המחלקה AVLNode

**\_\_init\_\_(self)** – מאתחל את האובייקט עם השדות key, value, left, right, parent, height. במידה והעלה הוא וירטואלי המפתח והערך הם None. זמן הריצה O(1).

**Is\_real\_node(self)** – פונקציה בוליאנית הבודקת לפי השדה key אם צומת נתון אמיתי (בעל מפתח שהוא מספר, ואז תחזיר true) או וירטואלי (המפתח הוא None ואז תחזיר false).  
 זמן הריצה - O(1).

**balance\_factor(self)** – פונקציית עזר שנקראת ע"י הפונקציות balance\_tree, balance\_after\_insert, finger\_insert, join. בודקת את ה-balance factor (=הפרש גבהים של הבן השמאלי והימני) של צומת נתון ע"י גישה לשדות height של הבנים.  
 זמן הריצה - O(1).

**update\_height(self)** – מעדכנת את שדה height שמחזיק את גובה הצומת, בהתבסס על גישה לשדות height של הבנים שלו (מחזירה את הגבוה מביניהם+1). אם הצומת וירטואלי הגובה שלו מוגדר -1.  
 זמן הריצה - O(1).

המחלקה AVLTree

**\_\_init\_\_(self)** – מאתחלת את האובייקט, עם השדות root, size, max. זמן הריצה O(1).

**set\_root(self, node)** –מגדירה את node כשורש העץ. זמן הריצה O(1).

**set\_size(self, size\_to\_add)** – הפונקציה מעדכנת את שדה ה- size של העץ לפי size\_to\_add שקיבלה. זמן הריצה O(1).

**set\_max(self, node) –** מגדירה את node להיות הצומת המקסימלי. זמן הריצה O(1).

**search\_from\_node(self, node, key, e, is\_insert)** – פונקציית עזר שנקראת ע"י הפונקציות search ו finger\_search. הפונקציה מחפשת צומת בעל מפתח מסוים כאשר החיפוש מתחיל בצומת הקלט node באמצעות חיפוש סטנדרטי בעץ AVL. מחזירה זוג סדור (x,e), כאשר x הוא מצביע לצומת המתאים\*\* (לפי התנאים שמפורטים למטה), ו- e הוא אורכו בקשתות של מסלול החיפוש + 1.  
 \*\*אם הקריאה לפונקציה נועדה לחפש צומת בעץ (נקבע ע"י is\_insret = false) מחזירה מצביע לצומת שחיפשנו אם נמצא בעץ, ואחרת None. אם נועדה למצוא מקום אליו נכניס צומת חדש בעל המפתח k (is\_insert = true) המפתח בוודאות לא נמצא בעץ, והפונקציה מחזירה מצביע לצומת שישמש הורה לצומת שיתווסף.  
 זמן הריצה – O(log(n)). הפונקציה נעזרת במבנה עץ החיפוש בעת חיפוש הצומת ולכן מבקרת לכל היותר ב- O(logn) צמתים, בהם מבצעת פעולות בעלות סיבוכיות זמן קבועה (השוואות וכו').

**search(self, key)** - הפונקציה מחפשת איבר בעל מפתח k בעץ AVL החל מהשורש ומחזירה זוג סדור (x,e), כאשר x הוא מצביע לצומת המתאים (או None  אם לא קיים), ו- e הוא אורכו בקשתות של מסלול החיפוש + 1. קוראת לפונקציית העזר search\_from\_node כשמסלול החיפוש מתחיל בשורש, משמע node = root , key = k,  is\_insret = false (כלומר נחפש את הצומת עצמו בעץ). זמן הריצה – O(log(n)). לפי ניתוח הסיבוכיות של פונקציית העזר search\_from\_node.

**finger\_search(self, key, is\_insert=False)** – הפונקציה מחפשת איבר בעל מפתח k החל מהצומת המקסימלי. היא מחזירה זוג סדור (x,e), כאשר x הוא  מצביע לצומת המתאים (או None אם לא קיים), ו- e הוא אורכו בקשתות של מסלול החיפוש + 1. נתחיל את החיפוש בצומת המקסימלי, ונעלה לאורך העץ (עבור כל רמה שנעלה נגדיל את e ב-1) עד שנגיע לראשונה לצומת שהמפתח שלו קטן/שווה למפתח k

שקיבלנו. בשלב הזה הפונקציה קוראת לפונקציית העזר search\_from\_node עם node הצומת הנוכחי (משמע ממנו מתחיל החיפוש), key=k ו-is\_insret = false. זמן הריצה – O(log(n)). מבצעת חיפוש במעלה העץ אחר הצומת הראשון שהמפתח שלו קטן/שווה למפתח k שקיבלנו, המתבצע בO(log(n)), ולאחר מכן הפעולה של פונקציית העזר search\_from\_node מתבצעת גם היא ב- O(log(n)) (פירוט בפונקציה עצמה) ולכן סה"כ הסיבוכיות נותרת O(log(n)).

**insert(self, key, val)** – הפונקציה מייצרת צומת חדש עם val ו-key ומכניסה אותו לעץ, ומחזירה שלשה (x,e,h) כאשר x מצביע לצומת שנוצר, e מספר הקשתות על מסלול ההכנסה ו-h מספר מקרי שינוי גובה (promote) שנדרשו במהלך האיזון. אם העץ ריק הפונקציה מגדירה את הצומת החדש כשורש. אחרת, מוצאת את הצומת שישמש ההורה ע"י קריאה לפונקציית העזר search\_from\_node כאשר החיפוש מתחיל בשורש ו- is\_insret = true (משמע תחזיר מצביע להורה ואת אורך המסלול עד אליו מהשורש). בשלב זה מכניסה את הצומת החדש כבן הימני או השמאלי בהתאם ליחס הסדר בין המפתחות. לאחר מכן קוראת לפונקציית העזר balance\_after\_insert שמחזירה את כמות מקרי שינוי הגובה ומעדכנת את גבהי הצמתים. לבסוף מעדכנת את גודל העץ ע"י קריאה לפונקציה set\_size.  
 סיבוכיות –O(logn) . הקריאה לפונקציה search\_from\_node מתבצעת ב- O(logn)(פירוט בפונקציה עצמה), הכנסה לעץ ועדכון שדה הsize מתבצעות בזמן קבוע, והפונקציה balance\_after\_insert מתבצעת גם היא ב- O(logn)

**balance\_after\_insert(self, new\_node, node, h=0)** – הפונקציה מאזנת את העץ החל מהצומת החדש ע"י תיקון לוקלי בכל רמה עד לאיזון העץ (פירוט למטה) ולבסוף מחזירה זוג סדור (new\_node, h) כאשר new\_node מצביע לצומת החדש שנכנס לעץ וh סופר את כמות מקרי הpromote שבוצעו עד לאיזון העץ (אם הגובה של הצומת השתנה התבצע promote ולכן h גדל ב-1).  
 הפונקציה מאזנת את העץ באמצעות לולאה שמתחילה בצומת החדש ועולה במעלה העץ כמידת הצורך עד שמגיעה לשורש או עד מקרה של איזון העץ. בכל איטרציה על צומת נתון הטיפול בו נקבע לפי ה-balance factor והאם הגובה שלו השתנה במהלך איזון תת העץ של הבן שלו–

1. אם הגובה של הצומת לא השתנה בעקבות השינויים בעץ והbalance factor שלו חוקי (קטן מ2 בערך מוחלט) נחזיר את h ומצביע לצומת החדש.
2. אם הגובה של הצומת השתנה בעקבות השינויים בעץ והbalance factor שלו חוקי (קטן מ2 בערך מוחלט) אנחנו במקרה של promote ולכן נחזור על הלולאה עם ההורה של הצומת.
3. אחרת, נבצע סיבוב/גלגול ע"י קריאה לפונקציית העזר balance\_tree.

זמן הריצה – O(logn). הפונקציה עוברת על מסלול אחד מעלה לשורש לכל היותר ובכל צומת תבצע פעולות בעלות זמן ריצה קבוע (Promote). קריאה לbalance\_tree מתבצעת פעם אחת לכל היותר והפעולות בbalance\_tree מתרחשות גם הן בזמן קבוע ולכן בסך הכל נקבל זמן ריצה של O(logn).

**balance\_tree(self, node, bf) –** הפונקציה בודקת את הbalance factor שקיבלה כפרמטר ופועלת באחת משתי האפשרויות הבאות:

1. אם הוא גדול מ1 זה אומר שהצומת "כבד" משמאל לכן נבדוק את הbalance factor של הבן השמאלי שלו. אם קטן מאפס, נבצע סיבוב שמאלה על הילד השמאלי של הצומת ולאחר מכן סיבוב ימינה על הצומת. אם גדול מאפס נבצע סיבוב ימינה על הצומת בלבד.
2. אם הוא קטן מ(-1) זה אומר שהצומת "כבד" מימין לכן נבדוק את הbalance factor של הבן הימני שלו. אם גדול מאפס, נבצע סיבוב ימינה על הילד הימני של הצומת ולאחר מכן סיבוב שמאלה על הצומת. אם קטן מאפס נבצע סיבוב שמאלה על הצומת בלבד.

זמן הריצה של הפונקציה הוא קבוע מכיוון שהבדיקות נעשות בזמן קבוע והפונקציות שעושות סיבובים גם הן מתבצעות בזמן קבוע.

**rotate\_left(self, node) -** מקבלת כקלט צומת, ומבצעת עליו סיבוב שמאלה (הופכת אותו לבן השמאלי של הבן הימני הנוכחי, כך שהבן הימני של הצומת הנתון הפוך לאחר הפעלת הפונקציה לשורש של תת העץ). הפונקציה מעדכנת את שדות הצמתים (גובה וגודל) במקביל.  
 זמן הריצה O(1) שכן הפונקציה מבצעת כמות קבועה של פעולות.

**rotate\_right(self, node)** - מקבלת כקלט צומת, ומבצעת עליו סיבוב ימינה (הופכת אותו לבן הימני של הבן השמאלי הנוכחי, כך שהבן השמאלי של הצומת הנתון הפוך לאחר הפעלת הפונקציה לשורש של תת העץ). הפונקציה מעדכנת את שדות הצמתים (גובה וגודל) במקביל.  
 זמן הריצה O(1) שכן הפונקציה מבצעת כמות קבועה של פעולות.

**finger\_insert(self, key, val) –** הפונקציה מכניסה לעץ איבר בעל ערך v ומפתח k (שלא נמצא בעץ לפני כן) החיפוש למיקום ההכנסה הוא החל מהצומת המקסימלי. הפונקציה מחזירה שלשה  (x,e,h) x מצביע לצומת שנוצר,  eמספר הקשתות על מסלול ההכנסה, ו-ℎ מספר מקרי שינוי גובה (promote ) שנדרשו במהלך האיזון. מבחינת לוגיקה מלבד החיפוש החל מהצומת המקסימלי, הלוגיקה זהה לinsert ולכן לא נפרט בשנית. זמן הריצה של finger\_search הוא O(logn) כפי שהוסבר בתיעוד של הפונקציה, מתבצעות פעולות השוואה בO(1) וקריאה לbalance\_after\_insert שגם מתרחשת בO(logn) ולכן בסך הכל נקבל זמן ריצה של O(logn)

**delete(self, node) –** הפונקציה מקבלת מצביע לצומת למחיקה. הפונקציה מתמודדת עם אחד משלושה מקרים אפשריים:

1. לצומת אין ילדים, במקרה כזה נהפוך את הצומת לעלה וירטואלי. אם הצומת זה השורש נאתחל את שורש העץ לNone
2. אם לצומת יש ילד אחד בלבד, נשים את הילד במקום הצומת שנמחק. אם הצומת זה השורש נהפוך את הילד היחיד להיות השורש החדש.
3. לצומת יש שני ילדים, נחפש את הsuccessor של הצומת, נשים אותו במקום הצומת הנוכחי ואז נמחק את הsuccessor בקריאה נוספת לdelete.

לאחר מכן נבצע איזון לעץ בלולאת while שמעדכנת לכל צומת החל מהצומת שנמחקה ועד השורש את הגובה, בודקת את הbalance factor ובמידת הצורך קוראת לפונקציה balance\_tree.

לבסוף נקרא לset\_size עם -1 כדי לעדכן את גודל העץ לאחר המחיקה, ובמידה והצומת שמחקנו היה הצומת המקסימלי בעץ נקרא לupdate\_height כדי לעדכן מצביע חדש לצומת המקסימלי הנוכחי.

זמן הריצה של הפונקציה הוא O(logn) מכיוון שהבדיקה באיזה מקרה אנו נמצאים קורית בO(1), אם אנחנו במקרה שלוש חיפוש הsuccessor מתבצע בO(logn) כפי שראינו בהרצאה. נשים לב שיכולה להיות לכל היותר קריאה רקורסיבית אחת לdelete עם הsuccessor כי אם היו לו שני ילדים הוא לא היה היורש ולכן עומק הרקורסיה O(1)

לאחר מכן לולאת האיזון מתבצעת בO(logn) מכיוון שנלך לכל היותר על מסלול אחד לשורש בעץ ופעולות האיזון מתבצעות בזמן קבוע. לבסוף, קריאה לupdate\_max גם היא מתבצעת בO(logn) מכיוון שנחפש על מסלול אחד מהשורש לעלה הימני ביותר. לכן בסך הכל זמן הריצה של הפונקציה הוא O(logn)

**get\_successor(self, node) –** הפונקציה מקבלת צומת ומחזירה את הצומת הקטן ביותר בתת העץ של אותו צומת שקיבלה כקלט. מכיוון שנלך לכל היותר על מסלול אחד מהשורש לעלה המינימלי נסיק שזמן הריצה הוא O(logn).

**join(self, tree2, key, val) –** הפונקציה מקבלת עץ נוסף, מפתח וערך לייצר מהם מפתח חדש ולחבר בעזרתו את שני העצים כאשר ידוע כי המפתח הוא קטן מכל המפתחות בעץ אחד וגדול מכל המפתחות בעץ השני.

הפונקציה מטפלת במקרה שבו אחד או יותר מהעצים הוא ריק. במידה ושניהם לא ריקים הפונקציה בודקת מה העץ הגבוה יותר ומחפשת בו את הצומת שהגובה שלה זהה לגובה של השורש של העץ הנמוך יותר. חיפוש הצומת הזאת מתבצע על המסלול הימני ביותר או השמאלי ביותר בהתאם ליחסים בין גודל המפתח שקיבלנו כקלט למפתחות בעץ הגבוה.

לאחר מכן מחברים את הצומת החדש שיצרנו להיות אבא של השורש של העץ הקצר יותר ושל הצומת שמצאנו בעץ הגבוה (כיוונות הילדים תלויה בגדלים של המפתחות בעצים ביחד למפתח בקלט) והאבא (במידה ויש) של הצומת שמצאנו בעץ הגבוה יותר יהפוך להיות האבא של הצומת החדש.

הפעולות מתבצעות על העץ הגבוה יותר ולכן אם Tree2 הוא הגבוה, נקרא לself.set\_root עם השורש של tree2 כדי שהחיבור יבוא לידי ביטוי בעץ שלנו. בנוסף, במידה ויש צורך נעדכן גם את הצומת המקסימלי בקריאה לset\_max. לבסוף נבצע איזונים החל מהצומת החדש שקיבלנו כקלט ועד השורש ע"י עדכון הגובה בכל צומת וקריאה לbalance\_tree במקרה הצורך. לבסוף נעדכן את גודל העץ ונאתחל את השורש של Tree2 להיות None.

זמן הריצה הוא O(logn) מכיוון שחיפוש הצומת לחיבור העץ הגבוה זה לכל היותר מעבר על מסלול אחד מהשורש לעלה, והאיזונים גם הם יהיו על לכל היותר מסלול אחד מעלה לשורש כאשר איזון בפני עצמו מתרחש בO(1). כל שאר הפעולות בפונקציה כמו בדיקה של גבהים, טיפול במקרים של עץ ריק וחיבורי צמתים מתבצע בזמן קבוע ולכן בסך הכל זמן הריצה יהיה O(logn).

**split(self, node) –** הפונקציה מקבלת מצביע לצומת בעץ. עליה לפצל את העץ לשניים ולהחזיר (t1,t2) כך ש-t1 יכיל את המפתחות הקטנים מ-x, ו-t2  את הגדולים. לאחר הפעולה העץ הנוכחי וגם הצומת שקיבלנו כקלט אינם שמישים. הפונקציה עושה את זה ע"י מעבר על מסלול מהצומת שקיבלנו ועד לשורש ולפי הילדים של כל צומת במעבר היא מבצעת insert במקרה של צומת בודד להוספה או join במקרה של תת עץ גדול מ1 לt1 ול t2 לפי הצורך. ראינו בהרצאה שזמן הריצה הוא O(logn).

**in\_order\_to\_arr(self, node, arr) –** הפונקציה מכניסה את האיברים בעץ לפי האלגוריתם inroder - באופן רקורסיבי החל מהאיברים השמאליים לימניים.  
 זמן הריצה הוא O(n) שכן נבקר בכל צומת בעץ בין פעם לשלוש פעמים כדי להכניסו למערך ובעת המעבר לבנים שלו.

**avl\_to\_array(self) -** הפונקציה מחזירה מערך ממוין (ע״פ המפתחות) של האיברים במילון כאשר כל איבר מיוצג ע״י זוג סדור של (value ,key). קוראת לפונקציית העזר in\_order\_to\_arr עם שורש ומערך ריק ומחזירה את המערך.  
 זמן הריצה הוא O(n) בהתבסס על זמן הריצה של פונקציית העזר in\_order\_to\_arr.

**max\_node(self)  -** הפונקציה מחזירה מצביע לצומת בעל המפתח המקסימלי בעץ.

**update\_max(self) –** הפונקציה מעדכנת את המצביע לצומת המקסימלי בעץ ע"י לולאה שמבצעת מעבר לבן הימני כל עוד לצומת יש בן שאינו וירטואלי. אם העץ ריק מחזירה None.  
זמן הריצה O(logn) שכן נבקר לכל יותר ב-O(logn) (גובה העץ) צמתים בהם נבצע פעולות בעלות זמן ריצה קבוע.

**size(self)  -** מחזירה את הגודל את העץ ע"י גישה לשדה ה-size של שורש העץ. זמן הריצה O(1).

g**et\_root(self) –** מחזירה את הצומת שמשמש כשורש המבנה; השורש שמור כשדה במבנה ולכן גישה לשדה מתבצעת ב-O(1).

.