מגישים:

1. Amit Sandler
324172972
Amitsandler
2. David Mensher
212779920
Davidmensher

מדידות:

חלק ראשון:

א. נציג את המדידות:

מספר סידורי	גודל המערך	כמות החילופים במיון רגיל עבור מערך אקראי	כמות החילופים במיון רגיל עבור מערך ממוין הפוך	עלות החיפושים במיון AVL עבור מערך אקראי	עלות החיפושים במיון AVL עבור מערך ממוין הפוך
1	10,000	24652108	49995000	217580	211338
2	20,000	99255081	199990000	487906	462666
3	30,000	224818881	449985000	766756	728090
4	40,000	400810048	799980000	1043582	1005322
5	50,000	622170636	124997500	1298075	1286170
6	60,000	905222236	1799970000	1600401	1576170
7	70,000	1223244734	2449965000	1923098	1870634
8	80,000	1601517244	3199960000	2317180	2170634
9	90,000	2023010644	4049955000	2566974	2470634
10	100,000	2496942473	4999950000	2879500	2772330

ניתוח הנתונים :

נשים לב שכמות החילופים האלגוריתם הרגיל הן עבור מערך אקראי והן עבור מערך ממויין הפוך גדולים משמעותית מכמות החיפושים בשני המקרים. כמו כן, ניתן להבחין בכך שבכל שורה (גודל מערך), כמות החילופים במיון הרגיל עבור מערך ממויין‏הפוך גדולה בכ- פי 2 מכמות החילופים עבור מערך אקראי, בעוד שבאלגוריתם המשתמש בAVL העלות כמעט זהה לכל גודל במערך.

נעבור לנתיוח התיאורטי:

תחילה, נשים לב שאם נסמן לכל קלט אפשרי Xt (שהוא מערך) ב Ht את כמות הרוטציות (החילופים במערך) באלגוריתם הרגיל, מתקיים שסיבוכיות הזמן של האלגוריתם insertion_sort הרגיל עבור הקלט Xt הינו n+Ht.

נעבור לנתח את סיבוכיות האלגוריתם המשתמש ב*AVL*.

בדומה לאלגוריתם הרגיל, נרצה תחילה לחשב תחילה את סיבוכיות הזמן עבור קלט *Xt*. נסמן שוב ב*Ht* את כמות החילופים במערך הקלט *Xt*. כמו כן, נסמן בh_i את כמות החילופים עבור האיבר המוכנס באיטרציה ה-*i*. נבחין בכך שכאשר מבצעים הכנסה מהמקסימום (כפי שהתבקשנו), אנחנו בכל פעם עולים עד לאיבר הכי גבוה שניתן להגיע מהמקסימום שקטן מהאיבר אותו אנחנו רוצים להכניס, ולאחר מכן מבצעים חיפוש מטה. נשים לב שכל התהליך הנ"ל מתבצע אך ורק בתת העץ של העץ המקורי, שמכיל את כל האיברים שגדולים מהאיבר שכרגע אנחנו מכניסים. למעשה, כמות האיברים בתת העץ הזה הוא h_i, שכן זהו מספר *swaps*, שזה מספר האיברים במערך ברגע הכנסת האיבר הנוכחי. כמו כן, נשים לב שאנחנו מבצעים לכל היותר $2 * subtreeHight$ בכל חיפוש שכזה (עולים עד לאיבר הראשון שקטן, ולאחר מכן רק יורדים). במקרה שלנו, גובה תת העץ הוא $\log(h_i)$, ולכן סיבוכיות החיפוש עבור האיבר במקום ה*i* היא לכל היותר $2 * \log(h_i)$. כעת נסמן ב-

$S = \sum_{i=0}^{n-1} \log(h_i)$. זוהי למעשה עלות כל החיפושים באלגוריתם זה. נרצה לתת חסם עליון לביטוי זה .

נשים לב שפונקציית ה *log* הינה פונקצייה קעורה, ונזכר בעובדה הבאה:

לכל פונקצייה קעורה *f*, מתקיים:

f(x1) + ... f(xn))/n <= f((x1+...xn)/n)). ובמקרה שלנו, נקבל בעזרת חוקי לוגריתמים ש-

$$\frac{\sum_{i=0}^{n-1}\log(h_i)}{n} \leq log\left(\frac{\sum_{i=0}^{n-1}\mathrm{h_i}}{n}\right) \to \frac{S}{n} \leq log\left(\frac{H_t}{n}\right) \to \frac{S}{n} \leq \log(H_t) - \log(n)$$

נכפיל ב*n* ונקבל את החסם העליון $S \leq nlog(H_t) - nlog(n)$.

לכן, עבור קלט *Xt* נקבל:

זמן הריצה של האלגוריתם על קלט זה הינו : $O(nlog(H_t) - nlog(n))$.

לכן, מהניתוח שביצענו לעיל, עבור המקרים של מערך אקראי ומערך ממויין יורד, יהיה סביר להניח שכמות החילופים עבור קלט *Xt* תהיה משמעותית יותר גדולה מעלות החיפושים, שכן הפקטור המרכזי בהערכות האסימפטוטיות הללו הינו *Ht* , שמופיע בצורה לינארית בעלות של האלגוריתם הראשון, ובצורה לוגריתמית בעלות הניתוח עבור האלגוריתם השני. באלגוריתם השני הביטוי הלוגריתמי *log(Ht)* מוכפל ב*n*, אך גם מחסרים את *nlogn*.

כמו כן, ניתן ניתן להסיק כי העובדה הטוענת שציינו שבמיון הרגיל, כמות החילופים הוא כ-פי 2 במערך ממויין הפוך מאשר במערך אקראי, נובעת מכך שהגורם *Hi* נמצא כפקטור לינארי בסיבוכיות הרגילה, וגודלו מקסימלי כאשר המערך ממויין הפוך.

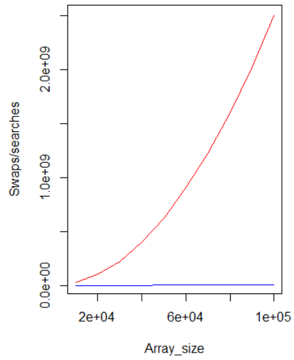
ואכן, ניתן לראות מהדגימות שמספר החילופים גדל בצורה הרבה יותר מאסיבית ומהירה מאשר מספר החיפושים באלגוריתם השני.

להמחשה:

הקו האדום- מספר החילופים באלגוריתם הרגיל כתלות בגודל המערך.

הקו הכחול- מספר החיפושים באלגוריתם המבוסס *AVL* כתלות בגודל המערך.

ציר ה*X* – גודל המערך.



ב. נשים לב שסיבוכיות זמן של *insertion-sort* באמצעות *AVL* הינה סיבוכיות כלל החיפושים שהתבצעו + סיבוכיות הריבלנסים + סיבוכיות טעינת העץ למערך. קודם לכן הראנו שכמות החיפושים היא $O(nlog(H_t) - nlog(n))$. כמו כן, הוכחנו בכיתה שעבור *n* הכנסות, זמן הריצה של כל הריבלנסים הינו *O(n)* (באמורטייזד *O(1)* לפעולה). טעינת המערך מתבצעת ב *O(n)* גם כן, ולכן מחישובי אסימפטוטיקה נקבל שסיבוכיות הזמן של *insertion-sort* היא $O(nlog(H_t) - nlog(n))$.

חלק שני:

נציג את המדידות:

מספר סידורי	עלות *join* ממוצע עבור *split* אקראי	עלות *Join* מקסימלי עבור *split* אקראי	עלות *join* ממוצע עבור *split* של איבר מקס בתת העץ השמאלי	עלות *join* מקסימלי עבור *split* של איבר מקס בתת העץ השמאלי
1.	*2.7*	*5*	*2.5*	*12*
2.	*2.21*	*4*	*2.58*	*13*
3.	*2.8*	*7*	*2.2*	*14*
4.	*2.7*	*3*	*2.64*	*15*
5.	*2.7*	*6*	*2.6*	*15*
6.	*2.3*	*4*	*2.76*	*18*
7.	*2.5*	*5*	*2.8*	*16*
8.	*2.6*	*5*	*2.5*	*16*
9.	*2.3*	*6*	*2.4*	*17*
10.	*2.1*	*7*	*2.18*	*15*

ניתוח הנתונים:

מתוצאות המדידות, ניתן להבחין כי עלות ה*join* הממוצע נע בין 2.1 ל2.8 בשני מקרי ה*split* על כל גדלי המערך האפשריים. למרות זאת, ניתן לראות הבדלים משמעותיים בין עלות *join* המקסימלי עבור *split* אקראי לבין עלות *join* המקסימלי עבור *split* של

האיבר המקסימלי בתת העץ השמאלי- העלות עבור איבר אקראי נעה בין 3 ל-7, בעוד שהעלות עבור האיבר המקסימלי בתת העץ השמאלי נע בין 12 ל18.

נעבור לניתוח האסימפטוטי:
תחילה , נשים לב שכאשר אנו מבצעים *split* על האיבר המקסימלי בתת העץ השמאלי, אנחנו למעשה בכל עלייה עד לשורש (לא כולל)מבצעים ספליט שעלותו היא 1 או 2, שכן בכל עלייה המפתח של הצומת הנוכחי בהכרח קטן מהמפתח של האיבר המקסימלי בתת העץ הימני- ולכן מבצעים *join* של תת העץ הימני של הצומת הנוכחיץ איתו ועם העץ שמחזיק את כלל המפתחות הקטנים מהאיבר שלנו. העלות של פעולת ה *join* בכל מקרה שכזה (שהגדרנו אותה להיות ההפרש בין גבהי הצמתים) היא 1 או 2, שכן מדובר בעצי *AVL* והצמתי הם מסוגים *1,1 or 2,1 or 1,2*.
לעומת זאת, כאשר מגיעים לשורש מבצעים *join* של השורש עם תת העץ הימני שלו והעץ שמכיל את המפתחות שגדולים יותר מהצומת שלנו (שכרגע ריק) , ולכן במקרה זה עלות ה-*join* תהיה כגובה העץ, כלומר *O(log(n))* הבחנה שמסבירה את ההבדל המשמעותי בין העלות המקסימלית בין שני המקרים שציינו קודם לכן.
לעומת זאת, נשים לב בשחירה אקראית של איבר כלשהו מהעץ שקולה לבחירה של מסלול אקראי כלשהו. בחירה למסלול אקראי היא בעצם בחירה של כיוון בכל פעם מהקבוצה {שמאל,ימין}. ההתפלגות כאן היא כמובן אחידה, והתוחלת יוצאת 0.5. כלומר בכל שתי עליות בממוצע אנחנו נחליף כיוון, ולכן העלות הממוצעת היא העלות כאשר מבצעים *Join* בכל שתי עליות, כלומר בערך 3. יחד עם זאת, הנחה סבירה היא שייתכנו מקרים של 3 עליות ברצף לאותו כיוון על פני קשתות שה- *rank differnce* שלהן הוא 2, ולכן קיימים מקרים בהם ה-*join* המקסימלי מקבל ערכים עד 7.

תיעוד:
```
public AVLTree(IAVLNode root)(*)
public AVLTree()
```
שני בנאים למחלקת AVLTree. הראשון מקבל צומת ובונה את העץ ששורשו הוא root ויש בו איבר אחד. השני בונה עץ ריק.
סיבוכיות: O(1).

```
public static void rotate(IAVLNode x, IAVLNode y)(*)
```
מתודה שמבצעת סיבוב על הקשת y→x. x ו-y הינם מסוג AVLNode, ובהכרח מתקיים שx הוא ההורה של y . אם y הוא בן שמאלי של x מתבצע סיבוב לצד ימין, ואם y הוא בן ימני אז מתבצע סיבוב לצד שמאל. הסיבוב מתבצע בזמן O(1).

```
public boolean empty()(*)
```
מתודה שבודקת האם העץ ריק, כלומר האם השורש הוא עלה אמיתי.
סיבוכיות: O(1)

```
public String search(int k)(*)
```
מתודה שמקבלת כקלט מספר k, ובודקת האם קיים בעץ איבר שזהו המפתח שלו. משתמשת בפונקציית העזר שעליה נרחיב מייד.
סיבוכיות: O(log(n)).

```
private static IAVLNode search(int k,IAVLNode node)(*)
```
מתודת עזר לsearch. הפונקציה מקבלץ מספר k וצומת node ועובדת בצורה רקורסיבית. בכל פעם מחפשת האם בתת העץ ש node משריש יש צומת עם מפתח k. קוראת רקורסיבית על הבן

המתאים שמתאים לכיוון המפתח, ועוצרת אם הגענו לצומת כנ"ל ומחזירה את הצומת. אם לא הגענו- מחזירה null.
סיבוכיות: O(log(n)).

public int insert(int k, String i)(*)
הפונקציה מקבלת מפתח k וערך סטרינג i, ומכניסה צומת חדש עם מפתח k וערך i לעץ הנוכחי. תחילה מחפשים את מקום ההכנסה, לאחר מכן מכניסים את הערך והמפתח לצומת ומעדכנים לה בנים שיהיו עלים חיצוניים (לא אמיתיים), ולבסוף נעזרים בפעולת ה- insertRebalance על מנת לאזן את העץ בדיוק כפי שלמדנו בכיתה. לאחר מכן, מבצעים פעולת update לכל הצמתים במסלול מהצומת אליו הגענו עד לשורש (פעולת הupdate מתוארת בהמשך). הפעולה מחזירה את מספר צעדי הריבלנס שבוצעו.
סיבוכיות: O(log(n)).

public static int insertRebalance(IAVLNode y)(*)
פעולה זו היא פעולה רקורסיבית. הפעולה מקבלת כקלט צומת כלשהי, ומבצעת את פעולת ה- InsertRebalnce כפי שלמדנו בכיתה- ישנם שלושה מקרים אפשריים שאינם תקינים case1,case2,case3 , ומטפלים בכל אחד לפי הצורך. כמו כן מתווספת לפעולה זו פעולת rebalance מיוחדת שיכולה להווצר רק בפעולת הjoin, שכן גם פעולת join תשתמש במתודה זו. ההנחה היא שהמקרים הם בדיוק כפי שראינו בכיתה, אך מייד לאחר אתחול הצמתים הרלוונטים מבצעים בדיקה האם זהו המקרה הסימטרי. הפעולה מחזירה את מספר צעדי הריבלנס שבוצעו.
סיבוכיות זמן: O(log(n)).

public int delete(int k)(*)
פעולה זו מקבלת כקלט מספר k ומוחקת את הצומת בעל מפתח זה. אם לא קיים, מחזירה 2- ולא מבצעת כלום. תחליה מבצעים חיפוש של האיבר עם מפתח k , ואם לא נמצא כזה מחזירים 1-.
אחרת , מחלקים לשני מקרים:
מקרה אחד- העץ הוא עץ פנימי. נצטרך לעשות את מה שעשינו בכיתה- נחליף אותו בצומת עם המפתח העוקב לו בעץ (successor) ונמחק את העוקב מהעץ (כמובן שנעדכן את כל המצביעים הנדרשים). לאחר מכן, מבצעים את פעולת הrebalance באמצעות מתודת העזר deleteRebalnce. המקרה השני הוא שהצומת שמצאנו הינו עלה או צומת אונארי, ואז במקרה כזה נסיר את הצומת מהעץ בפשטות כפי שלמדנו, נעדכן את המצביעים הנדרשים, ונבצע את פעולת הdeleteRebalance . הפעולה מחזירה את עלות צעדי הrebalance.
סיבוכיות: O(log(n)).

public static int deleteRebalance(IAVLNode y)(*)
פעולה זו מבצעת את פעולות הrebalance הדרושות בממעלה העץ, כפי שלמדנו בכיתה במקרה של delete. ישנם אבעה מקרים (case1,case2,case3,case4) ואנחנו מטפלים בכל אחד מהם במידת הצורך, בדיוק באותו האופן שטיפלנו בכיתה. כאשר הבעיה נפתרה/הגיעה לשורש, מוחזר מספר פעולות ה rebalace שהתבצעו בסה"כ. אם הצומת אינו צומת תקין, ונדרש תיקון, אנחנו תחילה מניחים שמדובר באותו המקרה שראינו בהרצאה (מבחינת הכיוונים) , ולאחר מכן בודקים מייד האם זהו המקרה הסימטרי, ומשנים את השדות במידת הצורך.
סיבוכיות: O(log(n)).

public static IAVLNode recupdate (IAVLNode x)(*)
פעולה שמטרתה לעדכן (לעשות update) לשאר הצמתים במעלה העץ את שדות max,min,size שלהם, בכל פעם שהעץ משתנה.
סיבוכיות: O(log(n))

public String min()(*)
מחזירה את האיבר המינימלי בעץ , אם ישנו. עושה זאת באמצעות החזרת השדה min של השורש.
סיבוכיות: O(1).

public String max()(*)
מחזירה את האיבר המקסימלי בעץ , אם ישנו. עושה זאת באמצעות החזרת השדה max של השורש.
סיבוכיות: O(1).

(*)public int[] keysToArray()
מכניסה את כל המפתחות בסדר ממויין למערך. משתמשת בפעולת העזר loadToArray שמקבלת מערך int.
סיבוכיות: O(n).

(*)public String[] infoToArray()
מכניסה את כל המפתחות בסדר ממויין לפי המפתחות שלהם למערך. משתמשת בפעולת העזר loadToArray שמקבלת מערך String.
סיבוכיות: O(n).

(*) private static int loadToArray(IAVLNode node,String[] arr, int start)
טוענת באופן רקורסיבי את מפתחות המערך בinorder. הפונקציה רקורסיבית. הפונקציה מקבלת צומת, מערך אינטים, ואינקס בו צריכים להכניס את האיבר הבא, ומבצעת רקורסיבית את הטעינה למערך.
סיבוכיות: O(n)

(*)private static int loadToArray(IAVLNode node,int[] arr, int start)
טוענת באופן רקורסיבי את ערכי המערך בinorder לפי סדר המפחות המתאימים. הפונקציה רקורסיבית. הפונקציה מקבלת צומת, מערך סטרינגים, ואינקס בו צריכים להכניס את האיבר הבא, ומבצעת רקורסיבית את הטעינה למערך.

(*)public int size()
מחזירה את גודל העץ. עושה זאת ע"י החזרת שדה ה-size של השורש.
סיבוכיות: O(1).

(*)public IAVLNode getRoot()
מחזירה את השורש של העץ, ע"י החזרת השדה המתאים.
סיבוכיות : O(1).

(*)public AVLTree[] split(int x)
הפעולה split מקבלת ערך של מפתח בעץ, ועל הפעולה להחזיר מערך בגודל 2 שמכיל במקום הראשון את העץ עם המפתחות הקטנים מx ובמקום השני את העץ עם המפתחות הגדולים מx.
הפעולה ממושת בדיוק כפי שראינו אותה בכיתה- מחפשים את הצומת בעל המפתח k, מאתחלים את תת העץ הימני להיות העץ עם המפתחות הגדולים (למעשה הוא עדיין צומת), את תת העץ הימני שלו להיות העץ עם המפתחות הקטנים, ולאחר מכן "מטיילים" עד לשורש , ובכל עלייה שלנו לצומת אנחנו מוודאים האם עלינו ימינה או שמאלה, ובהתאם לכך מבצעים פעולת join לתת העץ הימני/ שמאלי בהתאמה עם העץ הגדול/קטן בהתאמה, וכך ממשיכים עד מגיעים לשורש.
הפעולה כאמור מחזירה את המערך מגודל 2 שצפורט לעיל.
סיבוכיות: O(log(n)).

(*)public int join(IAVLNode x, AVLTree t)
הפעולה מבצעת join על העץ הנוכחי עם t בעזרת הצומת x. הפעולה תחילה בודקת איזה תת עץ הוא עם המפתחות הקטנים ואיזה עם הגדולים, ולאחר מכן משתמשת בפעולת העזר joinnodes.
הפעולה מחזירה את הפרשי גבהי העצים +1.
סיבוכיות: O(log(n)).

(*)private int joinnodes(IAVLNode x,IAVLNode a, IAVLNode b)
פעולת עזר, שמטרתה לחבר את העצים שהשורשים שלהם הםa,b עם הצומת x. מחלקים לשני מקרים-לפי איזה תת בעל גובה יותר גדול. לאחר מכן יורדים שמאל (ימין , באופן סימטרי) בתת העץ הגבוה יותר, עד שמגיעים לצומת מגובה קטן שווה לגובה העץ הקטן. מחברים את הצומת x כפי שלמדנו בכיתה, ומבצעים update ל-x. לאחר מכן, מפעילים את פעולת insertRebalance כדי לאזן את העץ. מחזירים את הפרשי גבי העצים +1.
סיבוכיות: O(log(n)).

(*)`private static IAVLNode successor(IAVLNode t)`
הפעולה מחזירה את האיבר עם המפתח העוקב למפתח של האיבר שהוכנס כקלט(successor).

(*)במחלקת AVLNode – הוספנו שדות min,max,size. משמעות min/max זהו האיבר המקסימלי.מינימלי בתת העץ שהצומת הנוכחי משריש. כנ"ל לגבי size- גודל תת העץ שהצומת הנוכחי משריש.

(*)`public void update()`
מעדכן את שדות הmin,max,size של הצומת הנוכחי בזמן קבוע, לפי הערכים של השדות הללו בילדי הצומת הנוכחי, אם קיימים. הפעולה תקרא כאשר מבצעים שינויים בעץ ורוצים לעדכן את השדות הללו בכלל הצמתים הרלוונטים, שייתכן והשתנו.
סיבוכיות: O(1)

(*) שאר הפעולות- מבצעים החזרה של השדות הרצויים בזמן קבוע. O(1).

(*)`public boolean isRealNode()`
בודקת האם העלה שעליו מבצעים את הפעולה הוא עלה אמיתי אם כן נחזיר true ואם הוא עלה חיצוני -לא אמיתי, נחזיר False.

חלוקת העבודה:

<u>דוד מנשר:</u>

מימוש פעולות rotate,insert,insert rebalance,delete וכל החלק של המדידות.

<u>עמית סנדלר:</u>

מימוש פעולות: deleterebalance,search,,keystoarray,infotoarray,split,join,successor,update

וכל מתודות העזר בהן פעולות אלו משתמשות, פעולות המחלקה AVLNode.