מבני נתונים – תרגיל מעשי 1

מגישים: איתמר גולדברגר ויואב לוי (מספרי ת.ז: 303078752, 20432618)

**תיעוד**

**WAVLTree (class)**

זהו עץ WAVL כפי שהוגדר לנו בכיתה. כל איבר בעץ הוא מסוג WAVLNode והעץ הינו בעל מספר שירותים, ביניהם שדה מסוג String השומר מידע ושדה מסוג int השומר key (פירוט אודות המחלקה WAVLNode בהמשך המסמך). מיון האיברים נעשה לפי הkey בדומה לעצים בינאריים.

כל מופע של המחלקה מכיל 3 שדות: **root, minode, maxnode** אשר מפורטים מטה. בעת אתחול

ראשוני וכאשר העץ "מתרוקן" (נמחק האיבר האחרון) שלושת שדות אלו מצביעים חזרה ל-null. בנוסף כאשר מכניסים איבר ראשון לעץ אז השדות הנ"ל מצביעים כולם אליו.

**Root (field)**

שדה זה שומר מצביע אל WAVLNode שנקרא שורש העץ ומאותחל בעץ יצירת העץ ל-null. במידה ומוחקים את השורש אז פונקציית Delete מחפשת לו מחליף - אשר הוגדר כ- successor. במידה וה- successor לא קיים (השורש הוא node אונרי) – אזי מוגדר כי root יצביע על ה-predecessor (ישנו רק מקרה אחד אפשרי כזה והוא המקרה בו קיימים 2 Nodes בעץ).

זהו למעשה "ראש" העץ, וניתן להגיע לכל איבר בעץ אם מתחילים ממנו ומחפשים את האיבר השמאלי/ימני כדרוש.

**Minode (field)**

שדה זה שומר מצביע ל- WAVLNodeבעל המפתח המינימלי בעץ וגם הוא מאותחל לnull. בכל הכנסת איבר בודקים האם האיבר החדש קטן מהמינימלי. אם כן, זהו האיבר המינימלי החדש ו-Minode יצביע עליו. כאשר נמחק האיבר המינימלי עובר המצביע לאיבר המינימלי הבא (successor). באופן זה נשמרת תמיד הגישה לערך ה-Node המינימיאלי ב-O(1).

**Maxnode (field)**

שדה זה שומר מצביע ל- WAVLNodeבעל המפתח המקסימלי בעץ וגם הוא מאותחל לnull. בכל הכנסת איבר בודקים האם האיבר החדש גדול מהמקסימלי. אם כן, זהו האיבר המקסימלי החדש. כאשר נמחק האיבר המקסימלי עובר המצביע לאיבר המקסימלי הבא (predecessor). בדומה לעיל גם כאן נשמרת גישה ל-Node המקסימאלי ב-O(1).

**Size: (field)**

שדה זה הינו מסוג int ומייצג את מספר ה-WAVLNode-ים בעץ. השדה מאותחל ל-0 כאשר בכל פעולת insert (הכנסת איבר לעץ) הוא גדל באחד ובכל פעולת delete (מחיקת איבר מהעץ) הוא קטן באחד.

הגישה ל-size גם היא דרך שדות העץ ולכן מתבצעת גם ב-O(1).

**Empty (method) O(1)**

פעולה זו בוחנת האם העץ ריק (ללא איברי WAVLNode). היא עושה זאת על ידי נסיון גישה לשורש העץ דרך השדה root - אם הוא null, משמע שהעץ ריק (אם הוא תוחזק כראוי) ולכן היא תחזיר true. במידה ולא היא תחזיר false.

זוהי כמות פעולות קבועה ולכן סיבוכיות הפונקציה היא O(1).

**Search (method) O(log(n))**

פונקציה זו מקבלת ארגומנט מסוג int, מחפשת WAVLNode בעץ בעל מפתח עם ארגומנט זהה ומחזירה את שדה ה-info שלו. מאחר ומימשנו פונקציית חיפוש (אשר תתואר בהמשך בשם searchNode) אשר מחזירה מצביע לWAVLNode הרלונטי – הפונקציה משתמשת בפונקציית העזר כדי למצוא מצביע לWAVLNode הרלוונטי, ואז ניגשת לשדה הinfo שלו כדי להחזיר ערך זה.

זוהי כמות פעולות קבועה בנוסף לכמות הפעולות של searchNode ולכן יש להן אותה סיבוכיות אסימפטוטית.

**Insert (method) O(log(n))**

תיאור כללי:

פעולה זו יוצרת WAVLNode עם מפתח מסוג int ומידע מסוג String (המוזנים לו) ומוסיפה אותו במקומו המתאים לפי מבנה WAVLTree. לאחר פעולת הקישור (ההכנסה) היא מאזנת את העץ כך שיישאר חוקי. לבסוף, היא מחזירה ערך מספרי של כמות פעולות האיזון שעשתה.

תיאור מפורט:

תחילה הפונקצייה שומרת מצביע לשורש העץ ויוצרת את האיבר המבוקש באמצעות הפרטים שהוזנו לו. לאחר מכן היא נעזרת במתודה empty כדי לבחון האם העץ ריק –

אם כן היא מאתחלת את האיבר החדש להיות השורש, האיבר המינימלי והאיבר המקסימלי בעץ. לבסוף היא מחזירה 0 משום שלא ביצעה כלל פעולות איזון.

אם לא היא ממשיכה הלאה בקוד.

בשלב זה היא בוחנת האם מפתח האיבר החדש גדול ממפתח האיבר המקסימאלי או קטן ממפתח האיבר המינימלי. במידה וכן, היא משנה את המצביעים של שדות אלו בהתאם (כמתואר לעיל ב-Minode ו-Maxnode).

בשלב הבא מבוצע חיפוש בינארי רגיל למקום בו הWAVLNode החדש אמור להימצא, זאת בעזרת לולאת while.

אם מצאנו WAVLNode בעל מפתח זהה אז יוחזר -1 מכיוון שמפתח זה כבר נמצא בעץ ואין להרשות כפילויות.

אם המפתח החדש קטן מהמפתח הנוכחי אז נחפש באגף שמאל של תת העץ, אחרת, נחפש באגף ימין. כאשר המצביע הבא הוא null נבין שבמיקום זה אמור להימצא האיבר החדש, נזכור לשמור מצביע לאיבר האמיתי האחרון (שאינו null, כלומר ה-parent שלו) ונקשר את האיבר החדש במקומו. לאחר מכן נגדיל את שדה size ב-1 ונקרא לפונקציית העזר rebalanceInsert על האיבר שהוספנו כדי לאזן את העץ על מנת שיישאר במבנה תקין.

יש לציין כי פעולות ה-insert בהכרח תבוצע בעלה (leaf) או ב-unary node, זאת מתוקף הגדרת הפעולות הבסיסיות על עץ בינארי.

גובה עץ WAVL בעל n איברים הוא לכל היותר בסדר גודל של log(n). מאחר ובכל מעבר על לולאת הwhile ירדנו רמה בעץ, לכל היותר עד קצה העץ, אזי כמות הפעולות האסימפטוטית שביצענו היא log(n). לאחר מכן נקראת פונקציית העזר של פעולת האיזון (rebalanceInsert) – אם פונקצייה זו בעלת סיבוכיות של לא יותר מlog(n) פעולות באופן אסימפטוטי אז סיבוכיות הריצה המצטברת של Insert הוא אכן log(n), זאת נראה בהמשך.

**rebalanceInsert (method) O(log(n))**

פונקצייה זו מקבלת כקלט את האיבר שנוסף לעץ, משמשת כמעיין פונקציית מעטפה שמבצעת איזונים בסיסיים וקוראת לפונקציית עזר נוספת רקורסיבית כדי לוודא שהעץ מאוזן ולאזן אותו. כמו-כן מתודה זו מחזירה לבסוף מספר פעולות האיזון.

ראשית, מאותחל מספר פעולות האיזון ל- 0 (משתנה cnt). פונקצייה זו מקבלת מצביע אל הורה האיבר שנוסף ובוחנת באיזה מקרה בסיס מדובר (מצביע זה לא יקבל ערך null מכיוון שבהכרח יש לו הורה, שכן התמודדנו עם מקרה הבסיס בו העץ ריק ולא קיים שורש קודם לכן).

כעת, אם הוספנו איבר להורה מדרגה 1 (כלומר unary node - מקרה B במצגת) אז העץ כבר מאוזן וניתן להחזיר את כמות פעולות האיזון (0) ללא המשך טיפול.

במקרה האחר (כלומר כאשר מוסיפים איבר לעלה -מקרה A במצגת), יש להעלות את דרגת ההורה, לספור את פעולת האיזון ולקרוא לפונקציית העזר הרקורסיבית rebalanceInsertRec המקבלת מצביע להורה של האיבר שהוכנס ואת מספר פעולות האיזון שנעשו עד כה (בשלב זה 1).

בסה"כ מדובר בכמות פעולות קבועה, מלבד אלו הנעשות על ידי פונקציית העזר, ולכן סיבוכיות הריצה של פונקציה זו תהיה זהה לסיבוכיות הריצה של הפונקציה rebalanceInsertRec.

**rebalanceInsertRec (method) O(log(n))**

במידה ופונקצייה זו נקראה, היא מקבלת כקלט מצביע לאיבר בעץ החל ממנו יש לאזן ואת מספר הפעולות שנעשו עד כה בעץ (כלומר תקבל מצביע להורה של ה-Node שהוכנס). הפונקציה מאזנת באופן רקורסיבי את העץ (כשדרוש היא קוראת לעצמה עם האיבר הבא אותו יש לאזן) ומחזירה את כמות הפעולות החדשה שנעשו עד כה.תנאי העצירה לפוקציה רקורסיבית זו הוא אם האיבר שהוזן הוא שורש העץ או כאשר הסתיימה פעולת האיזון באמצעות rotate). במקרה כזה פעולת האיזון הושלמה והמתודה תחזיר את מספר פעולות האיזון שנעשו.

כאמור, תחילה ישנו תנאי עצירה ראשון – אם ה-node שהוכנס כקלט הינו שורש העץ.

במידה ולא, הפונקציה ממשיכה לשלב הבא.

בשלב זה, הפונקצייה נעזרת בפונקציית העזר caseInsrt אשר מקבלת את האיבר עליו נדרש לבצע את פעולות האיזון ובוחנת באיזה מקרה מדובר (כלומר, הפונקציה בודקת את הפרשי הדרגות בסביבתו הקרובה של תת העץ). פונקציית עזר זו מחזירה משתנה מסוג int אשר מסמן את מספרו הסידורי של המקרה על מנת שנדע כיצד להתמודד איתו (תיאור מפורט אודות הסימונים בהסבר על פונקציית העזר).

אם הפונקצייה **החזירה** **0** משמע כי לא קיימת בעיה בעץ וניתן לסיים את התהליך ולהחזיר את מספר פעולות האיזון הכולל.

**במקרה 1** הפונקציה תעלה את דרגת ההורה של האיבר הנבחן, תספור זאת כפעולת איזון ותקרא מחדש לעצמה (rebalanceInsertRec). כעת מוזן לה (במקום ה-node הנוכחי) מצביע להורה של האיבר הנבחן (כלומר "גילגלנו" את הבעיה במעלה העץ) וכמות פעולות האיזון שנעשו עד כה. המקרה הסימטרי זהה מבחינת טיפול.

**\*הבהרה**: כאשר נתייחס למקרים סימטרים נעזר בסימן ' / ' כדי להסביר את הפעולה הנעשות בכל מקרה בנפרד, בהתאמה. למשל אם נתאר את מקרים 3/6 נכתוב שנעשית פעולה3/פעולה6. כלומר פעולה3 נעשית במקרה 3 ופעולה6 נעשית במקרה6.

**במקרים 2/5** הפונקציה תבצע סיבוב ימינה/שמאלה על האיבר בעזרת פונקציית העזר rotate (תפורט בהמשך), תוריד את דרגתו של האיבר שנמצא מימין/שמאל לאיבר הנבחן ותסיים על הפעולה על ידי החזרת כמות פעולות האיזון שנעשו עד כה.

**במקרים 3/6** הפונקציה תשמור מבעוד מועד מצביע ל-node הילד מימין/שמאל לאיבר הנבחן (צ'אק/נוריס בקוד), תבצע סיבוב סביב צ'אק/נוריס שמאלה/ימינה ואז ימינה/שמאלה בעזרת פונקציית העזר rotate ואז תשנה את הדרגות הדרושות: דרגת האיבר הנבחן והאיבר מימין/שמאל לצ'אק/נוריס תרד, ודרגתו של צ'אק/נוריס תעלה. לבסוף תסיים הפונקצייה את פעולתה ותחזיר את כמות פעולות האיזון שנעשו עד כה.

במקרה והפונקציה לא עשתה דבר או במקרים אחרים בהם לא קיימת בעיה שהתגלתה - היא תחזיר 0.

לסיכום, מדובר בכמות פעולות קבועה בכל קריאה לפונקציה ולכל היותר הבעיה "גולגלה" מתחילת העץ ועד שורש העץ, כלומר בכל רמה של העץ נעשתה כמות פעולות גבוהות וגובה העץ חסום ע"י log(n) ולכן סיבוכיות הזמן של פונקציה זו הוא log(n) (בידיעה שפונקציות העזר אינן גורעות מזמן ריצה זה).

**caseInsrt (method) (O(1)**

פונקציה זו מקבלת מצביע לאיבר אותו יש לבחון ומחזירה את מספרו הסידורי של המקרה המדובר לפי הפרשי הדרגות בסביבה המיידית שלו.

הפונקציה פועלת בדרך הבאה:

תחילה, היא שומרת מצביע נוסף להורה (parent) של ה-node שהוכנס כקלט לפונקציה. לאחר מכן היא משתמשת בפונקציות העזר rankLeft ו- rankRight (יפורטו בהמשך) כדי לבחון מה הפרש הדרגות בין האיבר שמוזן אליהן (parent) והאיבר שמשמאל ומימין (הילדים) בהתאמה.

המספרים שהמתודה מחזירה מייצגים את מספרי המקרים כפי שהוצגו במצגת, כאשר למקרים הסימטריים הוספנו 3 למספר הסידורי (המקרה הסימטרי של 2 הוא 5 ושל 3 הוא 6 בהתאמה).

* הפונקציה בודקת אם הפרשי הדרגות של ההורה הם 1-0. במידה וכן, מדובר במקרה 1.
  + במידה ולא, אם הפרש הדרגות של ה-parent מימין ושמאל הוא 2 ו0 בהתאמה :
    - אם הפרש הדרגות של ה-node מימין הוא 2 מדובר במקרה 2.
      * אחרת אם הפרש הדרגות של האיבר משמאל הוא 2 מדובר במקרה 3.
  + אחרת, אם הפרש הדרגות של ההורה משמאל ומימין הוא 2 ו0 בהתאמה:
    - אם הפרש הדרגות של האיבר מימין הוא 2 מדובר במקרה 6.
      * אחרת אם הפרש הדרגות של האיבר משמאל הוא 2 מדובר במקרה 5.
* אם לא הגענו לאף אחד מהמקרים הבעייתיים נחזור 0.

סיבוכיות פונקצייה זו היא O(1) משום שהיא מבצעת כמות פעולות סדורה קבועה.

**Rotate O(1):**

פונקצייה זו מקבלת איבר מסוג WAVLNode וערך בוליאני ומבצעת על האיבר פעולת "סיבוב" שמאלה אם הוזן לה "true" וימינה אחרת (false). פונקציה זו אינה מחזירה אף ערך (void) ורק מבצעת את פעולות הרוטציה. המתודה, בהתאם לסיבוב שמאלה/ימינה, שומרת מצביע לאיבר משמאל/ימין לאיבר הנבחן (b במצגת ולכן נקרא לו b מעתה), מצביע נוסף להורה של האיבר (z במצגת) ולהורה של z (מחוץ למצגת, נקרא לו סבא). בשלב זה, הפונקציה משימה את z בתור הילד השמאלי/ימני של האיבר ולאחר מכן משימה את הסבא בתור ההורה של האיבר. אם הסבא אינו null (מבוצעת בדיקה) אז המתודה משימה גם את האיבר בתור הילד החדש של הסבא (לאחר שבחנה האם הוא צריך להיות ילד ימין או שמאל שלו). אם הסבא הוא null, משמע ש-z היה השורש, ונשים את האיבר בתור השורש ונשים את null בתור ההורה שלו. לאחר מכן נשים את האיבר להיות ההורה של z ונשים את b להיות הילד הימני/שמאלי של z. אם b אינו null אז נשים את z להיות ההורה של b. בסה"כ בכל פעולת rotate מבוצעות מספר קבוע וסופי של פעולות ועל כן הסיבוכיות הינה O(1).

\*\* בסה"כ קיבלנו כי כל המתודות עזר אשר insert משתמשת בהן בעת פעולת הכנסה אחת אינן יותר מ-log(n) כל אחת (O(log(n)) או O(1)) ועל כן, כולל פעולת חיפוש האיבר – insert אינה מבצעת יותר מ-log(n) פעולות.

**Delete (method):**

מטרת מתודה זו הינה למחוק איברים מהעץ (nodes), זאת בהנחה שהם קיימים. הפונקציה מקבלת כקלט את ה-key של ה-node הרלוונטי ומחזירה ערך מסוג int עבור מספר פעולות האיזון (rotate,demote & promote) שבוצעו. לאחר מכן היא משתמש בפונקציית searchNode (פירוט אודות המתודה בהמשך) על מנת להשיג מצביע רלווטי ל-Node אותו היא אמורה למחוק. פעולות ה-searchNode מוערכת ב-Worst Case כ-log(n). בשלב זה, אם חוזר מהמתודה Null, משמע כי האיבר לא קיים והמתודה תחזיר את הערך (1-).

במידה ואותר node המסומן למחיקה שאינו null תבדוק הפונקציה האם node זה הינו ה-minode או ה-maxnode (או שניהם). במידה וכן, מוגדרים מחדש המצביעים של Minode ו-maxnode להיות ה-successor וה-predecessor בהתאמה (ובהתאם למאורע). אם הילד הרלוונטי אינו קיים (כלומר האלגוריתם לחיפוש האיבר הקודם או העוקב מחזיר Null) משמע כי ההורה הינו האיבר ה- successor או ה- predecessor בהתאמה ועל כן יוגדרו ההורים בהתאם למקרה להיות ה-maxnode או ה-minode (או שניהם).

בסופו של הליך זה ישנה קריאה למתודה rebalanceDelete אשר מחזירה ערך int (של מספר פעולות האיזון) ומטרתה לבצע את ההסרה בפועל של האיבר וכן את פעולות האיזון הנובעות מכך. כל פעולות ה- rebalanceDelete וכן שאר המתודות בהן היא משתמשת מבצעות את פעולתן במקרה הגרוע ב-log(n) פעולות (הסבר אודות כל מתודה בהמשך). מכיוון שכך ומכיוון שפעולת חיפוש ה-node למחיקה גם היא ב-log(n) פעולות, אזי נקבל כי סך הפעולות המבוצעות ב-delete מוערכות בסיבוכיות של O(log(n)).

**rebalanceDelete (method):**

לאחר כל פעולת delete נקראית מתודה זו. מתודה זו מקבלת כקלט את המצביע ל-node שאותר בפונקציית ה-delete ותפקידה העיקרי הינו לזהות איזה סוג של פעולת מחיקה נדרשת (מחיקת עלה, מחיקת unary node או מחיקת binary node) ולהתמודד עם המקרה בהתאם.

דרך פעולת המתודה:

ראשית, המתודה מאתחלת משתנה מסוג int בשם cnt עם ערך 0 אשר אמור לספור את פעולות האיזון.

לאחר מכן, המתודה מתמודדת עם מקרי הקצה בהם node הקלט שהוזן הינו שורש העץ. לאחר מכן המתודה בודקת האם השורש הוא עלה או unary node. במידה והשורש הינו עלה – מתבצעת הסרה של השורש, כלומר הפניית כל מצביעי העץ ל-null והנמכת ה-size של העץ. במקרה ומדובר ב-unary node מבוצעת החלפה של השורש עום הילד הרלוונטי (לאחר שמאותר אם הוא הילד השמאלי או הימני) ובדרך זאת מוסר השורש.

במידה וה-node שהוזן למחיקה אינו השורש מבוצעים התהליכים הבאים:

כאשר ה-node הינו עלה:

* אם זהו עלה אשר ההורה שלו בעל הפרש דרגות מסוג 1-1, תיקרא הפונקציה removeLeaf (ר' פירוט בהמשך) אשר תסיר את העלה, יוחזר הערך 0 ותסתיים המחיקה.
* אם זהו מקרה בו ההורה של הילד בעל קשתות מסוג 1-2 והילד השני של ההורה אינו קיים (external leaf), אזי תיקרא הפונקציה removeLeaf שתסיר את העלה הרלוונטי, תנמיך את דרגת ההורה ותיקרא הפונקציה הרקורסיבית rebalanceDeleteRec (תפורט בהמשך) על "סבא" של העלה אותו מחקנו (כמובן שנשמר מבעוד מועד מצביע להורה בכדי לא לאבדו בעת מחיקת העלה).
* בכל שאר המקרים שבו מוחקים עלה – תבוצע הסרה של העלה באמצעות removeLeaf והפונקציה הרקורסיבית rebalanceDeleteRec תיקרא הפעם על ההורה (ולא על הסבא).

כאשר ה-node הינו binary node:

* במידה וה-node אותו נרצה למחוק הינו binary node, כלומר בעל 2 ילדים קיימים אזי ננקוט בגישה אחרת. ראשית נשמור מצביע ל-successor (חייב להיות לו כזה מכיוון שיש לו שני ילדים). לאחר מכן נבצע העברה של המידע מה-successor node ל-node אותו נרצה למחוק, כלומר נעדכן את את ערך ה-key וה-info של ה-node אותו התכוונו למחוק בערכי ה-key וה-info של ה- successor node. לאחר מכן נשמור מצביע להורה של ה- successor nodeונפעיל מחדש את הפונקצייה rebalanceDelete עם המצביע ל- successor node המקורי. בנוסף נבדק התנאי שהוא לא ה-maxnode (במידה וכן, מעודכן המצביע ל-node "הראשוני" אותו התכוונו למחוק אשר כעת מכיל את המידע של ה-successor.

כאשר ה-node הינו unary node:

* ראשית, נבדוק האם הילד של ה-node אותו נרצה למחוק הינו הילד השמאלי או הימני ונגדירו באמצעות משתנה.
* לאחר מכן נבדוק את המקרה הרלוונטי:
  + במידה ולהורה של ה-node יש קשת 1-1, תבוצע מחיקה של ה-node באמצעות המתודה replaceUnary (תפורט בהמשך) ויוחזר cnt.
  + במידה וההפרש דרגות בינו לבין ההורה הינו 1 - נקראית הפונקציה replaceUnary ומוחזר cnt.
  + במידה וההפרש דרגיות בינו לבין ההורה הינו 2 – נקראית הפונקציה replaceUnary ולאחר מכן

rebalanceDeleteRec על ההורה.

\*\* סך הפעולות המבוצעות במתודה זו הינן פעולות קבועות ולכן אינן משפיעות על סיבוכיות זמן פונקציית ה-delete (בהנחיה שכל מתודות העזר בהן השתמשנו הינן גם לכל היותר O(log(n)).

**rebalanceDeleteRec:**

מתודה זו מטרתה לבצע פעולות איזון הרלוונטיות לעץ לאחר הסרת האיבר הרלוונטי שבוצעה בפונקציה rebalanceDelete וכן ספירתן.

המתודה מקבלת כקלט את ה-node עליו נרצה להתחיל לבצע את פעולות האיזון ומחזירה את cnt (אשר מסוג int) המסמל את מספר פעולות האיזון.

אחד מתנאי העצירה של מתודה רקורסיבית זו הינו האם ה-node שהוזן הינו Null. במקרה זה מחזירים את ערך ה-cnt המקורי. לאחר מכן משתמשת המתודה בפונקציה caseDelete (בדומה ב-insert) אשר בודקת איזה מקרה איזון יש לבצע ומחזירה מספר מקרה רלוונטי. בהתאם למספר המקרה – מבוצעות פעולות האיזון ופעולות ההמשך הרלוונטיות לאותו מקרה (פירוט בפונקציית ה-caseDelete).

* **מקרה 1:**

במקרה זה נבצע demote ל-node ו"נגלגל" את פונקציית ה- rebalanceDeleteRec להורה שלו.

* **מקרה 2:**

במקרה זה נבדוק תחילה אם ה-node אם הפרש דרגות של 1 הינו מהילד השמאלי או הימני (בכדי שנדע על איזה ילד לבצע demote כלומר אם אנחנו במקרה הסימטרי). לאחר שביצענו על הילד demote נבצע זאת גם על ההורה שלו (ה-node המקורי) – כלומר בסה"כ double demote. כמו-כן, לאחר מכן "נגלגל" את פונקציית ה- rebalanceDeleteRec להורה שלו.

* **מקרה 3:**

במקרה זה נבצע demote ל-node ולילד הימני שלו ולאחר מכן נבצע rotate לכיוון left על הילד הימני. במידה ולאחר ה-rotate אנו במקרה בו מדובר בעלה שהפרש הדרגות שלו משני ה-External Leaf שלו הינו 2-2, נבצע demote נוסף על ה-node. לאחרי מקרה זה מסתיים הטיפול ומחזירים את cnt הרלוונטי לאחר ששונה בהתאם למספר פעולות האיזון.

* **מקרה 13:**

מקרה זה הינו המקרה הסימטרי למקרה 3. את פעולות ה-demote נבצע על הילד השמאלי ואת פעולת ה-rotate נבצע לכיוון ימין.

* **מקרה 4:**

במקרה זה נשמור מצביע לנכד הימני-שמאלי של ה-node. ראשית נבצע עליו פעולת rotate-right, לאחר מכן rotate-left, נבצע demote ב-2 ל-node המקורי, נבצע promote ב-2 למצביע ששמרנו לנכד הימני-שמאלי (שכעת משמש כסבא לאחר פעולות ה-rotate) וכן נבצע demote נוסף לילדו הימני החדש. לאחר שלל פעולות אלה, מסתיים הטיפול ומחזירים את cnt הרלוונטי לאחר ששונה בהתאם למספר פעולות האיזון.

* **מקרה 14:**

זהו המקרה הסימטרי של 4. הטיפול הינו זהה ל-4 מלבד זאת שימין ושמאל כעת הפוכים.

* **בשאר המקרים:**

נחזיר את cnt.

\*\* מכיוון שבמקרה הגרוע אנו נבצע פעולות רקורסיביות מתחתית העץ ועד לשורש (למשל במצב בו יש מקרי 2 או מקרי 1 שחוזרים על עצמם) אזי מספר הפעולות יהיה ביחס לגובה העץ. מכיוון שגובה העץ הינו כ-log(n) אזי נקבל כי סיבוכיות הפעולות במתודה זו הינה לכל היותר log(n).

**caseDelete (Method):**

מטרת מתודה זו הינה לבדוק את סטאטוס העץ לאחר ביצוע הסרת ה-node הרלוונטי ולהחזיר ערך מספרי אשר יגדיר לנו אילו פעולות נרצה לבצע בפונקציית ה- rebalanceDeleteRec. פונקצייה זו מקבלת בתור קלט WAVLNode ומחזירה ערך int של המקרה הרלוונטי. המקרים המתוארים מתואמים עם המקרים הנלמדו בשיעור (מקרים, 1,2,3,4) כאשר המקרה הסימטרי של 3 הוגדר 13 ושל 4 – 14 בהתאמה. להלן הסבר אודות המקרים (המקרים הסימטריים יירשמו בסוגריים):

* אם הפרש הדרגות של ה-node מאחד מילדיו הוא 3 אזי:
  + אם הפרש הדרגות מאחד הילדים השני הוא גם 2 – משמע שאנו **במקרה 1**.
  + במידה ולא, נגדיר את הילד שהפרשו 3 מההורה בתור משתנה (child).
    - אם ל-child יש קשת 2-2 מילדיו אזי אנו **במקרה 2.**
    - אם הילד הינו ילד ימני (שמאלי) והפרש הדרגות שלו מילדו הימני (השמאלי) הינו 1, אזי אנו **במקרה 3 (מקרה 13).**
    - אם הילד הינו ילד ימני (שמאלי) והפרש הדרגות שלו מילדו הימני (השמאלי) הינו 2, אזי אנו **במקרה 4 (מקרה 14).**
* המשמעות שאנחנו לא באף באף אחד מהמקרים הנ"ל היא שלא קיימות בעיות בעץ ועל-כן במצב זה יוחזר **מקרה 0**.

\*\* בסה"כ מדובר בפעולה קבועה של פעולות ועל-כן סיבוכיות פעולת הפונקציה הינה O(1). כמו-כן, מכיוון שכלל מתודות העזר בהן השתמשנו בפונקציית ה-delete הינן עד סדר גודל של log(n) משמע כי פעולת ה-delete הינה O(log(n)).

**rankRight(method):**

מטרת מתודה זו הינה לשאול node כלשהו מהו הפרש הדרגות בינו לבין ילדו הימני. המתודה מקבלת כקלט node אשר ייקרא parent ומחזירה ערך int המייצג את הפרש הדרגות. המתודה פועלת באופן הבא:

המתודה שואלת תחילה האם ילדו הימני של ה-parent הינו null. במידה והתשובה לשאלה זו הינה true (כלומר מדובר ב-"External Leaf") יוחזר ערך ה-rank של ה-parent + 1. מכיוון ש-external leaf מקבלים ערך של (1-) ואנו מחפשים את הפרש הדרגות, אזי נקבל parent.rank+1.

במידה וילדו הימני אינו Null אזי יוחזר הפרש הדרגות בין ה-parent לילדו הימני באמצעות גישה לשדות ה-rank של כל node והחסרתם זה מזה. מכיוון שמבוצע מספר קבוע של פעולות, אזי סיבוכיות מתודה זו הינה O(1).

**rankLeft (method):**

מטרת מתודה זו הינה דומה ל-rankRight ופועלת באופן דומה למתודה זו, מלבד זאת שהפעולות מוצעות על ילדו השמאלי של ה-parent ולא הימני.

**isLeft (method):**

מטרת מתודה זו הינה לשאול האם ילד (node) הינו בנו השמאלי או בנו הימני של node אחר (ה-parent). הקלט המוכנס למתודה זו הינו node מסוג WAVLNode והערך המתקבל הינו true במידה והוא בנו שמאלי, false במידה והוא הימני. המתודה פועלת באופן הבא:

ראשית, שומרים משתנה זמני המצביע ל-parent של ה-node המוזן. לאחר מכן אנו שואלים האם ילדו השמאלי של ההורה הינו null. אם התשובה היא true משמע בוודאות כי מדובר בילד הימני ועל כן יוחזר הערך false. לאחר מכן שואלים את ה-parent האם ילדו השמאלי הינו ה-node המוזן. במידה והתנאי מתקיים נחזיר ערך true, במידה ולא, ערך false. מכיוון שמבוצע מספר קבוע של פעולות סיבוכיות המתודה הינה O(1).

**removeLeaf (method):**

מטרת מתודה זו הינה "להסיר" מהעץ עלה. מתודה זו מקבלת כקלט node לאחר שנבדק כי הוא "עלה". ראשית המתודה שומרת את ההורה בתור משתנה. לאחר מכן המתודה בודקת האם העלה הוא בנו הימני או השמאלי של ה-parent. בהתאם לכך, המתודה מחליפה ל-null את המצביע הרלוונטי לילד ובאופן דומה את מצביע parent אצל הילד. מכיוון שמדובר בכמות קבועה של פעולות הסיבוכיות הינה O(1).

**replaceUnari (method):**

מטרת מתודה זו הינה "להסיר" מהעץ unary node. מתודה זו מקבלת כקלט node ואת הילד שלו ("הנכד"), זאת לאחר שנבדק כי ה-node הוא מסוג unary. באופן דומה לעיל המתודה שומרת את ההורה ("סבא") בתור משתנה. לאחר מכן המתודה בודקת האם הילד הינו ילדו השמאלי או הימני ולבסוף משנה את המצביעים כך שה"נכד" יזהה בתור parent את ה-"סבא" וה"סבא" יזהה בתור ילד את "הנכד". מכיוון שמדובר בכמות קבועה של פעולות הסיבוכיות הינה O(1).

**findSuc (method):**

מטרת פונקצייה זו הינה למצוא את האיבר העוקב לאיבר המוזן ולהחזיר מצביע אליו. על-כן, פונקציה זו מקבלת כקלט WAVLNode ומחזירה כפלט מצביע לאיבר העוקב (גם WAVLNode). במתודה זו אנו משתמשים בעת פעולת מחיקה של איבר (כאשר אנחנו רוצים להחליף ביניהם את הערכים). מסיבה בו, מתודה זו מופעלת רק בעת מחיקת node בינארי (בעל 2 ילדים). לכן, מתודה זו פועלת באופן שהיא ניגשת לילד הימני של ה-node המוזן ולאחר מכן לילד הכי שמאלי (עד סוף העץ) עד שמגיעים ל-null. האיבר העוקב יהיה ה-node האחרון לפני ה-null ובאופן זה מובטח לנו כי node זה יהיה העוקב. מתודה זו מממשת בעצם את האלגוריתם הנלמד בשיעור לחיפוש איבר עוקב בעצים בינאריים. מכיוון שייתכן וה-node לו מחפשים איבר עוקב נמצא בשורש, קיימת האפשרות כי בעת מימוש האלגוריתם נרד לתחתית העץ עד לעלה ועל כן נרד בכל גובהו של העץ. מסיבה זו, סיבוכיות מתודה זו הינה O(log(n)) – כגובה העץ.

**findPre (method):**

מתודה זו זהה למתודת ה-findSuc, מלבד זאת שמטרתה להחזיר מצביע לאיבר הקודם בתור (איבר ה-predecessor) ושיטת חיפוש הפוכה. הקלטים אותה מקבלת הפונקציה זהים והפלט הינו מצביע לאיבר ה- predecessor. אופן החיפוש דומה למתודת האיבר העוקב מלבד זאת שהיא הפוכה – כלומר מתחילים בילד השמאלי ולאחר מכן מגיעים עד לילד הכי ימני. עקב כך, סיבוכיות פונקציה זו גם היא מבוצעת ב-log(n).

**searchNode(method):**

מתודה זו הינה פונקצייה אשר מקבלת בתור קלט ערך מסוג int ובודקת האם קיים node בעץ אשר לו key השווה לקלט ה-Int שהוכנס. אם קיים node אשר עונה על התנאים המתודה מחזירה הפנייה ל-node זה (כלומר מחזירה WAVLNode). במידה ולא, חוזר ערך null.

דרך פעולת המתודה הינה כדלקמן:

הפונקציה ניגשת לשורש העץ ושומרת מצביע ל-node הרלוונטי. לאחר מכן מתחילה לפעול לולאת while אשר ממשיכה כל עוד ה-node הנשאל אינו null. בכל פעם נשאל ערך ה-int המוזן אם הוא זהה לערך ה-key של ה-node הרלוונטי. אם הוא שונה, הוא נשאל האם הוא גדול או קטן ממנו. במידה והוא גדול משתנה ההפנייה להיות הבן הימני של ה-node הקודם, במידה והוא קטן – הבן השמאלי. באופן זה ממשיכים "להתקרב" ל-node המתאים. אם בסופו של תהליך לא נמצא כזה node מוחזר ערך null.

מכיוון שאנו נעים במסלול מהשורש ועד לעלה אזי סיבוכיות החיפוש הינה כגובה העץ, כלומר אם יש n איברים בעץ אזי חיפוש ה-node וסיבוכיות המתודה הינה (O(log(n). מכיוון שגם מתודת ה-search מבוססת עליה ומבצעת פעולה אחת נוספת גם מתודת ה-search הנ"ל הינה בסיבוכיות O(log(n)).

**min (method):**

מטרת מתודה זו הינה להחזיר את שדה ה-info (אובייקט מסוג String) של ה-node בעל ה-key המינימאלי בעץ. לאור כך שבעת פעולות ה-insert וה-delete מתוחזק שדה ה-minode של עץ ה-WAVL, כלומר באופן קבוע אנו מחזיקים מצביע ל-node בעל ה-key המינימאלי, מימוש המתודה מבוצע באמצעות גישה ישירה לשדה info של node זה והחזרתו. על-כן, סיבוכיות המתודה הינה O(1).

**Max (method):**

מטרת מתודה זו הינה להחזיר את שדה ה-info (אובייקט מסוג String) של ה-node בעל ה-key המקסימאלי בעץ. לאור כך שבעת פעולות ה-insert וה-delete מתוחזק שדה ה-maxnode של עץ ה-WAVL, כלומר באופן קבוע אנו מחזיקים מצביע ל-node בעל ה-key המקסימאלי, מימוש המתודה מבוצע באמצעות גישה ישירה לשדה info של node זה והחזרתו. על-כן, סיבוכיות המתודה הינה O(1).

**nodesArrayRec (method):**

מתודה זו הינה מתודת עזר המשמשת את הפונקציות keysToArray ו-infoToArray. מתודה זו מקבלת שלושה קלטים – איבר מסוג WAVLNode (node), מערך מסוג WAVLNode (arr\_rec) ואינדקס מסוג int (index) ומחזירה כפלט ערך int (index). המתודה פועלת על פי אלגוריתם חיפוש שנלמד בשיעור למיון nodes מהקטן לגדול בעץ בינארי בשיטת (left, middle, right). המתודה פועלת באופן רקורסיבי כאשר תנאי העצירה הוא האם ה-node המוזן הינו Null. במידה ולא המתודה תפעיל את הפונקציה הרקורסיבית על node.left ויוחזר אינדקס מתאים. לאחר מכן נשתמש באינדקס שחזר כאינדקס בו נשים את האיבר node במערך ה-node-ים, נעלה את index ב-1 ונפעיל שוב את הפונקציה הרקורסיבית על node.right. באופן זה אנו יוצרים מערך של node-ים המסודרים מהקטן לגדול (על-פי מיקום בעץ אשר נקבע על פי ה-key). לבסוף חוזר ביטוי ה-index.

מכיוון שבשיטה זו, אם נזין בתור ה-node הראשון את שורש העץ נגיע לכל האיברים בעץ – אזי סיבוכיות פונקציה זו במקרה הגרוע הינה כגודל העץ, כלומר O(n), או בחסם הדוק יותר כ-, כאשר h הוא גובה ה-node.

**keysToArray (method):**

מתודה זו מוציאה כפלט מערך ממויין מהקטן לגדול של כל המפתחות הקיימים בעץ WAVL.

המתודה תחילה יוצרת מערך WAVLNode ריק בגודל ה-size של העץ וכן מערך זהה בגודל מסוג int. לאחר מכן המתודה מפעילה את מתודת העזר nodesArrayRec, על הפרמטרים הבאים:

* שורש העץ
* מערך ה-WAVLNode שאותחל לעיל.
* ערך int השווה ל-0.

לאחר סיום מתודת העזר הרקורסיבית, מערך ה-WAVLNode מתמלא. בשלב זה נעבור בלולאה על כל המערך ובכל פעם נשים את ערך ה-key של כל node במערך ה-int שיצרנו. לבסוף נקבל מערך int ממויין בכל מפתחות העץ מסודרים מהקטן לגדול.

מכיוון שהפונקציה הרקורסיבית פועלת ב-O(n) והלולאה להעתקת ה-keys גם פועלתת ב-O(n) נקבל בסה"כ כי סיבוכיות הפעלת מתודה זו הינה O(n).

**infoToArray (method):**

מתודה זו מוציאה כפלט מערך של כל ה-info המתאים למפתחות הרלוונטיים ב-node החל מהמפתח הקטן ביותר ועד למפתח הגדול ביותר בעץ ה- WAVL. מתודה זו פועלת בדיוק כמו מתודת ה- keysToArray מלבד העובדה שניצור מערך מסוג String ולא int וכן בעת השמת הערכים, נפנה לערך ה-info של ה-node ולא ערך ה-key. מאותן סיבות נקבל גם במקרה זה כי סיבוכיות הפעלת המתודה הינה O(n).

**Size (method):**

פונקציה זו מחזירה את גודל העץ (אובייקט מסוג int), כלומר את מספר ה-node-ים הקיימים בכל העץ. לאור כך ששדה ה-size מתוחזק באופן קבוע כאחד משדות מופע ה-WAVLTree, כלומר בעת פעולת ההכנסה אנחנו מוסיפים אחד לגודל ובעת פעולת מחיקה מקטינים באחד את הגודל, אזי שדה ה-size נשאר מעודכן עבור כל פעולה. על-כן, מתודה זו מחזירה את שדה עץ ה-WAVL (השדה size) ומבצעת זאת ב-O(1).

**WAVLNode(Class):**

זוהי מחלקת עזר אשר יוצרת איברים מטיפוס WAVLNode (כשמה). כל node הינו בעל מספר שירותים (אשר יפורטו מטה) וכל node משמש כאבן הבניין של עץ ה-WAVL. העץ מחזיק 3 שירותים המחזירים ערך (Key, Rank ו-info) ו-4 שירותים המחזיקים מצביעים (right, left ו-parent). בעת יצירת node חדש מאותחלים שדות אלה בהתאם לטיפוס הרלוונטי הדיפלוטיבי. כאשר מבצעים insert של node לעץ – מאתחלים node ומחברים את המצביעים במקומות הרלוונטיים. כאשר מבצעים delete ל-node מבצעים – לא מוחקים את המופע עצמו מהזיכרון אלא משנים את מצביעיו כך שלא ניתן יהיה לגשת אליהם.

**Key (field):**

ה-Key הינו שדה השייך לWAVLNode. שדה זה מכיל את המפתח ל-node אשר אמור להוות ערך חח"ע בעץ ועל פיו ניתן לבצע את המתודות השונות אותן מממש העץ. ערך ה-Key הינו מסוג int כאשר יוצרים node חדש והוא נקבע בהתאם לקלט הרלוונטי שמוזן מהמשתמש. ניתן לגשת ל-key בסיבוכיות O(1).

**Info (field):**

Info הינו המידע שה-node מכיל בעת יצירתו. גם שדה זה נוצר בהתאם לקלט שמתקבל מהמשתמש, אך בניגוד לשדה ה-key - אין עליו הגבלה שיהיה ערך חח"ע בעת הכנסתו לעץ. שדה זה הינו מסוג String וגם אליו ניתן לגשת בסיבוכיות O(1).

**Rank (field):**

השדה Rank הינו השדה אשר קובע את דרגתו של כל node. שדה זה מאותחל בעת יצירתו להיות 0 (מכיוון שכל איבר שמוכנס לעץ חייב להיות עלה בדרגה 0). כאשר מבצעים פעולות איזון, הוספה ומחיקה של איברים שדה זה מתושאל בכדי לדעת באיזה סיטואציה נמצא העץ וכן דרגתו של כל node מתעדכנת (promote או demote) בהתאם לסיטואציה הרלוונטית. במהלך שהייתו בעץ השדה rank לא יקבל לעולם ערכים שליליים ובדומה לשאר השדות, ניתן לגשת אליו בסיבוכיות O(1).

**Right (field):**

שדה זה הינו מסוג WAVLNode ובעת אתחול ה-node נקבע כי יקבל Null. בעת פעולות האיזון, הכנסה או מחיקה המצביע אשר מחזיק שדה זה יכול להשתנות בהתאם (להצביע ל-node אחר או שיוסר ממנו node). שדה זה מכיל בעצם מצביע לבנו הימני של ה-node בהתאם למודל העצים הבינאריים. סיבוכיות הגישה למצביע זה הינה O(1).

**Left (field):**

בדומה לשדה Right, גם שדה זה הינו מסוג WAVLNode ובעת אתחול ה-node נקבע כי יקבל Null. בעת פעולות האיזון, הכנסה או מחיקה המצביע אשר מחזיק שדה זה יכול להשתנות בהתאם (להצביע ל-node אחר או שיוסר ממנו node). שדה זה מכיל בעצם מצביע לבנו השמאלי של ה-node בהתאם למודל העצים הבינאריים. סיבוכיות הגישה למצביע זה הינה O(1).

**Parent (field):**

גם שדה זה הינו מסוג WAVLNode ובעת אתחול ה-node גם הוא יקבל Null. השדה דומה בפעולתו לשדות Left ו-right ותפקידו לקשר בין node הילד (ה-right או ה-left) לאביהם לצורך ביצוע פעולות שונות וכן קישוריות העץ. גם גישה למצביע זה הינה O(1).