תיעוד למחלקת $AVLTree$:

<u>תיאור:</u>

המחלקת AVLTree ממשת עץ איזור עצמי שהוא עץ חיפוש בינארי המאזן את עצמו. הוא שומר על התכונה שהגבהים של תת-העץ השמאלי והימני של כל צומת שונים באחד לכל היותר, מה שמבטיח פעולות חיפוש, הוספה ומחיקה יעילים**.**

<u>בנאי:</u>

<u>AVLTree():</u> מאתחל עץ AVL ריק. הוא מגדיר את השורש ל-$None$ ויוצר צומת עם מפתיח שוול להערך השלם המקסימלי כצומת המינימום.

<u>תכונות:</u>

root: מייצג את צומת השורש של עץ ה-AVL.

min: מייצג את הצומת המינימלי בעץ AVL

<u>מתודות:</u>

<u>פונקציית update_min_field(self, node):</u> מעדכן את שדה הצומת המינימלי על ידי השוואת המפתח של הצומת הנתון עם הצומת המינימלי הנוכחי.$O(1)$

<u>פונקציית search(self, k):</u> מחזירה את האיבר בעץ בעל מפתח k במילון.

מתחילים החיפוש מהשורש של העץ. אם המפתח הנוכחי גדול מ-k ממשיכים לחפש בתת העץ השמאלי של השורש, אם המפתח הנוכחי קטן מ-k ממשיכים לחפש בתת העץ הימני של השורש ואחרית נסיק שהמפתח הנוכחי שווה ל-k וחזירים את האיבר הנוכחי.

חיפוש בעץ AVL עולה במקרה גרוע: $O(logn)$ כלומר כעומק העץ.

<u>פונקציית insert(self, key, val):</u>

מייצרים איבר $AVLNode$ עם הערכים key ו val ואז קוראים לפונקציה $insert\_node$ עם האיבר שייצרנו.

<u>פונקציית insert_node(self, node) :</u>

הוספת צומת חדש עם המפתח והערך הנתונים לעץ ה- AVL תוך שמירה על איזון העץ. מחזירה את מספר פעולות האיזון מחדש שבוצעו במהלך איזון עץ AVL. וכמובן מעדכנת השדות בצמתים במידת הצורך

<u>מיתודות עזר:</u> $insert\_bts,\ rotate,\ update\_min$

ראשית הפונקציה $insert()$ מכניסה צומת חדשה עם ערך val ומפתיח key לעץ $self$ כמו באלגורתים של הכנסה לעץ BST, לצורך זה נשתמש הפונקציית $insert\_bts()$. אחרי כך העדכן את המינימום אם צריך.
בעת לצורך איזון העץ נתחיל לרוץ עם משתנה $curr$ מהצומת מחדשה לשורש העץ, ולצורך ספירת מספר פעולות האיזון נאתחל מונה חדש ל-0 :

1. מעדכנים את השדות $height$ ו- $size$ ומחשבים את $BF(curr)$ (גורם האיזון של $curr$)
2. **אם** $|BF(curr)| < 2$ וגם גובה $curr$ נשמר בעקבות ההכנסה **אז** מסיימים ומחזירים את ערך המונה
3. **אם** $|BF(curr)| < 2$ וגם גובה $curr$ השתנה בעקבות ההכנסה **אז** מוסיפים 1 למונה ממשיכים בלולאה עם ההורה של $curr$
4. **אחרת** בהכרח ייתקיים $BF(curr) = 2$ ואז צריך לאזן את העץ ולהוסיף למונה את מספר פעולות האיזון, לצורך זה נשתמש בפונקצייה $rotate$: $counter \leftarrow counter + rotate(self, curr)$ ואז נמשיך עם ההורה של $curr$.

הכל מקרה ממשיכים לעדכן את השדות $height$ ו- $size$ עד לשורש העץ. בסוף ההרצה מחזירים את הערך של המונה.

<u>סובביות זמן:</u> לולאת ה-while רצה $O(\log n)$ איתרציות לכן רצה בזמן של $O(\log n) = \log n \cdot T(rotate)$.
לכן הלולאה מסתיימת ב- $T(inset\_BTS) + O(\log n) = O(\log n)$ זמן.

<u>פונקציית rotate(self, node, node_BF):</u> מבצע סיבוב על הצומת הנתון בהתבסס על גורם האיזון שלו (BF) וגורמי האיזון של הצמתים הצאצאים שלו. מחזירה את מספר הסיבובים שבוצעו.

<u>מיתודות עזר:</u> $right\_rotation, left\_rotation$

ראשית נחשה את גורם האיזון של הצאצאים של $node$: $BF(left),\ BF(right)$

1. **אם** $node\_BF = 2$ **אז**
   a. **אם** $BF(left)$ אי-שלילי (שווה ל- 1 או 0) נבציע גלגון ימינה על הצומת $node$ באמצעות הפונקציה $right\_rotation$ ומחזירים 1.
   b. **אחרת** $BF(left) = -1$ לכן נבציע גלגון שמולה על הבן השמלי באמצעות הפונקציה $left\_rotation$ ואז גלגול ימינה על הצומת $node$ ומחזירים 2.

2. **אחרת:** $node\_BF = -2$ **אז**
   a. **אם** $BF(right)$ אי-חיובי (שווה ל- 1- או 0) נבציע גלגון שמולה על הצומת $node$ באמצעות הפונקציה $left\_rotation$ ומחזירים 1.
   b. **אחרת** $BF(right) = 1$ לכן נבציע גלגון ימינה על הבן הימני באמצעות הפונקציה $right\_rotation$ ואז גלגול שמולה על הצומת $node$ ומחזירים 2.

<u>סובביות זמן:</u> במקרה הגרוע הפונקציה מסתיימת ב- $O(\max\{left\_rotation, right\_rotation\}) = O(1)$ כלומר בזמן קבוע

<u>פונקציית right_rotation(self, node):</u> מבצע סיבוב ימינה בצומת שצוין.

הפונקציה $right\_rotation$ לוקחת פרמטר בודד, שהוא הצומת עליו יתבצע הסיבוב. ראשית, הוא בודק אם הצומת הנתון חוקי ויש לו ילד שמאלי. אם לא, לא ניתן לבצע את הסיבוב, והפונקציה חוזרת. אם לצומת יש ילד שמאלי, הסיבוב מתנהל באופן הבא:
א. הילד השמאלי של הצומת הנתון הופך לשורש החדש של תת-העץ.
ב. הילד הימני של השורש החדש (אם קיים) הופך לילד השמאלי של הצומת המקורי, תוך שמירה על מאפיין עץ החיפוש הבינארי.
ג. הצומת המקורי הופך לילד הימני של השורש החדש.

לאחר עדכון הקישורים, הפונקציה $right\_rotation$ מעדכנת את הגבהים של הצמתים המושפעים.
א. מעדכנים תחילה את גובה וגודל ($size$ & $height$) הצומת המקורי $node$ בהתבסס על הגבהים של

הילדים השמאליים והימניים החדשים שלו.
ב. לאחר מכן, הוא מעדכן את גובה וגודל השורש החדש על ידי השוואת הגבהים של ילדיו הימני ($node$) והשמאלי החדשים.

הסיבוב הושלם כעת, והפונקציה $right\_rotation$ מסתיימת בזמן קבוע $O(1)$.

__פונקציית left_rotation(self, node):__ מבצע סיבוב שמולה בצומת שצוין.

מבצע סיבוב ימינה בצומת שצוין.

הפונקציה $left\_rotation$ לוקחת פרמטר בודד, שהוא הצומת עליו יתבצע הסיבוב. ראשית, הוא בודק אם הצומת הנתון חוקי ויש לו ילד ימני. אם לא, לא ניתן לבצע את הסיבוב, והפונקציה חוזרת. אם לצומת יש ילד ימני, הסיבוב מתנהל באופן הבא:
א. הילד הימני של הצומת הנתון הופך לשורש החדש של תת-העץ.
ב. הילד השמאלי של השורש החדש (אם קיים) הופך לילד הימני של הצומת המקורי, תוך שמירה על מאפיין עץ החיפוש הבינארי.
ג. הצומת המקורי הופך לילד השמאלי של השורש החדש.

לאחר עדכון הקישורים, הפונקציה $left\_rotation$ מעדכנת את הגבהים של הצמתים המושפעים.
א. מעדכנים תחילה את גובה וגודל ($size$ & $height$) הצומת המקורי $node$ בהתבסס על הגבהים של הילדים השמאליים והימניים החדשים שלו.
ב. לאחר מכן, הוא מעדכן את גובה וגודל השורש החדש על ידי השוואת הגבהים של ילדיו השמאלי ($node$) והימני החדשים.

הסיבוב הושלם כעת, והפונקציה $left\_rotation$ מסתיימת בזמן קבוע $O(1)$.

__פונקציית insert_bts(self, node):__ הוספת צומת לעץ AVL באמצעות אלגוריתם ההכנסה של עץ החיפוש הבינארי (BST)

זה מתחיל ביצירת צומת חדש עם המפתח והערך הנתונים. אם העץ ריק, הצומת החדש הופך לשורש העץ, ותהליך ההכנסה הושלם. אם ה-BST אינו ריק, הפונקציה $insert\_bts$ עוקבת אחר תהליך הכנסת ה-BST הרגיל: זה מתחיל מהשורש ומשווה את המפתח של הצומת החדש למפתח של הצומת הנוכחי.

1. **אם** המפתח החדש קטן מהמפתח של הצומת הנוכחי, **אז** עוברים לילד השמאלי.
2. **אם** המפתח החדש גדול מהמפתח של הצומת הנוכחי, **אז** עוברים לילד הימני.
3. **חוזרים** על שלבים 1 ו- 2 עד שהוא מגיע לצומת המתאים לצומת החדש, שהוא צומת וירטואלית. ברגע שנמצא מיקום מתאים לצומת החדש, נחליף את הצומת הווירטואלית בצומת החדש.

הפונקצייה מחזירה את הצומת החדש שהוכנס במידת הצורך, ומאפשרת שינויים או פעולות נוספות באותו הצומת.

__סובביות זמן:__ במקרה הגרוע הפונקציה מסתיימת ב- $O(\log n)$ כלומר כעומק העץ.

__פונקציית delete(self, node):__ מקבלת מצביע לצומת מסויימת בעץ הנוכחי ומוחקת את הצומת מהעץ תוך שמירה על איזון העץ. מחזירה את מספר פעולות האיזון מחדש שבוצעו במהלך איזון עץ AVL. וכמובן מעדכנת השדות בצמתים במידת הצורך

מיתודות עזר: $delete\_bts,\ rotate,\ update\_min,\ successor$

ראשית הפונקציה $delete()$ מוחקת את הצומת $node$ מהעץ $self$ כמו באלגוריתמים של מחיקה מעץ BST, לצורך זה נשתמש הפונקציית $delete\_bts()$ שמחזירה את האב של הצומת שנמחיקה פיזית מהעץ ואת הגווה הישן שלו (לפני פעולת המחיקה). אחרי כך העדכן את המינימום אם צריך לעוקב ($successor$) של $node$.
כעת לצורך איזון העץ נתחיל לרוץ עם משתנה $curr$ מהאב של הצומת שנמחיקה פיזית לשורש העץ, ולצורך ספירת מספר פעולות האיזון נאתחל מונה חדש ל-0 :

1. מעדכנים את השדות $height$ ו- $size$ ומחשבים את $BF(curr)$ (גורם האיזון של $curr$)
2. **אם** $|BF(curr)| < 2$ וגם גובה $curr$ נשמר בעקבות ההכנסה **אז** מסיימים ומחזירים את ערך המונה
3. **אם** $|BF(curr)| < 2$ וגם גובה $curr$ השתנה בעקבות ההכנסה **אז** מוסיפים 1 למונה ממשיכים בלולאה עם ההורה של $curr$
4. **אחרת** בהכרח ייתקיים $BF(curr) = 2$ ואז צריך לאזן את העץ ולהוסיף למונה את מספר פעולות האיזון, לצורך זה נשתמש בפונקצייה $rotate$: $counter \leftarrow counter + rotate(self, curr)$ ואז נמשיך עם ההורה של $curr$.

הכל מקרה ממשיכים לעדכן את השדות $height$ ו- $size$ עד לשורש העץ. בסוף ההרצה מחזירים את הערך של המונה.

סובביות זמן: לולאת ה-while רצה $O(\log n)$ איתרציות לכן רצה בזמן של $\log n \cdot T(rotate) = O(\log n)$.
לכן הלולאה מסתיימת ב- $T(delete\_BTS) + O(\log n) = O(\log n)$ זמן.

פונקציית (delete_bts(self, node: מוחקת את הצומת שצוין מעץ ה-AVL באמצעות אלגוריתם המיחקה של עץ החיפוש הבינארי (BST). מחזירה את צומת האב של הצומת שנמחק פיזית ואת הגובה הישן שלו (לפני מחיקה).

מיתודות עזר: $delete\_case\_1, delete\_case\_2, delete\_case\_3$

אם הצומת $node$ הוא עלה מוחקים אותו באמצעות הפונקיצ $delete\_case\_1$. אם יש לצומת בין יחיד מוחקים אותו באמצעות הפונקיצה $delete\_case\_2$. אם יש שני לעץ שני בנים מוחקים אותו באמצעות הפונקיצה $delete\_case\_3$.

סובביות זמן: $O(\max\{delete\_case\_1, delete\_case\_2, delete\_case\_3\}) = O(1)$

פונקציית (delete_case_1(self, node: מטפל במקרה המחיקה באלגוריתם המיחקה של עץ החיפוש הבינארי (BST) שבו הצומת הוא צומת עלה (אין לו ילדים).

כאן פשוט מוחקים מהעץ. כלומר מנתכים את $note$ מהאב שלו, אם אין לו אב אז הוא השורש של העץ ואז מעדקינים את העץ להיות עץ ריק. מסתיים ב- $O(1)$.

פונקציית (delete_case_2(self, node: מטפל במקרה המחיקה באלגוריתם המיחקה של עץ החיפוש הבינארי (BST) שבו לצומת יש רק ילד אחד.

במקרה זה יצריך גם לעקוב על הצומת $node$: ניגשים לאב אש $node$ וגדירים את הבן שלו (בצד המתאים) להיות הבן היחיד של $node$. ואז מגדירים האב של $node$ ושני הבנים שלו להיות $None$.

מסתיים ב- $O(1)$.

פונקציית (delete_case_3(self, node: מטפל במקרה המחיקה באלגוריתם המיחקה של עץ החיפוש הבינארי (BST) שבו לצומת יש שני ילדים.

מיתודות עזר: $delete\_case\_1, delete\_case\_2, successor$

במקרה זה נחזיק מצביע $succ \leftarrow successor(node)$. נבחין כאן שלוש הבחנות:

1. $succ$ **יש לו צומת אב:** זה נכון כי הינחנו שיש שני בנים ל- $node$ לכן העוקב שלו חייב להתמקם בתת-עץ הימני של $node$.
2. $succ$ **הוא צומת אמיתי (קיים ואינו וורטואלי):** זות מסקנה מיידית מ- 1
3. $succ$ **אין לו בן שמאלי:** כי אם כן אז הוא קיימת צומת x עם מפתיח קטן מ- $succ$ אך גדול מ- $node$. ואז $succ$ לא היה העוקב של $node$ בסתירה.

אחרי ההבחנות אפשר להמשיך באופן הבא: נמחק $succ$ באמצעות $case1$ או $case2$ ואז מחליפים אותו עם $node$.

מסתיים ב- $O(\max\{delete\_case\_1, delete\_cas\_2\}) + O(successor) = O(\log n)$.

פונקציית (successor(self, node: מחזירה מצביע לעוקב בסדר ממויין של הצומת $node$ בעץ.

אם לצומת $node$ יש ילד ימני **אז** העוקב יהיה המפתח המינימלי בתת-עץ הימני. אז הפונקציה מחזירה את הצומת השמאלי ביותר (המינימום) בתת העץ הימני על ידי מעבר שוב ושוב אל הילד השמאלי עד שמגיע לצומת עלה .**אחרת** לצומת אין ילד ימני **אז** העוקב נמצא בצמתים הקדמוניים של $node$. הפונקציה חוצה במעלה העץ, החל מהצומת הנתון, עד שהיא מגיעה לצומת שהוא הילד השמאלי של ההורה שלו (או עד שהוא מגיע לשורש). האב של הצומת הזה יהיה היורש.

סובביות זמן: במקרה הגרוע $node$ הוא עלה והעוקב שלו הוא השורש של העץ, במקרה זה מסיימים ב- $O(\log n)$ זמן.

פונקציית (min_in_tree(self:

פונקציה מקבלת עץ ומחזירה את הצומת המינימלית בעץ.

אם העץ ריקה: נחזיר *None* אחרת קיים מינימום בעץ נחפש עליו.

נבצע לולאת *while* מהשורש של העץ עד שנגיע לצומת וירטואלית וכשהגענו לצומת וירטואלית זאת אומרת שהאב שלה הוא המינימום בעץ. בתוך הלולאה אנו עוברים מהצומת הנוכחית אל הבן השמאלי שלה.

סיבוכיות הזמן: אנו מבצעים מסלול מהשורש עד העלה שהוא בעומק לכל היותר *h* ולכן עולה *O(log(n))* .

פונקציית (avl_to_array(self:

מתודת עזר:

מיתודות עזר: $successor, \min\ \_in\_tree, size$

מאתחלים רשימה ריקה באורך $self.size()$ ומשתנה בשם curr מצביע על הצומת המינימלי בעץ (נקבל אותו ב-O(logn) מקראה לפונקציה min_in_tree).

בבנסה ללולאת for שרצה מ-0 עד n ובכל הרצה- i בלולאה מעדכנים את הרשימה במקום i להיות $(curr.key, curr.val)$ ואחריה מעדכנים $curr$ להיות העוקב שלו.

סיבוכיות הזמן :כיוון שהתחלנו מאיבר מינימלי ו שביצענו n קריאות ל- $successor$ בעץ מגובה $\log n$ מקבלים שזמן הריצה – $O(n)$ לפי הנותוח מהתרגול.

<u>פונקציית size(self) :</u>

מחזירים את ה-$size$ של השורש בעץ. $O(1)$

<u>פונקציית split(self, v) :</u>

<u>מיתודות עזר:</u> $join\_node$

מקבלים צומת בעץ ומחזירים שני עצי AVL, אחד מכיל כל הצמתים בעלי מפתח יותר קטן ממפתח של הצומת הנתון-v והשני בעל על הצמתים היותר גדולים.

בהתחלה מייצרים שני עצי $AVLTree(), min\_tree$ – תת עץ שמאלי של הצומת הנתון ו max_tree -תת עץ ימני של הצומת הנתון ובמידה **ואין** תת עץ ימני/ שמאלי נאתחל עץ ריק משורש בצומת וירטואלי. אתחול העצים עולה $O(1)$ זמן. נאתחל משתנה $curr = v$, ונכנס ללולאת $while$ שממשיכה לרוץ עד ש $curr$ הוא $None$.(עד שנגיע לשורש העץ)

בתוך לולאת $while$:

- אם המפתח של $curr$ יותר קטן ממפתח של הצומת הנתון, זאת אומרת שהצומת ותת עץ שמאלי שלו קטנים מ-v ולכן נוסיף אותם לעץ min_tree :
 נבצע פעולת $join$ לתת עץ השמאלי של $curr$ ולעץ min_tree על ידי הצומת $curr$. ובמידה ואין עץ שמאלי, מייצרים עץ שהשורש שלו צומת וירטואלי ומבצעים $join$ לשני העצים אלה.
- אחרת, אם המפתח של $curr$ יותר גדול ממפתח של הצומת הנתון, זאת אומרת שהצומת ותת עץ ימני שלו גדולים מ-v ולכן נוסיף אותם לעץ max_tree :
 אותה פעולת $join$ עם תת עץ ימני של $curr$ ו max_tree . מקרה סימטרי למקרה שלפני.
- נעדכן $curr$ להיות האב שלו.

בסוף מחזירים רשימה מכילה שני העצים min_tree ו max_tree .

<u>סיבוכיות הזמן</u> היא $O(\log n)$ לפי הניתוח מההרצאה.

<u>פונקציית join(self, tree, key, val):</u>

הפונקציה $join$ לוקחת שני עצים $self$ ו- $tree$ עם ההנחה שכל המפתחות בעץ $self$ קטנים מ- key שבטן יותר מכל המפתחות בעץ $tree$.

<u>מיתודות עזר:</u> $join\_node$

מייצרים איבר $AVLNode$ עם הערכים key ו val ואז קוראים ל $join\_node$ עם הפרמתרים המתאימים

<u>סבביות זמן:</u> $T(join) = T(join\_node) = O(\log n)$

<u>פונקציית join_node(self, tree, node):</u>

מיתודות עזר: $insert$

1. הפונקציה בודקת תחילה אם $self$ או $tree$ ריקים. אם אחד מהם ריק, הוא מחזירה $self$ שמעודכן להיות העץ הלא ריק ביניהם (אם קיים) מוסף לו הצומת $node$ (באמצעות $insert$).
2. אם שני העצים אינם ריקים, אנו משווים את האורכים של שני העצים ונעים בעץ עם האורך הארוך יותר מהשורש לתחתית ומחפשים את הצומת הראשון על השדירה השמלית/ ימנית בעץ בעל גובה שווה לגובה העץ הקצר. צומת זו נקרה לה $curr$
3. נשלה את הילד של $node$ (בביוון הנכון) להיות $curr$ ונתחיל לתקן את העץ מההורה הקודם של $curr$ עד לשורש כמו אחרי הכנסה אן מחיקה (באמצעות $rotate$)

סיבוכיות הזמן: כדו למצוא $curr$ (הראשון) עולה לנו $O(height\ difference)$. וגם העליה שוב למעלה עם תיקונים עולה $O(height\ difference)$.
בסה"כ: $O(height\ difference)$

פונקציית rank(self, node):

מקבלת צומת בעץ ומחזירה את הדרגה שלה בעץ, הדרגה של צומת מוגדרת כמספר הצמתים בעץ שיש להם מפתחות נמוכים או שווים למפתח של הצומת הנתון.

מאתחלים מונה עם הערך $left.size + 1$. נחזיק מצביע $curr$ שמאותחל ל-$node$ ועולים עד שמגיעים לשורש. הכל שלב, אם עולים שמולה מוספים למונה את הערך $curr.left.size + 1$ ואם עולים ימינה לא מוספים כלום. בסוף מחזירים את ערך המונה

סובביות: בכל שלב מבלים זמו קבוע (חישוב $size$ לוקח זמו קבוע) ויש $O(\log n)$ שלבים לכן בסה"כ: $O(\log n)$

פונקציית select(self, k):

מקבלת מספר טבעי $1 \leq k \leq tree.size()$ ומחזירה מצביע לצומת בעץ בעלת הדרגה k.

נחזיק מצביע $curr$ לשורש ומונה שמאתחל ל- $curr.left.size + 1$. אם ערך המונה קטן מ- k אז יורדים ימינה ומחסירים מ- k את ערך המונה. אחרת רק יורדים שמולה ובכל שלב מעדכנים את המונה ל- $curr.left.size + 1$. נמשיך עד שנגיע למצב שהמונה שווה ל- k ואז נחזיר את הצומת $curr$.

סובביות: בכל שלב מבלים זמו קבוע (חישוב $size$ לוקח זמו קבוע) ויש $O(\log n)$ שלבים לכן בסה"כ: $O(\log n)$

פונקציית get_root(self): מחזירה מצביע לשורש העץ $O(1)$.

חלק ב: חלק ניסויי/תיאורטי

שאלה 1

סעיף 1:

מספר סידורי i	מספר חילופים במערך ממוין -הפוך	עלות מיון AVLעבור מערך ממוין -הפוך-	מספר חילופים במערך מסודר אקראית	עלות מיון AVLעבור מערך מסודר אקראי	מספר החילופים במערך כמעט ממוין	עלות מיון AVLעבור מערך כמעט ממוין
1	4498500	63810	2258773	63362	448500	67520
2	17997000	139618	8965303	139216	897000	154598
3	71994000	303234	35717217	304448	1794000	345554
4	28798800	654466	144279567	655372	3588000	754352
5	1151976000	1404930	575794575	1402556	7176000	1622510

סעיף 2:

מספר החילופים במערך:

בכל הכנסה במערך ממוין הפוך באיטרציה ה-j , כל הצמתים שכבר בעץ הם גדולים מצומת שמכניסים וכיוון שבשלב ה-j גודל העץ הוא j-1 ולכן מספר החילופים כעט הוא j-1. כלומר בשלב j אנו סופרים j-1 חילופים.

עבור n הכנסות נקבל שמספר החילופים הוא:

$$\sum_{j=1}^{n}\sum_{k=1}^{j-1} 1 \ = \sum_{j=1}^{n} j-1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

עלות המיון:

כשנרצה להכניס את הצומת ה-i כבר יש בעץ i-1 צמתים] כולם גדולים מ-i .

מתחילים בחיפוש מהאיבר המקסימלי שהוא עלה, בהכרח נעלה עד השורש ואז נמשיך במסלול מהשורש לאיבר המינימלי בעץ שהוא עלה ונוסיף i. כיוון שהוכחנו שכל העלים הם בעומק לפחות $h/2$ כאשר h הוא גובה העץ, נקבל לפחות $\log n/2$ קשתים ממקסימום עד השורש ועוד $logn/2$ מהשורש עד המינימום וסכה $\log n$ קשתות במסלול. ולכל היותר $\log{(n)}$ קשתות עד השורש ו $\log{(n)}$ קשתות מהשורש עד המינימום.

הוכחנו כי בכל הכנסה נצטרך לכל היותר לבצע גלגול, עולה לנו O(1) . ונצטרך לעדכן את הגובה לכל היותר $\log i$ ולפחות 0(יש מקרים לא משתנה הגובה בכל הצמתים).

עבור n הכנסות נקבל שעלות המיון F(n) היא:

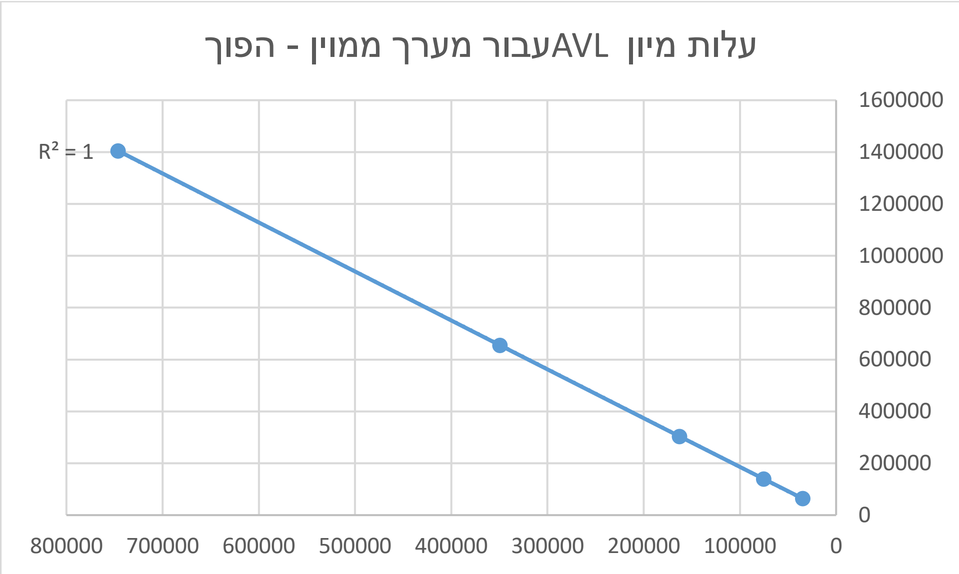
$$\Omega(n*\log(n)) = \log(n!) = \sum_{k=1}^{n}\log k \le \boldsymbol{F(n)} < \sum_{k=1}^{n}\log k + \log k + 1 = 3\sum_{k-1}^{n}\log k + n$$
$$\le 3\log(n!) \le n + 3n\log n = O(n\log n)$$

ולכן $F(n) = \theta(n * \log(n))$.

<u>סעיף 3:</u>

נבחר ציר x להיות $n * log(n)$ וציר y הוא עלות המיון ב AVL עבור מערך ממוין הפוך לפי הטבלה בסעיף קודם.

היחס בין שני הצרים צריך להיות לינארי לפי הסעיף הקודם ואכן זה מה שמקבלים בגרף ומקבלים גם ש $R^2 = 1$ כמו שציפינו.



שאלה 2

<u>סעיף 1:</u>

מספר סידורי i	גודל העץ n	**ניסוי 1:** split אקראי		**ניסוי 2:** split של איבר מקסימלי בתת העץ השמאלי	
		עלות join **ממוצע** עבור split **אקרא**	עלות join **מקסימלי** עבור split **אקראי**	עלות join **ממוצע** עבור split של **האיבר מקסימלי** בתת העץ השמאלי	עלות join **מקסימלי** עבור split של **איבר מקסימלי** בתת העץ השמאלי
1	3K	1.8	5	1.455	11
2	6K	1.3	5	1.75	13
3	12K	1.727	6	1.385	14
4	24K	1.9	3	1.615	15
5	48K	1.692	5	1.643	17
6	96K	1.5625	7	1.786	18
7	192K	1.647	4	1.5625	19
8	384K	1.5	7	1.5	20
9	768K	1.583	4	1.765	21
10	1536K	1.6315	9	1.55	22

<u>סעיף 2 (ניתוך עלות join ממוצע):</u>

הבה נבחן את העלות הממוצעת של הצטרפות בתרחיש פיצול אקראי. ברור שמספר החיבורים שבוצעו מתאים לגובה הצומת שבו מתרחש הפיצול. בסעיף זה, נציג את הגובה כ-*O(d)*, כאשר *d* נבחר באקראי. כפי שצפינו, העלות המפוצלת היא *O(d)*, וכתוצאה מכך עלות *join* ממוצעת של *O(d)/O(d)=O(1)*. רואים שהתשובה לא מושפעת מהצומת הספציפי שבו מתרחש הפיצול. באופן דומה, אם נבחר את הצומת עם האיבר הגבוה ביותר בתת העץ השמאלי, התוצאה תהיה זהה. זה מתיישב עם הנתונים שנאספו מהניסויים, שכן הממוצע נשאר כמעט קבוע לאורך כל הניסויים.

<u>סעיף 3 (ניתוך עלות join ממוצע):</u>

הבה נבחן את עלות *join* המקסימלית לניסוי מספר 2. בתרחיש זה, אנו מתחילים מהצומת המקסימלי בתת העץ השמאלי של השורש ועולים באופן עקבי שמאלה עד שמגיעים לבן השמאלי של השורש. לבסוף, אנחנו עולים פעם אחת ימינה, מגיעים לשורש. כתוצאה מכך, עלות *join* של תת-העצים עם מפתיחות גדולות יותר ממפתיח הצומת נשארת קבועה, בעוד שעלות *join* לתת-העץ הימני של השורש היא לוגריתמית (*log(n)*) בהתאם לניתוח שלנו