – מקבלת צומת שצריך למחוק, בודקת כמה ילדים יש לצומת ולפי זה מבצעת פעולות מתאימות וקריאה לפונקציית מחיקה רלוונטית (deleteLeaf, deleteOneChild). פונקציות המחיקה מתקנות את העץ ומחזירות לפנקציה deleteNode את כמות ההיפוכים שנדרשו והיא מחזירה גם היא את כמות ההיפוכים.

  סיבוכיות: $O(logn)$ מכיוון שקוראת לפונקציה deleteLeaf או deleteOneChild

  הערה: במקרה שבו יש לצומת שצריך למחוק 2 ילדים, הפונקציה מוצאת את

ה-Successor של הצומת והוא ימחק ממקומו הנוכחי ויחליף את הצומת הנ"ל. הפונקציה מניחה של-sucsessor יש לכל היותר ילד אחד ופועלת לפי שני המקרים האפשריים (ה-successor עלה / בעל ילד אחד). ניתן להבין את הסיבה לכך שיש ל-successor לכל היותר בן אחד לפי האלגוריתם למציאת ה-successor של צומת שיש לה בן ימני: בשביל למצוא את העוקב במקרה זה פונים ימינה מהצומת ומשם ממשיכים שמאלה כל עוד ניתן. לכן בהכרח אין ל successor בן שמאלי.

- **deleteLeaf** – פונקציה שמוחקת צומת במקרה בו הוא עלה, מחזירה את כמות ההיפוכים שנדרשו. משתמשת בפונקציית עזר disconnect שמנתקת את הצמתים המבוקשים (שינוי pointers).

  סיבוכיות: $\boldsymbol{O(logn)}$ מכיוון שקוראת לפונקציה rebalanceUpwards.

- **disconnect (AVLNode parent, boolean side)**
  הפונקציה מנתקת את parent מהילד שלו בצד ה-side (הצד מיוצג כערך בוליאני- אמת = ימין, שקר = שמאל). היא עושה זאת ע"י שינוי ערך המצביעים הרלוונטיים של הצמתים שמנותקים ל-null בעזרת פונקציות עזר של המחלקה AVLNode.

  סיבוכיות: $\boldsymbol{O(1)}$.

- **deleteNodeOneChild** - פונקציה שמוחקת צומת במקרה בו הוא יש לצומת בן יחיד, מחזירה את כמו ההיפוכים שנדרשו. משתמשת בפונקציה rebalanceUpwards כדי לתקן את העץ, connect לשינוי המצביעים הרלוונטיים ובפונקציות עזר של המחלקה AVLNode.

  סיבוכיות: $\boldsymbol{O(logn)}$ מכיוון שקוראת לפונקציה rebalanceUpwards.

- **connect(AVLNode parent, boolean side, AVLNode newChild)**
  משנה את המצביעים הרלוונטיים כך שparent יהיה ההורה של newChild בצד המבוקש (הצד מיוצג כערך בוליאני- אמת = ימין, שקר = שמאל), כמו כן אם היה קודם לכן ילד אחר במקום הנ"ל, הוא מנותק מparent (כלומר מצביע הparent שלו הופך ל-null).

  סיבוכיות: $\boldsymbol{O(1)}$.

3) **סיבוכיות:** $\boldsymbol{O(logn)}$, כסכום סיבוכיות פונקציות העזר.

ח. keyToArray(AVLNode node), infoToArray(AVLNode node):

1) **הסבר:** הפונקציה מחזירה מערך מספרים/מחרוזות המכיל את כל המפתחות/הערכים בעץ, ממוינים על פי סדר המפתחות, או מערך ריק אם העץ ריק. כל אחת מהפונקציות קוראת לפונקציה רקורסיבית מתאימה שמחזירה את המערך המבוקש.
   אופן הפעולה של הפונקציה הרקורסיבית: הפונקציה קוראת לעצמה בכדי לחשב את המערך הנוצר מהבן הימני של הצומת שנקראה ומהבן השמאלי.
   היא מחזירה מעין איחוד של המערכים האלו כך שקודם יהיה המערך שנוצר מהבן השמאלי, לאחר מכן יהיה המפתח/הערך של הצומת עצמה ואז המערך שנוצר מהבן הימני.
2) **סיבוכיות:** $\boldsymbol{O(n)}$. הפונקציה הרקורסיבית נקראת עם כל צומת שבעץ פעם אחת בדיוק.

ט. getSuccessor(AVLNode node), getPredecessor:

1) **הסבר - getSuccessor:** מחזיר את העוקב של node אם קיים (אחרת יוחזר null).
   דרך פעולה- אם יש לnode בן ימני, הפונקציה תלך לבן הימני וממנו בצורה איטרטיבית תתקדם לבן השמאלי של אותו בן עד שתגיע לבן השמאלי האחרון שקיים במסלול זה. בן זה יהיה העוקב של node.

אם ל node אין בן ימני, באופן איטרטיבי הפונקציה תעלה מ node עד שתגיע לפניה ימינה. הצומת שאליה הפונקציה הגיעה לאחר הפניה ימינה הוא העוקב של node.
***באופן דומה ממומש getPredecessor** (מציאת הקודם של node) רק שהכיוונים בדיוק הפוכים.

2) **סיבוכיות:** $O(logn)$ במקרה הגרוע בכל רמה מבקרים לכל היותר פעם אחת.

י. <u>min(), max()</u>:

1) **הסבר:** מחזירות את הערך (=String) של הצומת המקסימלי / מינימלי ע״י גישה לשדות min ו-max ואחזור ערכם בזמן קבוע או null אם העץ ריק.
2) **סיבוכיות:** $O(1)$.
3) **הערה:** סיבוכיות איחזור מינימום ומקסימום הינה בזמן קבוע בזכות תיחזוק שדות אלה במהלך הפונקציות השונות במחלקה, תחזוק זה גורם אך ורק לגידול של סיבוכיות פונקציות אלה **בקבוע**. אם כן היינו רוצים להימנע מגידול זה, ומבחינתנו גישה מהירה למיני׳ ומקסי׳ מיותרת, היה ניתן לממש את הפונקציות הנ״ל ע״י מציאת הצומת הקיצוני משמאל / מימין בהתאמה בסיבוכיות לוגריתמית במקרה הגרוע.

הפונקציות הבאות בעיקר נועדו למימוש TreeList:

יא. <u>SelectNodeByRank(int k)</u>:

1) **הסבר:** הפונקציה תחזיר את הצומת בדרגה ה-k ע״י מימוש איטרטיבי של הפונקציה select שראינו בכיתה. תסייר מהשורש במורד העץ ותרד בכל צומת בהתאם לדרגתו ביחס לבניו השמאליים (חישוב בזמן קבוע). כאשר תפגוש בצומת עם הדרגה המבוקשת תחזיר אותו. אם תגיע לצומת ריק לפני שמצאה את הדרגה המתאימה תחזיר null (=אחת ההנחות הופרה).
2) **הנחות:** $1 \leq k \leq this.size() \ \land \ tree \ is \ not \ empty$
3) הגדרנו גם את הפונקציה select**Item**ByRank שמשתמשת בפונקציה הזאת, אך מחזירה את תוכן הצומת, איבר מסוג Item, ע״מ שתשמש אותנו במחלקה TreeList.
4) **סיבוכיות:** $O(logn)$. במקרה הגרוע הצומת המבוקשת תהיה עלה. במקרים בהם תבוקש הדרגה ה-1 או הדרגה המקסימלית סיבוכיות הפונקציה תהיה **קבועה** (כסיבוכיות ()max, ()min).

יב. <u>insertByIndex(int i, int k, String s)</u>:

1) **הסבר:** פונקציה שנועדה לשימוש של מחלקת TreeList.
פונקציה זו לא שומרת על הסדר של העץ לפי ערכי המפתחות אלא פועלת לפי אינדקס מבוקש (כאשר $index \ of \ node \ = \ rank(node) \ - 1$).
אם $i < 0 \ \lor \ i \geq n$ הפונקציה תחזיר 1-. אחרת, הפונקציה תבצע insert במקום המתאים לפי i (כפי שראינו בכיתה). בנוסף, אם האיבר שנוסף הוא בדרגה מינימלית / מקסימלית השדות min / max יעודכנו בהתאם. בסיום הפונקציה קוראת ל-rebalanceUpwards לטובת איזון העץ ועדכון השדות שהושפעו מההכנסה, לאחר הכנסה מוצלחת יוחזר (1).
2) **סיבוכיות**: $O(logn)$. כסיבוכיות הפונקציה rebalanceUpwards.

יג. deleteByIndex(int i):
1) **הסבר:** פונקציה שנועדה לשימוש של מחלקת TreeList. (**אינה** שומרת על תכונות עץ AVL לפי מפתחות, בדומה לפונקציה הקודמת).
הפונקציה מוחקת את הצומת באינדקס i. היא מניחה ש- $0 \le i < n$ (כלומר שזהו אינדקס קיים וכמו כן מכך גם נובע שהעץ לא ריק). הפונקציה קוראת לפונקציה selectNodeByRank שמחזירה את הצומת שצריך למחוק. לאחר מכן, נקראת הפונקציה deleteNode שמוחקת את הצומת הנ"ל.
2) **סיבוכיות:** $\boldsymbol{O(logn)}$, כסיבוכיות הפונקציה deleteNode.

יד. **<u>הערה:</u>** אמנם בחרנו לממש את פונקציה *יב'* במחלקה **זו**, אך הפונקציה נועדה בעיקר לשימוש במחלקות כמו TreeList בהן עץ ה-AVL מעוצב שונה וההכנסה נעשית **לפי אינדקס** ולא לפי מפתח. על כן, מבחינתנו (כפי שגם ציינו והזהרנו בתיעוד פנימי בקוד) אנו מניחים כי כאשר נעשה שימוש ב-AVLTree כעץ הממוין לפי מפתחות, המשתמש **לא** יבצע שימוש לא חוקי בפונקציה *יב'* (=insertByIndex).

טו. **<u>הנחות עבור חישובי הסיבוכיות לעיל:</u>**
1) הסיכוביות הינה סיבוכיות worst case ההדוקה ביותר.
2) גישה לשדות של המחלקות וחישובים אריתמטיים מתבצעים בזמן קבוע.
3) גובה עץ AVL חסום ע"י $O(log\ n)$.

<u>תיעוד מחלקה – TreeList</u>

1. **<u>תיאור כללי:</u>** מימוש ADT רשימה באמצעות עץ AVL.
הרעיון - דרגת צומת בעץ מייצג אינדקס ברשימה ($index := rank - 1$)

2. **<u>שדות המחלקה:</u>**
א. **tree:** מצביע למופע ממחלקה AVLTree, העץ בו מיוצגת הרשימה.

3. **<u>מתודות המחלקה:</u>**
א. <u>TreeList():</u>
1) **הסבר:** בנאי המחלקה המאתחל את העץ המייצג את הרשימה (עץ ריק).
2) **סיבוכיות:** $\boldsymbol{O(1)}$.

ב. <u>Retrieve (int i):</u>
1) **הסבר:** גישה לאיבר במיקום ה-i ברשימה, כלומר גישה לדרגה ה-i+1 בעץ.
הפונקציה קוראת לפונקציית העזר selectItemByRank, שפירטנו לעיל, אשר מחזירה את ה-Item שמוכל בצומת מהדרגה i+1.
2) **מקרי קצה:** i לא מקיים $0 \le i < len$ (בפרט אם הרשימה ריקה) אין משמעות לקלט ויוחזר null.
3) **סיבוכיות:** $\boldsymbol{O(log\ n)}$. כסיבוכיות המקרה הגרוע ב-selectItemByRank.
סיבוכיות $O(1)$ **אם** האינדקס המבוקש הוא האיבר הראשון או האחרון ברשימה.

ג. length():

1) **הסבר:** הפונקציה מחזירה את אורך הרשימה, משתמשת בפונקצייה size() של מחלקת AVLTree.

2) **סיבוכיות:** $\boldsymbol{O(1)}$.

ד. insert(int i, int k, String s):

1) **הסבר:** הכנסת איבר בעל מפתח k וערך s לאינדקס i ברשימה. הפונקציה מחזירה -1 אם $i < 0 \lor i > n$ ולא מבוצעת הכנסה. אחרת, הפונקציה תבצע הכנסה באמצעות קריאה ל לפונקציה insertByIndex ממחלקת AVLTree.

2) **סיבוכיות:** $\boldsymbol{O(log\ n)}$, כסיבוכיות המקרה הגרוע ב- insertByIndex.

ה. delete(int i):

1) **הסבר**: מחיקת איבר באינדקס ה-i ברשימה. הפונקציה מחזירה -1 אם $i < 0$ או ש- $\mathrm{i} > \mathrm{n} - 1$. אחרת הפונקציה קוראת ל- deleteByIndex ממחלקת AVLTree שמוחקת את הצומת באינדקס i. לאחר מחיקה מוצלחת תחזיר הפונקציה 0.

2) **סיבוכיות:** $\boldsymbol{O(log\ n)}$, כסיבוכיות המקרה הגרוע ב- deleteByIndex.

חלק ב' - מדידות

1. **מדידה (1) – רשימה מעגלית < רשימה עצית:**

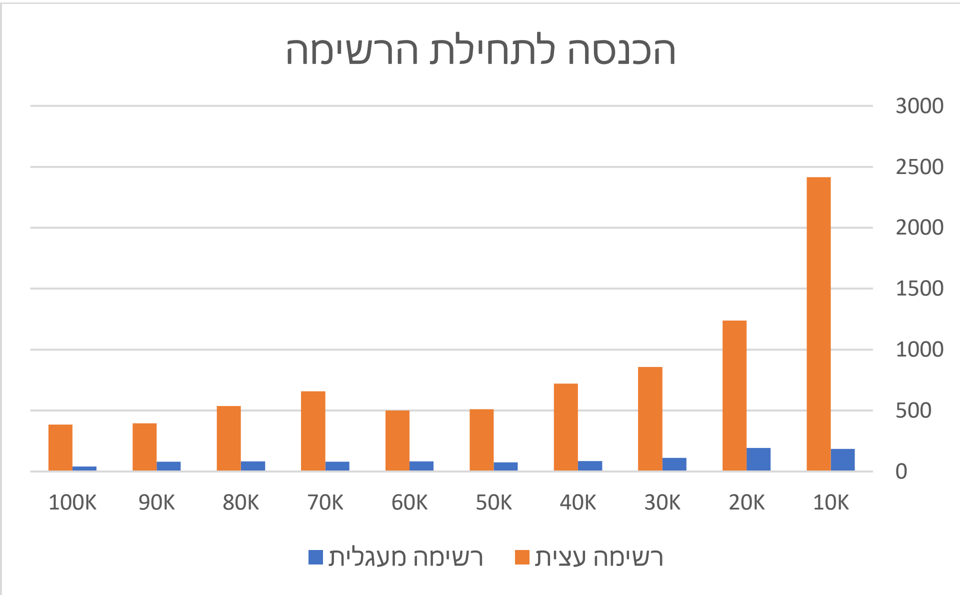
א. **הניסוי:** ע"מ להראות את יתרונה של הרשימה המעגלית, נבצע הכנסות **לתחילת** הרשימה (דהיינו כל הכנסה תהיה לאינדקס ה-0).

ב. **סיבה לבחירה:** אלו כפי שציינו בסיבוכיות זמן קבועה ברשימה מעגלית. זאת לעומת רשימה עצית, אשר בכל מצב סיבוכיותה תהיה לוגריתמית לגודל הרשימה.

ג. **תוצאות המדידה:**

מידות	מספר הכנסות	זמן הכנסה ממוצע (ננו-שניות - 10^{-9}sec)		כמות גלגולים ממוצעת (פר הכנסה, רשימה עצית)	
		רשימה מעגלית	רשימה עצית	ימינה	שמאלה
1	10K	184	2416	0.998	0
2	20K	192	1238	0.999	0
3	30K	113	858	0.999	0
4	40K	86	722	0.999	0
5	50K	76	510	0.999	0
6	60K	83	500	0.999	0
7	70K	81	659	0.999	0
8	80K	84	537	0.999	0
9	90K	80	396	0.999	0
10	100K	42	385	0.999	0

ד. **תרשים:**



ה. **תוצאות:**

1) זמן ההכנסה הממוצע של **רשימה מעגלית** בערך פי 10 יותר **מהיר** מזמן ההכנסה של רשימה עצית.
2) מתבצע בממוצע גלגול אחד להכנסה, ללא תלות במספר ההכנסות הכולל.
3) המגמה של זמן ההכנסה הממוצעת ביחס לכמות ההכנסות ברשימה העצית וכן ברשימה מעגלית היא יורדת. כלומר ככל שיש יותר הכנסות לרשימות, הזמן הממוצע להכנסה יורד.

ו. **מסקנה:**

1) כפי שצפינו, הכנסת איברים לתחילת רשימה מעגלית פעלה במהירות ממוצעת **גבוהה יותר** מאשר הכנסת אותה כמות איברים לתחילת רשימה עצית. הדבר נובע מכך שבעוד שהכנסת איברים לתחילת רשימה מעגלית פועלת **בזמן קבוע** ללא תלות בגודל הרשימה (שקולה לגישה וכתיבה למערך), ברשימה העצית קיימת **תלות בגובה** הרשימה (אשר הוא פונקציה לוגריתמית של גודלה).
2) צפינו שיהיו רק גלגולים ימינה ושכמות הגלגולים לא תהיה גדולה מ-1. זאת מכיוון שבתיקון עץ AVL לאחר הכנסה מבוצע לכל היותר גלגול אחד כפי שראינו בשיעור ובנוסף, מכיוון שמדובר תמיד בהכנסת צומת בקצה העץ השמאלי, הגיוני שמהר מאוד העץ יצא מאיזון ויידרש גלגול כדי לתקן זאת ולכן כמות הגלגולים הממוצעת קרובה ל- 1. כמו כן, מה שיפר את האיזון של העץ בניסוי זה הוא תמיד עודף איברים בתת עץ שמאלי ולכן מבוצעים רק גלגולים ימינה.
3) בנוסף, מהניסוי גילינו שככל שיש יותר הכנסות לרשימות, הזמן הממוצע להכנסה **יורד**. ניתן **לשער** ולהסביר זאת ברשימה מעגלית, בין השאר, ע"י מיצוע של קבוע מסוים על פני יותר איברים המוכנסים, וברשימה עצית ייתכן כי במשתנים כמו זיכרון, ובכל מקרה ככה"נ גם ממימושים פנימיים, שאיננו חשופים אליהם, של ג'אווה.

2. <u>**מדידה (2)**</u> – **רשימה עצית** > **רשימה מעגלית:**

א. **הניסוי:** ע"מ להראות את יתרונה של הרשימה המעגלית, נבצע הכנסות **לאמצע** הרשימה (דהיינו כל הכנסה תהיה לאינדקס ששוה למחצית מאורכה).

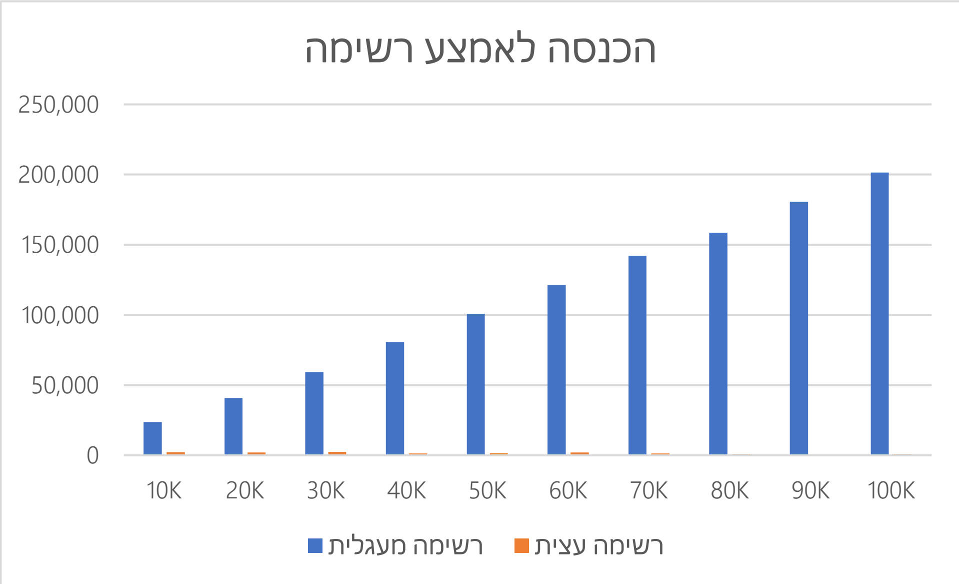
ב. **סיבה לבחירה:** הכנסות אלו יהיו בסיבוכיות לינארית לגודל הרשימה ברשימה מעגלית, לעומת רשימה עצית אשר תהיה בסיבוכיות לוגריתמית לגודל הרשימה.

ג. **תוצאות המדידה:**

סידורי	**מספר הכנסות**	**זמן הכנסה ממוצע** (ננו-שניות - 10^{-9}sec)		**כמות גלגולים ממוצעת** (פר הכנסה, רשימה עצית)	
		רשימה מעגלית	**רשימה עצית**	**ימינה**	**שמאלה**
1	10K	23,652	2365	0.81	0.81
2	20K	40,863	2123	0.81	0.81
3	30K	59,317	2409	0.81	0.81
4	40K	80,718	1285	0.81	0.81
5	50K	100,856	1496	0.81	0.81
6	60K	121,322	1986	0.81	0.81
7	70K	142,181	1321	0.81	0.81
8	80K	158,513	820	0.81	0.81

9	90K	180,708	637	0.81	0.81
10	100K	201,501	878	0.81	0.81

ד. **תרשים:**



ה. **תוצאות:**

1) זמן ההכנסה הממוצע ברשימה מעגלית גָדֵל במגמה ליניארית ביחס לכמות ההכנסות שבוצעו.
2) זמן ההכנסה לרשימה עצית מהיר בערך פי 72 מזמן ההכנסה לרשימה מעגלית (בממוצע על פני כל ההכנסות).
3) כמות הגלגולים הממוצעת היא 1.6 וכמות הגלגולים מתחלקת בצורה שווה בין גלגולים ימינה ושמאלה.

ו. **מסקנה:**

1) הסיבוכיות של הכנסה לאמצע רשימה מעגלית באופן תיאורטי היא $O(\frac{n}{2}) = O(n)$ (כאשר n הוא אורך הרשימה). התוצאות של הניסוי נתנו מדדים מאוד הגיוניים להשערותינו - קל לראות בעזרת הגרף שהמגמה של זמן ממוצע להכנסה כתלות בכמות הכנסות ברשימה מעגלית היא מגמה ליניארית. זמן ההכנסה הממוצע לרשימה עצית נמוך משמעותית וזו מכיוון שהכנסה לרשימה עצית היא בסיבוכיות $O(\log n)$ במקרה הגרוע. מכיוון שבמדידה אנו עוסקים ברשימות ארוכות מאוד, הפער בין הזמנים של הרשימה העצית והמעגלית מתעצם.
2) כמות הגלגולים גדולה מ-1 ודבר זה מצביע על כך שהיו מקרים של "גלגולים כפולים" (דהיינו גלגול מסוג LR או RL שנספרים כשני גלגולים). בנוסף, מספר הגלגולים התחלק בצורה שווה בין שני הצדדים. ככה"נ דבר זה נובע מאיזון של עץ AVL ומכך שהאיברים תמיד הוכנסו לאמצע הרשימה ולכן בערך בחצי מהמקרים שהאיזון הופר, הוא "הופר לכיוון" ימין ובחצי מהמקרים "הופר לכיוון" שמאל.

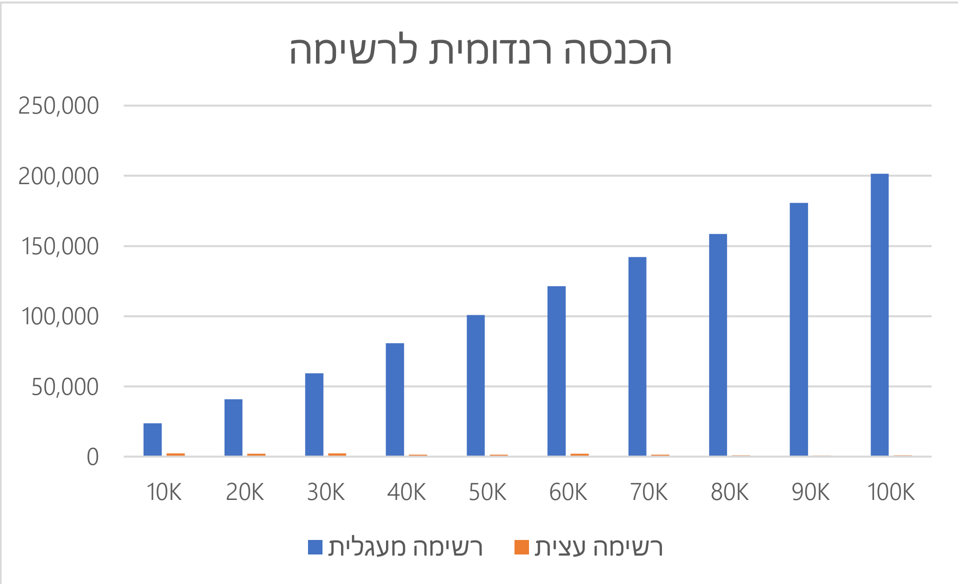
3. **<u>מדידה (3)</u> – רשימה עצית VS רשימה מעגלית:**

א. **הניסוי:** נבצע את ההכנסות כך שאינדקס כל הכנסה ייקבע בצורה **אחידה** (=יוגרל) בין כל המיקומים האפשריים עת אותה הכנסה.

ב. **תוצאות המדידה:**

סידורי	מספר הכנסות	זמן הכנסה ממוצע (ננו-שניות - 10^{-9}sec)		כמות גלגולים ממוצעת (פר הכנסה, רשימה עצית)	
		רשימה מעגלית	**רשימה עצית**	**ימינה**	**שמאלה**
1	10K	11,140	2278	0.355	0.354
2	20K	25,677	1935	0.351	0.351
3	30K	40,341	2316	0.352	0.353
4	40K	49,257	1345	0.348	0.348
5	50K	54,308	1580	0.349	0.348
6	60K	65,736	2297	0.346	0.349
7	70K	85,011	1868	0.351	0.350
8	80K	105,035	1088	0.349	0.349
9	90K	106,760	1056	0.349	0.351
10	100K	111,403	1393	0.349	0.351

ג. **תרשים:**



ד. **תוצאות:**

1) זמן ההכנסה הממוצע ברשימה מעגלית גדל במגמה ליניארית ביחס לכמות ההכנסות שבוצעו.
2) זמן ההכנסה לרשימה עצית מהיר בערך פי 38 מזמן ההכנסה לרשימה מעגלית (בממוצע על פני כל ההכנסות).
3) כמות הגלגולים הממוצעת היא בערך 0.7 ומתחלקת בצורה שווה בין גלגולים ימינה ושמאלה.

ה. **מסקנה:**

1) גם בניסוי זה **הרשימה העצית מנצחת** את הרשימה המעגלית. הסיבה לכך טמונה בעובדה שהאינדקסים מוגרלים אקראית ולכן במקרים רבים תהיה הכנסת איבר לא בקצוות הרשימה. במקרים אלו, הסיבוכיות תהיה תלויה בגודל הרשימה המעגלית, נזכיר שהסיבוכיות לפעולת הכנסה ברשימה מעגלית היא: $O(min\{i+1, len-i+1\})$ וזה קרוב לצורה ליניארית כתלות באורך הרשימה כאשר ערך i אקראי. לעומת זאת, הסיבוכיות של הכנסה ברשימה עצית היא במקרה הגרוע $O(\log n)$ ולכן כשמדובר ברשימות גדולות כפי שיש בניסוי זה, הפער בין הביצועים הופך למשמעותי.
2) כפי שצפינו, מספר הגלגולים התחלק בצורה שווה בין שני הצדדים. דבר שככה"נ נובע מאיזון של עץ AVL ומכך שמיקומי ההכנסות הן אקראיים. כמו כן, אנו יודעים שבהכנסה מבוצע לכל היותר גלגול אחד ולכן מדד כמות הגלגולים הממוצעת (~0.7) מסתדר עם הניתוח התיאורטי שעשינו בשיעור על עצי AVL.

■