AVL TREES PROJECT

Documentation

Gal Nadler(206964090) and Tal Baram (209036524)

I. מחלקה- AVLNode

תיאור המחלקה-המחלקה מגדירה את האובייקט node והוא ישמש כצומת בעץ ה- AVL בנוסף המחלקה תגדיר פונקציות שונות הנוגעות לגישה לנתונים של nodes.

a. פונקציות שנתבקשו-

i. init (self, key=None, value=None, parent=None)-

מה הפונקציה עושה: מאתחלת node חדש

כיצד הפונקציה פועלת: הפונקציה מקבל באופן דיפולטי מפתח, ערך והורה כאשר כולם מאותחלים לNone, ניתן ליצור node רגיל, כלומר צומת שמכניסים לה ערך ומפתח וניתן לאתחל node וירטואלי, כאשר הערך והמפתח שלו מאותחלים ל-None. בעת יצירת node רגיל, אנו אוטומטית מייצרים לו בן ימני ובן שמאלי וירטואליים.

השדות שמתאחלת הפונקציה הינם-

Self. Parent- "ההורה" של ה-node

Self. Left- ה"ילד" השמאלי של הnode, בעלה הנ"ל הוא צומת וירטואלי

Self. Right- ה"ילד" הימני של ה-node, בעלה הנ"ל הוא צומת וירטואלי

Self. Height- הגובה של ה-node, יאותחל ל-0 באופן ראשוני, או ל-1 בצומת וירטואלי

Self. Size- גודל העץ שתחת ה-node, יאותחל ל-1 באופן ראשוני (ה-node עצמו) וכן ל-0 בצומת וירטואלי.
self.Oldheight- מסמל "גובה ישן" וישמש אותנו בהמשך.

פירוט סיבוכיות זמן ריצה:**O(1)**, השמות, חישובים פשוטים, ותחזוקת השדות.

ii. Get_key(self)-

מה הפונקציה עושה: מקבלת node ומחזירה את המפתח שלו

כיצד הפונקציה פועלת: ניגשת לשדה המפתח ומחזירה משם את ערך המפתח

פירוט סיבוכיות זמן ריצה:**O(1)**

iii. Get_value(self)-

מה הפונקציה עושה: מקבלת node ומחזירה את הערך שלו

כיצד הפונקציה פועלת: ניגשת לשדה הערך ומחזירה משם את הערך

פירוט סיבוכיות זמן ריצה:**O(1)**

iv. **Get_left(self)-**

<u>מה הפונקציה עושה:</u> מקבלת node ומחזירה מצביע לבן השמאלי שלו

<u>כיצד הפונקציה פועלת:</u> ניגשת לשדה הבן השמאלי ומחזירה משם מצביע לבן השמאלי

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)**

v. **Get_right(self)-**

<u>מה הפונקציה עושה:</u> מקבלת node ומחזירה מצביע לבן הימני שלו

<u>כיצד הפונקציה פועלת:</u> ניגשת לשדה הבן הימני ומחזירה משם מצביע לבן הימני

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)**

vi. **Get_parent(self)-**

<u>מה הפונקציה עושה:</u> מקבלת node ומחזירה מצביע להורה שלו

<u>כיצד הפונקציה פועלת:</u> ניגשת לשדה ההורה ומחזירה משם מצביע להורה

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)**

vii. **Get_height(self)-**

<u>מה הפונקציה עושה:</u> מקבלת node ומחזירה את הגובה שלו

<u>כיצד הפונקציה פועלת:</u> אם הnode וירטואלי תחזיר 1-, אחרת תחזור את הגובה משדה הגובה.

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)**

viii. **Get_size(self)-**

<u>מה הפונקציה עושה:</u> מקבלת node ומחזירה את גודל העץ שתחתיו

<u>כיצד הפונקציה פועלת:</u> אם הnode וירטואלי תחזיר 0, אחרת תחזור את הגודל משדה הגודל.

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)**

ix. **Set_key(self,key)-**

<u>מה הפונקציה עושה:</u> מקבל מפתח וnode נתון ומשימה את המפתח לnode

<u>כיצד הפונקציה פועלת:</u> ניגשת לשדה המפתח ומשנה שם את ה-key לזה שקיבלה הפונקציה

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)**

x. **Set_value(self,value)-**

<u>מה הפונקציה עושה:</u> מקבל ערך וnode נתון ומשימה את הערך לnode

כיצד הפונקציה פועלת: ניגשת לשדה הערך ומשנה שם את ה-value לזה שקיבלה הפונקציה

פירוט סיבוכיות זמן ריצה:**O(1)**

xi. -Set_left(self, node)

מה הפונקציה עושה: משימה את node כבן השמאלי של self

כיצד הפונקציה פועלת: מקבלת מצביע לnode כלשהו, ניגשת לשדה הבן השמאלי של self ומשימה אותו כבן השמאלי של הצומת self.

פירוט סיבוכיות זמן ריצה: **O(1)**

xii. -Set_right(self, node)

מה הפונקציה עושה: משימה את node כבן הימני של self

כיצד הפונקציה פועלת: מקבלת מצביע לnode כלשהו, ניגשת לשדה הבן הימני של self ומשימה אותו כבן הימני של הצומת self.

פירוט סיבוכיות זמן ריצה: **O(1)**

xiii. -Set_parent(self,node)

מה הפונקציה עושה: משימה את node כהורה של self

כיצד הפונקציה פועלת: מקבלת מצביע לnode כלשהו, ניגשת לשדה ההורה של self ומשימה אותו כהורה של הצומת self.

פירוט סיבוכיות זמן ריצה: **O(1)**

xiv. -Set_height(self,h)

מה הפונקציה עושה: משימה את הערך h כגובה של self

כיצד הפונקציה פועלת: ניגשת לשדה הגובה של self ומשנה שם את הגובה לh.

פירוט סיבוכיות זמן ריצה:**O(1)**

xv. -Set_size(self,s)

מה הפונקציה עושה:משימה את הערך s כגודל של self

כיצד הפונקציה פועלת: ניגשת לשדה הגודל של self ומשנה שם את הגודל לs.

פירוט סיבוכיות זמן ריצה:**O(1)**

xvi. -is_real_node(self)

מה הפונקציה עושה:בודקת האם ה-node וירטואלי או רגיל

כיצד הפונקציה פועלת: ניגשת לשדה המפתח של self ובודקת האם הוא None, אם כן- self הוא וירטואלי, אחרת רגיל.

פירוט סיבוכיות זמן ריצה:**O(1)**

b. **<u>פונקציות שהוספנו-</u>**

i. **<u>BFF(self)-</u>**

<u>מה הפונקציה עושה</u>:מחשבת ה-balance factor

<u>כיצד הפונקציה פועלת:</u> הפונקנציה מחשבת את הגבהים של הבן הימני והבן השמאלי של self והרי זהו ה-BF.

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)** בהסתמך את פונקציות שהוסברו מעלה.

ii. **<u>setHeightAndSize(self)-</u>**

<u>מה הפונקציה עושה</u>:מעדכנת את הגובה והגודל על פי הבן השמאלי והימני

<u>כיצד הפונקציה פועלת</u>:את הגודל הפונקציה מחשבת על ידי סכימה של גודל הבן הימני, הבן השמאלי ועוד 1. את הגובה הפונקציה מחשבת על ידי בחירה במקסימום מבין הגובה של הבן השמאלי והימני וכן הוספת 1 (עבור self עצמו)

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)** בהסתמך את פונקציות שהוסברו מעלה.

iii. **<u>Successor_specific (self)-</u>**

<u>מה הפונקציה עושה: </u> מוצאת את ה – **<u>successor</u>** של הצומת הנוכחית (כלומר הצומת בעלת המפתח הכי קטן, מתוך כל הצמתים בעלי המפתחות הגדולים יותר מהנוכחית) , אך רק על צמתים בעלי שני בנים לא וירטואליים

<u>כיצד הפונקציה פועלת: </u>הפונקציה פועלת רק על צמתים בעלי בן ימני ובן שמאלי, לא וירטואליים. נרד לבן הימני של הצומת, ואז נרד שמאלה עד שנגיע לעלה (כאשר הצומת לא וירטואלית)

<u>פירוט סיבוכיות זמן ריצה: </u>**O(log n)** . במקרה הכי גרוע, בו אנו מחפשים את ה **<u>successor</u>** של שורש העץ, נרד במורד כל העץ, שגובהו במקרה הגרוע הוא log n.

II. **<u>מחלקה-AVLTree</u>**

<u>תיאור המחלקה-</u>המחלקה מממשת את האובייקט עץ AVL המורכב מה-nodes וכן מאפשר ביצוע מספר פעולות המגולמות בפוקנציות המחלקה ומפורטות מטה.

a. **<u>פונקציות שנתבקשו-</u>**

i. **<u>Init-</u>**

<u>מה הפונקציה עושה: </u>הפונקציה מאתחלת עץ AVL

<u>כיצד הפונקציה פועלת</u>:הפונקציה מאתחלת את העץ עם השדה root כ-None

<u>פירוט סיבוכיות זמן ריצה:</u>**O(1)**

ii. **<u>search(self, key)-</u>**

<u>מה הפונקציה עושה: </u>מחפשת node המתאים ל-key הנתון ומחזירה מצביע אליו

<u>כיצד הפונקציה פועלת</u>:הפונקציה עוברת על העץ בחיפוש בינארי ובכל פעם מחפשת את המפתח הנתון, כאשר הפונקציה מוצאת היא מחזירה את הערך של אותו ה-node. אם הפונקציה לא מוצאת את המפתח היא תחזיר None

פירוט סיבוכיות זמן ריצה: O(log n) שכן הפעולה המרכזית היא עצם החיפוש הבינארי, בסה"כ עוברים על כמות מפתחות כגובה העץ, בעץ AVL מדובר בlog n nodeים כמובן.

iii. -insert(self, key, val)

מה הפונקציה עושה:הפונקציה לוקחת את הערך והמפתח המוכנסים , מכינה מהם node חדש ומכניסה אותו לעץ כך שהעץ נותר עץ AVL

כיצד הפונקציה פועלת:ראשית הפונקציה יורדת לסוף העץ למקום שבו יש להכניס את ה-node (לפי key), לאחר מכן הפונקנציה משנה מצביעים ומקשרת את ה-node שלנו לעץ, לאחר מכן תפעיל הפונקנציה את Tree fixer, מה שיתקן את העץ ויגלגל במידת הצורך. בנוסף תטפל הפונקנציה במקרי קצה שבהם העץ ריק.

פירוט סיבוכיות זמן ריצה:O(log n) הסיבוכיות מושפעת מסיבוכיות Tree fixer וכן מהלולאה הראשנית בinsert שיורדת עד סוף העץ (מסלול באורך log n).

iv. delete(self,node)

מה הפונקציה עושה: הפונקציה מוחקת צומת מתוך עץ.

כיצד הפונקציה פועלת: ראשית,נטפל במקרי קצה בהם נדרש למחוק צומת "וירטואלית" או צומת שאינה קיימת בעץ, לאחר שבחנו את מקרי הקצה. הפונקציה בודקת כמה "ילדים " שאינם וירטואלים יש לצומת.

לאחר מכן נפריד לארבעה מקרים- הצומת הוא עלה, הצומת הוא בעל ילד ימני בלבד, הצומת הוא בעל ילד שמאלי בלבד, הצומת בעל שני בנים.

במידה והצומת עלה – אם הצומת היא גם השורש, נמחק את הצומת, נעדכן את השורש להיות None, ונחזיר 0 כיוון שאין פעולות גלגול.
אחרת נמחק את העלה בלבד.

נפריד בין המקרה של בן ימני לבן שמאלי לשני בנים
בהתאם למקרה, נחליף את המצביעים בהתאם.

כעת נקבל עץ לאחר מחיקת צומת אך טרם הגלגולים, לאחר כל זאת נקרא ל-TreeFixer וכן נחזיר את התוצאה שמחזיר TreeFixer.

פירוט סיבוכיות זמן ריצה: הסיבוכיות הינה O(log n) , נסביר -במהלך כלל הפוקנציה אנו משנים מצביעים ומבצעים קריאות לפונקציות ממחלקת ה-AVLnodes, כלל הפעלות עולות O(1) וכמות הפעולות אינה תלוי בגודל העץ, ועל כן כלל הפעולות הללו יהיו בעלות סיבוכיות של O(1), לבסוף, אנו קוראים לTreeFixer שעובד בסיבוכיות (O(log n ועל כן גם deleteעובדת בסיבוכיות זו.

v. AVL_to_array(self)

מה הפונקציה עושה: הפונקציה מקבל את העץ ומחזירה רשימה של טאפלים של המפתחות והערכים הנמצאים בעץ, הרשימה תמוין לפי ערך המפתחות.

כיצד הפונקציה פועלת: הפונקציה קוראת לפונקציה רקורסיבית שמבצעת את הפעולה, הסבר על הפונקציה הרקורסיבית מפורט מטה

פירוט סיבוכיות זמן ריצה: O(n) נובע מפעולת הפונקציה הרקורסיבית ומוסבר מטה

vi. **size(self)**

מה הפונקציה עושה: מחזירה את מספר ה-nodes בעץ self

כיצד הפונקציה פועלת: הפוקנציה תחזיר את הגודל של השורש (שהוא הגודל של כלל העץ) אם השורש וירטואלי או שאינו קיים, משמע שהעץ ריק ועל כן יוחזר 0.

פירוט סיבוכיות זמן ריצה:**O(1)** מכיוון שכלל הפעולות הן קריאה למספר סופי של פונקציות בגודל O(1)

vii. **split(self, node)**

מה הפונקציה עושה: הפוקנציה מקבלת עץ AVL -self וכן צומת שנמצאת בעץ node ומבקשת לפצל את העץ לשני עצי AVL נפרדים- עץ אחד שכולל בו את כלל המפתחות הקטנים מnode ועץ שני שכולל בו את כלל המפתחות הגדולים מnode.

כיצד הפונקציה פועלת:הפוקנציה מאתחלת שני עצים, שמאלי וימני, תחילה כל אחד מהם יהיה בהתאמה הבן הימני והבן השמאלי של node. ננתק מהם את node,

כעת נתחיל בלולאת while המקדמת בכל פעם את node להיות ההורה שלו:

בכל פעם נבדוק עבור node האם הוא ילד ימני או ילד שמאלי של ההורה שלו, אם הוא ילד שמאלי- ניצור עץ AVL חדש r כאשר השורש שלו יאותחל להיות הילד הימני של ההורה של node, על אותו הr נבצע join ביחד עם right tree , הjoin יהיה לפי המפתח של ההורה , כעת נקדם את node להיות ההורה. (באופן דומה עובדת הפעולה במידה וnode הוא ילד ימני)

לבסוף נדפיס רשימה של right_tree וכן left_tree.

פירוט סיבוכיות זמן ריצה:**O(log n)** לפי הניתוח שביצענו בכיתה, קריאה ל-join לוקחת O(height difference+1) ועל כן כאשר אנו מחברים בכל פעם שני עצים לאחד וקוראים ל-join , נקבל בסופו של דבר סיבוכיות O(log n)

viii. **join(self, tree, key, val)**

מה הפונקציה עושה: הפונקציה מחברת בין שני עצי AVL , וצומת נוספת, כאשר כל המפתחות של אחד העצים קטנים ממש מהמפתח של הצומת הנוספת, שקטן מכל המפתחות בעץ השני.
הפונקציה מחזירה את הפרש הגבהים בין העצים המקוריים שקיבלנו, הפרש שמתאר גם את עלות הזמן של הפונקציה.

כיצד הפונקציה פועלת: ראשית , התייחסנו למקרי הקצה בו אחד מהעצים ריק, ושם השתמשנו גם בפונקציה INSERT , שכן רק צריך להכניס את הצומת המיועדת לעץ.
במקרה הכללי יותר, הפרדנו בקוד בין 4 מקרים, בדקנו מי מבין שני העצים גדול יותר, ובתוך כל מקרה האם מפתחות העץ הגדול קטנים או גדולים מהצומת החיצונית הנוספת. התחלנו מהשורש של העץ הגדול, ובהתאם, ירדנו במורד העץ עד שאנו מגיעים לצומת שמהווה שורש של תת עץ בגובה העץ הקטן.
חיברנו בעזרת מצביע את שורש תת העץ שמצאנו לצומת החדשה (שהיא עכשיו ההורה של אותו שורש) תוך ניתוק מצביע מההורה המקורי שלו. לצומת החדשה הוספנו בן מהצד השני – שורש העץ הקטן.
לאחר מכן השתמשנו בפונקציה שיצרנו Tree Fixer , מהצומת החדשה עד מעלה העץ המאוחד, ולבסוף עדכנו את השורש והחזרנו את הפרשי הגבהים המקוריים.

פירוט סיבוכיות זמן ריצה: **O(log n)**

הירידה במורד העץ תעלה לנו כגובה העץ-O(log n) , לאחר מכן חיבור וניתוק

הצמתים ירוץ בסיבוכיות זמן קבועה o(1). הפונקציות TreeFixer ו-Update_root רצות גם הן בlog(n) במקרה הגרוע (כאשר אנו נמצאים בעלה), ספציפית בפעולת ה-join נתחיל את הפעולות מהצומת המקשרת בין העצים ועל כן הסיבוכיות אף תהיה פחותה מ- log n אלא תכלול את הפרשי הגבהים בין בעצים ועוד 1.

ix. **rank(self, node: AVLNode)**

מה הפונקציה עושה: הפונציה מוצאת את ה-rank של איבר

כיצד הפונקציה פועלת: נאתחל counter שסופר את גודל תת העץ השמאלי של ה-node הנתון ועוד אחד,ונעלה במעלה העץ, בכל פעם שהצומת הוא בן ימני של ההורה שלו , נעלה ונוסיף את גודל תת העץ השמאלי של האבא ועוד אחד. לבסוף נחזיר את ה-counter לאחר שנגיע לשורש.

פירוט סיבוכיות זמן ריצה:O(log n) כיוון שכלל הפעולות אריתמטיות, החישוב של size קורה ב-O(1) ועל כן זמן הריצה מושפע בעיקר מהעלייה במעלה העץ שהיא במקסימום בגודל של גובה העץ והוא log n בעץ AVL

x. **select(self, i)**

מה הפונקציה עושה:מחפש את האיבר ה-i קטן בעץ ומחזיר מצביע אליו

כיצד הפונקציה פועלת: הפוקנציה משתמשת ברקורסיה שבכל פעם מקדמת את ה-node ימינה או שמאלה במורד העץ ומחיפשת מתי ה-i שאנו מחפשים שווה לגודל העץ השמאלי ועוד 1- כלומר האיבר שבו אנו נמצאים הוא בדיוק האיבר ה-i.

הפונקציה מבצעת זאת על ידי הפעלה בכל פעם על הבן השמאלי או הימני של צומת, כאשר אם הפעלנו על הבן השמאלי זה מכיוון שi עדיין גדולה מגודל העץ השמאלי ועוד 1. אם הפעלנו על הבן הימני זה אומר שi קטנה ולכן צריך לגשת לתת העץ בו האיברים גדולים יותר ולחפש שם.

פירוט סיבוכיות זמן ריצה:**O(log n)**, שכן בתוך הפעולה הרקורסיבית אנו מבצעים בדיקות שעולות O(1) והמסימום שנרד בעץ יהיה כגובה העץ- log n.

xi. **get_root(self)**

מה הפונקציה עושה:מחזירה מצביע לשורש העץ

כיצד הפונקציה פועלת:הפונקציה ניגשת לשדה ה-root בעץ ומחזירה מצביע לצומת

פירוט סיבוכיות זמן ריצה:**O(1)**

b. **פונקציות שהוספנו-**

i. **max_tree(self)-**

מה הפונקציה עושה: מוצאת הצומת בעלת המפתח הגדול ביותר בעץ

כיצד הפונקציה פועלת: הפונקציה מקבלת צומת, משווה את המפתח שלה למפתח של המקסימלי הנוכחי של העץ, אם המפתח של הצומת קטן יותר, נשאיר את המקסימלי הקודם, אחרת נעדכן את המקסימלי לצומת החדש.

פירוט סיבוכיות זמן ריצה: **O(1)** כיוון שאנו מתחזקים שדה.

ii. **update_root(self)**

מה הפונקציה עושה: מעדכנת את שורש העץ

כיצד הפונקציה פועלת: הפונקציה מתחילה מהשורש האחרון המעודכן של העץ, עולים במעלה העץ עד שמגיעים לצומת שההורה שלה הוא None, ומעדכנים לפיה את השורש

פירוט סיבוכיות זמן ריצה: **O(log n)** במקרה הגרוע, נעלה במעלה כל גובה העץ גדי לעדכן את השורש. לכן סיבוכיות הזמן כגובה העץ - O(log n).

iii. **AVL_to_array_rec(self, node)**

מה הפונקציה עושה:הפונקציה היא הגרסא הרקורסיבית של AVL to array ומחזירה רשימה עם טאפלים של מפתח וערך הממויינת לפי מפתח וערך.

כיצד הפונקציה פועלת:בכל פעם הפונקציה מתפצלת ל-
left_list,right_list,node_list כאשר בכל פעם מפעילה על left_list ו-right_list את הפוקנציה בשנית, כאשר ה-node המוכנס יהיה בהתאמה
node.get_right(),node.get_left(), בתנאי node_list יוחזר בסופו של דבר עבור כל node המפתח והערך שלו בטאפל. לבסוף יוחזרו כלל הרשימת באופן משורשר.

Node וירטואלי או ששוה None לא יוחזר

פירוט סיבוכיות זמן ריצה: **O(n)** ,הסיבוכיות הינה O(n) שכן בסה"כ אנו עוברים בכל הפונקציה על n עברים ומחזירים עבורם ערך ומפתח.

iv. **TreeFixer(self, node)**

מה הפונקציה עושה:דואגת לתיקון של העץ על ידי ארבעת סוגי הגלגולים

כיצד הפונקציה פועלת:ראשית הפונקציה דואגת לתיקון של כלל הגבהים והגדלים של העצים החל מה-node המוכנס לפוקנציה ועד לשורש וכן משמרת את הגובה הישן בשדה משלו לשם בדיקה .נתקן את ה-node ממנו מתחילים לפי הקביעה האם אנו בהכנסה או בכל פעולה אחרת, כעת נחל בלולאת while אשר מקדמת את node בכל פעם מעלה ובודקת עבורו ועבור ילדיו האם עברני AVL בעזרת הפונקציה BFF , במידה והם אכן עבריינים, הפונקציה מפעילה את הגלגול המתאים וממשיכה לקדם את node להורה שלו, לפי סוג הגלגול תוסיף הפונקציה לספירת פעולות האיזון את הכמות המתאימה. במידה והילדים אינם עברייני AVL-

אם הגובה הישן שווה לגובה החדש ואני בהכנסה - תפסיק הפונקציה לעדכן את פעולות האיזון ותצא מהלולאה. במידה והגובה הישן שונה מהגובה החדש- הפוקנציה תוסיף פעולת איזון לספירה (כמתבקש)לבסוף הפוקנציה מעדכנת את שורש העץ ומחזירה את מספר פעולות האיזון.

פירוט סיבוכיות זמן ריצה:(**O(log n**באופן כללי הפוקנציה עובדת ב-O(log n) שכן היא מעדכנת את כל הגבהים עד השורש (רצה בלולאה עד השורש) בנוסף גם עצם הלולאה שבודקת את עברייני ה-AVL רצה עד השורש. נשים לב כי O(log n) מדבר על המקרה הגרוע, בו אנו מתחילים את הבדיקה של TreeFixer מעלה בתחתית העץ.

v. **left_rotation(self, node)**

מה הפונקציה עושה:מבצעת גלגול שמאלה

כיצד הפונקציה פועלת:הפונקציה משנה את המצביעים של node ושל בנו הימני כך שיבצעו גלגול שמאלה.

פירוט סיבוכיות זמן ריצה:**O(1)**, כלל הפעולות הן פעולות השמה

vi. **right_rotation(self, node)**

מה הפונקציה עושה:מבצעת גלגול ימינה

כיצד הפונקציה פועלת:הפונקציה משנה את המצביעים של node ושל בנו השמאלי כך שיבצעו גלגול ימינה.

פירוט סיבוכיות זמן ריצה:**O(1)**, כלל הפעולות הן פעולות השמה

vii. **leftright_rotation(self, node)**

מה הפונקציה עושה: גלגול שמאלה ואז ימינה

כיצד הפונקציה פועלת: הפוקנציה מפעילה על node את פונקציית left_rotation ולאחר מכן את פונקציית right_rotation

פירוט סיבוכיות זמן ריצה:**O(1)**, הסיבוכיות נובעת מסיבוכיות שתי הפונקציות מעלה.

viii. **rightleft_rotation(self, node)**

מה הפונקציה עושה: גלגול ימינה ואז שמאלה

כיצד הפונקציה פועלת: הפוקנציה מפעילה על node את פונקציית right_rotation ולאחר מכן את פונקציית left_rotation

פירוט סיבוכיות זמן ריצה:**O(1)**, הסיבוכיות נובעת מסיבוכיות שתי הפונקציות מעלה.

ix. **join_for_split (self,tree,key,val)**

מה הפונקציה עושה:הפונקציה מחברת שני עצים בעזרת node מסויים שיהווה המחבר בניהם.

כיצד הפונקציה פועלת: הפוקנציה פועל כמו join הרגיל מעלה, אך מחזירה את העץ המוכן self.

פירוט סיבוכיות זמן ריצה:**O(log n)** כפי שהוסבר ב-join.

Theoretic part

שאלה 1

מספר סידורי i	מספר חילופים במערך ממוין-הפוך	עלות מיון AVL עבור מערך ממוין-הפוך	מספר חילופים במערך מסודר אקראית	עלות מיון AVL עבור מערך מסודר אקראי	מספר החילופים במערך כמעט ממוין	עלות מיון AVL עבור מערך כמעט ממוין
1	4,498,500	67,805	2,307,672	57,367	448,500	52,552
2	17,997,000	147,635	8,945,888	130,280	897,000	112,628
3	71,994,000	319,297	35,776,208	293,908	1,794,000	232,845
4	287,988,000	686,623	143,569,144	594,217	3,588,000	473,282
5	1,151,976,000	1,469,277	576,140,857	1,358,512	7,176,000	954,159

1. נתחו באופן תיאורטי את מספר החילופים וגם את עלות החיפושים של ה-AVL במקרה של מערך ממוין-הפוך. התשובות צריכות להיות כתלות בגודל המערך n. את מספר החילופים יש לתת באופן מפורש, עלות החיפושים יכולה להיות מתוארת במונחי $\Theta(\cdot)$ (הציגו חסמי $O(\cdot)$ ו- $\Omega(\cdot)$)

במערך ממוין הפוך, נשים לב כי מספר החילופים , הוא כסכום סדרה חשבונית בעלת n-1 איברים, שכן אנו מתחילים את ההכנסה מהמקסימום של העץ.
כל איבר במערך הממוין הפוך, מעלה את מספר החילופים במספר האיברים שלפניו.
לכן לאיבר הראשון והמקסימלי יהיו 0 חילופים, לאיבר השני חילוף אחד, וכך הלאה עד שלאיבר ה-n יהיו n-1 חילופים.
נוסחה לסכום סדרה חשבונית מ-0 עד n-1 = $S = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$
כאשר נציב את n לפי האינדקסים שאותם היינו צריכים לבדוק:
$$n_1 = 3{,}000\ , n_2 = 6{,}000\ , n_3 = 12{,}000\ , n_4 = 24{,}000\ , n_5 = 48{,}000$$
ולכן נצפה לראות מספר חילופים השווה ל –
$$Sn_1 = 4{,}498{,},500$$
$$Sn_2 = 17{,}997{,}000$$
$$Sn_3 = 71{,}994{,}000$$
$$Sn_4 = 287{,}988{,}000$$
$$Sn_5 = 1{,}151{,}976{,}000$$

עלות המיון –
כיוון שבכל הכנסה לעץ, נכניס את האיבר הקטן ביותר, נצטרך בכל פעם לעלות בכל מעלה העץ ואז לרדת בכל מורד העץ.
מספר האיברים במערך - n
גובה העץ - $\log n$
<u>חסם עליון –</u> $\sum_{i=1}^{n} 2\cdot\log i \le 2\cdot\sum_{i=1}^{n}\log n = 2n\cdot\log n = O(n\cdot\log n)$
כאשר $\sum_{i=1}^{n} 2\cdot\log i$ מתאר את העלייה והירידה בהתאם למספר האיברים בעץ.

<u>חסם תחתון –</u> נתבונן בהכנסת $\frac{n}{2}$ האיברים האחרונים. באותו אופן כמו שהסברנו לחסם העליון נסכום את מספר הצעדים במורד ומעלה העץ.
כאשר $\frac{n}{2}$ מאיברי המערך מוכנסים לעץ, גובהו $\log\frac{n}{2}$ ולכן , כל חיפוש מקום בעץ לאיבר יהיה $\log\frac{n}{2}$.
אנו מתבוננים ב $\frac{n}{2}$ האיברים האחרונים במערך לכן לכל הפחות סיבוכיות זמן ההכנסה שלהם תהיה:
$\sum_{i=1}^{n} 2\cdot\log i \ge \sum_{i=\frac{n}{2}}^{n}\log i \ \ge \frac{n}{2}\log\frac{n}{2}$ כלומר $\Omega(n\cdot\log n)$
ולכן סך סיבוכיות הזמן של חיפושים במערך ממוין הפוך - $\theta(n\cdot\log n)$

2. האם הערכים בטבלה מסעיף א' והניתוח מסעיף ב' מתאימים? נמקו.

נשים לב כי הערכים מסעיף א **זהים** למספר החילופים בפועל.

```
/usr/local/bin/python3.10 /Users/galnadler/Desktop/DataStructures_project1/
number of switch when i= 1 is 4498500
number of switch when i= 2 is 17997000
number of switch when i= 3 is 71994000
number of switch when i= 4 is 287988000
number of switch when i= 5 is 1151976000
```

לכן כמובן בהצבת המספרים בגרף היינו מקבלים $R^2 = 1$.

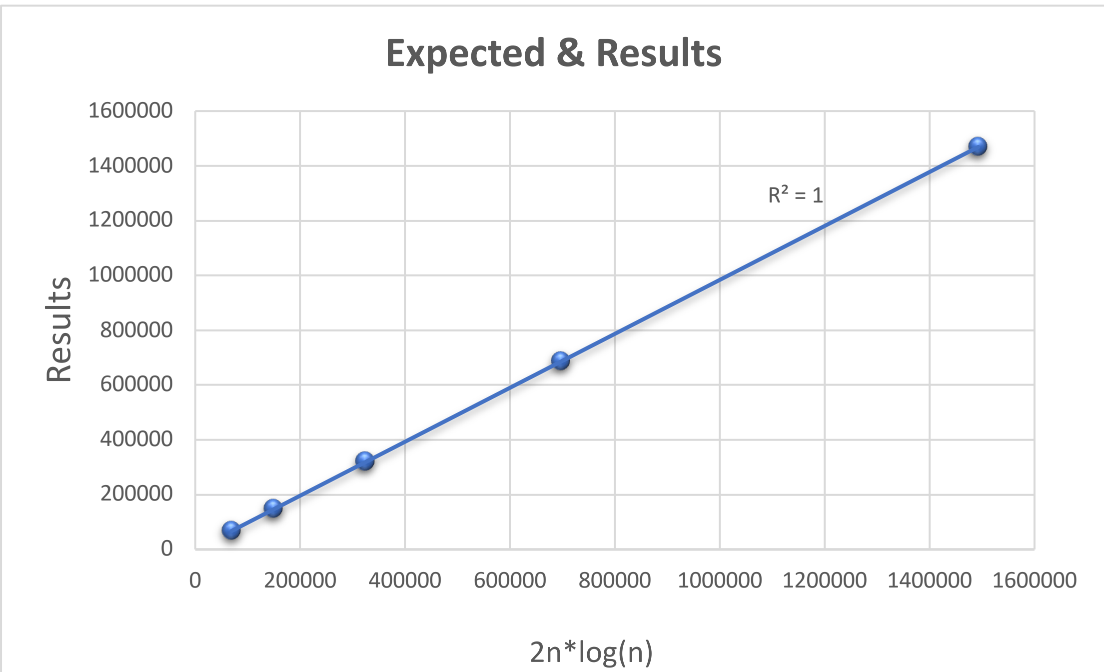
הערכים של עלות המיון:

```
/usr/local/bin/python3.10 /Users/galnadler/Desktop/DataStructures_project1/
cost of sort when i= 1 is 67805
cost of sort when i= 2 is 147635
cost of sort when i= 3 is 319297
cost of sort when i= 4 is 686623
cost of sort when i= 5 is 1469277

Process finished with exit code 0
```

נראה גם מידת התאמה על ידי גרף:
נשים לב שלפני ההסבר בסעיף הקודם,
העלות המשוערת היא $\theta(n \cdot \log n)$
הכפלנו ב2 כיוון שבכל הכנסה גם נעלה
למעלה וגם נרד למטה בעץ.

n	cost of sort	2*n*log(n)
3,000	67,805	69,304.4807
6,000	147,635	150,608.961
12,000	319,297	325,217.923
24,000	686,623	698,435.846
48,000	1,469,277	1,492,871.69



שאלה 2

1. להלן טבלת הערכים הפורטת את העלות הממוצע והיקרה ביותר של פעולת ה-Join:

מספר סידורי i	עלות join ממוצע עבור split אקראי	עלות join מקסימלי עבור split אקראי	עלות join ממוצע עבור split של האיבר המקסימלי בתת העץ השמאלי	עלות join מקסימלי עבור split של האיבר המקסימלי בתת העץ השמאלי
1	2.8	4	2.5454545454545454	13
2	2.5	6	2.4166666666666665	15
3	2.6923076923076925	8	2.8333333333333335	16
4	2.7142857142857144	8	3	18
5	3	6	2.1176470588235294	19
6	2.6875	5	2.6875	20
7	2.75	5	3.0625	21
8	2.411764705882353	7	2.4210526315789473	22
9	2.4375	9	2.6842105263157894	23
10	2.65	7	2.409090909090909	24

2. נתחו באופן תיאורטי את העלות של **join** ממוצע לשני הניסויים (split אקראי או על האיבר המסוים שבחרנו), והסבירו אם התוצאות מתיישבות עם ניתוח הסיבוכיות התאורטי.
מותר להשתמש ללא הוכחה בכך שסיבוכיות פיצול אסימפטוטית היא כעומק הצומת.

ראינו בהרצאה כי הסיבוכיות האסימפטוטית של הפונקציה Split , כאשר בעץ יש n איברים, היא $O(\log n)$.
כאשר אנו עושים Split על איבר אקראי, או על האיבר המקסימלי של תת העץ השמאלי, מספר הקריאות לפונקציה Join היא כעומק הצומת ממנה אנו מפצלים את העץ. נזכר כי גובה העץ הוא בערך $\log n$.
כיוון שבשני המקרים, ההפרש בין גובה השורש לגובה הממוצע של הצומת ממנה נפצל הוא בערך $\log n$, וליתר דיוק $O(\log n)$, נקבל כי הסיבוכיות הממוצעת לכל פעולת Join, היא קבועה – O(1) (עד כדי קבוע c)
נשים לב גם כי ערכי העלות הממוצעת בשני המקרים, כפי שניתן לראות בטבלה, נעים בערך בין 2 ל-3, דבר המתיישב עם ניתוח הסיבוכיות.

3. נתחו באופן תיאורטי את העלות של **join** מקסימלי בניסוי השני של split על האיבר המקסימלי בתת העץ השמאלי של השורש. הסבירו האם התוצאות מתיישבות עם ניתוח הסיבוכיות התאורטי.

גובה העץ הוא בערך $\log n$, ולכן גובה תת העץ השמאלי הוא $\log(n)$ - 1 או $\log(n)$ - 2 .
כלומר, עומק הצומת המקסימלית בתת העץ השמאלי היא לכל היותר $\log(n)$ - 1, כלומר $O(\log n)$.
בכל פעם שנפעיל את Join , כשנעלה במעלה העץ, הפונקציה Join תעלה לנו כהפרש הגבהים של העצים אותה היא מחברת, ועוד 1. כלומר 1 או 2. ההפעלה האחרונה של Join תהיה המקסימלית, כיוון שהיא מחברת עץ ריק (או בן וירטואלי לפי התרגיל המעשי) , לתת העץ הימני של הורש, והשורש עצמו, שגובהו בערך $\log n$.
לכן, נצפה לראות שהמספר המקסימלי הוא בסדר גודל של $\log_2 1500 \cdot 2^i$.
הניתוח התיאורטי מתיישב עם התוצאות. שכן נצפה לראות טווח מספרים בין $\log_2 1500 \cdot 2^1 = 11.5507$, לבין $\log_2 1500 \cdot 2^{10} = 20.5507$. (כמובן שכיוון שהאיברים הוכנסו לעץ בצורה אקראית, מלכתחילה, נראה חריגות ולא נוכל להגיע לדיוק של 100 אחוז, לכן נתייחס לזה כ-$O(\log_2 1500 \cdot 2^i)$
כפי שניתן לראות בטבלה, קיבלנו כי העלות המקסימלית היא בין 13 ל24 וזה סדר הגודל לו ציפינו.