AVL Tree list

פרויקט ראשון בקורס מבני נתונים –
עמוד ראשון – פרטי המגישים
עמודים 2-12 – תיעוד המחלקות
עמודים 13-15 – ניסויים ובדיקות

מגישים : עומר שפי, יונתן בניזרי לוי
תעודות זהות : עומר שפי – 205923154 יונתן בניזרי לוי – 314802992
שמות במודל : עומר שפי – omersheffi יונתן בניזרי לוי – yonatanbenizri

תיעוד המחלקות –

השתמשנו בשתי מחלקות על מנת לממש רשימה באמצעות עץ avl, אחת המתארת צומת ואחת את העץ שמשמש כרשימה בפני עצמו . כעת נפרט על תיעוד שתי המחלקות, המתודות הנדרשות ומתודות העזר שהשתמשנו בהן על מנת לממש את הרשימה כנדרש. בניתוח הסיבוכיות לעתים לא נפרט יותר מידי, כאשר אנו מבצעים פעולות גישה לשדה למשל לא תמיד נסביר שזמן הריצה הוא קבוע כיוון שהטענה תחזור על עצמה הרבה .

תיעוד מימוש של List באמצעות עץ AVL

תיעוד מחלקת AVLNode:

מחלקה זו מייצגת צומת בעץ.

1. __init__(self,value) :

מהות: הבנאי של המחלקה,זהו הבנאי הדיפולטי של המחלקה ואנו איננו משתמשים בו , נשתמש בבנאי משדורג עם דגל על מנת לזהות צמתים מדומים (העמסנו עוד בנאי שמקבל עוד פרמטר)

__init__(self,value,virutal=0)

מהות: הבנאי האמיתי של מחלקת nodes .

לכל איבר במחלקה יש את השדות: value, parent, left, right, size, height, balance_factor,virtual.

על מנת לשמור על מבנה עץ AVL חוקי, אנחנו מתחזקים את שדה ה-height, ובאמצעותו מחשבים את שדה ה-balance_factor.

אופן הפעולה: הבנאי האמיתי של כל הצמתים שניצור – נשים לב לדגל הvirtual = 0 . בעת יצירת צומת ע''י המשתמש, הדגל תמיד יהיה 0 וזה יהיה אמצעי הזיהוי שלנו לבדיקת צמתי דמה וצמתים אמיתיים . בעת יצירת צומת ע''י המשתמש, הצומת יאותחל עם הערך שהמתמש נתן .לאחר מכן נבדוק בעזרת isrealnode אם הצומת אמיתי או לא, מתודה זו משתמש בדגל הדיפולטי ובעת יצירת צומת ע''י המשתמש תמיד נדע שזהו צומת אמיתי . לאחר בדיקה האם הצמת אמיתי או לא – אם לא, נאתחל את הבנים ל none , גובה ל 1- גודל וגורם איזון ל-0 . אם הצומת אכן אמיתי, נרצה ליצור לו בנים וירטואלים בעזרת הבנאי הנוכחי ובעזרת הדגל שלנו, הבנים יווצרו כפי שתיארנו כרגע . בנוסף, נעדכן את הורי הבנים הוירטואלים שיצרנו להיות הצומת הנוכחי, את גודל הצומת הנוכחי להיות 1, את הגובה להיות 0 ונחשב את גורם האיזון . אנו עושים מספר קבוע של פעולות, שכוללות חישובים והזנה של ערכים בשדות ולכן הסיבוכיות היא זמן ריצה קבוע .

סיבוכיות: $O(1)$

2. UpdateNode(self):

מהות: המתודה מעדכנת את ערכי השדות: *size, height, balance_factor* לאחר שינוי מבנה העץ.

אופן הפעולה: בודקים האם הצומת שאנו נדרשים לעדכן היא צומת אמיתית. במידה שכן מבצעים בדיקה האם הצומת עלה(ראה סעיף 16). אם כן מעדכנים את הערכים height, balance_factor ל-0, ואת size ל-1. במידה והצומת איננו עלה, כלומר יש לו לפחות בן אחד אמיתי, מעדכנים את שדה ה-height לפי ההגדרה שלמדנו ע"י חישוב ה-max בין שני הבנים והוספת 1. עדכון שדה size ע"י חישוב גודל תת העץ השמאלי ותת העץ הימני והוספת 1 לסכום זה. ועדכון שדה ה-balance_factor ע"י קריאה למתודה calculateBF(ראה סעיף 17)

סיבוכיות: $O(1)$

3. getBalanceFactor(self) :

מהות: המתודה מחזירה את ה balance factor-של הצומת. מתודה זו משמשת עבור בדיקת התקינות של הצומת כצומת בעץ AVL במתודות המבצעות תיקונים על העץ.

אופן הפעולה: המתודה נקראת על צומת מסוים ומחזירה את ערכו של השדה balance_factor שמחושב ע"י קריאה למתודה calculateBF(ראה סעיף 17).

סיבוכיות: $O(1)$

4. getLeft(self) :
מהות: המתודה מחזירה את הבן השמאלי של הצומת שעליו היא הופעלה. אם הצומת הוא וירטואלי, היא תחזיר None.
אופן הפעולה: המתודה נקראת על צומת מסוים ומחזירה את שדה ה-left שלו.
סיבוכיות: $O(1)$

5. getRight(self) :
מהות: המתודה מחזירה את הבן הימני של הצומת שעליו היא הופעלה. אם הצומת הוא וירטואלי, היא תחזיר None.
אופן הפעולה: המתודה נקראת על צומת מסוים ומחזירה את שדה ה-right שלו.
סיבוכיות: $O(1)$

6. getSize(self):
מהות: מחזיר את הגודל של תת-העץ שהצומת עליו נקראה המתודה הוא השורש שלו, או -1 אם הצומת וירטואלי.
אופן הפעולה: המתודה נקראת על צומת מסוים ומחזירה את שדה ה-size שלו.
סיבוכיות: $O(1)$

7. getParent(self):
מהות: מחזירה את ההורה של הצומת עליו היא הופעלה. אם הצומת הוא השורש, הפונקציה תחזיר None .
אופן הפעולה: המתודה נקראת על צומת מסוים ומחזירה את שדה parent שלו .
סיבוכיות: $O(1)$:

8. getValue(self) :
מהות: מחזיר את הערך של הצומת או None אם הצומת הוא וירטואלי.
אופן הפעולה: המתודה נקראת על צומת מסוים ומחזירה את שדה ה-value שלו.
סיבוכיות : $O(1)$

9. getHeight(self) :
מהות : מחזיר את הגובה של הצות או 1- אם הצומת הוא וירטואלי.
אופן הפעולה: המתודה נקראת על צומת מסוים ומחזירה את שדה ה-height שלו.
סיבוכיות: $O(1)$

10. setLeft(self, node) :
מהות ואופן הפעולה: מתודה זו מקבלת מצביע לצומת ומגדירה את הצומת כבן שמאלי של הצומת שעליה המתודה מופעלת. בנוסף מגדירה את הצומת שעליה הופעלה כאבא של הצומת שקיבלה כמצביע ע"י קריאה למתודה setParent(ראה סעיף 12).
סיבוכיות: $O(1)$

11. setRight(self,node) :

מהות ואופן הפעולה: מתודה זו מקבלת מצביע לצומת ומגדירה את הצומת כבן ימני של הצומת שעליה המתודה מופעלת. בנוסף מגדירה את הצומת שעליה הופעלה כאבא של הצומת שקיבלה כמצביע ע"י קריאה למתודה setParent(ראה סעיף 12).
סיבוכיות: $O(1)$

12. setParent(self,node) :
מהות ואופן הפעולה: מתודה זו מקבלת מצביע לצומת ומגדירה את שדה ה-parent של הצומת שעליה היא נקראה למצביע של הצומת.
סיבוכיות: $O(1)$

13. setValue(self,value) :
מהות ואופן הפעולה: מתודה זו מקבלת ערך, ומגדירה אותו כערך של הצומת עליו היא מופעלת באמצעות עדכון שדה ה-value שלו.
סיבוכיות: $O(1)$

14. setHeight(self,h) :
מהות ואופן הפעולה: מתודה זו מקבלת מספר שלם, ומגדירה אותו כגובה של הצומת עליו היא מופעלת באמצעות עדכון שדה ה-height שלו.
סיבוכיות: $O(1)$

15. isRealNode(self) :
מהות: מחזירה True אם הצומת מייצג צומת אמיתי בעץ (כלומר צומת שאינו וירטואלי) False אחרת .
אופן הפעולה: הבדיקה נעשית באמצעות בדיקת שדה ה – virtual שמעודכן לפי האם המשתמש עצמו יצר את הצומת – במקרה זה תמיד הצומת יהיה אמיתי או אם הצומת הוא צומת דמה שאנו יוצרים בעצמנו . אנו בודקים שדה של המחלקה ולכן זמן הריצה קבוע
סיבוכיות: $O(1)$

16. isLeaf(self) :
מהות : המתודה בודקת האם הצומת שעליה הופעלה היא עלה(כלומר אין לה בנים אמיתיים). מחזירה True אם עלה, False אחרת.
אופן הפעולה: המתודה בודקת האם שני הבנים של הצומת הנוכחי הם צמתי דמה – במקרה ששניהם צמתי דמה נחזיר true כיוון שאז אנו מביטים בעלה, אחרת נחזיר false
אנו מפעילים מתודה על שני צמתים, המתודה עובדת בזמן ריצה קבוע ולכן זמן הריצה הכללי קבוע .
סיבוכיות: $O(1)$.

17. calculateBF(self) :
מהות: המתודה מחשבת את ערך ה-balance_factor של צומת שעליה הופעלה.
אופן הפעולה: המתודה ניגשת לשדה ה-height של הבן הימני והשמאלי של הצומת שעליה הופעלה ע"י קריאה למתודה getHeight(ראה סעיף 9), ומחשבת את ערך ה-BF בהתאם להגדרות שנלמדו בקורס.
סיבוכיות: גישה לשדה של 2 בנים בסיבוכיות $O(1)$ לכן, $O(1)$.

תיעוד מחלקת AVLTreeList :

מחלקה זו מייצגת את העץ AVL אשר מממש את ה-ADT רשימה.

18. (self) __init__ :

מהות ואופן הפעולה: הבנאי של המחלקה.

מייצר עץ ריק עם השדות root, firstItem, lastItem,size אשר מצביעים בהתאמה על השורש של העץ, על הצומת שדרגתה היא 1, ועל הצומת שדרגתה היא כגודל העץ.גודל העץ יאותחל ל-0 . כאשר קוראים לבנאי על מנת ליצור רשימה ריקה, כלל השדות מאותחלים ל-None. בנוסף שדה ה size יהיה שדה הגודל של העץ, הוא יעודכן בכל פעולה שמשנה את העץ הנתון – הכנסה מחיקה ושרשור עצים, ובכל פעולה כזו נעדכן אותו באמצעות self.length() שמטפלת במקרי הקצה השונים .

סיבוכיות: $O(1)$.

19. getTreeHeight(self) :

מהות: מתודה שמטרתה לחשב את הגובה של העץ.

אופן הפעולה: אם העץ ריק, כלומר השורש הוא None המתודה תחזיר 1-. אחרת, המתודה תקרא למתודה getHeight(ראה סעיף 9) על השורש של העץ ותחזיר את ערכו.

סיבוכיות: $O(1)$. – במקרה הגרוע אנו בודקים שדה של השורש ובדיקת שדה תיקח זמן ריצה קבוע .

20. empty(self) :

מהות: המתודה מחזירה True אמ"מ הרשימה ריקה.

אופן הפעולה: ע"י בדיקת שדה ה-root של העץ. אם השורש של העץ הוא None אזי העץ ריק והמתודה מחזירה True, אחרת False.

סיבוכיות: $O(1)$.

21. retrieve(self,i) :

מהות: הפונקציה מקבלת מספר שלם i, ותחזיר את הערך של האיבר באינדקס i ברשימה, אם קיים. אחרת, תחזיר None.

אופן הפעולה: תחילה, המתודה בודקת העם העץ ריק, אם כן, תחזיר None שכן אין איברים ברשימה. על מנת לדעת האם האינדקס קיים, המתודה בודקת האם i קטן מאורך הרשימה(ראה סעיף 29) והאם i גדול/שווה ל-0. במידה וכן, תתבצע קריאה למתודה treeSelect(ראה סעיף 36) עבור i+1 ותחזיר את ערכו של הצומת שמחזירה treeSelect ע"י קריאה למתודה getValue(ראה סעיף 8).

סיבוכיות: כפי שניתן לראות בניתוח סיבוכיות של treeSelect, ומכיוון שיתר הפעולות הן פעולות אריתמטיות ב-$O(1)$ וקריאה למתודה שגם כן פועלת ב-$O(1)$, הסיבוכיות של מתודה זו זהה לסיבוכיות של treeSelect, כאמור, $O(\log(i))$.

22. insert(self,I,val) :

מהות: הכנסת איבר בעל ערך val לרשימה במקום ה-i, במידה וקיימים לפחות i איברים ברשימה. המתודה מחזירה את מספר פעולות האיזון שנדרשו בסה"כ בשלב תיקון העץ על מנת לשמר את תכונת האיזון.

אופן הפעולה: מתודה זו שמטרתה להכניס איברים איברים לרשימה , היא משתמשת במתודת עזר בשם avlTreeInsert שמכניסה כרצוי איברים שאנו מתחזקים (ראה סעיף 44) . avlTreeInsert מלבד להכניס לעץ גם מחזירה את מספר הגלגולים שהיא עושה , כך פונקציית insert שלנו תחזיר את מספר הגלגולים שנעשו בעת ההכנסה כנדרש .

סיבוכיות: כפי שנפרט בתיאור של avlTreeInsert סיבוכיות הפונקציה היא log(n) וכיוון שכל מה ש insert עושה זה לקרוא לפונקציית העזר , נקבל שהסיבוכיות של insert יהיה $O(log(n))$

23. append(self,val):

מהות: הכנסת איבר לסוף הרשימה

אופן הפעולה: מתודת append כפי שאנו מכירים בפייתון, מכניסה איבר לסוף הרשימה. הוספנו אותה כמתודה עזר אבל בעיקר לנוחות המשתמש . *מתודה זו משמשת אותנו במתודה* listToArrRecusive*(ראה סעיף 28)*
סיבוכיות: מתודה זו קוראת למתודה insert עבור i=n, וכמו שניתן לראות בסעיף 22, סיבוכיות זו היא $O(log(n))$.

24. *(delete(self,i:*
מהות: המתודה מקבלת אינדקס i למחיקה מהרשימה, האיבר באינדקס i יימחק מהרשימה והמתודה תחזיר את מספר פעולות האיזון שנדרשו על מנת לאזן את העץ לאחר המחיקה. במידה והאינדקס i אינו קיים ברשימה, המתודה תחזיר 1- (ובמקרה זה לא תמחק כלום) .
אופן הפעולה: מתודה זו קוראת למתודת העזר avlDelete(i)(ראה סעיף 42) שמוחקת את האיבר ה – i מהעץ
סיבוכיות: כפי שנפרט בתיאור של *avlDelte(i)* זמן הריצה של מחיקה מהעץ הוא $O(\log(n))$.

25. *(first(self:*
מהות: המתודה מחזירה את האיבר הראשון ברשימה, או *None* אם הרשימה ריקה.
אופן הפעולה: המתודה מבצעת בדיקה האם שדה השורש של העץ הוא *None*. במידה ולא, תחזיר את ערכו של השדה *firstItem* ע"י קריאה למתודה *getValue* (ראה סעיף 8).
סיבוכיות: $O(1)$.

26. *(last(self) :*
מהות: המתודה מחזירה את האיבר האחרון ברשימה או *None* אם הרשימה ריקה.
אופן הפעולה: המתודה מבצעת בדיקה האם שדה השורש של העץ הוא *None*. במידה ולא, תחזיר את ערכו של השדה *lastItem* ע"י קריאה למתודה *getValue* (ראה סעיף 8).
סיבוכיות: $O(1)$.

27. *(listToArray(self:*
מהות: המתודה מחזירה מערך המכיל את איברי הרשימה לפי סדר האינדקסים או מערך ריק אם הרשימה ריקה.
אופן הפעולה: המתודה מאתחלת מערך ריק. אם הרשימה ריקה המתודה מחזירה מערך ריק. אחרת, היא מבצעת סיור inorder רקורסיבי בעץ המייצג את הרשימה עליה היא הופעלה, כאשר הפעולה אשר מתבצעת במהלך הסיור היא הכנסת הערך של הצומת בסוף הרשימה באמצעות פעולתappend (ראה סעיף 23).
סיבוכיות: המתודה מבצעת קריאה למתודה רקורסיבית *listToArrRecursive* שהסיבוכיות שלה היא $O(n)$*(*ראה סעיף 28), לכן סיבוכיות $O(n)$.

28. *(listToArrRecursive(self,node,arr :*
מהות: המתודה מבצעת סיור *inorder* בעץ, ומכניסה למערך את ערכי הצמתים .
אופן הפעולה: המתודה מבצעת סיור *inorder* בעץ כפי שנלמד בקורס. והכנסת ערכה של הצומת (ע"י קריאה לכל צומת למתודה *getValue*(ראה סעיף 8)) למערך באמצעות פעולת *append*(עלות קבועה).
סיבוכיות: כפי שנלמד בקורס, עלות הסיור *inorder* בעץ היא $O(n)$, רק נותר לוודא שבכל צומת זמן הריצה הוא קבוע – אכן, בכל צומת אנו בודקים אם הצומת אינו דמה, בודקים מה ערכו ומוסיפים לסוף המערך הנתון את ערך הצומת . מכיוון ששני הפעולות הראשונות לבסוף רק ניגשים לשדות והוספה בסוף מערך עולה זמן ריצה קבוע, נקבל שבכל צומת זמן הפעולה קבוע כפי שרצינו ולכן סה"כ סיבוכיות $O(n)$.

29. *(length(self:*
מהות: מתודה זו מחזירה את אורך הרשימה שעליה היא הופעלה.

אופן הפעולה: בדיקה האם הרשימה ריקה בעלות קבועה(ראה סעיף 20), אם ריקה תחזיר 0. אחרת תחזיר את גודל השורש ע"י גישה לשדה ה-*size* של השורש בעלות קבועה.
סיבוכיות: $O(1)$.

30. *sort(self)*:
מהות: המתודה תחזיר עץ אשר ממויין ע"פ ערכי ה-*value* של הרשימה.
אופן הפעולה: המתודה תבצע בדיקה האם העץ ריק, אם כן תחזיר רשימה ריקה. אחרת, המתודה תמיר את העץ למערך ע"י קריאה למתודה *listToArray*(ראה סעיף 27). לאחר מכן תמיין את הרשימה ע"י קריאה למתודת עזר *quicksort*, ולאחר מכן תבצע הכנסה לעץ חדש של כלל האיברים בסדר ממויין. המתודה תחזיר את העץ החדש הממויין.
סיבוכיות: כפי שראינו בסעיף 27, *listToArray* פועלת ב- $O(n)$, לאחר מכן, כפי שנלמד בקורס מבוא מורחב למדמ"ח, מתודת *quicksort* ממיינת את הרשימה בסיבוכיות של $O(nlog(n))$, ולבסוף מכיוון שאנחנו מבצעים סדרה של הכנסות ממויינת, אנחנו תמיד מכניסים את האיבר המקסימלי בכל רגע נתון, כלומר הפעולה המשמעותית ביותר היא פעולת המיון ולכן והסיבוכיות הכוללת של מתודת *sort* היא : $O(nlog(n))$.

31. *permutation(self) :*
מהות: המתודה תחזיר עץ חדש אשר מייצגים פרמוטציה רנדומלית של העץ המקורי.
אופן הפעולה: המתודה תבצע בדיקה האם העץ, אם כן, תחזיר עץ חדש ריק. אחרת, המתודה תבצע קריאה ל-*listToArray* (ראה סעיף 27), לאחר מכן תשנה את מיקומי כלל האיברים ברשימה בצורה רנדומית ע"י שימוש ב-*random*. לאחר מכן תבצע הכנסה לעץ חדש של כלל האיברים באופן טורי בדומה להכנסה במתודת *sort*.
סיבוכיות: כפי שראינו בסעיף 27, *listToArray* פועלת ב- $O(n)$, לאחר מכן, מעבר על כל איברי המערך פועל ב-$O(n)$, ולבסוף הכנסה כסדרת הכנסות למקסימום תבוצע גם כן ב-$O(n)$.
שפי שראינו בשיעור ניתוח amortzed לסדרת הכנסות או מחיקות של n פעולות כאשר יודעים תמיד להיכן מכניסים (כאן נכניס למקסימום אליו יש תמיד מצביע) תקרה בזמן ריצה amorted של *O(1)* לפעולה ולכן הסיבוכיות היא כנ"ל . סה"כ נקבל סיבוכיות ריצה של $O(n)$.

32. *concat(self)*:
מהות: המתודה מקבלת רשימה *lst* המיוצגת בתור *AVLTreeList* ומשרשרת אותה בסיום הרשימה המיוצגת גם היא בתור *AVLTreeList* עליה היא הופעלה. בנוסף המתודה מחזירה את הפרש הגבהים(בערכו המוחלט) הראשוני בין 2 העצים המייצגים את הרשימות. **נציין – כפי שהובהר בפורום – אם אחד העצים ריק נתייחס אליו כאל עץ בגובה 1- ולכן הפרש הגבהים יהיה הגובה של העץ השני 1+ . אם שני העצים ריקים נחזיר 0 שכן זה הפרש הגבהים ביניהם**
אופן הפעולה : המתודה בודקת תחילה האם הרשימה שעליה המתודה הופעלה היא ריקה. **במידה וכן –** המתודה תגדיר את הרשימה הראשונה להיות הרשימה השנייה, lst שהוכנסה כקלט, להיות הרשימה השנייה באמצעות עדכון בעלות קבועה של השדות firstItem, lastItem, root. אם גם הרשימה השנייה ריקה אז המתודה תחזיר 0 כהפרש הגבהים, אחרת תחשב את ערכו המוחלט של הפרש גובה הרשימה הראשונה הריקה(1- ע"פ הגדרה) וגובה השורש של הרשימה שנקלטה כקלט.
במידה ולא- המתודה תשמור בשני משתנים זמניים את גבהי שני העצים, ובמשתנה נוסף את הצומת האחרונה של העץ הראשון. כלל פעולות אלה הינן בעלות קבועה. לאחר מכן, תמחק המתודה את האיבר האחרון ברשימה הראשונה(שנשמר כפרמטר זמני). ותתבצע קריאה למתודת joinTrees(ראה סעיף 52) עם פרמטר הצומת האחרונה של הרשימה הראשונה והרשימה השנייה. מתודה זו כפי שמוסבר בסעיף 52 מבצעת חיבור של שני עצים כפי שנלמד בקורס. לבסוף, מתבצע בדיקה האם הרשימה השנייה ריקה. אם ריקה, מחושב הפרש הגבהים של הרשימה הראשונה והרשימה השנייה(1- ע"פ הגדרה), אחרת מחושב הפרש הגבהים על סמך שני המשתנים הזמניים שנשמרו שמייצגים את גבהי העצים.

סיבוכיות: נשים לב שמירב הפעולות שמתבצעות במתודה זו הינן בעלות קבועה. ננתח את יתר הפעולות.
מחיקת האיבר האחרון מהעץ הראשון: כפי שראינו בסעיף 24 פעולה זו מתבצעת בסיבוכיות של $O(\log(n))$.
בנוסף, כפי שראינו בקורס ומוסבר בסעיף 52 פעולת join ל-2 עצים לוקחת גם כן $O(\log(n))$. ולכן סה"כ סיבוכיות: $O(\log(n))$.

33. search(self,val) :
מהות: החזרת האינדקס הראשון ברשימה בו מופיע הערך *val*, או 1- אם לא קיים ערך כזה.
אופן הפעולה וניתוח סיבוכיות: נעבור על כל איברי הרשימה באופן הבא:
1. נאתחל מצביע לצומת שעליו מצביע השדה *firstItem* ומשתנה *index* לערך 0 . עלות $O(1)$
2. כל עוד המצביע לא מצביע על *None* נבצע את הפעולות הבאות:
   - אם הערך של הצומת שעליו מצביע המצביע הוא *val* נחזיר את *index*
   - אחרת, נקדם את המצביע להצביע על ה-*successor* של הצומת הנוכחי שעליו הוא מצביע ונקדם את *index* ב-1 .נבצע את פעולת ה-*successor* באמצעות המתודה *successor(node)*(ראה סעיף 40)

בשלב זה נבצע לכל היותר *n* פעולות *successor* וראינו בקורס הוכחה לכך שהעלות של *n* פעולות *successor* ברצף שמתחילות בצומת המינימלי בעץ חיפוש בינארי היא $O(n)$
מכיוון ששאר פעולות שמתבצעות בשלב 2 הינן בעלות זמן קבוע, עלות שלב 2 היא $O(n)$.
3. אם הגענו לכאן, סימן שלא קיים האיבר ברשימה שערכו *val* ולכן נחזיר 1-.

סה"כ סיבוכיות זמן ריצה במקרה הגרוע: $O(n)$

34. *getRoot(self):*
מהות: המתודה מחזירה את השורש של העץ המייצג את הרשימה עליה הופעלה המתודה.
אופן הפעולה: החזרת השדה *root* של העץ.
סיבוכיות: $O(1)$.

35. treeRank(self,node) :
מהות: המתודה מקבלת כקלט צומת ומחזירה את דרגתה.
אופן הפעולה: המתודה פועלת בהתאם לאלגוריתם שנלמד בקורס. תחילה, מאתחלת משתנה *r*, אשר מייצג את גודל תת העץ השמאלי, ומוסיפה לו 1 עבור הצומת שהמתודה הופעלה עליה. לאחר מכן המתודה עולה עד לשורש ובכל פעם בעלייה שאנחנו פונים שמאלה מוסיפים את מספר הצמתים שבתת העץ השמאלי של הצומת הנוכחית בטיפוס העץ. לבסוף מחזירים את *r*.
סיבוכיות: כלל הפעולות של קריאות למתודות *getLeft().size, y.getParent(), getRight()* מתבצעות בסיבוכיות של $O(1)$ כפי שהוסבר במסמך זה כבר. מכיוון שגובה העץ חסום ע"י $O(\log(n))$ ומתבצע סיור לכל היותר מעלה של העץ עד לשורש, סה"כ סיבוכיות: $O(\log(n))$.

36. *treeSelect(self,k) :*
מהות: מתודת עזר, שמחזירה את הצומת שדרגתה היא *k*. המתודה מחזירה את האיבר שנמצא ברשימה באינדקס ה-*(k-1)*.
אופן הפעולה: המתודה קוראת למתודת עזר רקורסיבית *selectRec(self.root,k)*. נסביר בסעיף 37 על אופן פעולתה.
סיבוכיות: המתודה קוראת רק למתודת עזר בעלת סיבוכיות $O(\log(n))$.

37. *selectRec(self,node,k) :*
מהות: המתודה מחזירה את הצומת שדרגה *k*

אופן הפעולה: המתודה פועלת על בסיס האלגוריתם שנלמד בקורס. למתודה מועבר השורש של העץ, בנוסף לפרמטר *k* שמייצג את דרגתו של הצומת המבוקשת. תחילה, נחשבת את גודל תת העץ השמאלי של השורש, נסמנו *r*. אם דרגת השורש שווה ל-*k* הרי שהצומת המבוקשת הינה השורש וסיימנו. לאחר מכן נבצע השוואה של דרגת השורש לעומת פרמטר *k*. אם מתקיים *k<r* הרי שאנחנו צריכים לחפש את האיבר ה-*k* הקטן ביותר בתת-העץ השמאלי. לכן תבוצע קריאה רקורסיבית למתודה זו עם הפרמטרים: *node.getLeft()*- כלומר הבן השמאלי של הצומת הנוחכית(במקרה ההתחלתי, הבן השמאלי של השורש), ועם *k*.
אם מתקיים *k>r* הרי שאנחנו צריכים לחפש את האיבר ה-(*k-r*) הקטן ביותר בתת-העץ הימני. לכן תבוצע קריאה רקורסיבית למתודה זו עם הפרמטרים: *node.getRight()*, *k-r*.
סיבוכיות: בכל קריאה אנחנו מבצעים פעולות אריטמטיות וקריאה למתודות אשר מבוצעות בזמן קבוע. לכל היותר אנחנו עוברים בכל העץ, ומכיוון שגובה העץ חסון ע"י $O(\log(n))$, נקבל סיבוכיות כוללת של $O(\log(n))$.

38. *treeMin(self,node)* :
מהות: המתודה מחזירה את הצומת בעץ שמייצג את הרשימה שמכיל את האיבר הראשון ברשימה, ללא שימוש בשדה המצביע אליו.
אופן הפעולה: המתודה ממומשת באמצעות ירידה מהשורש עד הסוף שמאלה והחזרת הצומת שהוא עלה במסלול זה.
סיבוכיות: מכיוון שמספר האיטרציות חסום ע"י גובה העץ- $O(\log(n))$, ובכל איטרציה מבוצעות קריאה למתודות *getLeft, getValue* שמבוצעות בזמן קבוע, לכן עלות המתודה בסה"כ היא $O(\log(n))$.

39. *treeMax(self,node)* :
מהות: המתודה מחזירה את הצומת בעץ שמייצג את הרשימה שמכיל את האיבר האחרון ברשימה, ללא שימוש בשדה המצביע אליו.
אופן הפעולה: המתודה ממומשת באמצעות ירידה מהשורש עד הסוף ימינה והחזרת הצומת שהוא עלה במסלול זה.
סיבוכיות: מכיוון שמספר האיטרציות חסום ע"י גובה העץ- $O(\log(n))$, ובכל איטרציה מבוצעות קריאה למתודות *getRight, getValue* שמבוצעות בזמן קבוע, לכן עלות המתודה בסה"כ היא $O(\log(n))$.

40. *successor(self, node)* :
מהות: המתודה מקבלת צומת ששייך לעץ. אם הדרגה של הצומת היא *i* אז המתודה תחזיר את הצומת שדרגה היא *i+1*. אם הצומת הוא הצומת המייצג את האיבר האחרון ברשימה, המתודה תחזיר *None*.
אופן הפעולה: המתודה פועלת בהתאם לאלגוריתם שראינו בקורס. אם לצומת יש בן ימיני אמיתי, אזי המתודה תחזיר את הצומת עם הדרגה המינימלית בתת-העץ של הבן הימני בעזרת קריאה למתודה *treeMin(node,getRight())* (ראה סעיף 38). במידה ולצומת אין בן ימני אמיתי, נעלה במעלה העץ עד הפנייה ימינה הראשונה בעץ. אם הגענו ל-*None* נחזיר *None*, אחרת, נחזיר את הצומת.
סיבוכיות: מספר האיטרציות חסום ע"י גובה העץ, $O(\log(n))$. בכל שלב מתבצעת קריאה למתודות אשר פועלות בזמן קבוע ולכן סה"כ זמן הריצה הוא $O(\log(n))$.

41. *predecessor(self,node)* :
מהות: המתודה מקבלת צומת ששייך לעץ. אם הדרגה של הצומת היא *i* אז המתודה תחזיר את הצומת שדרגה היא *i-1*. אם הצומת הוא הצומת המייצג את האיבר הראשון ברשימה, המתודה תחזיר *None*.
אופן הפעולה: המתודה פועלת בהתאם לאלגוריתם שראינו בקורס. אם לצומת יש בן שמאלי אמיתי, אזי המתודה תחזיר את הצומת עם הדרגה המינימלית בתת-העץ של הבן הימני בעזרת קריאה למתודה

treeMax(node,getLeft()) (ראה סעיף *39*). במידה ולצומת אין בן שמאלי אמיתי, נעלה במעלה העץ עד הפנייה שמאלה הראשונה בעץ. אם הגענו ל-*None* נחזיר *None*, אחרת, נחזיר את הצומת.
סיבוכיות: מספר האיטרציות חסום ע"י גובה העץ, $O(\log(n))$. בכל שלב מתבצעת קריאה למתודות אשר פועלות בזמן קבוע ולכן סה"כ זמן הריצה הוא $O(\log(n))$.

42. *avlDelete(self,i) :*
מהות: מתודה זו מוחקת מעץ avl כפי שלמדנו בכיתה את האיבר באינדקס *i*.
אופן הפעולה: ראשית, נעזר ב *treeSelect*(ראה סעיף 36) על מנת למצוא את הצומת הרצוי למחיקה .
מקרי הקצה שנרצה לבדוק כאן הם שלושת מקרי הקצה הבאים :
האם הצומת הוא הצומת הראשון בעץ – נרצה לשנות את שדה המינימום בעץ להצביע על האיבר העוקב.
האם הצומת הוא האחרון – נרצה לשנות את שדה המקסימום להצביע על קודמו
האם הצומת הוא הקודם למקסימום – נרצה לשנות את המצביע לשדה המקסימום לאיבר שאנו מוחקים
כל אלו משיקולי מחיקה בעץ בינארי – נזכור שבמקרים בהם לצומת יש שני בנים, לא נמחק את הצומת עצמו אלא את העלה שהוא עוקבו , לכן המקרה קצה השלישי הוא ככתוב
בנוסף שלושת מקרי קצה אלו נועדו לשמור מצביעים נכונים במהלך ריצת התוכנית לכיוון המינ' והמקס' .
בהמשך נקרא לפונקציית *bstDelete* (ראה סעיף 50) שאחראית על מחיקה מעץ בינארי כפי שנלמד ועובדת בזמן ריצה לוגריתמי (נפרט בהמשך) , נשמור את צומת האב של הצומת שבאמת נמחק על מנת להמשיך בתיקונים בהמשך העץ ונקרא בנוסף גם ל *fixAfterDeletion* (ראה סעיף 46) שתתקן בצורה דומה ל *fixAfterInsertion* , ונקרא לה עם צומת ההורה שקיבלנו ממתודת המחיקה מעץ בינארי.
סיבוכיות: כפי שניתן לראות המתודות *fixAfterDeletion , bstDelete* (ראה סעיף 50, סעיף 46 בהתאמה) עובדות בזמן ריצה לוגריתמי , וכיוון שמלבדן אנו קוראים ל- *predecessor, successor* שגם זמן ריצה הוא לוגריתמי ונבצע פעולות בזמן קבוע עליהם , נקבל שזמן הריצה של מתודה זו הוא $O(\log(n))$.

43. *treeInsert(self,node,i) :*
מהות: מתודה זו עוסקת בהכנסה לעץ .
אופן הפעולה: ראשית נבצע מספר בדיקות עבור מקרי קצה בהן נבדוק אם העץ ריק , או אם נכניס למקסימום או למינימום . במקרה שהעץ ריק נכניס צומת כאיבר הראשון, עלות ריצה קבועה.. בשני מקרי הקצה האחרים , אנו מכניסים למקסימום ולמינימום אליהם נתחזק מצביעים במהלך ריצת התוכנית , ולכן נמנע מלחפש את הצומת הרלוונטי אליו נצטרך להכניס ובעצם במקרים אלה, זמן הריצה יהיה גם כן קבוע $O(1)$. כשאיננו באף אחד מן המקרים הנ''ל , אנו מכניסים איבר בעץ במקום שאינו ידוע מראש . נשתמש ב treeSelect (ראה סעיף 36) על מנת למצוא את האיבר הנוכחי שנמצא באינדקס בו אנו רוצים להכניס את האיבר החדש . אם אין לו בן שמאלי , נכניס את הצומת החדש כבן שמאלי ואחרת נלך לקודמו ע''י predeccessor (ראה סעיף 41) ונכניס את הצומת החדש כבן ימני עבורו . כך נדאג למעשה שהאיבר החדש יהיה קטן ב-1 מהאיבר שהיה באינדקס הרצוי עד כה , וכך נזיז למעשה את כל האיברים שגדולים ממנו "קדימה" באינדקס .
סיבוכיות: סה''כ מתודה זו מבצעת קריאה ל- treeSelect , predeccessor ושתיהן במקרה הגרוע פועלות ב $O(\log(n))$. לכן סה''כ חסם עליון על סיבוכיות זמן הריצה של $O(\log(n))$.

44. avlTreeInsert(self,item,i) :
מהות: מתודה זו בונה צומת עבור הערך שקיבלנו, לאחר מכן קוראת לפונקציה treeInsert (ראה סעיף 43), מתודת עזר שמטרתה להכניס צומת לעץ. לסיום , המתודה הנ''ל קוראת למתודת עזר נוספת בשם fixAfterInsertion (ראה סעיף 45) שמהותה לתקן את העץ לאחר הכנסה ובכך לשמור עליו עץ avl חוקי , היא סופרת בנוסף את מספר הגלגולים שנעשו בהכנסה .
המתודה תחזיר את מספר הגלגולים ש fixafterinsertion ביצעה.

סיבוכיות: treeInsert כפי שמפורט בסעיף 43 מכניסה לעץ בינארי וכיוון שאנו מתחזקים עץ AVL תקין סיבוכיות זמן הריצה שלה יהיה $O(\log(n))$, ובנוסף fixAfterInsertion שמתקנת לאחר ההכנסה , עולה במעלי העץ שגם כן חסום ע"י $O(\log(n))$ ומבצעת מספר קבוע של גלגולים , סופרת אותם ועולה עד השורש על מנת לעדכן שדות . כיוון שגלגולים (ראה סעיפים 47-49) הם רק הזזה של מצביעים ולכן זמן הריצה לגלגול הוא קבוע.
מתבצעים רק עד שני גלגולים בתיקון ולכן סך התיקונים קבוע, בנוסף עדכון השדות במעלה העץ קבוע. לכן זמן הריצה הינו קבוע כמספר הצמתים שעברנו (שהוא גובה העץ). נקבל שזמן הריצה של המתודה הוא: $O(\log(n))$.

45. fixAfterInsertion(self,node) :
מהות: זוהי מתודת עזר של המתודה avlTreeInsert, שמטרתה לתקן את העץ לאחר הכנסת צומת חדש אליו על מנת לשמור על תכונות העץ AVL.
אופן הפעולה: מתודה זו בפשטות עוברת בלולאה מהצומת הנתון לאביו עד שמגיעה אל השורש. בכל צומת שעוברת בו , מעדכנת את שדותיו, ובודקת את גורם האיזון שלו . במקרה שגודל גורם האיזון קטן מ-2 , ממשיכה לאיטרציה הבאה , אחרת היא נעזרת בפונקציית fixWithRotates (ראה סעיף 47) על מנת לתקן את גורם האיזון .
סיבוכיות: כפי שניתן לראות בסעיף 47 fixWithRotates פועלת בזמן קבוע, בנוסף אנו מעדכנים שדות ועושים בדיקה של שדה לכל צומת ולכן בכל איטרציה זמן הריצה הוא קבוע . אנו מבצעים $O(\log(n))$ איטרציות כגובה העץ ולכן זמן הריצה הכולל הוא : $O(\log(n))$.

46. fixAfterDeletion(self, parent) :
מהות: זוהי מתודת עזר של המתודה avlDelete, שמטרתה לתקן את העץ לאחר מחיקת צומת על מנת לשמור על תכונות העץ AVL.
אופן הפעולה: מתודה זו עובדת בצורה זהה למתודה fixAfterInsertion(ראה סעיף 45), אך נציין שמתודה זו מתחילה לעבוד על הצומת הראשון שמקבלת ולא על צומת האב שלה כמו המתודה fixAfterInsertion.
סיבוכיות: מכיוון שהמתודה כמעט זהה למתודה fixAfterInsertion, מתבצעת איטרציה אחת נוספת, סה"כ סיבוכיות זמן הריצה הכולל הוא : $O(\log(n))$.

47. fixWithRotates(self,node) :
מהות: מתודה זו מתקנת בעזרת גלגולים את גורם האיזון של צמתים כפי שראינו בכיתה.
אופן הפעולה: המתודה בודקת לכל צומת את גורם האיזון שלו, בהתאמה את גורם האיזון של צומת הבן שלו שמצידו נובע חוסר האיזון ובהתאם לזה מבצעת גלגולים. למשל, אם גורם האיזון של צומת הוא +2 ושל בנו יהיה +1 או 0, ראינו בכיתה במקרה זה נבצע גלגול ימינה , ולכן תבצע קריאה למתודת העזר rightRotation (ראה סעיף 48). מתודה זו עוברת על כל מקרי הקצה שנלמדו בקורס ומטפלת בגלגולים בהתאם ע"י קריאה למתודות עזר rightRotation, leftRotation.מתודה זו לבסוף תחזיר את מספר הגלגולים שבצעה, וכיוון שכל מה שעושה זה מספר בדיקות אריתמטיות וקוראת לפונקציית גלגול שזמן הריצה שלה קבוע , נקבל שזמן הריצה של fixwithrotates הוא : $O(1)$.

48. rightRotation(self,node) :
מהות: מתודה זו מקבלת צומת שהוא עבריין AVL בעל balance_factor = |2| ומבצעת גלגול ימינה על מנת לתקן את ההפרה.
אופן הפעולה: מתודה זו פועלת ע"ב האלגוריתם שנלמד בקורס לסיבובים של עץ AVL. למשל , מתודת הגלגול ימינה תיקח את הבן השמאלי של הצומת שהמתודה נקראה עליו, ותשנה את המצביעים של העץ בין הצמתים כך שלאחר גלגול ימינה הבן השמאלי שהיה מקודם , יהיה האב של הצומת שקראנו עליו את המתודה .

סיבוכיות: כל הפעולות הן פעולות קבועות אשר מעדכנות שדות בעלות קבועה או משנות מצביעים, לכן סה"כ סיבוכיות $O(1)$.

49. leftRotation(self,node) :
מתודה זו זהה וסימטרית למתודה rightRotation, ראה סעיף 48.

50. bstDelete(self,node) :
מהות: מתודה זו באה לממש מחיקה מעץ בינארי כפי שנלמד בכיתה.
אופן הפעולה: המתודה מקבלת צומת שאותו נרצה למחוק, ועושה מספר בדיקות :
1. בודקת האם הוא עלה ע"י קריאה למתודה isLeaf (ראה סעיף 16), נמחק אותו ואת המצביעים אליו.
2. אם איננו עלה, נבדוק אם לצומת יש בן יחיד. אם כן, נבצע מעקף ונקשר את אב הצומת הנמחק ובנו של הצומת הנמחק כאב ובן חדשים.
3. אם איננו עלה ויש לו שני בנים – נחפש את עוקבו ע"י קריאה למתודה successor(ראה סעיף 40), נחליף בין הערכים שלו ושל עוקבו ואז נקרא למתודה bstdelete עם עוקבו. מכיוון שיש לו שני בנים, עוקבו הוא עלה בתת העץ הימני שלו ולכן bstdelete תמחק אותו כיוון שיהיה אחד מבין שני המקרים הראשונים שהעלינו .

בנוסף הוספנו כאן מקרה של הסרת צומת השורש מעץ בגודל 2 , כי אז רק נותר להגדיר את הצומת האחרונה כשורש החדש וזמן ריצה יהיה קבוע .
סיבוכיות: סה''כ זמן הריצה הגרוע ביותר עבור מתודה זו יהיה במקרה השלישי, בו נצטרך לחפש את העוקב לצומת הנ''ל. מלבד זה ישנן רק השוואות ובדיקות מצביעים ועל כן זמן הריצה יהיה כמו של מתודת successor, וזמן הריצה הסופי יהיה :$O(\log(n))$.

51. treeHeight(self):
מהות: מתודה שמחזירה את גובה העץ.
אופן הפעולה: אם העץ ריק, המתודה תחזיר 1-. אחרת, תחזיר את הגובה של השורש העץ באמצעות שימוש בשדה המצביע לשורש.
סיבוכיות: $O(1)$.

52. joinTrees(self,x,tree2) :
מהות: מתודה זו, בהינתן שני מופעים של עצי AVL המייצגים רשימות, וצומת X , מאחדת את הרשימות כך שאיברי הרשימה עליה הופעלה המתודה הם האיברים הראשוניים, לאחר מכן X, ולבסוף כלל איברי העץ tree2.
אופן הפעולה וניתוח סיבוכיות: המתודה פועלת על בסיס האלגוריתם שראינו בקורס לאיחוד 2 עצים. נסמן גובה העץ של העץ שעליו הופעלה המתודה כ-h1, גובה עץ tree2 כ-h2.
נחלק למקרים:
1. אם tree2 מייצג רשימה ריקה: יבוצע הכנסה של האיבר X לעץ הראשון באינדקס האחרון. סה"כ סיבוכיות $O(h1-h2+1)$
2. אם העץ שעליו מופעלת המתודה מייצג רשימה ריקה: המתודה תכניס את האיבר X למקום הראשון בעץ tree2. ותגדיר את tree2 להיות הרשימה שעליו הופעלה המתודה ע"י שינוי שדות המצביעים. סה"כ סיבוכיות $O(h2-h1+1)$.
3. אם שתי הרשימות אינן ריקות:
   a. אם גבהי העצים זהים: המתודה מגדירה את השורש של העץ המייצג את self להיות בנו השמאלי של X. ואת השורש של העץ המייצג את tree2 להיות בנו הימני. לאחר מכן המתודה מגדירה את ההורה של X להיות None ואת X להיות השורש.

b. אם $h1 < h2 \ or \ h2 < h1$: נסביר כיצד פועלת המתודה כאשר $h2 > h1$ כאשר המקרה ההפוך ממומש בצורה סימטרית .נסייר החל מהשורש של העץ המייצג את self כלפי מטה(ימינה) ונשמור את הצומת הראשון, (נסמנו curr) אשר הגובה שלו קטן או שווה לגובה של השורש tree2 . נשמור גם את ההורה של curr . בלולאה זו יתבצעו $O(h1 - h2 + 1)$ איטרציות שכן הגובה של הצומת curr שווה h2 או h2-1 , ומשום שבכל איטרציה עלות הפעולות קבועה, העלות הכוללת של הלולאה במקרה הגרוע היא $O(h1 - h2 + 1)$. נגדיר את בנו השמאלי של X להיות curr, ואת בנו הימני להיות השורש של העץ המייצג את tree2 . נגדיר את X להיות הבן של ההורה המקורי של curr . עלות פעולות אלו היא $O(1)$.
*במקרה ההפוך בו tree2 הוא העץ הגבוה יותר, נגדיר את השדה root של self להצביע על השורש של tree2.

נגדיר את השדה lastItem של self להיות הצומת שמוגדר כ-lastItem של tree2. נבצע עדכון לשדות של הצומת X. ככל שיש ל-X הורה, נתקן את העץ self שעליו הופעלה המתודה עד לשורש באמצעות קריאה למתודה fixAfterDeletion(ראה סעיף 46).

סה"כ סיבוכיות זמן הריצה תלויה בגובה העצים, מכיוון שחסם עליון על גובה העצים הוא לוגריתמי נקבל סיבוכיות של $O(\log(n))$ כאשר n הוא מספר האיברים הכולל בשתי הרשימות.

חלק ניסויי/תיאורטי

שאלה 1

.1

מספר סידורי -i	ניסוי 1- הכנסות n	ניסוי 2- מחיקות n	ניסוי 3- סירוגין, רק הכנסות n/2	ניסוי 3- סירוגין n/2(סופר רק מחיקות והכנסות)
1	2087	1086	1028	804
2	4273	2237	2058	1570
3	8393	4517	4157	3133
4	16892	9001	8446	6287
5	33343	18072	16698	12712
6	67228	35959	33235	25493
7	133893	71458	66910	50776
8	268207	143300	133363	101307
9	537002	287340	268229	203173
10	1072815	572439	535426	405345

2. ניתן לראות בבירור כי הביטוי האסימפטומטי התואם כל עמודה הוא $\theta(n)$

שאלה 2

תחילה, נתאר את ציפיות הניסוי שלנו. כאמור הניסוי בודק זמני ריצה של 3 סוגי של הכנסות למבני הנתונים השונים: הכנסה בהתחלה, הכנסה בסוף והכנסה במקום רנדומלי. ראשית ניזכר כי כשאנו מדברים במובני O(f(n)) אנו מדברים על קצב הגדילה של זמני הריצה ביחס לגדלי הקלט השונים .
ננתח כל הכנסה בנפרד.

<u>עבור הכנסה בהתחלה:</u>
בהתבסס על סיבוכיות זמני הריצה שלמדנו עבור פעולות הכנסה בהתחלה לשלושת מבני הנתונים השונים נצפה כי קצב הגדילה הטוב ביותר יהיה של רשימה מקושרת בזמן ריצה O(1), לאחר מכן הכנסה בהתחלה לעץ avl קצב הגדילה יהיה O(log(n)) ולסיום במערך נקבל O(n) . אכן התוצאות גם נראות מתאימות- ראשית התוצאות נראות מתעתעות כיוון שזמני הריצה במערך הם הקצרים ביותר, אך כמובן שהדבר נובע מכך שגישה לזיכרון במקומות שונים ארוכה מגישה למערך בה במחשב התוכן יושב רציף בזיכרון – כלומר מהמימוש הפיזי של רשימות מקושרות או עצים הדבר ייקח יותר זמן . אך ברגע שאנו מסתכלים בפרטים, ניתן לראות שקצב הגדילה של זמני הריצה במערך הוא המשמעותי ביותר כפי שציפינו, לאחר מכן במקום השני נמצא את קצב הגדילה של עצי ה avl שכתבנו ובמקום הראשון קצב הגדילה של הרשימה המקושרת יהיה הטוב ביותר .

<u>עבור הכנסה בסוף:</u>
במקרה זה, נצפה כי במקום הראשון המהיר ביותר יהיה **מערך** שכן על סמך הנלמד בקורס זמן הריצה של פעולה זו היא: $O(1)$.
לאחר מכן במקום השני נצפה למצוא את **עץ AVL**, שכן סיבוכיות זמן הריצה היא $O(logn)$. ולבסוף במקום השלישי נצפה למצוא את **רשימה מקושרת** שכן הסיבוכיות עבור פעולה זו היא $O(n)$.

<u>עבור הכנסה ברנדומליות:</u>
במקרה הרנדומלי, קשה לחזות את זמני הריצה השונים של עבודה עם הכנסה במערך, זאת כיוון שיש מקרים בהם מערך יכול לעבוד בזמן ריצה קבוע ובחלק מהמקרים יהיה תלוי בגודל המערך . אף על פי כן נצפה לראות את המערך במצב יותר טוב מהרשימה המקושרת, שזמני הריצה שלה צפויים להיות דומים ביותר לזמני ריצה של הכנסות בסוף בסיבוכיות זמן ריצה של O(n). לסיום נצפה שמבנה הנתונים **עץ AVL** יניב את התוצאות העקביות ביותר כיוון שזמן הריצה שלו אמור להיות תמיד O(log(n)) .
נציג את תוצאות מדידה של הניסויים, נציין כי כלל זמני המדידה שמוצגים בטבלה הינם עבור סך כל פעולות ההכנסה לעץ בגודל n מטעמי נוחות הצגת הנתונים. מכיוון שבכל בדיקה מבוצעות אותן מספר פעולות הכנסה ל-3 מבני הנתונים, חישוב הממוצע הוא גם כן זהה(חלוקה ב-n) ועל כן אין לכך חשיבות.
להלן תוצאות הניסוי:

זמן ריצה בממוצע מספר סידורי i	עץ AVL הכנסות להתחלה	רשימה מקושרת הכנסות להתחלה	מערך הכנסות להתחלה
1	0.06226530000000041	0.00178939999999983	0.0007536999999970817
2	0.15451309999999996	0.0032464000000000937	0.0023669000000055007
3	0.20106380000000001	0.004740300000000364	0.005402500000002419
4	0.3548800999999999	0.006572300000000197	0.008543099999997139
5	0.4093417000000006	0.009302200000000482	0.012728899999999044
6	0.4474532	0.02338249999999853	0.018002899999999045
7	0.5734732999999999	0.015515999999999863	0.02298730000000404
8	0.6708229999999986	0.013349900000001469	0.03355269999999422
9	0.768465299999999	0.02788680000000099	0.03698539999999895
10	0.851274851637001	0.02914459999999508	0.05286040000000014

זמן ריצה בממוצע מספר סידורי i	עץ AVL הכנסות אקראיות	רשימה מקושרת הכנסות אקראיות	מערך הכנסות אקראיות
1	0.10080329999999993	0.06409509999999985	0.0018862000000012813
2	0.23155389999999976	0.18101449999999986	0.0053082000000017615
3	0.2651458999999998	0.4366300999999999	0.00736779999999726
4	0.39635149999999975	0.7002682	0.010744600000002436
5	0.5331552999999998	1.0763914999999997	0.015319200000000421
6	0.5974012000000002	1.528432500000001	0.01981349999999793
7	0.6875588000000015	2.1157724999999985	0.02549799999999891
8	0.7922732000000003	2.7435772000000007	0.027155100000001653
9	0.8945873000000013	3.4980831000000023	0.03220380000000489
10	0.9852127999999993	4.4959605000000025	0.04574340000000632

זמן ריצה בממוצע מספר סידורי i	עץ AVL הכנסות בסוף	רשימה מקושרת הכנסות בסוף	מערך הכנסות בסוף
1	0.08423649999999983	0.07707950000000041	0.0002980000000007976
2	0.1784024999999998	0.29828469999999996	0.0006902999999951476
3	0.25790489999999977	0.6150564999999997	0.001041200000003073
4	0.3745631999999999	1.1042201	0.0012698000000028742
5	0.38147410000000015	1.6491242999999995	0.001880200000002219
6	0.4096985000000011	2.640180899999999	0.0018378999999981716
7	0.5956487999999993	3.2928870999999997	0.002497399999995764
8	0.5824772000000014	4.271066100000002	0.002395700000001
9	0.7739799000000005	5.292582100000001	0.0028920999999968444
10	0.904460799999999	6.462514399999996	0.0036893000000048914

כפי שניתן לראות, כלל תוצאות הניסוי תואמות את השערותינו.