**תיעוד הפרויקט**

מגישים:

רואי עזר

דביר סלומון

**Enum Node Direction**

Enum ציבורי. מקבל ערכים Right ו-Left.

נועד להגדיר את הכיוון של צומת ביחס לאבא שלו.

**Class WAVLTree**

שדות:

**Private WAVLNode externalNode** – מופע של צומת חיצוני של העץ. כל בן שהוא צומת חיצוני של צומת מפנה אליו.

**Private WAVLNode root** – מצביע לשורש של העץ.

**Private WAVLNode min** – מצביע לצומת עם המפתח המינימלי בעץ.

**Private WAVLNode max** – מצביע לצומת עם המפתח המקסימלי בעץ.

מתודות:

**Public WAVLTree()** – בנאי ריק של המחלקה. מאתחל עץ ריק כך שהשורש מצביע לצומת חיצוני. סיבוכיות O(1).

**Public WAVLTree(int key,String info)** – בנאי שמקבל key, info ומאתחל עץ עם שורש שמכיל key, info. סיבוכיות O(1).

**public WAVLNode createWavlNode(int k,String i)** – מחזירה WAVLNode עם מפתח k וערך i. יצירה של WAVLNode לוקחת O(1) ולכן הסיבוכיות O(1).

**public boolean empty()** – בודקת האם העץ ריק, אם כן מחזירה True, אחרת False. הפונקציה בודקת אם דרגת השורש היא -1 ולכן הסיבוכיות היא O(1).

**public String search(int k)** – הפונקציה מחפשת את האיבר עם מפתח k בעץ ומחזירה את הערך שלו. אם הוא לא קיים היא מחזירה null. הפונקציה מחפשת את המפתח בעזרת private WAVLNode search(int k, WAVLNode x) שהסיבוכיות שלה O(logn) ולכן גם הסיבוכיות של הפונקציה היא O(logn).

**private WAVLNode search(int k, WAVLNode x)** – מבצעת חיפוש בינארי של המפתח k בתת עץ של x. אם המפתח נמצא היא תחזיר את הצומת שלו ואם המפתח לא נמצא היא תחזיר את הצומת האחרון שבו היא נעצרה. במקרה הגרוע ביותר היא תחפש מפתח של עלה/שלא נמצא ותתחיל מהשורש ולכן הסיבוכיות O(logn).

**public int insert(int k, String i)** - הפונקציה יוצרת עצם חדש מסוג Tree-Node עם ערכי המפתח k והמחרוזת i שקיבלה מהקלט, מכניסה את הצומת לעץ שעליו הופעלה הפונקציה במיקום המתאים (לפי חוקי עצי חיפוש בינאריים) ובהנחה שהעץ היה עץ WAVL תקין לפני ההכנסה הפונקציה תבצע פעולות שיאזנו את העץ לאחר ההכנסה כך שגם בסופה העץ יישאר עץ WAVL תקין. בתהליך האיזון של העץ הפונקציה סופרת כמה פעולות איזון (promotion, rotation) שהיא מבצעת ומחזירה את הכמות.

לצורך הניתוח אפריד את הפונקציה לשלושה חלקים:

*חלק 1* – הכנסה: מתחיל בהערה ###insertion start ומסתיים בהערה

###insertion done.

*חלק 2* – איזון ראשוני: מתחיל בהערה ###initial insertion balance start ומסתיים בהערה ###initial insertion balance done

*חלק 3* – איזון: מתחיל בהערה ###balancing the tree start וממשיך עד סוף הפונקציה (return).

*חלק 1:*

1. ניצור צומת חדש עם שדות כמתבקש מהקלט, עם 2 עלים חיצונים כבנים ועם שדה size מתאים לפי הפונקציה updateSubtreeSize. עלות במקרה הגרוע: **O(1).**
2. נבדוק אם העץ ריק, אם כן נגדיר את הצומת החדש בתור שורש, נעדכן את השדות min, max (מצביעים על הצומת המינימלי והמקסימלי) להיות השורש החדש ונחזור מבלי שביצענו פעולות איזון return 0. עלות במקרה הגרוע: **O(1).**
3. אם העץ לא ריק נעדכן את השדות min, max כנדרש לפי המפתח של הצומת שהוכנס ונבצע חיפוש בינארי בעזרת פונקציית העזר SearchForInsertשתחפש צומת עם מפתח זהה לזה של הצומת החדש ואם לא תמצא אחד כזה תחזיר את הצומת שצריך להיות האבא של הצומת החדש. עלות במקרה הגרוע: **O(log(n))** כעלות פונקציית העזר.
4. אם בשלב ג' מצאנו צומת עם מפתח k אז קיים כבר צומת עם מפתח k בעץ והתוכנית תבצע return -1 כמתבקש, אחרת היא תכניס את הצומת החדש כבן הימני/שמאלי (לפי השוואה עם מפתח האבא) של האבא המתאים שמצאנו בשלב ג'. עלות במקרה הגרוע: **O(1).**

סה"כ סיבוכיות חלק 1 היא

פה מסתיים חלק 1 של הפונקציה, סה"כ עלות חלק זה במקרה הגרוע היא O(log(n)).

*חלק 2:*

1. נאתחל מונה למספר פעולות האיזון. **O(1).**
2. נבדוק באיזה "מצב איזון ראשוני" נמצא העץ שלנו לאחר שהכנסנו את הצומת החדש. כאשר המצבים האפשריים הם A או B המייצגים את המצבים A, B המתוארים במצגת WAVL בשקפים 18,20. לצורך הבדיקה אשתמש בפונקציית העזר initialInsertionCase שמקבלת את הצומת החדש שהוכנס ומחזירה מחרוזת שמייצגת את המצב בעזרת בדיקה של הפרש הדרגות בין הצומת החדש לאח שלו. **O(1).**
3. אם מצב האיזון הראשוני הוא B אז אין צורך בפעולות איזון, נבצע return 0 אם המצב הוא A נבצע promote לאבא של הצומת החדש ונעבור לחלק 3 עלות:

סה"כ עלות חלק 2 היא

*חלק 3:*

בחלק זה נקרא לפונקציה balanceTheTreeAfterInsert שתבצע את כל פעולות האיזון הנדרשות כדי שהעץ יהיה תקין. לאחר מכן נעדכן את השדה size של הצומת המוכנס בעזרת updateSubtreeSize בעלות , נוסיף למונה פעולות האיזון שלנו את מספר פעולות האיזון שהחזירה balanceTheTreeAfterInsert ונחזיר את המונה.

סיבוכיות חלק זה היא כסיבוכיות פונקציית העזר balanceTheTreeAfterInsert והיא **O(log(n).**

*סיכום סיבוכיות Insert:*

נסכום את הסיבוכיות של החלקים ש-insert נעזרת בהם ונקבל **שהסיבוכיות של insert היא O(log(n).**

**public int balanceTheTreeAfterInsert(WAVLNode pnode)** - הפונקציה מבצעת את כל פעולות האיזון הדרושות בהנחה שבחלק 2 הגענו למצב איזון ראשוני A. פונקציה זו נעזרת בפונקציית העזר BalancingCase שמחזירה מספר המייצג את מצב העץ כמתואר בשקפים 23,26,27 במצגת WAVL ומטפלת בכל מצב באופן שיש לטפל בו על פי המצגת כלומר:

מצב 1 מטופל על ידי קידום דרגת האבא O(1) והתחלת אותו התהליך (של חלק 3) על האבא (שמתבטא בקריאה רקורסיבית ל-balanceTheTreeAfterInsert על צומת האבא כשתנאי העצירה הוא הגעה לשורש).

מצב 2 מטופל ע"י סיבוב בעזרת הפונקציה upperRotation בעלות O(1).

מצב 3 : מטופל ע"י סיבוב כפול בעזרת קריאה כפולה ל-upperRotation בעלות O(1).

הוספתי לפונקציה גם שני מצבים שלא מהמצגת, מצבים 0,4.

מצב 0 הוא מצב שבו קידום הדרגה הקודם שבצענו הוביל לכך שהפרש הדרגות של הצומת ה"בעייתי" (pnode) עם אח שלו הוא 0 ולכן הצומת כלל לא בעייתי והעץ שלנו הוא עץ WAVL תקין, במקרה זה תהליך האיזון עוצר ונחזיר 0 במונה פעולות האיזון.

מצב 4, הוא מצב שבו קידום הדרגה הקודם שביצענו הוביל צומת שהפרש הדרגות שלו עם בניו הוא 2,2 להפוך להיות 2,1 או 1,2 כלומר העץ תקין, במקרה זה תהליך האיזון עוצר ונחזיר 0 במונה פעולות האיזון.

הפונקציה כמובן סופרת כל פעולת איזון במונה שיתווסף למונה שאותחל בחלק 2 ובסוף כאשר נגיע לפעולה סופנית כמו במצבים 0,2,3,4 או שנגיע לשורש במצב 1 נדע שהעץ מאוזן ונחזיר את מספר פעולות האיזון שהתבצעו.

סיבוכיות כל קריאה ל-balanceTheTreeAfterInsert היא O(1) (סה"כ כמות קבועה של בדיקות וקריאה לפונקציות עזר שסיבוכיותן ) אבל כיוון שזו פונקציה רקורסיבית וייתכן שקראה לעצמה מספר קריאות השווה למספר הצמתים שבין העלה לשורש (במקרה של promotion שנגרר מעלה לשורש) ייתכן שהיא נקראה כמות פעמים השווה לגובה העץ שהוא O(log(n)) כיוון שהעץ היה מאוזן לפני ההכנסה. לכן סה"כ סיבוכיות הפונקציה במקרה הגרוע היא O(log(n)).

**public int BalancingCase(WAVLNode pnode)** – קובע באיזה "מצב בעייתי" אנחנו נמצאים בזמן פעולות האיזון לאחר insert. כאשר רשימת המצבים האפשריים היא 0,1,2,3,4 והמצבים 1,2,3 הם המצבים המתוארים בשקפים 23,26,27 במצגת WAVL.

המצבים 0,4 אינם מהמצגת והם מוגדרים כך:

מצב 0 הוא מצב שבו קידום הדרגה הקודם שבצענו הוביל לכך שהפרש הדרגות של הצומת ה"בעייתי" (pnode) עם אח שלו הוא 0 ולכן הצומת כלל לא בעייתי והעץ שלנו הוא עץ WAVL תקין.

מצב 4, הוא מצב שבו קידום הדרגה הקודם שביצענו הוביל צומת שהפרש הדרגות שלו עם בניו הוא 2,2 להפוך להיות 2,1 או 1,2 כלומר העץ תקין.

הפונקציה קובעת מהו המצב המתאים לפי הדרגות של pnode, של אבא שלו ושל אח שלו ולפי המיקום של pnode כבן ימני או שמאלי.

**public String initialInsertionCase(WAVLNode node)** - בודקת מה הפרש הדרגות בין הצומת node לאח שלו ולפי זה מחליטה איזה intialinsertioncase כאשר המצבים האפשריים הם A או B המייצגים את המצבים A, B המתוארים במצגת WAVL בשקפים 18,20.

הסיבוכיות שלה היא O(1).

**public int upperRotation(WAVLNode x,int ind)** - מבצעת סיבוב יחיד כפי שמתואר בשקף 26 במצגת (סה"כ עדכון של מספר קבוע של מצביעים) תוך עדכון כל השדות הרלוונטיים כמו rank, size בעזרת הפונקציה updateSubtreeSize. הארגומנט ind נותן אינדיקציה האם זה סיבוב ראשון או סיבוב שני בצמד של .double rotation יש צורך בארגומנט זה בגלל שהטיפול ב2 המקרים הוא קצת שונה מבחינת עדכון הדרגות. (עוד פירוט בעניין זה ניתן למצוא בהערות בקוד). הפונקציה תחזיר את מספר פעולות ה-promotion/demotion שהיא ביצעה ותוסיף לסכום 1 כי הסיבוב עצמו גם הוא פעולה שנרצה לספור כפעולת איזון. הפונקציה סה"כ מעדכנת מספר קבוע של מצביעים ודרגות לכן הסיבוכיות שלה היא O(1).

**public void edgeUpdate(WAVLNode parent,WAVLNode child,String childType)** – מעדכנת את השדות הרלוונטיים כך שלאחר החזרה מהפונקציה הצומת parent יהיה האבא של child. היא מחליטה אם child יהיה בן שמאלי או ימני לפי הארגומנט childType. סה"כ שינוי מספר קבוע של מצביעים לכן הסיבוכיות היא O(1).

**public void replaceChild(WAVLNode parent,WAVLNode oldChild,WAVLNode newChild)** –

מעדכנת את newChild להיות הבן הימני/שמאלי (לפי המיקום של oldchild) של parent במקום oldchild. סה"כ מספר קבוע של בדיקות ושינוי מצביעים לכן הסיבוכיות היא O(1).

**public int promote(WAVLNode node)** – מגדילה את הדרגה של node ב-1 ומחזירה 1. הפונקציה משתמשת ב-WAVLNode.Promotion() שהסיבוכיות שלה O(1) ולכן הסיבוכיות שלה היא O(1).

**public int[] nodeType(WAVLNode node)** – הפונקציה מחזירה את הפרשי הדרגות בין הצומת לבנים שלו (הקשתות) ע"י חיסור בין הדרגות. סיבוכיות O(1).

**public WAVLNode SearchForInsert(WAVLNode StartingNode,int k)** – הפונקציה מחפשת את המפתח k בתת עץ של StartingNode. הפונקציה משתמשת ב- private WAVLNode search(int k, WAVLNode x) שהסיבוכיות שלה היא O(logn) ולכן הסיבוכיות שלה היא O(logn).

**private void updateSubtreeSizeToRoot(WAVLNode node)** – הפונקציה מבצעת עדכון של ה-size של כל צומת מ-node עד השורש בצורה רקורסיבית ונעזרת ב-WAVLNode. updateSubtreeSize() שלוקחת O(1), אבל בגלל שיכול להיות ש-node הוא עלה נצטרך לבצע logn קריאות רקורסיביות, לכן הסיבוכיות שלה היא O(logn).

**public int delete(int k)** – מוחקת את האיבר עם מפתח k בעץ. ומחזירה את מספר פעולות האיזון שנדרשו לביצוע המחיקה. אם המפתח לא נמצא מחזירה -1. הפונקציה עושה זאת על ידי בדיקה של מספר מקרים:

אם העץ ריק, מחזירה -1. (סיבוכיות O(1)).

אחרת, מעדכנים את min, max ואם יש צורך מחליפים את הצומת עם ה-successor/predecessor שלו ואז מוחקים אותו ע"י קריאה ל-delete של root. (סיבוכיות O(logn)).

סה"כ סיבוכיות O(logn).

**public String min()** – הפונקציה מחזירה את הערך של הצומת המינימלי ומשתמשת לצורך כך בשדה min של המחלקה. אם העץ ריק היא מחזירה null. הסיבוכיות O(1).

**public String max()** – הפונקציה מחזירה את הערך של הצומת המקסימלי ומשתמשת לצורך כך בשדה max של המחלקה. אם העץ ריק היא מחזירה null. הסיבוכיות O(1).

**public int[] keysToArray()** – הפונקציה מחזירה את המפתחות של העץ לפי הסדר ונעזרת לשם כך בפונקציה getKeysWithOrder שהסיבוכיות שלה היא O(n). אם העץ ריק הפונקציה מחזירה מערך ריק. הסיבוכיות שלה היא O(n).

**private void getKeysWithOrder(WAVLNode x, int[] arr, int[] i)** – הפונקציה שמה את המפתחות של תת העץ שמתחיל ב-x במערך arr לפי הסדר, כאשר i שומר את המקום הפנוי הראשון במערך. הפונקציה עוברת על כל הצמתים בעץ ולכן לוקחת O(n).

**public String[] infoToArray()** – הפונקציה מחזירה את הערכים של כל הצמתים במערך מסודרים לפי המפתחות שלהם. היא נעזרת בפונקציה getInfoWithOrder שהסיבוכיות שלה היא O(n). אם העץ ריק היא מחזירה מערך ריק. הסיבוכיות שלה היא O(n).

**private void getInfoWithOrder(WAVLNode x, String[] arr, int[] i)** – הפונקציה שמה את הערכים של הצמתים בתת העץ של x במערך arr לפי הסדר של המפתחות שלהם, כאשר i שומר את המקום הפנוי הראשון במערך. הפונקציה עוברת על כל הצמתים בעץ ולכן לוקחת O(n).

**public int size()** – הפונקציה מחזירה את גודל העץ, כלומר את root.size. סיבוכיות O(1).

**public WAVLNode getRoot()** – המתודה מחזירה את root. סיבוכיות O(1).

**public String select(int i)** – הפונקציה מחזירה את הערך של האיבר ה-i הכי קטן בעץ, או null אם יש פחות מ-i צמתים בעץ או ש-i קטן מ-1. הפונקציה נעזרת ב-private String select(WAVLNode x, int i) שהסיבוכיות שלה O(logn) ולכן הסיבוכיות של הפונקציה היא O(logn).

**private String select(WAVLNode x, int i)** – הפונקציה מחזירה את הערך של האיבר ה-i הכי קטן בעץ. הפונקציה מבצעת בדיקה של גודל התת עץ השמאלי של x ולפי זה בוחרת האם להמשיך בתת עץ השמאלי או הימני. במקרה הגרוע ביותר נתחיל בשורש ונגיע לאחד העלים ולכן הסיבוכיות היא O(logn).

**public boolean isLeaf(WAVLNode node)** – הפונקציה בודקת האם node הוא עלה. היא בודקת אם שני הבנים שלו הם צומת חיצוני ולכן הסיבוכיות היא O(1).

**private void setRoot(WAVLNode newRoot)** – המתודה מגדירה את הצומת newRoot כשורש של העץ.

**Class WAVLNode**

שדות:

**Private WAVLNode left** – מצביע לבן השמאלי של הצומת.

**Private WAVLNode parent** – מצביע לאבא של הצומת.

**Private WAVLNode right** – מצביע לבן הימני של הצומת.

**Private int rank** – דרגת הצומת.

**Private int key** – מפתח הצומת.

**Private String info** – הערך של הצומת.

**Private int size** – גודל תת העץ של הצומת.

מתודות:

**Public WAVLNode(int key, String info)** – בנאי שמקבל key, info ומאתחל את השדות right, left, parent ל-null, את rank ל-(-1) ואת size ל-0. סיבוכיות O(1).

**public int getKey()** – מחזירה את key. סיבוכיות O(1).

**public String getValue()** – מחזירה את info. סיבוכיות O(1).

**public int getRank()** – מחזירה את rank. סיבוכיות O(1).

**public WAVLNode getLeft()** – מחזירה את left. סיבוכיות O(1).

**public WAVLNode getRight()** – מחזירה את right. סיבוכיות O(1).

**public void promotion()** – מעלה את rank ב-1. סיבוכיות O(1).

**public void demotion()** – מורידה את rank ב-1. סיבוכיות O(1).

**public boolean isInnerNode()** – בודקת האם הצומת הוא צומת פנימי ע"י בדיקה האם הדרגה שלו גדולה מ-(-1). סיבוכיות O(1).

**public boolean isExternalNode()** – בודקת האם הצומת הוא צומת חיצוני ע"י החזרת הערך ההפוך מ-isInnerNode(). סיבוכיות O(1).

**private boolean isRoot()** – בודקת האם הצומת הוא השורש של העץ ע"י בדיקה האם this==getRoot(). סיבוכיות O(1).

**private boolean isLeaf()** – בודקת האם הצומת הוא עלה של העץ (הכוונה לצומת ששני הבנים שלו הם עלים חיצוניים) ע"י בדיקה האם שני הבנים שלו הם עלים חיצוניים. סיבוכיות O(1).

**public int getSubtreeSize()** – מחזירה את size. סיבוכיות O(1).

**public void updateSubtreeSize()** – מעדכנת את גודל תת העץ של הצומת (size). אם הצומת עלה חיצוני אז הגודל 0. אחרת היא סוכמת את size של שני הבנים שלו ומוסיפה 1. סיבוכיות O(1).

**public void setSubtreeSize(int newSize)** – מעדכנת את גודל תת העץ של הצומת ל-newSize. סיבוכיות O(1).

**public void increaseSubtreeSize()** – מוסיפה 1 לגודל תת העץ של הצומת ע"י הגדלה של size ב-1. סיבוכיות O(1).

**public void decreaseSubtreeSize()** – מחסירה 1 מגודל תת העץ של הצומת ע"י החסרה של 1 מ-size. סיבוכיות O(1).

**private WAVLNode getMinNode()** – מחזירה את האיבר המינימלי של העץ ע"י החזרה של min. סיבוכיות O(1).

**private WAVLNode getMaxNode()** – מחזירה את האיבר המקסימלי של העץ ע"י החזרה של max. סיבוכיות O(1).

**private WAVLNode getParent()** – מחזירה את האבא של הצומת ע"י החזרה של parent. סיבוכיות O(1).

**private void setParent(WAVLNode node)** – מעדכנת את parent ל-node אם הצומת הנוכחי לא צומת חיצוני. סיבוכיות O(1).

**private WAVLNode getChild(NodeDirection direction)** – מחזירה את הבן השמאלי/ימני של הצומת בהתאם ל-direction. סיבוכיות O(1).

**private void setChild(NodeDirection direction, WAVLNode newChild)** – מעדכנת את הבן שמאלי/ימני של הצומת ל-newChild בהתאם ל-direction ואת ה-parent של newChild ל-this. סיבוכיות O(1).

**private WAVLNode successor()** – מחזירה את ה-successor של הצומת הנוכחי ע"י קריאה ל-goToNode(NodeDirection.Right). סיבוכיות O(logn).

**private WAVLNode predecessor()** – מחזירה את ה-predeccessor של הצומת הנוכחי ע"י קריאה ל-goToNode(NodeDirection.Left). סיבוכיות O(logn).

**private WAVLNode goToNode(NodeDirection direction)** – מחזירה את ה-successor/predecessor של הצומת (בהתאם ל-direction). במקרה הגרוע ביותר של successor, הצומת הנוכחי הוא המקסימום של תת העץ השמאלי של השורש ולכן ה-successor שלו הוא השורש ונצטרך לעלות עד אליו וזה O(logn). במקרה הגרוע ביותר של predeccessor הצומת הנוכחי הוא המינימום של תת העץ הימני של השורש ולכן ה-predeccessor שלו הוא השורש ונצטרך לעלות עד אליו וזה O(logn). סה"כ הסיבוכיות היא O(logn).

**private void switchNode(WAVLNode other)** – מחליפה את הצומת הנוכחי עם צומת other שהוא אחד הבנים של הצומת הנוכחי (נשתמש בה אך ורק להחלפת הצומת עם ה-successor שלו לפני מחיקה אם צריך). הפונקציה משנה את המצביעים של הצומת הנוכחי ושל other ונעזרת בפונקציות getParent(), getRank(), isRoot(), getParentDirection(), getRight(), getLeft(), setChild, setParent, setRoot, decreaseSubtreeSize() שהסיבוכיות של כולן היא O(1). אחרי ההחלפה היא מעדכנת את גודל תת העץ של כל אחד מהצמתים במסלול שמקשר ביניהם ובמקרה הכי גרוע היא תעלה עד לשורש. לכן הסיבוכיות O(logn).

**private int delete(int key)** – הפונקציה מוחקת את הצומת עם מפתח key ומחזירה את מספר פעולות האיזון שנדרשו. אם המפתח לא נמצא, מחזירה -1. כדי למצוא את הצומת שצריך למחוק, הפונקציה מבצעת חיפוש בינארי ברקורסיה ומורידה 1 מה-size של כל צומת. אם המפתח לא נמצא והגענו לצומת חיצוני אז הפונקציה עולה בחזרה לשורש ומעלה בחזרה את הדרגה של כל צומת ב-1. אם המפתח נמצא הפונקציה מחלקת לפי מצבים איך למחוק את הצומת:

1. אם הצומת עלה:
2. אם הוא גם השורש, כלומר יש איבר אחד בעץ, אז נגדיר את השורש להיות צומת חיצוני ונחזיר 0.
3. אחרת, נחליף את הצומת עם צומת חיצוני ונקרא ל- deletionBalance() של האבא של הצומת שמחקנו.
4. אם הוא צומת אונרי:
5. אם הוא גם השורש, כלומר יש 2 איברים בעץ, נגדיר את הבן שלו בתור הצומת ונמחק אותו ונחזיר 0.
6. אחרת, בהנחה ש-p האבא של הצומת ו-c הוא הבן שלו, נגדיר את c להיות הבן של p ונקרא ל- deletionBalance() של p.
7. אם יש לצומת 2 בנים: נחליף אותו עם ה-successor, נמחק אותו ואז נקרא ל- deletionBalance() של האבא של הצומת האמיתי שמחקנו.

החיפוש של המפתח, הקריאה ל- deletionBalance() והמציאה והחלפה עם ה-successor לוקחות O(logn). שאר הפעולות שהן החלפות מצביעים, בדיקות של שורש או עלה חיצוני לוקחות O(1). לכן בסה"כ הסיבוכיות של הפונקציה היא O(logn).

**private int deletionBalance()** – הפונקציה מחזירה את מספר פעולות האיזון שנדרשו לאחר מחיקה מהעץ. הפונקציה בודקת לפי מקרים (עד כדי סימטריה) מהם פעולות האיזון שנדרשות:

1. אם הקשתות של הצומת הן 2,1/1,1 או 2,2 והצומת לא עלה: העץ חוקי ולא נדרשות פעולות איזון ולכן יוחזר 0.
2. אם הצומת עלה והקשתות שלו 2,2 אז נקרא ל-demotion() ונבדוק האם לאבא של הצומת נדרשות עוד פעולות איזון עד שנגיע לשורש, במצב כזה נשמור 1 ונבצע קריאה רקורסיבית.
3. אחרת נבדוק מה מקרה מתוך ארבעת המקרים האפשריים:
4. אם הקשתות הן 3,2 נבצע demotion() ונבדוק האם לאבא של הצומת נדרשות עוד פעולות איזון עד שנגיע לשורש, במצב כזה נשמור 1 ונבצע קריאה רקורסיבית.
5. אם הקשתות הן 3,1 נבדוק את הקשתות של הבן של הקשת 1:

* אם הקשתות שלו הן 2,2 נוריד את הדרגה של הבן ושל האבא נשמור 2 ונבצע קריאה רקורסיבית כדי לבדוק אם נדרשות עוד פעולות איזון.
* אם הקשת הימנית היא 1, נבצע פעולת rotate אחת לכיוון השני. אם כעת האבא הפך להיות עלה עם קשתות 2,2 נוריד לו את הדרגה. לבסוף נחזיר 3.
* אם הקשתות שלו הן 1,2 נבצע 2 פעולות rotate ונחזיר 5.

בגלל שהפונקציה במקרה הגרוע ביותר תצטרך לבצע קריאות רקורסיביות עד השורש ושאר הפעולות שהיא מבצעת לוקחות O(1) הסיבוכיות שלה היא O(logn).

**private NodeDirection getParentDirection()** – מחזירה את הכיוון של הצומת הנוכחי ביחס לאבא. הפונקציה משתמשת בפעולות getParent(), getChild שלוקחות O(1). לכן הסיבוכיות O(1).

**private NodeDirection getOppositeDirection(NodeDirection direction)** – מחזירה את הכיוון ההפוך ל-direction. סיבוכיות O(1).

**public int getRankDifference(NodeDirection direction)** – מחזירה את הפרש הדרגות (דרגת הקשת) בין צומת לבן שלו בכיוון direction. סיבוכיות O(1).

**private void rotate(NodeDirection direction)** – מבצעת פעולת סיבוב לצומת סביב קשת בכיוון direction. הפונקציה מבצעת שינוי של המצביעים (שזה O(1)) ולכן הסיבוכיות היא O(1).

**מדידות**

לאחר ביצוע המדידות לפי ההוראות קיבלנו את הטבלה:



על סמך ההסבר התיאורטי של עצי WAVL מתקיים:

לכן היינו מצפים שמספרי פעולות האיזון המקסימליים של insert/delete יהיו חסומים ע"י מספר קבוע כפול (כאשר n הוא מספר הצמתים בעץ) ושמספרי פעולות האיזון הממוצע עבור סדרת פעולות insert/delete יהיה חסום ע"י קבוע.

התוצאות אכן תואמות את הציפיות שלנו, קל לראות שמספר פעולות האיזון המקסימליים של insert/delete אכן חסומות למשל ע"י 50 כאשר

כאשר

וגם מספרי פעולות האיזון הממוצע עבור סדרה של פעולות insert/delete חסום ע"י מספר קבוע, למשל 5.

משמעות המדידות שביצענו היא אישוש התיאוריה וההוכחות בכיתה על החסמים על מספר פעולות האיזון בעצי WAVL.