מיה מרום mayamarom 209162544
קשת ברנדט keshetbrandt 207004078

קובץ תיעוד: תרגיל מעשי 1 – קורס מבני נתונים

חלק מעשי – מימוש עץ AVL בשפת python:

את ההסברים על הפונקציות השונות וסיבוכיות זמן הריצה שלהן נפריד ל3 קטגוריות – הפונקציות שהתבקשנו לממש, פונקציות נוספות שאנחנו הוספנו למחלקה, פונקציות בזמן ריצה קבוע ושדות שהוספנו.

1. פונקציות שהתבקשנו לממש:

- **retrieve(i): סיבוכיות זמן ריצה O(logn)**
  הפונקציה מחזירה את הערך של האיבר במקום ה-i ברשימה. היא מבצעת זאת על ידי קריאה לפוקנציה Select(i+1). בעץ האיברים מסודרים מ-1 עד n ולכן יש לבצע Select(i+1) ולא ל-i.
  סיבוכיות זמן הריצה של הפונקציה היא O(log n). כזכור, עץ AVL הוא עץ חיפוש בינארי ולכן מספר הצמתים המקסימלי שנבקר בו כדי למצוא צומת מסוים הוא כגובה של העץ, ועץ AVL שומר על גובה מקסימלי שהוא log n (כאשר n הוא מספר הצמתים בעץ).
- **search(value): סיבוכיות זמן ריצה O(n)**
  הפונקציה Search מחזירה לנו את האינדקס של הצומת עם הערך value. אם אין כזה, תחזיר את הערך 1-. סיבוכיות זמן הריצה של פונקציה זו היא O(n). הפונקציה יוצרת מערך (רשימה של פייתון) עם איברי העץ (בעזרת פונקציית listToArray() שסיבוכיות הזמן שלה היא O(n)). לאחר מכן, עוברת בעזרת לולאת for על כל איברי המערך (גם כן, סיבוכיות O(n)) ומשווה את הערך שבתא i במערך ובין value.
- **sort(): סיבוכיות זמן ריצה O(n logn)**
  הפונקציה משתמשת במערך שמייצג את איברי העץ (בעזרת פונקציית listToArray() שסיבוכיות הזמן שלה O(n)). לאחר מכן, היא משתמשת בפונקציית mergesort (שהכרנו בקורס מבוא מורחב – מימשנו בעצמנו לפי מה שנלמד בקורס). פונקציה זו פועלת בעזרת אלגוריתם שמחלק בכל פעם את הרשימה לשני חצאים, ממיין כל אחד מהחצאים ומחבר את החלקים הממוינים חזרה (בעזרת פונקציית merge). זהו אלגוריתם רקורסיבי, בעל סיבוכיות O(n log n). לכן, סיבוכיות הזמן של פונקציית sort היא O(n log n).
- **concat(T): סיבוכיות זמן ריצה O(logn)**
  הפונקציה מקבלת כאינפוט עץ AVL אחד ומשרשרת אותו לסוף של עץ הAVL השני (עליו מופעלת הפונקציה). ראשית, היא בודקת אם אחד העצים ריק. במידה וכן, רק מעדכנת את המצביעים של העצים. במידה ולא, הפונקציה בודקת מי מהעצים גבוה יותר. אם self גבוה יותר אז נמצא את האיבר הראשון בself שגובהו קטן/שווה לגובה של T (ונמצא ב"ענף" הימני ביותר של העץ. נסמנו בx לצורך ההסבר), נשלוף את האיבר המקסימלי של העץ ונבצע עדכון כך שx (וכל האיברים תחתיו) יהיו תת העץ השמאלי של האיבר המקסימלי, וT יהיה תת העץ הימני של האיבר המקסימלי. נחבר את האיבר המקסימלי כבן הימני של צומת האב של x. כך נחבר את שני העצים בלי לפגוע בסדר של הרשימות אותם העצים מממשים. במידה וT עץ גבוה יותר מself, נבצע את הפעולה ההפוכה סימטרית (נמצא את האיבר הראשון בעל גובה קטן/שווה משל self שנמצא על הענף השמאלי של T, ונחבר אותו בעזרת הצומת המינימלי של T). אם שני העצים באותו הגובה, נשלוף את האיבר המקסימלי של self ונקבע אותו כשורש של העץ החדש, כאשר self הוא תת העץ השמאלי וT הוא תת העץ הימני). בסיום, הפונקציה מעדכנת את המצביעים של שני העצים שיצביעו על העץ המאוחד וכן

את הגדלים והגבהים של התאים הרלוונטיים בעץ. סיבוכיות זמן הריצה של הפונקציה היא O(log n) כאשר n הוא מספר הצמתים בעץ הגבוה מבין העצים שמאחדים. הפונקציה מחפשת (בעץ הגבוה יותר) את הצומת בעל הגובה המתאים לחיבור בעזרת לולאת while. מספר האיטרציות המקסימלי של לולאה זו הוא גובה העץ שהוא log n. לאחר מכן, היא קוראת לפונקציה minNode\maxNode שגם היא בעלת סיבוכיות זמן של O(log n). שאר הפעולות בפונקציה הן שינוי מצביעים (זמן עבודה קבוע O(1)). לסיום, עדכוני הגבהים והגדלים הוא בעזרת פונקציות Update שלהן סיבוכיות זמן O(log n), וכן איזון העץ החדש בעזרת פעולת Balance, שסיבוכיות הזמן שלה היא O(log n). נפרט על פונקציות אלו בהמשך. לכן סיבוכיות הזמן היא O(log n).

- **delete(index): סיבוכיות זמן ריצה O(logn)**

הפונקציה מוחקת מהעץ את האיבר באינדקס ה-i ברשימה שהעץ מממש ומחזירה את מספר האיזונים שבוצעו לאחר המחיקה. ראשית הפונקציה בודקת אם יש מספיק איברים בעץ (כלומר, אם הקלט תקין). במידה ולא היא מחזירה 1-. במידה והוא כן נמצא בעץ, נשתמש בפונקציית Select(i+1) כדי למצוא את האיבר ה-i שאותו אנו רוצים למחוק. לאחר מכן נפריד למקרים בהתאם לצומת אותו אנו רוצים למחוק:

- אם הצומת הוא השורש של העץ – נמצא את הsuccesor של הצומת ונגדיר אותו כשורש החדש של העץ (נעדכן את המצביעים, גבהים וגדלים בהתאם).
- אם הצומת הוא עלה – נמחק אותו על ידי עדכון המצביעים של צומת האב, ואת הגבהים והגדלים בהתאם.
- אם לצומת יש בן יחיד – נמחק אותו על ידי עדכון המצביעים של האב להצביע לבן של הצומת שמחקנו ולהפך. נעדכן גבהים וגדלים בהתאם.
- אם לצומת יש שני בנים – נמצא את הsuccesor של הצומת ונמקם אותו במקום של הצומת שמחקנו. במצב זה, יש לעדכן את המצביעים של האב והבנים של הצומת שמחקנו וכן למקם מחדש את הבנים של הsuccesor (במידה והיו).

בסיום, הפונקציה קוראת לפונקציית Balance כדי לאזן את העץ לאחר המחיקה (מחזירה את מספר האיזונים).
סיבוכיות זמן הריצה של הפונקציה היא O(log n). הפונקציה קוראת למספר פונקציות כאשר מריצים אותה (succesor, Select, Balance, updateHeights, updateSizes), כולן בעלות סיבוכיות זמן ריצה של O(log n). שאר העבודה היא שינוי מצביעים, שזה בעלות של O(1).

- **permutations(): סיבוכיות זמן ריצה O(n)**

הפונקציה יוצרת עץ AVL חדש ובו הערכים שהיו בעץ AVL המקורי בסדר מעורבב. מימשנו את הפונקציה על ידי העתקת המבנה של העץ המקורי self (בעזרת קריאה לפונקציית עזר copyTree). כמו כן, יצרנו רשימה של איברי העץ בעזרת listToArray וערבבנו את סדר האיברים (לולאת for יורדת בעלת n איטרציות = סיבוכיות O(n)). לבסוף, קראנו לפונקציה shuffleTree אשר מחליפה את הvalues בצמתים הקיימים לערכים אחרים בעץ שנמצאים במיקום שונה (פירוט על אופן המימוש של הפונקציה בהמשך). סיבוכיות זמן הריצה של הפונקציה היא O(n). 2 פונקציות העזר הן בסיבוכיות O(n) (יפורטו בהמשך), ובנוסף פעולת Select פועלת בסיבוכיות O(log n) ולכן סיבוכיות הזמן היא O(n).

- **insert(index, value): סיבוכיות זמן ריצה O(logn)**

הפונקציה מכניסה צומת חדש עם ערך value באינדקס הi ברשימה. ראשית הפונקציה בודקת האם העץ ריק, אם כן היא יוצרת AVLNode חדש כך שהשדות root, First, Last של העץ יצביעו עליו. אם אינדקס ההכנסה הוא במקום האחרון ברשימה, הפונקציה מחברת את הצומת כבן ימני לצומת הגדול ביותר בעץ. אחרת, הפונקציה מוצאת את הצומת במקום של האינדקס (בעזרת פונקציית Select שנסביר עליה בהמשך וסיבוכיות הזמן שלה O(logn)), ומכניסה את הצומת החדש כבן שמאלי שלו או כבן ימני של הpredecessor שלו. לאחר הכנסת הצומת החדש, הפונקציה מעדכנת את שדות ה size, height בעזרת פונקציות עזר שנסביר עליהן בהמשך (סיבוכיות הזמן שלהן היא O(logn)) וכן מעדכנת את שדות First, Last בעזרת פונקציית Select. לסיום מתבצעת קריאה לפונקציית Balance על מנת לאזן חזרה את העץ ולהחזיר את מספר האיזונים שבוצעו (מספר הגלגולים) שסיבוכיות הזמן שלה היא גם כן O(logn).

- **listToArray(): סיבוכיות הזמן של הפונקציה היא O(n)**

הפונקציה מעבירה את ערכי הצמתים במבנה הנתונים של עץ הAVL שמימשנו לרשימה של פייתון. הפונקציה בודקת האם העץ ריק, במידה וכן תוחזר רשימה ריקה. אחרת, היא קוראת לפונקציה הרקורסיבית listToArrayMem(node, temp) שמבצעת את ההכנסה לרשימה. היא עושה זאת בעזרת מעבר in-order על צמתי העץ. מקרה הבסיס של הרקורסיה הינו הגעה לצומת וירטואלי, במקרה זה הפונקציה חוזרת (עם return ריק). סיבוכיות הזמן היא O(n) משום שנבקר בכל צומת בעץ בדיוק פעם אחת (כאשר נכניס אותו למערך) והעבודה בכל צומת היא קבועה. לכן סה״כ סיבוכיות הזמן היא כמספר הצמתים בעץ, O(n). נציין שזוהי סיבוכיות הזמן הטובה ביותר עבור הפונקציה משום שבכל מקרה על מנת להכניס את הערכים למערך נצטרך לעבור על כל הערכים בעץ.

2. פונקציות שאנחנו הוספנו למחלקה:

- **Select(i) – סיבוכיות זמן ריצה O(logn)**

פונקציה זו מחזירה את האיבר הi בגודלו בעץ. כלומר באינדקס הi-1 ברשימה. ראשית הפונקציה בודקת האם העץ ריק, במידה וכן יוחזר None. אחרת, כפי שראינו בהרצאה, הפונקציה משתמשת בקריאות רקורסיביות בעזרת פונקציית TreeSelect מהשורש מטה בחיפוש אחר הצומת, כאשר הקריאה היא שמאלה או ימינה בהתאם לגודל תת העץ והערך שמחפשים. במקרה הגרוע, נחפש עלה, כך שיהיו לנו לכל היותר logn קריאות רקורסיביות כגובה העץ, כאשר כל פעולה מתבצעת בזמן קבוע ולכן סה״כ סיבוכיות הזמן היא O(logn).

- **Rank(node) – סיבוכיות זמן ריצה O(logn)**

פונקציה זו מחזירה את האינדקס של הnode שנשלח כקלט לפונקציה. ראשית הפונקציה בודקת האם העץ ריק, במידה וכן יוחזר (-1). אחרת, כפי שראינו בהרצאה, הפונקציה משתמשת בקריאות רקורסיביות מהצומת הנתון במסלול מעלה לשורש: אם הצומת בן ימני של אביו, נוסיף למשתנה עזר את גודל תת העץ השמאלי שלו (פלוס אחד), אחרת הוא ממשיך מעלה מבלי להגדיל את משתנה העזר. לבסוף יוחזר הערך של משתנה העזר שמייצג את הrank. הפונקציה מתחילה מהצומת ועולה עד השורש, במקרה הגרוע נשלח צומת שהוא עלה, כך שיהיו לנו לכל היותר logn קריאות רקורסיביות כגובה העץ כאשר כל פעולה מתבצעת בזמן קבוע ולכן סה״כ סיבוכיות הזמן היא O(logn).

- **Succesor(node): סיבוכיות זמן ריצה O(logn)**

הפונקציה מחזירה את האיבר הבא בעץ (עבור צומת באינדקס i, תחזיר את האיבר באינדקס i+1). במידה ונשלח האיבר המקסימלי, יוחזר None. אם לצומת יש בן ימני (לא וירטואלי), היא תחזיר את המינימום שבתת העץ הימני של הצומת (בעזרת פונקציית minNode() שסיבוכיות הזמן שלה O(log n). אם אין לצומת תת עץ ימני, אז הפונקציה תעלה למעלה עד שתגיע לצומת הראשונה שהיא בן שמאלי של צומת האב שלה. במקרה הזה, הsuccesor יהיה האב (הצומת ש node נמצא בתת העץ השמאלי שלה). סיבוכיות זמן הריצה של הפונקציה היא O(log n), זאת משום שאנחנו ״זזים״ על גבי העץ למעלה/למטה (בהתאם למקרה – אבל תמיד רק באחד מהכיוונים). מספר הצעדים המקסימלי שנוכל לבצע שווה לגובה העץ, שהוא log n.

- **Predecessor(node): סיבוכיות זמן ריצה O(logn)**

הפונקציה מחזירה את האיבר הקודם בעץ (עבור צומת באינדקס i, תחזיר את האיבר באינדקס i-1. במידה ונשלח האיבר המינימלי, יוחזר None. אם לצומת יש בן שמאלי (לא וירטואלי), היא תחזיר את המקסימום שבתת העץ השמאלי של הצומת (בעזרת פונקציית maxNode() שסיבוכיות הזמן שלה O(log n)). אם אין לצומת תת עץ שמאלי, אז הפונקציה תעלה במעלה העץ עד שתגיע לצומת הראשונה שהיא בן ימני של צומת האב שלה. במקרה הזה, הpredeccesor יהיה האב (הצומת ש node נמצא בתת העץ הימני שלה). סיבוכיות זמן הריצה של הפונקציה היא O(log n), זאת משום שאנחנו ״זזים״ על גבי העץ למעלה/למטה (בהתאם למקרה – אבל תמיד רק באחד מהכיוונים). מספר הצעדים המקסימלי שנוכל לבצע שווה לגובה העץ, שהוא log n.

- **minNode(node): סיבוכיות זמן ריצה O(logn)**

פונקציה זו מחזירה את הצומת המינימלי (צומת מינימלי = הצומת בעל האינדקס הנמוך ביותר ברשימה שהעץ ממשו) בתת העץ שהשורש שלו הוא הnode שהועבר. היא עושה זאת על ידי תזוזה שמאלה על גבי העץ עד שנגיע לאיבר שאין לו בן שמאלי (כלומר, אין שום איבר שקטן ממנו) ולכן הוא

הצומת המינימלי. סיבוכיות זמן הריצה של הפונקציה היא O(log n) כי מספר האיטרציות של לולאת הwhile, שווה למס׳ הצעדים המקסימלי שנוכל לבצע שמאלה בעץ, כלומר גובה העץ, שהוא לכל היותר log n.

- **maxNode(node): סיבוכיות זמן ריצה O(logn)**
  פונקציה זו מחזירה את הצומת המקסימלי (הצומת בעל האינדקס הגבוה ביותר ברשימה שהעץ מממש) בתת העץ שהשורש שלו הוא הnode שהועבר. היא עושה זאת על ידי תזוזה ימינה על גבי העץ עד שנגיע לאיבר שאין לו בן ימני (כלומר, אין שום איבר שגדול ממנו) ולכן הוא הצומת המקסימלי. סיבוכיות זמן הריצה של הפונקציה היא O(log n) כי מספר האיטרציות של לולאת הwhile, שווה למס׳ הצעדים המקסימלי שנוכל לבצע ימינה בעץ, כלומר גובה העץ, שהוא לכל היותר log n.
- **buildTree(lst): סיבוכיות זמן ריצה O(n logn)**
  הפונקציה מקבלת כאינפוט מערך ומחזירה עץ AVL שהצמתים בו הם איברי המערך. הפונקציה נעזרת בפונקציית עזר רקורסיבית כדי לבנות את העץ. הפונקציה הרקורסיבית מחלקת את המערך לשני חצאים, בונה צומת AVLNode שהערך שלו הוא הערך של החציון ברשימה (שמפריד בין שני החצאים), ואז בונה רקורסיבית את תתי העץ הימני והשמאלי בעזרת חצאי הרשימות. הפונקציה גם מעדכנת את הגובה והגודל של הצומת (פעולות בO(1)) וקוראת לפונקציות updateHeights ו-updateSizes כדי לעדכן את שאר הגבהים והגדלים בעץ. סיבוכיות זמן הריצה היא O(n log n) כי נבקר בכל תא במערך (נכניס לעץ) פעם אחת בדיוק ומספר הקריאות הרקורסיביות הוא לוגריתמי בהתאם לאורך המערך (בכל פעם מחלקים את המערך ל-2, והקריאות מפסיקות כאשר הגענו לרשימה באורך 0 או 1). סיבוכיות זמן הריצה של פונקציות update הן O(log n) ולכן לא מעלות את זמן הריצה.
- **copyTree(): סיבוכיות זמן ריצה O(n)**
  הפונקציה נעזרת בפונקציית עזר copyTreeMem. פונקציה זו מעתיקה באופן רקורסיבי את עץ AVL עליו הפעלנו את הפעולה. היא עושה זאת בעזרת מעבר in-order על הצמתים בעץ, ובכל קריאה יוצרת AVLNode חדש (זמן קבוע) ומעדכנת את המצביעים בהתאם (זמן קבוע). הפונקציה עוברת רקורסיבית על כל הצמתים בעץ ולכן יש n קריאות רקורסיביות. מכאן, סיבוכיות זמן הריצה של הפונקציה היא O(n).
- **shuffleTree(node, lst): סיבוכיות זמן ריצה O(n)**
  הפונקציה מקבלת צומת ומערך שמכיל את הערכים של כל הצמתים בעץ המקורי בסדר מעורבב. הפונקציה עוברת על הצמתים בעץ באופן רקורסיבי (in-order traverses) ובכל צומת מעדכנת את הvalue של הצומת להיות הערך האחרון במערך ״המבולגן״ ולאחר מכן מוחקת את התא מהמערך (כדי למנוע הכנסה כפולה של ערכים). התהליך הנ״ל ממשיך עד שכל התאים בעץ עודכנו וכל התאים בlst נמחקו. סיבוכיות זמן הריצה היא O(n) כי מבצעים n קריאות רקורסיביות (ביקור בכל הצמתים בעץ) ובכל קריאה מבוצעת עבודה בעלת זמן קבוע (עדכון הvalue בצומת, מחיקת התא מהמערך). נציין כי ביצוע pop() לאיבר האחרון במערך הוא O(1).
- **Balance(i): סיבוכיות זמן ריצה O(logn)**
  פונקציה זו מאזנת את העץ לאחר פעולות שעשויות להוציא אותו מאיזון (הכנסה, מחיקה וכו׳). הפונקציה מקבלת אינדקס של צומת שממנו היא תתחיל לאזן את העץ (ועוברת בלולאה ממנו מעלה עד השורש). האיזונים נעשים על עברייני AVL בעזרת פונקציות עזר של רוטציות כפי שראינו

בהרצאה (נסביר על פונקציות אלו בהמשך). על אף שסיבוכיות הזמן של פונקציות אלו היא O(logn) נבצע לכל היותר שני גלגולים כי יהיה לכל היותר עבריין AVL אחד (כפי שראינו בהרצאה) ולכן סיבוכיות הזמן נשארת O(logn). הפונקציה מחזירה את מספר הגלגולים שנעשו. במקרה הגרוע ישלח צומת שהוא עלה, כך שהלולאה תרוץ logn פעמים כגובה העץ כאשר כל פעולה מתבצעת בזמן קבוע ולכן סה״כ סיבוכיות הזמן היא O(logn).

- **Rotate(node): סיבוכיות זמן ריצה O(logn)**
הפונקציה מקבלת צומת שהוא כרגע עבריין AVL, בודקת איזה רוטציה יש לעשות (לפי הBF של הצומת הנתון ושל הבן שלה) וקוראת לפונקציית הרוטציה המתאימה (שסיבוכיות הזמן שלהן היא O(logn)). לבסוף, הפונקציה מחזירה את מספר הגלגולים שבוצעו.
- **RotateLeft(node), RotateRight(node): סיבוכיות זמן ריצה O(logn)**
הפונקציות מקבלות צומת שאבא שלו הוא עבריין AVL ומבצעות גלגול על הצומת. הגלגול נעשה בעזרת חילופי מצביעים בין הצמתים, שנעשה בזמן קבוע. משום ששמרנו שדה גובה בכל צומת, נבצע לבסוף קריאה לפונקציית UpdateHeights שמעדכנת את הגבהים של הצמתים עד השורש. נראה בהמשך שסיבוכיות הזמן שלה היא O(logn).
- **UpdateSizes(node), UpdateHeights(node): סיבוכיות זמן ריצה O(logn)**
הפונקציה עוברת בלולאת while מהצומת הנתון ועד השורש ומעדכנת את שדות הsize/height בהתאמה, בהתאם לבנים שלה. במקרה הגרוע ישלח צומת שהוא עלה, כך שהלולאה תרוץ logn פעמים כגובה העץ כאשר כל פעולה מתבצעת בזמן קבוע ולכן סה״כ סיבוכיות הזמן היא O(logn).

מיה מרום mayamarom 209162544
קשת ברנדט keshetbrandt 207004078

3. פונקציות get\set ונוספות (ללא ניתוח סיבוכיות – כולן בזמן קבוע (O(1)):

- שדות שהוספנו למחלקה AVLNode (עם ערכים דיפולטיים):
  - size: גודל תת העץ שהצומת הוא השורש שלו. ערך דיפולטי: 0.
  - BF: הBalance factor של הצומת. ערך דיפולטי: 1-.
  - real: שדה בוליאני המתאר האם זהו צומת וירטואלי או אמיתי. ערך דיפולטי: True.
- שדות שהוספנו למחלקה AVLTreeList (עם ערכים דיפולטיים):
  - First: מצביע לצומת המינימלי בעץ (אינדקס 0 ברשימה אותה העץ מממש). ערך דיפולטי: None.
  - Last: מצביע לצומת המקסימלי בעץ (אינדקס אחרון ברשימה אותה העץ מממש). ערך דיפולטי: None.
- **getLeft()**: הפונקציה מחזירה את הבן השמאלי של הצומת עליה הופעלה הפונקציה. במידה והצומת היא וירטואלית, הפונקציה מחזירה None.
- **getRight()**: הפונקציה מחזירה את הבן הימני של הצומת עליו הופעלה הפונקציה. במידה והצומת היא וירטואלית, הפונקציה מחזירה None.
- **getParent()**: הפונקציה מחזירה את ההורה של הצומת עליו מופעלת הפונקציה. במידה ואין הורה (שורש העץ), יוחזר None.
- **getValue()**: הפונקציה מחזירה את הערך של הצומת עליו הופעלה הפונקציה, עבור צומת וירטואלי יוחזר None.
- **getHeight()**: הפונקציה מחזירה את הגובה של הצומת עליו הופעלה הפונקציה, עבור צומת וירטואלי יוחזר 1-.
- **getSize()**: הפונקציה מחזירה את הגודל של תת העץ שהשורש שלו הוא הצומת עליו הופעלה הפונקציה, עבור צומת וירטואלי יוחזר 0.
- **getBF()**: הפונקציה מחזירה את הbalance factor של הצומת עליו הופעלה הפונקציה, עבור צומת וירטואלי יוחזר 1-.
- **isRealNode()**: הפונקציה מחזירה ערך בוליאני שמציין האם הצומת הוא צומת אמיתי או צומת וירטואלי.
- **setLeft(node)**: הפונקציה קובעת את הצומת node כבן השמאלי של הצומת עליו הופעלה הפונקציה.
- **setRight(node)**: הפונקציה קובעת את הצומת node כבן הימני של הצומת עליו הופעלה הפונקציה.
- **setParent(node)**: הפונקציה קובעת את הצומת node כהורה של הצומת עליו הופעלה הפונקציה.
- **setValue(val)**: הפונקציה מעדכנת את הערך של הצומת עליו הופעלה הפונקציה להיות val.
- **setHeight(h)**: הפונקציה מעדכנת את הגובה של הצומת עליו הופעלה הפונקציה להיות h.
- **setSize(new)**: הפונקציה קובעת את הגודל של הצומת (כלומר, הגודל של תת העץ שמתחיל בצומת עליו הופעלה הפונקציה) להיות new.
- **setBF()**: הפונקציה מעדכנת את הbalance factor של הצומת עליו הופעלה הפונקציה. היא עושה זאת על ידי חישוב הגובה של תת העץ השמאלי מינוס הגובה של תת העץ הימני. במידה והצומת עליו הופעלה הפונקציה הוא וירטואלי, היא קובעת את הBF להיות 1-.

- **empty()**: הפונקציה מחזירה True אם העץ ריק.
- **first()**: הפונקציה מחזירה את הערך של האיבר הראשון ברשימה שהעץ מממש (=את האיבר שמתקבל בSelect(1)). אם העץ ריק, מחזיר None.
- **last()**: הפונקציה מחזירה את הערך של האיבר האחרון ברשימה שהעץ מממש (=את האיבר שמתקבל בSelect(self.size)). אם העץ ריק, מחזיר None.
- **length()**: הפונקציה מחזירה את האורך של הרשימה שהעץ AVL מממש.
- **getRoot()**: מחזיר את הצומת שהיא השורש של העץ עליו הופעלה הפונקציה.

מיה מרום 209162544 mayamarom
קשת ברנדט 207004078 keshetbrandt

חלק ניסויי-תיאורטי:

1.

		חלק 1 - הכנסות	חלק 2 - מחיקות	חלק 3 - הכנסות ומחיקות לסירוגין		
				n/2 הכנסות	n/4 הכנסות ומחיקות לסירוגין	סך הכל
n=3,000	i=1	2,061 איזונים	1,098 איזונים	1,007 איזונים	816 איזונים	1,823 איזונים
n=6,000	i=2	4,221 איזונים	2,177 איזונים	2,059 איזונים	1,606 איזונים	3,731 איזונים
n=12,000	i=3	8,440 איזונים	4,427 איזונים	4,211 איזונים	3,180 איזונים	7,391 איזונים
n=24,000	i=4	16,823 איזונים	8,924 איזונים	8,335 איזונים	6,313 איזונים	14,672 איזונים
n=48,000	i=5	33,291 איזונים	17,930 איזונים	16,989 איזונים	12,483 איזונים	29,472 איזונים
n=96,000	i=6	66,670 איזונים	35,594 איזונים	33,433 איזונים	25,341 איזונים	58,774 איזונים
n=192,000	i=7	133,772 איזונים	71,702 איזונים	67,294 איזונים	50,427 איזונים	117,721 איזונים
n=384,000	i=8	267,688 איזונים	143,188 איזונים	134,406 איזונים	100,533 איזונים	234,939 איזונים
n=768,000	i=9	537,661 איזונים	284,736 איזונים	268,014 איזונים	258,110 איזונים	526,124 איזונים
n=1,536,000	i=10	1,073,576 איזונים	573,567 איזונים	575,360 איזונים	428,876 איזונים	1,004,236 איזונים
ביטוי אסימפטוטי:		O(n)	O(n/2) = O(n)	O(n)		

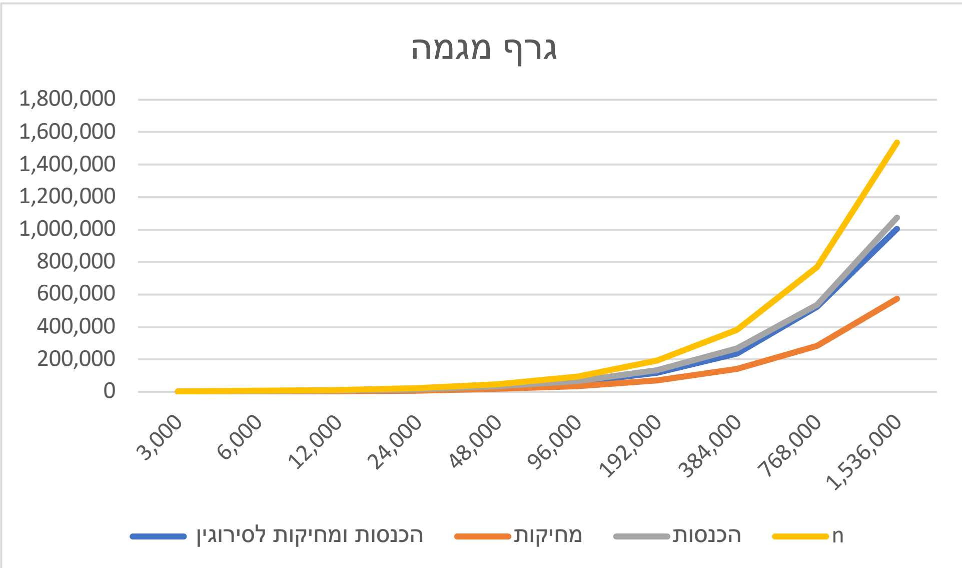
יצרנו בexcel גרף מגמה כך שבציר הX יש את מספר ההכנסות/מחיקות שביצענו (n) ובציר הY יש את מספר האיזונים שבוצעו. השיפוע של גרפי המגמה מתאר את היחס בין מספר פעולות האיזונים למספר פעולות ההכנסה ומחיקה. מכאן הסקנו את הביטוי האסימפטוטי.

חלק 1 – הכנסות: שיפוע הגרף הינו 0.699182112 ולכן הביטוי האסימפטוטי הוא 0.69n

חלק 2 – מחיקות: שיפוע הגרף הינו 0.373027611 ולכן הביטוי האסימפטוטי הוא 0.37n

חלק 3 – הכנסות ומחיקות לסירוגין: שיפוע הגרף (עבור סך כל הפעולות) הינו 0.659508544 ולכן הביטוי האסימפטוטי הוא 0.65n

ניתן לראות בכל אחת מן העמודות, שמספר האיזונים הוא לינארי לגודל העץ ולכן הביטוי האסימפטוטי המתאים לכל אחת מהפעולות שבדקנו הוא O(n) (כאשר n מייצג את מספר הצמתים ברשימה). כמו כן, צירפנו גרף מגמה לשלמות ההסבר:



מיה מרום mayamarom 209162544
קשת ברנדט keshetbrandt 207004078

.2

		עץ AVL הכנסות להתחלה	רשימה מקושרת הכנסות להתחלה	מערך הכנסות להתחלה
n=1,500	i=1	7.62E-05	1.14E-06	8.62E-07
n=3,000	i=2	9.41E-05	1.22E-06	1.45E-06
n=4,500	i=3	8.71E-05	1.08E-06	1.48E-06
n=6,000	i=4	8.71E-05	8.77E-07	1.69E-06
n=7,500	i=5	9.41E-05	1.11E-06	2.44E-06
n=8,000	i=6	0.000114867	1.50E-06	3.25E-06
n=9,500	i=7	9.36E-05	9.61E-07	2.91E-06
n=11,000	i=8	0.000106933	1.10E-06	3.59E-06
n=13,500	i=9	0.000118382	1.37E-06	5.34E-06
n=15,000	i=10	0.000111783	1.14E-06	4.39E-06

		עץ AVL הכנסות בסדר אקראי	רשימה מקושרת הכנסות בסדר אקראי	מערך הכנסות בסדר אקראי
n=1,500	i=1	0.000101133	3.20E-05	2.63E-06
n=3,000	i=2	9.57E-05	5.30E-05	2.56E-06
n=4,500	i=3	0.000110494	0.000142344	3.97E-06
n=6,000	i=4	0.000110384	0.000127902	3.32E-06
n=7,500	i=5	0.000104505	0.000148974	3.21E-06
n=8,000	i=6	0.000110015	0.000198177	3.66E-06
n=9,500	i=7	0.000109368	0.000246517	4.18E-06
n=11,000	i=8	0.000117861	0.000277775	4.40E-06
n=13,500	i=9	0.000133671	0.000539619	1.04E-05
n=15,000	i=10	0.000143564	0.000572918	7.57E-06

		עץ AVL הכנסות בסדר בסוף	רשימה מקושרת הכנסות בסוף	מערך הכנסות בסוף
n=1,500	i=1	8.19E-05	7.26E-05	5.81E-07
n=3,000	i=2	8.48E-05	1.37E-04	5.39E-07
n=4,500	i=3	9.20E-05	0.000230938	6.82E-07
n=6,000	i=4	0.000105279	0.000353987	8.63E-07
n=7,500	i=5	0.00012933	0.000769729	1.29E-06
n=8,000	i=6	0.000120267	0.000715765	1.06E-06
n=9,500	i=7	0.000129346	0.001081406	1.30E-06
n=11,000	i=8	0.000155325	0.001538147	1.67E-06
n=13,500	i=9	0.000143479	0.001305128	1.33E-06
n=15,000	i=10	0.000148384	0.001760291	1.53E-06

התוצאות שהתקבלו אכן תואמות למה שציפינו. חשוב לציין שמיקום ההכנסה למבנה הנתונים משפיע על הפרשי הזמנים בין מבני הנתונים השונים.

ציפינו שהמימוש של מערך של פייתון יהיה היעיל ביותר. מעבר לעובדה שפייתון ככל הנראה מממש את מבנה הנתונים באופן הכי יעיל שמתאפשר, במערך יש אפשרות לגשת בקלות לאינדקסים ספציפיים (שאינם רק בקצוות המבנה). כמו כן, מערך (בשונה מעצי AVL ורשימה מקושרת) שמור בבלוק צמוד בזיכרון ולכן נדרשות פחות פעולות I/O. כפי שלמדנו בהרצאה, סיבוכיות הזמן של פעולות אלו גבוהה יחסית ואנחנו מתעלמים מהם בחישוב הסיבוכיות.

מערך של פייתון פחות יעיל כאשר רוצים לבצע הכנסות במרכז הרשימה, שכן צריך לבצע הזזה של כל שאר האיברים בהתאם לפעולה שבוצעה. אך גם בעצי AVL וברשימה מקושרת נדרשים למצוא את המיקום המתאים (האינדקס הספציפי) ולכן עדיין מערך יעיל יותר. בהמשך לכך, רשימה מקושרת היא פחות יעילה כאשר צריך לגשת לאיברים באינדקס ספציפי ברשימה, שכן צריך לעבור על איברי הרשימה למציאת האינדקס המתאים. לכן, נראה כי בהכנסה בסוף הרשימה, עצי ה-AVL שמימשנו יעילים יותר מהרשימה המקושרת (שצריכה לעבור על כל האיברים ברשימה להגיע לאינדקס המתאים).

עצי AVL יעילים בהכנסה ומחיקה בזכות האיזון שנשמר לאורך הפעולות השונות בעץ, אך בשל האיזונים הללו הכנסות "בסיסיות" עלולות לקחת O(logn) כפי שראינו בניתוח הסיבוכיות בסעיפים שלעיל.