<u>מסמך תיעוד פרויקט AVL Tree - 1</u>

יונתן ניצן (212825715), אורטל סימני (208163055)

<u>AVLNode:</u>

המחלקה AVLNode מייצגת צומת בעץ AVL. כל צומת כוללת את השדות מופע הבאים:

- **key** – מפתח ייחודי לצומת.
- **value** – לכל צומת שמור ערך פרטי.
- **left** – בן שמאלי של הצומת.
- **right** – בן ימני של הצומת.
- **parent** – ההורה של הצומת.
- **height** – גובה הצומת.

למחלקה AVLNode יש את הפונקציות הבאות:

- **__init__(self, key, value)** – מאתחל צומת חדשה עם value,key ללא בנים או הורה בגובה -1.
- **is_real_node(self)** – מחזיר True אם הצומת self צומת אמיתית או False אם וירטואלית, הבדיקה לפי המפתח של הצומת (None מייצג צומת וירטואלית).
- **is_leaf_node(self)** – מחזיר true אם הצומת הוא עלה (אין לו ילדים אמיתיים).
- **is_left_child(self) –** מחזיר true אם self הוא בן שמאלי להורה שלו.
- **is_right_child(self)** – מחזיר true אם self הוא בן ימני להורה שלו.
- **get_balance_factor(self)** – מחזיר את ה-balance factor של הצומת (הפרש הגבהים של הבן השמלי והבן הימני).
- **calc_height(self)** – מעדכן את גובה הצומת להיות המקסימום בין הבנים השמאלי והימני, בתוספת 1.
- **insert_child(self, child, right)** – מוסיפה בן לself ומעדכנת את שדה ההורה של הילד להיות self.
- **insert_right(self, child)** – מעדכנת את הבן הימני של הצומת עם הבן החדש, בנוסף מעדכנת את שדה האב של הבן להיות self.
  - קוראת לפונקציה: insert_child(self, child, right)
- **insert_left(self, child)** – מעדכנת את הבן השמאלי של הצומת עם הבן החדש, בנוסף מעדכנת את שדה האב של הבן להיות self.
  - קוראת לפונקציה: insert_child(self, child, right)

- **add_virtual_children(self)** – מעדכנת לכל צומת שאין לה בן ימני או בן שמאלי בן וירטואלי.
  - קוראת לפונקציות: insert_right(self, child), insert_left(self, child)

- **replace_child(self, other)** – מחליף בין הצומת הנוכחית לצומת אחרת ומעדכן את ההורים.

- **create_leaf(self, spot)** – מחליף בין בין צומת וירטואלית ל-self ומעדכן את הגובה של self להיות 0.
  - קוראת לפונקציה: replace_child(self, other)

- **__str__(self)** – מחזיר מחרוזת של המכילה מפתח והערך של הצומת בצורה (key,value).

AVLTree:

המחלקה AVLTree מייצגת עץ AVL שמבנהו מאוזן באופן אוטומטי. כל עץ מכיל את שדות המופע הבאים:

- **root** – שורש העץ.
- **t_size** – גודל העץ.
- **max** – הצומת בעת המפתח המקסימלי בעץ.

למחלקה AVLTree יש את הפונקציות הבאות:

- **__init__(self, node=None)** – אם node=None איתחול עץ AVL ריק בגודל 0, אחרת בונה עץ מצומת קיימת.
  - קוראת לפונקציה tree_from_root(self, node).

- **tree_from_root(self, node)** – בונה עץ מצומת נתונה. הפונקציה אינה מעדכנת את הגודל של העץ או את המקסימום, אלו יקרו במקומות אחרים.
  - קוראת לפונקציות AVLNode.is_real_node(self).

- **search(self, key)** – מבצע חיפוש עבור מפתח בעץ החל מהשורש ומחזיר את הצומת המתאים ומספר הצעדים שבוצעו במהלך + 1, אם הצומת לא קיים בעץ יחזיר None ומספר הצעדים שבוצעו בעץ (לא כולל צמתים וירטואלים).
  - קוראת לפונקציות: search_result_wrapper(self, tup), search_from(self, key, node, dist).
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שקוראת לפונקציות הפועלות בסיבוכיות זמן זו או פחות.

- **finger_search(self, key)** – זהה לפעולת search רק שמתחיל את החיפוש מהמקסימום.
  - קוראת לפונקציות: search_from_max(key), search_result_wrapper(self, tup)
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שקוראת לפונקציות הפועלות בסיבוכיות זמן זו או פחות.

- **search_result_wrapper(self, tup)** – פונקציית מעטפת עבור הפונקציה search_from, מחזירה את אותם ערכים שמחזירה הפונקציה - אלא אם הצומת וירטואלית ואז מחזירה None במקומה.
  - קוראת לפונקציה: AVLNode.is_real_node(self)

- **search_from(self, key, node, dist=1)** – מחפשת צומת בעלת המפתח key, החיפוש נעשה החל מהצומת הנתונה node, בנוסף בעזרת הפרמטר dist ניתן להתחיל את החיפוש עם מספר צעדים גדול מאפס כלומר להמשיך ספירת צעדים שנעשו כבר. ההחזרה זהה לזו של search רק שמחזירה פוינטר לצומת גם אם הינה וירטואלית.
  - קוראת לפונקציה AVLNode.is_real_node(self).
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שבעץ בינארי משך החיפוש חסום ע"י גובה העץ, וגובה עץ AVL חסום ע"י O(log(n)).

- **search_from_max(self, key)** – זהה לפעולת search_from רק שמתחיל את החיפוש מהמקסימום.
  - קוראת לפונקציה: search_from(self, key, node, dist), max_node(self).
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שקוראת לפונקציות הפועלות בסיבוכיות זמן זו או פחות.

- **insert(self, key, val)** – יוצרת צומת חדשה עם key ו-val. אם העץ ריק, מאתחלת את העץ עם הצומת החדשה, אחרת מוצאת את הצומת הוירטואלית במקום בו צריך להכניס את הצומת החדשה ומכניס אותה שם. הפונקציה מחזירה שלשה המכילה את הצומת החדשה, המרחק מהשורש לפני איזון ומספר מקרי הPROMOTE במהלך איזון העץ, כלומר מספר הפעמים בהם עדכנו את הגובה ולא בוצע גלגול.
  - קוראת לפונקציות:
    search_from(self, key, node, dist), insert_at(self, spot, node), AVLNode.add_virtual_children(self).
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שקוראת לפונקציות הפועלות בסיבוכיות זמן זו או פחות.

- **finger_insert(self, key, val)** – זהה לפעולת insert מלבד זאת שמתחילים את הבדיקה החל מהצומת המקסימלית בעץ.
  - קוראת לפונקציות:
    search_from_max(self, key), insert_at(self, spot, node), AVLNode.add_virtual_children(self).
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שקוראת לפונקציות הפועלות בסיבוכיות זמן זו או פחות.

- **insert_at(self, spot, node)** – מכניסה צומת במיקום מסוים נתון בעץ. הפונקציה מקבלת את הצומת הוירטואלית אותה מחליפים בצומת החדשה ואת הצומת החדש להוספה, מאזנת את העץ אם נדרש, מתחזקת את הצומת המקסימלי ומחזירה את מספר הPROMOTES שבוצעו (לפי ההגדרה לעיל).
  - קוראת לפונקציות:
    rebalance(self, node), AVLNode.is_leaf_node(self), AVLNode.create_leaf(self, spot) , max_node(self).
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שקוראת לפונקציה הפועלת בסיבוכיות זו.

- **delete(self, node)** – הפונקציה בודקת מספר מקרים על הצומת ומבצעת מחיקה בהתאם:
  1. אם הצומת שיש למחוק הוא הצומת היחיד בעץ, אז מעדכנת את השורש והצומת המקסימלי להיות None.
  2. אם הצומת היא עלה פשוט מוחקת אותו (עבור ההורה מחליפים אותו בצומת וירטואלית).
  3. אם לצומת בן יחיד, מעדכנת את הבן להחליף את הצומת (עקיפה). אם מוחקת את השורש, מעדכנת את המצביע למי שהחליף אותו.
  4. אם לצומת 2 בנים מוצאת את ה-successor שלו, מחליפה בין המפתחות והערכים של הצומת וה-successor שמצאה ומוחקת את ה-successor. יהיה לו לכל היותר בן אחד (ימני) ולכן קוראת בריקורסיה לפונקציה שמבצעת מחיקה לפי אחד המקרים הקודמים.

  בכל מקרה הפונקציה בודקת אם הצומת הנמחקת היא המקסימלית בעץ, אם כן מוחקת את המצביע למקסימום (ראו מימוש עצל של מצביע זה בmax_node). לבסוף הפונקציה מבצעת איזונים בעץ מהאב של הצומת הנמחקת עד לשורש.
  - קוראת לפונקציות:
    AVLNode.is_real_node(self), AVLNode.add_virtual_parents(self), AVLNode.is_leaf_node(self), AVLNode.is_right_child(self), AVLNode.is_left_child(self), AVLNode.insert_child(self, child, right), delete(self, node), rebalance(self, node) , max_node(self).
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שמתבצעת קריאה לפונקציית איזון הפועלת בסיבוכיות O(log(n)). הקריאה הרקורסיבית יכולה לקרות לכל היותר פעם אחת ולכן לא משפיעה על הסיבוכיות.

- **join(self, tree2, key, val)** – ראשית הפונקציה בודקת אם אחד העצים ריק, במידה וכן מוסיפה צומת חדשה עם המפתח והערך שנתונים לעץ השני (כלומר שאינו ריק). אם העץ הריק היה self, בונה את self מהשורש של הtree2 (לאחר הוספת האיבר החדש), מעדכנת את הגודל והמצביע למקסימום ומסיימת. אחרת, הפונקציה מעדכנת את גודל העץ self להיות חיבור של גודלו עם גודל העץ אותו אנחנו מחברים אליו בתוספת 1 עבור הצומת שנוסיף, עורכת השוואה בין גובה העץ הראשון לעץ השני ומחפשת נקודה בעץ הגבוה, לאורך הענף השמאלי/הימני ביותר תלוי באיזה מן העצים המפתחות הגדולים, שם נוכל לבצע את האיחוד. יוצרת צומת חדשה עם הערכים שנשלחו אליה (key, val) מבצעת מיזוג כך שצומת זו תהיה צומת המחברת בין שני העצים, ובמידת הצורך מבצעת איזונים בעץ החל מההורה של הצומת החדשה שהוכנסה ועד לשורש. לבסוף מעדכנת את שורש העץ המאוחד ומוחקת את המצביע למקסימום (ראו מימוש עצל של מצביע זה בmax_node).
  - קוראת לפונקציות:
    insert(self, key, val), rebalance(self, node), tree_from_root(self, node), max_node(self), AVLNode.insert_right(self,child), AVLNode.insert_left(self,child).
  - סיבוכיות זמן ריצה: O(log(n)) החיפוש אחר נקודת החיבור על העץ הגבוה חסום ע"י גובה העץ, פעולת החיבור עצמה הינה בזמן קבוע, ואז האיזון מתחיל מנקודת החיבור והולך מעלה ולכן חסום גם הוא ע"י הגובה.

- **split(self, node)** – הפונקציה מקבלת מצביע לצומת node בעץ, מפצלת את העץ לשניים (left_tree, right_tree) כך שleft_tree מכיל את המפתחות הקטנים מnode וright_tree מכיל את המפתחות הגדולים מnode. מאתחילים את עצים עלו מהתתי-העץ השמאלי והימני של הצומת המבוקשת בהתאם. כעת הפונקציה מחלחלת מעלה ומחברת חלקים של העץ לleft_tree או right_tree (בעזרת פעולת join):
  1. אם עולה שמאלה, הרי שההורה ותת העץ השמאלי שלו קטנים מהצומת המקורי ולכן נכנסים לעץ השמאלי.
  2. אם עולה ימינה, הרי שההורה ותת העץ הימני שלו גדולים מהצומת המקורי ולכן נכנסים לעץ הימני.

  הפונקציה מחזירה את שני העצים החדשים left_tree, right_tree.
  יש לציין כי לעצים שיוחזרו לא יהיה גודל נכון, וכי לא יהיה להם מצביע למקסימום, עד שיקרא get_max לראשונה (ראו מימוש עצל של מצביע זה בmax_node).
  - קוראת לפונקציות: AVLNode.is_right_child(self), join(self, tree2, key, val).
  - סיבוכיות זמן ריצה: O(log(n)) כיוון שהפונקציה עולה למעלה ובכל שלב עושה פעולת join אחת שלא צורכת איזון כיוון שהעצים אותם מחברים תמיד באותו עומק (עד כדי balance factor), ולכן החיבור לוקח זמן קבוע.

- **rebalance(self, node)** – האיזון מתבצע החל מצומת נתונה ועולה למעלה עד השורש. הפונקציה מחשבת את ה-balance factor , אם הוא גדול מ1 היא בודקת אם צריך לגלגל ימינה או שמאלה ואז ימינה על מנת לאזן את העץ בעזרת השוואת גבהים של הילדים של הבן השמאלי, אחרת אם הוא קטן מ1- בודקת אם צריך לגלגל שמאלה או ימינה ואז שמאלה על מנת לאזן את העץ בעזרת השוואת גבהים של הילדים של הבן הימני. אם רק מתבצע עדכון גובה – זוהי פעולת PROMOTE. מחזירה את המספר הPROMOTES.
  - קוראת לפונקציות:
    rl_rotate(self, node), lr_rotate(self, node), r_rotate(self, node), l_rotate(self, node),
    AVLNode.get_balance_factor(self), AVLNode.calc_height(self).
  - סיבוכיות זמן ריצה: O(log(n)) פעולת האיזון מבצעת גלגול/העלאה, שהן פעולות בזמן קבוע, לכל היותר פעם אחת בכל גובה, ולכן חסומה ע"י גובה העץ.

- **rotate(self,node, right)** – מבצעת גלגול לצומת node ולבן הימני או השמאלי שלו (בהתאם ל-right) לבסוף מעדכנת את הגבהים של הצומת והבן. מעדכנת את המצביע לשורש אם השתנה.
  - קוראת לפונקציות:
    AVLNode.calc_height(self), AVLNode.insert_child(self, child, right), AVLNode.replace_child(self, node).

- **r_rotate(self, node)** – גלגול ימינה של הצומת והבן השמאלי שלו.
  - קוראת לפונקציה: rotate(self,node,right).

- **l_rotate(self, node)** – גלגול שמאלה של הצומת והבן הימני שלו.
  - קוראת לפונקציה: rotate(self,node,right).

- **lr_rotate(self, node)** – גלגול שמאלה של הבן השמאלי של הצומת והבן הימני שלו (של הבן השמאלי של הצומת) ואז גלגול ימינה של הצומת והבן השמאלי שלו.
  - קוראת לפונקציות: r_rotate(self, node), l_rotate(self, node).

- **rl_rotate(self, node)** – גלגול ימינה של הבן הימני של הצומת והבן השמאלי שלו (של הבן הימני של הצומת) ואז גלגול שמאלה של הצומת והבן הימני שלו.
  - קוראת לפונקציות: r_rotate(self, node), l_rotate(self, node).

- **in_order(self, node)** – הפונקציה מבצעת סריקה בסדר inorder על העץ, סורקת את העץ בצורה רקורסיבית כך שהצמתים יהיו בסדר הנכון ומחזירה מערך של הצמתים כך שכל איבר מורכב מהצמד (key, value).
  - קוראת לפונקציה in_order(self, node).
  - סיבוכיות זמן ריצה: O(n) כיוון שקוראת לפונקציה בסיבוכיות זו.

- **avl_to_array(self)** – המרת העץ למערך של זוגות (מפתח,ערך) מסודרים ע"פ מפתח.
  - קוראת לפונקציה: in_order(self,node).
  - סיבוכיות זמן ריצה: O(n) כיוון שהפונקציה עוברת על כל צומת בעץ פעם אחת ומבצעת עליה פעולות בזמן קבוע.

- **max_node(self)** – מחזירה את הצומת בעל המפתח הגדול ביותר בעץ בעזרת מצביע, אם קיים. אם לא, (והעץ אינו ריק) מבצעת חיפוש למקסימום – האיבר האחרון על הענף הימני.

- **size(self)** – מחזירה את גודל העץ.

- **get_root(self)** – מחזירה את השורש של העץ.

חלק ניסויי / תיאורטי

.1

מספר סידורי i	עלות איזון במערך ממויין	עלות איזון במערך ממויין-הפוך	עלות איזון במערך מסודר אקראית	עלות איזון במערך עם היפוכים סמוכים אקראיים	$2n$
1	430	430	386	425.1	444
2	873	873	774.95	859.7	888
3	1760	1760	1572.05	1741.2	1776
4	3535	3535	3160.45	3490.95	3552
5	7086	7086	6306.9	7008.55	7104
6	14189	14189	12633.55	14025.95	14208
7	28396	28396	25349.1	28081.05	28416
8	56811	56811	50688.1	56155.3	56832
9	113642	113642	101473.55	112354.85	113664
10	227305	227305	202920.35	224731.55	227328

עץ AVL הוא עץ חיפוש בינארי מאוזן שבו ההבדל בגובה בין הילד השמאלי והילד הימני של כל צומת לא יכול להיות גדול מ-1.
כל הכנסת איבר לעץ כזה עשויה לגרום לגלגולים על מנת לשמור על איזון העץ ותכונותיו.
הגלגולים מבוצעים על מנת לשמור על גובה העץ בסדר גודל של $O(\log n)$ כך שחיפוש והכנסה לעץ כזה מבוצעים בזמן $O(\log n)$ לכל פעולה.

<u>חסם עליון על עלות האיזון כולל גלגולים:</u>
כפי שראינו בכיתה ניתן להראות כי מספר פעולות האיזון בכל הכנסה הינן $amort\ O(1)$ משמע עבור n הכנסות מתקבל חסם של $O(n)$.

<u>האם הערכים בטבלה מתאימים לחסם העליון:</u>
נבחין בטבלה כי עבור $c = 2$ מתקבל חסם עליון הדוק מאוד.

<u>הסבר לכך שתוספת הגלגולים לספירה אינה משנה אסימפטוטית:</u>
תוספת הגלגולים אינה משנה את החסם האסימפטוטי של האלגוריתם מכיוון שלכל הכנסה יכולים להיות לכל היותר שני גלגולים כפי שראינו בכיתה, ולכן העלות לא משתנה בצורתה האסימפטוטית.

.2

מספר סידורי i	מספר היפוכים במערך ממוין	מספר היפוכים במערך ממוין-הפוך	מספר היפוכים במערך מסודר אקראית	מספר היפוכים במערך עם היפוכים סמוכים אקראיים
1	0	24531	12407.2	109.9
2	0	98346	48664.25	222.25
3	0	393828	198366.3	444.15
4	0	1576200	787617.75	891.2
5	0	6306576	3155100.55	1763.55

.3

מספר סידורי i	עלות חיפוש במערך ממוין	עלות חיפוש במערך ממוין-הפוך	עלות חיפוש במערך מסודר אקראית	עלות חיפוש במערך עם היפוכים סמוכים אקראיים
1	221	2694	2409.45	400.5
2	443	6272	5750.4	800
3	887	14316	13255.55	1610.9
4	1775	32180	29997.95	3232.4
5	3551	71460	67182.2	6442.35
6	7103	157124	147212.85	12891.3
7	14207	342660	342609	25742.4
8	28415	742148	707345.8	51492.55
9	56831	1597956	1535428.95	103044.15
10	113663	3423236	3311980.05	206052.9

4.

I. d_i מייצג את מספר האיברים לפני האיבר ה-i שיש להם ערך גבוה יותר ממנו.
מספר ההיפוכים עם האיבר הj, לפי ההגדרה בשאלה 2, הינו כל האיברים במיקומים i<j אשר גדולים ממנו, שזו בדיוק ההגדרה של d_j. מכאן, מספר ההיפוכים הכולל הוא סכום של d_i על כל האיברים כרצוי.

II. נראה כי הכנסת האיבר הi, שנסמנו x, בפעולת finger_insert (משמע מתחילים מהמקסימום) בהינתן שישנם d_i איברים בעץ שגדולים מx,
ניתן לחסום את עלות החיפוש ע"י $O(\max(1, \log d_i)) = O(\log(d_i+2))$. נוכיח:
את הכנסת האיבר יש לבצע כpredecessor של האיבר הקטן ביותר מבין הd_i הגדולים. על מנת למצוא את האיבר הזה מהמקסימום עלינו לעלות עד שנגיע לשורשו של תת-עץ בעל לפחות d_i איברים, שגובהו הינו $O(\log d_i)$ – משמע עלינו $O(\log d_i)$ צעדים. כעת עלינו לבצע הכנסה רגילה בתת-עץ זה שהיא גם, וודאי, תהיה $O(\log d_i)$. משמע בסה"כ עלות החיפוש הינה $O(\log d_i)$ למעט המצב ש$d_i = 0$ שבו עלות החיפוש הינה 1.
מכאן שניתן לחסום את סך עלות החיפוש ע"י $\sum_{i=1}^{n} O(\log(d_i+2)) = O(\log \prod_{i=1}^{n}(d_i+2))$.

III. האיבר הדומיננטי במכפלה $\prod_{i=1}^{n}(d_i+2)$ הינו $\prod_{i=1}^{n} d_i$ (בהנחה שאף d_i אינו אפס) שאותו ניתן לרשום כממוצע הנדסי בחזקת n -

$$\prod_{i=1}^{n} d_i = \sqrt[n]{\prod_{i=1}^{n} d_i}^{\,n} \le \left(\frac{\sum_{i=1}^{n} d_i}{n}\right)^n = \left(\frac{I}{n}\right)^n$$

נציב חזרה בביטוי ונקבל שהחסם העליון לעלות החיפוש הכוללת עבור $I \neq 0$ הינו:

$$O\left(\log\left(\frac{I}{n}\right)^n\right) = O\left(n \log\frac{I}{n}\right)$$

עבור חלק מהd_i אפס, ניתן פשוט להתייחס לאיבר בלוג של המכפלה רק של האיברים שאינם אפס במקום של כל האיברים ולהגיע לתוצאה זהה, כיוון שלI ההסרה של איברים שהינם אפס לא משנה דבר.
ועבור $I = 0$ (כולם אפס) נקבל הכנסה של רשימה ממויינת בה כל איבר הוא המקסימום החדש ולכן עלות החיפוש שלו היא 1 – משמע $O(n)$. בסה"כ:

$$O\left(n \log\left(\frac{I}{n}+2\right)\right)$$

וודאי שזהו הערך הדומיננטי אסימפטוטית מבין עלויות האיזון והחיפוש שכן $\text{n} = O\left(n\log\left(\frac{I}{n}+2\right)\right)$.

IV. עבור הרשימה המומיינת קיבלנו זמן לינארי כצפוי (פחות אחד כיוון שההכנסה של השורש עולה אפס).
עבור הרשימה ההפוכה, עבורה $I = \binom{n}{2} = \frac{n(n-1)}{2}$ מכאן $n \log\frac{I}{n} = n(\log(n-1) - 1)$. עבור $c = 2$ כבר נקבל חסם די הדוק למקרה הגרוע (רשימה ממויינת הפוך כפי שניתן לראות בטבלה הבאה:

מספר סידורי i	עלות חיפוש במערך ממוין-הפוך	$2n(log(n-1)-1)$
1	2694	3013
2	6272	6918
3	14316	15615
4	32180	34786
5	71460	76680

עבור הערכים הרנדומים קיבלנו משהו בין הרשימה הממויינת לממויינת הפוך, כצפוי.