1. **<u>תיעוד המחלקות:</u>**

**<u>מחלקה AVLTree</u>**

**שדות המחלקה:**

כלל השדות הם מטיפוס IAVLNode ובנראות private:

1. **root** – מצביע לצומת שהוא שורש העץ.
2. **max** – מצביע לצומת עם המפתח המקסימלי.
3. **min** – מצביע לצומת עם המפתח המינימלי.

**פירוט המתודות:**

1. **public AVLTree()**
בנאי – יוצר עץ ריק, כלומר יוצר עלה חיצוני דרך הבנאי המתאים במחלקה AVLNode, אשר מתבצע ב(O(1, אליו מצביע השורש. סה"כ (O(1.
2. **public boolean empty()**
בודקת אם העץ ריק, כלומר אם ערך הצומת שבשורש הוא null תחזיר true , אחרת false.
בדיקה שמתבצעת עבור מספר קבוע של מצביעים והשוואת כתובת מחרוזת לnull – (O(1.
3. **public String search(int k,int addToSize)**
קוראת ל- searchNode אשר מחזירה את הצומת עם המפתח k ואם לא קיים כזה תחזיר null. זמן הריצה של searchNode הוא ((O(log(n וכן שאר הפעולות במתודה ב-(O(1, לכן סה"כ ((O(log(n.
4. **private IAVLNode searchNode(int k,int addToSize)**
תחזיר את הצומת שהמפתח שלו הוא k, ואם הוא לא קיים בעץ אז תחזיר את הצומת שאחריו צומת חדש עם המפתח k צריך להיות ממוקם בעץ. addToSize = 1 אם הקריאה למתודה מתוך insert. אם המשתמש קורא לפונקציה מתוך delete, addToSize = 1-.
אם הוא לא מוחק ולא מכניס איבר חדש, אז addToSize = 0.
חיפוש זה הוא לאורך הגובה של העץ החל מהשורש, וייתכן שהצומת עם מפתח k יימצא לפני ההגעה לעלה – והגובה הוא ((O(log(n, לכן סה"כ ((O(log(n.
5. **private void rightRotation(IAVLNode x, IAVLNode y)**
מבצע רוטציה בין X לY כפי שנלמד בכיתה. משנים מספר קבוע של מצביעים ולכן פעולה זו מתבצעת ב-(O(1.
6. **private void leftRotation(IAVLNode y, IAVLNode x)**
מבצע רוטציה בין Y ל-X כפי שנלמד בכיתה. משנים מספר קבוע של מצביעים ולכן פעולה זו מתבצעת ב-(O(1.
7. **private void promote(IAVLNode node)**
מקדם ב-1 את השדה rank של node, הגישה לשדה והקידום שניהם ב-(O(1.
8. **private void demote(IAVLNode node)**
מוריד ב-1 את השדה rank של node, הגישה לשדה והקידום שניהם ב-(O(1.
9. **private int rebalanceInsertion(IAVLNode node)**
בהינתן צומת בודק האם יש לאזן את תת העץ שלה לפי המצבים שלמדנו בכיתה.
בפונקציה יש לולאה אשר רצה מהעלה החדש ובודקת בכל פעם אם העץ מאוזן. אם כן, הלולאה תפסיק. אחרת, בודקת את המקרים הבאים-
**<u>מצב א':</u>** <u>0,1 או 1,0</u> מבצע רק promote, לא מסתיים באיזון סופי, הלולאה ממשיכה לרוץ.
**<u>מצב ב':</u>** <u>0,2 – אחריה העץ מאוזן, מתחלק למקרים:</u>
   - אם 1,2 אז קוראים ל-case02And12Rebalance(IAVLNode node) אשר מבצעת מספר קבוע של פעולות לשינוי מצביעים,פועלת ב-(O(1.
   - אם 2,1 אז קוראים ל-case02And21Rebalance(IAVLNode node) אשר מבצעת מספר קבוע של פעולות לשינוי מצביעים,פועלת ב-(O(1.

**מצב ג':** 2,0 אחריה העץ מאוזן, מתחלק למקרים:

- אם 1,2 אז קוראים ל-case20And12Rebalance(IAVLNode node) אשר מבצעת מספר קבוע של פעולות לשינוי מצביעים,פועלת ב-O(1).
- אם 2,1 אז קוראים ל-case20And21Rebalance(IAVLNode node) אשר אשר מבצעת מספר קבוע של פעולות לשינוי מצביעים,פועלת ב-O(1).

במקרה הגרוע יבוצע promote לאורך כל הגובה מהעלה עד השורש וזה בסיבוכיות O(log(n)), כאשר כל שאר הפעולות הן ב-O(1).

**10. private void case02And11Rebalance(IAVLNode node)**
טיפול בcase1 – קוראים לפונקציה rightRotation אשר תבצע גלגול אחד ימינה- פעולה ב-O(1) וpromote אחד, סה"כ O(1).

**11. private void case20And11Rebalance(IAVLNode node)**
טיפול בcase1 – קוראים לפונקציה leftRotationאשר תבצע גלגול אחד שמאלה- פעולה ב-O(1) וpromote אחד, סה"כ O(1).

**12. private void case02And12Rebalance(IAVLNode node)**
טיפול בcase2 – קוראים לפונקציה rightRotation אשר תבצע גלגול אחד ימינה- פעולה ב-O(1) וdemote אחד, סה"כ O(1).

**13. private void case02And21Rebalance(IAVLNode node)**
טיפול בcase3 – קוראים לleftRotation ואז לrightRotation שזה גלגול כפול, כל אחת מהפעולות הן ב-O(1), וכן שני demote וpromote אחד בהתאמה. סה"כ O(1).

**14. private void case20And21Rebalance(IAVLNode node)**
טיפול בcase3 – קוראים לrightRotation ואז ל leftRotation שזה גלגול כפול, כל אחת מהפעולות הן ב-O(1), וכן שני demote וpromote אחד בהתאמה. סה"כ O(1).

**15. private void case20And12Rebalance(IAVLNode node)**
16. טיפול בcase2 – קוראים לפונקציה leftRotation אשר תבצע גלגול אחד שמאלה- פעולה ב-O(1) וdemote אחד, סה"כ O(1).

**17. public int insert(int k, String i)**
מאתחלים משתנה numOfRebalanceOpp שיספור את פעולות האיזון. לאחר מכן יוצרים צומת חדש עם המפתח והערך הנתונים. מחפשים את המיקום שבו הצומת החדש צריך להיכנס באמצעות קריאה ל -searchNode, אם קיים צומת עם מפתח זהה הפונקציה תחזיר 1- ותעצור. לאחר החיפוש מכניסים את הצומת למקום המתאים, וכן מבצעים rebalance לעץ באמצעות קריאה ל- rebalanceInsertion, אשר תחזיר את מספר הפעולות שבוצעו וערך וזה הערך שהפונק' תחזיר בסוף.
סיבוכיות – החיפוש ופעולת האיזון של העץ לוקחות O(log(n)) , שאר הפעולות הן בזמן קבוע ולכן סיבוכיות הזמן היא O(log(n)) .

**18. private IAVLNode createLeaf(int k, String val)**
יוצרת צומת- עלה אמיתי עם הבנאי AVLNode(k,val), וכן שני עלים חיצוניים, עם הבנאי AVLNode(), ומחברת בין הצומת לעלים החיצוניים בעזרת המתודות setLeft, setParent ו setRight. סה"כ כל מתודה היא בO(1) סיבוכיות, לכן סה"כ – O(1).

**19. public int delete(int k)**
תחילה מאתחלים משתנה numOfRebalance שיספור את מספר פעולות האיזון ל 0.
בודקים אם העץ ריק – אם כן מחזירים 1-. אם לא קוראים לפונקציה search(k) שתחפש אם הצומת קיים בעץ. אם הוא קיים קוראים לsearchNode(k,-1) , אשר תחזיר את הצומת שצריך למחוק ותקטין ב 1 את size של הצמתים שעוברים בהם בדרך. לאחר מכן בודקים אם הצומת הוא עלה באמצעות קריאה ל isLeaf(deleteNode) , אם כן מוחקים את הצומת באמצעות deleteLeaf(deleteNode) , מבצעים איזון לעץ באמצעות קריאה למתודה rebalanceDeletion ושומרים את מספר פעולות האיזון ב numOfRebalance .

אם הצומת הוא לא עלה, בודקים אם הוא צומת אונארי באמצעות קריאה למתודה isUnary ואז נקבל שני מקרים:

א. אם צומת אונארי - נמחק אותו באמצעות המתודה deleteUnary , ונאזן את העץ ונשמור את מספר פעולות האיזון שזה הערך שחוזר מ-rebalanceDeletion במשתנה numOfRebalance.

ב. אם הצומת הוא לא עלה ולא אונארי , נמצא את העוקב שלו באמצעות המתודה successor אשר תמצא את העוקב ותקטין את ה size של הצמתים שהיא עוברת בהם בדרך ב 1 , נחליף את הצומת עם העוקב שלו באמצעות המתודה switchSuccessor , נעדכן את ה size של העוקב שהחלפנו ונמחק את הצומת , אם הוא עלה באמצעות deleteNode ואם אונארי באמצעות deleteUnary.

נבצע איזון לעץ באמצעות rebalanceDeletion , נעדכן את הצומת המינימלי והמקסימלי בעץ באמצעות קריאה ל ()findMin , ()findMax ולבסוף נחזיר את numOfRebalance. אם הצומת לא קיים בעץ , numOfRebalance לא ישתנה ויחזיר 0.

סיבוכיות החיפוש היא O(log(n)) שכן לכל היותר עוברים על מסלול אחד בעץ ובכל צומת מבצעים O(1) פעולות. הבדיקה אם צומת הוא עלה או אונארי ומחיקה של צומת שהוא עלה או אונארי היא O(1) – שינוי מספר קבוע של מצביעים, ביצוע פעולות האיזון של העץ חסום על ידי גובה העץ ולכן גם O(log(n)) , ולבסוף מספר הפעולות בעדכון הצומת המינימלי והמקסימלי גם כן חסום על ידי גובה העץ ולכן גם כן O(log(n)) , ולכן בסה"כ הסיבוכיות היא O(log(n)) .

20. **switchSuccessorNotRightSon(IAVLNode node, IAVLNode successor)**

הפונקציה מקבלת צומת ואת העוקב שלו. תנאי קדם:העוקב של הצומת הוא לא הבן הימני של הצומת ומחליפה ביניהם. הפונקציה מעדכנת את הבן הימני, השמאלי והאבא של כל אחד מהצמתים ולכן הסיבוכיות היא O(1) .

21. **switchSuccessorRightSon(IAVLNode node, IAVLNode successor)**

הפונקציה מקבלת צומת ואת העוקב שלו. תנאי קדם:העוקב הוא הבן הימני של הצומת ומחליפה ביניהם באמצעות עדכן המצביעים לבן הימני, שמאלי והאבא של כל אחד מהם , כלומר מבצעים מספר קבוע של פעולות ולכן הסיבוכיות היא O(1) .

22. **switchSuccessor(IAVLNode node, IAVLNode successor)**

הפונקציה מקבלת צומת ואת העוקב שלו, בודקת אם העוקב הוא בן ימני של הצומת, אם כן- מחליפה ביניהם באמצעות קריאה ל switchSuccessorRightSon ואם לא באמצעות קריאה ל switchSuccessorNotRightSon , כאשר הסיבוכיות של כל אחת מהן היא O(1) ולכן בסה"כ הסיבוכיות היא O(1) .

23. **isLeaf(IAVLNode node)**

מחזירה true אם צומת הוא עלה אחרת false באמצעות בדיקה של הבן הימני והשמאלי של הצומת – סיבוכיות O(1) .

24. **isUnary(IAVLNode node)**

מחזירה true אם צומת הוא אונארי אחרת false, באמצעות בדיקה של האם צומת הוא לא עלה באמצעות קריאה ל isLeaf ואם חוזר false בדיקה אם אחד מהבנים הוא עלה חיצוני. סיבוכיות isLeaf היא O(1) ולכן בסה"כ O(1) .

25. **deleteLeaf(IAVLNode node)**

מחיקה של צומת שהוא עלה. תחילה בודקים אם הצומת הוא שורש העץ, אם כן מגדירים ששורש העץ יהיה עלה חיצוני חדש. אם לא , בודקים אם הצומת הוא בן ימני או שמאלי של אבא שלו ובהתאם מעדכנים את המצביעים של האבא ושל הבנים של הצומת (העלים החיצוניים). משנים מספר קבוע של מצביעים ולכן הסיבוכיות היא O(1) .

26. **deleteUnary(IAVLNode node)**

מחיקה של צומת אונארי. תחילה בודקים אם הבן שהוא לא עלה חיצוני הוא הבן הימני או השמאלי. נשמור מצביע לבן במשתנה newSon . אם הצומת היה שורש נגדיר את newSon

להיות השורש, אם לא נבדוק אם newSon הוא בן ימני או שמאלי של הצומת ובהתאם נעדכן את המצביעים ונמחק את הצומת. סיבוכיות O(1) .

**27. successor(IAVLNode node, int increaseSize)**

הפונקציה מקבלת צומת ומספר increaseSize ומחזירה את העוקב של הצומת ומשנה את ה size של הצמתים שעוברים בהם בדרך ב increaseSize . תחילה בודקים את לצומת יש בן ימני – אם בלולאה עוברים על הבן השמאלי של כל צומת בתת העץ הימני כדי לקבל את הצומת המינימלי בתת העץ הימני, אם אין בן ימני עוברים בלולאה מהצומת לאבא שלו עד שהצומת הוא בן שמאלי של האבא ומחזירים את האבא. לכל היותר עוברים על מסלול אחד בעץ ובכל צומת מבצעים מספר קבוע של פעולות ולכן הסיבוכיות היא O(logn) .

**28. case31And11Rebalance(IAVLNode node)**

ביצוע פעולות איזון כאשר צומת הוא מסוג 3-1 והבן הימני שלו הוא 1-1 - מבצעים איזון כפי שראינו בכיתה כלומר גלגול שמאלה באמצעות קריאה ל leftRotation , וביצוע demote ו promote בהתאמה – סיבוכיות O(1) .

**29. case13And11Rebalance(IAVLNode node)**

ביצוע פעולות איזון כאשר צומת הוא מסוג 1-3 והבן השמאלי שלו הוא 1-1 כפי שנלמד בכיתה קריאה ל rightRotation , וביצוע demote ו promote בהתאמה – סיבוכיות O(1) .

**30. case31And21Rebalance(IAVLNode node)**

ביצוע פעולות איזון כאשר צומת הוא מסוג 3-1 והבן הימני הוא 2-1 כפי שנלמד בכיתה – קריאה ל leftRotation לבצע גלגול שמאלה ו 2 פעולות demote , סיבוכיות O(1) .

**31. case13And12Rebalance(IAVLNode node)**

ביצוע פעולות איזון לצומת מסוג 1-3 והבן השמאלי שלו הוא מסוג 2-1. ביצוע rightRotation , ו2 פעולות demote . סיבוכיות O(1) .

**32. case31And12Rebalance(IAVLNode node)**

ביצוע פעולות איזון כאשר צומת הוא מסוג 3-1 והבן הימני הוא מסוג 2-1 – מבצעים גלגול כפול באמצעות קריאה ל rightRotation ול leftRotation ו מבצעים 4 פעולות demote בהתאם. סיבוכיות O(1) .

**33. case13And21Rebalance(IAVLNode node)**

ביצוע פעולות איזון כאשר צומת הוא מסוג 1-3 והבן הימני הוא 2-1. מבצעים גלגול כפול באמצעות קריאה ל leftRotation ול rightRotation ו מבצעים 4 פעולות demote בהתאם. סיבוכיות O(1) .

**34. rebalanceDeletion(IAVLNode node)**

ביצוע איזון לעץ לאחר מחיקה בהתאם למקרים השונים והחזרת מספר פעולות האיזון שהתבצע:

אם הצומת הוא null – היה בעץ צומת אחד ומחקנו אותו – הפונקציה תחזיר 0. כעת נאתחל cnt שיספור את מספר פעולות האיזון. נתחיל בלולאה שתתחיל בצומת שהפונקציה מקבלת כקלט, נחלק למקרים :

א. העץ מאוזן ולכן נצא מהלולאה.
ב. הצומת הוא מסוג 2-2 ולכן נבצע demote לצומת ונעלה את cnt ב 1.
ג. הצומת הוא 3-1 והבן הימני הוא 1-1 - נבצע איזון האמצעות קריאה ל case31And11Rebalance ונעלה את cnt ב 3.
ד. הצומת הוא 3-1 והבן הימני הוא 2-1 – נבצע איזון באמצעות קריאה ל case31And12Rebalance ונעלה את cnt ב 6.
ה. הצומת הוא 3-1 והבן הימני הוא 1-2 – נבצע איזון באמצעות קריאה ל case31And21Rebalance ונעלה את cnt ב 3.
ו. הצומת הוא 1-3 והבן השמאלי הוא 1-1 – נאזן באמצעות קריאה ל case13And11Rebalance ונעלה את cnt ב 3.
ז. הצומת הוא 1-3 והבן השמאלי הוא 1-2 – נאזן באמצעות קריאה ל case13And21Rebalance ונעלה את cnt ב 6.

ח. הצומת הוא 3-1 והבן השמאלי הוא 2-1 נאזן באמצעות קריאה ל case13And12Rebalance ונעלה את cnt ב 3. בסוף נחזיר את cnt .

סיבוכיות – בכל שלב עושים O(1) ולכל היותר עוברים במסלול מעלה לשורש ולכן הסיבוכיות היא O(log(n)) .

**35. public String min()**
מחזיר את הערך בשדה min של המחלקה אם קיים מינימום. אחרת מחזיר null. סה"כ O(1).

**36. public String max()**
מחזיר את הערך בשדה max של המחלקה אם קיים מינימום. אחרת מחזיר null. סה"כ O(1).

**37. private IAVLNode findMin()**
אם העץ לא ריק, מחפש את המינימום לאורך החלק השמאלי של העץ. סה"כ הולך על גובה העץ לכן O(log(n)).

**38. private IAVLNode findMin()**
אם העץ לא ריק, מחפש את המקסימום לאורך החלק הימני של העץ. סה"כ הולך על גובה העץ לכן O(log(n)).

**39. public int[] keysToArray()**
בהינתן עץ מחזירה מערך ממוין מהקטן לגדול עם המפתחות בעץ, בעזרת המתודה הרקורסיבית keysToArrayRec שרצה ב-O(n). סה"כ -O(n).

**40. private void keysToArrayRec(IAVLNode node, int[] keysArray, int[] index)**
מבצעת סריקת inorder על איברי העץ ומחזירה את מערך המפתחות הממוין, כפי שלמדנו מתבצע ב-O(n).

**41. public String[] infoToArray**
בהינתן עץ מחזירה מערך ממוין מהקטן לגדול עם המחרוזות בעץ, בעזרת המתודה הרקורסיבית infoToArrayRec שרצה ב-O(n). סה"כ -O(n).

**42. private void infoToArrayRec(IAVLNode node, String[] infoArray, int[] index)**
מבצעת סריקת inorder על איברי העץ ומחזירה את מערך הערכים הממוין, כפי שלמדנו מתבצע ב-O(n).

**43. public int size()**
קוראת למתודה getSize אשר מחזירה את הערך בשדה size, שזה גודל העץ. גישה לשדה זה ב-O(1).

**44. public IAVLNode getRoot()**
מחזירה את הצומת שהיא השורש בעץ, אם העץ לא ריק. אחרת תחזיר null. גישה לשורש הנה ב-O(1).

**45. split(int x)**
תחילה יוצרים מערך ריק של 2 איברים מטיפוס AVLTree . לאחר מכן מחפשים את הצומת שעליו מבצעים את ה split באמצעות קריאה ל search ושומרים מצביע לצומת זה במשתנה node . לאחר מכן מאתחלים שני עצים , אחד smaller , אשר יכיל את הצמתים עם המפתחות הקטנים מהצומת ואחד bigger אשר יכיל את צמתים עם המפתחות הגדולים מהצומת.
לאחר מכן מתחילים בלולאה מהצומת ועד לשורש, לכל צומת, אם הוא בן ימני של האבא שלו – מבצעים joinSetMinMax עם false (לא מעדכנים את המינימום והמקסימום של בעץ אלא רק בסוף הספליט) לsmaller עם האבא ותת העץ השמאלי שלו.
אם הצומת הוא בן שמאלי של האבא – מבצעים joinSetMinMax ל bigger עם האבא ותת העץ הימני שלו. בסיום התהליך לכל אחד מן העצים המתקבלים מחפשים את המינימום והמקסימום בעזרת קריאה לפונקציות findMin , findMax .

מעדכנים את המערך במקום ה 0 להיות smaller , המערך במקום ה 1 להיות bigger .
סיבוכיות – ראינו בהרצאה כי מתקבל בניתוח הסיבוכיות טור טלסקופי מהפרשי הגבהים של העצים שעליהם מבצעים את ה join ועוד מספר פעולות ה join שביצענו, לכן מתהליך ה split הסיבוכיות היא O(log(n)) הפרש הגבהים בין העץ הראשון בתהליך לאחרון הוא O(log(n)) ולכל היותר מספא פעולות ה join הוא כגובה העץ ולכן O(log(n)) , בנוסף בסוף התהליך מציאת המינימום והמקסימום הם בסיבוכיות של O(log(n)) , ולכן בסה"כ הסיבוכיות היא O(log(n)) .

**46. Public int join(IAVLNode x, AVLTree t)**

קוראת לפונק' joinSetMinMax . סה"כ O(log(n)).

**47. public int joinSetMinMax(IAVLNode x, AVLTree t,Boolean updateMinMax)**

ראשית, אם שני העצים ריקים – מגדירה את x להיות הצומת היחיד בעץ וכן השורש, ואז תחזיר 1 שזה הפרש הגבהים ועוד 1.
אם לא שניהם ריקים, נחלק למקרים:

א. העץ this צריך להיות משמאל והוא גם גדול יותר מt – נקרא למתודה joinBiggerTreeIsLeft אשר תחבר את העצים ואחריה this יהיה חיבור העצים וכן יהיה מאוזן. לאחר מכן נתאחל לעץ החדש את המינימום ואת המקסימום בעזרת findMin ו findMax, ואז יוחזר הפרש הגבהים של העץ this והעץ t ועוד 1.

ב. העץ this צריך להיות משמאל והוא שווה בגובהו לגובה של t- נקרא למתודה joinSameHeights אשר תחבר את this משאל לx ואת t מימינו. מקבלים עץ מאוזן. נחזיר 1 שזה בדיוק הפרש הגבהים ועוד 1.

ג. העץ this צריך להיות משמאל והוא גם קטן יותר מt – נקרא למתודה joinBiggerTreeIsRight אשר תחבר את העצים ואחריה this יהיה חיבור העצים וכן יהיה מאוזן. לאחר מכן נתאחל לעץ החדש את המינימום ואת המקסימום בעזרת findMin וfindMax , ואז יוחזר הפרש הגבהים של העץ this והעץ t ועוד 1.

ד. העץ this צריך להיות מימין והוא גם גדול יותר מt – נקרא לjoinBiggerTreeIsRight , אשר תחבר את העצים ואחריה this יהיה חיבור העצים וכן יהיה מאוזן. לאחר מכן נתאחל לעץ החדש את המינימום ואת המקסימום בעזרת findMin וfindMax , ואז יוחזר הפרש הגבהים של העץ this והעץ t ועוד 1.

ה. העץ this צריך להיות מימין והוא שווה בגובהו לגובה של t- נפעל כמו ב-ב' רק שהפעם נשלח את t ראשון בפונק'.

ו. העץ this צריך להיות מימין והוא קטן יותר -נקרא ל joinBiggerTreeIsLeft אשר תחבר את העצים ואחריה this יהיה חיבור העצים וכן יהיה מאוזן. לאחר מכן , אם updateMinMax הוא true (רק אם לא קוראים לjoin מתוך split) אז נתאחל לעץ החדש את המינימום ואת המקסימום בעזרת findMin וfindMax , ואז יוחזר הפרש הגבהים של העץ this והעץ t ועוד 1.

נשים לב ש joinBiggerTreeIsRight ו joinBiggerTreeIsLeft מתבצעות בO(log(n)) וכן גם findMin וfindMax- . joinSameHeights מתבצעת בO(1).

סה"כ הפונק' תתבצע בO(log(n)).

**48. private void joinSameHeights(AVLTree leftSon,IAVLNode x,AVLTree rightSon)**

מחבר את leftSon משמאל לx ואת rightSon מימין לx. מגדירים את this שהשורש שלו יהיה x .

**49. private void joinBiggerTreeIsLeft(AVLTree smaller,IAVLNode x,AVLTree bigger, boolean myTreeIsSmaller)**

בעזרת searchRankNodeLeftJoin אשר מתבצעת בO(log(n)), מוצאים את הבן השמאלי החדש של x בעץ bigger. מבצעים את החיבורים כך ש newLeftSon הוא הבן השמאלי של x והשורש של smaller הוא הבן הימני של x. אם צריך (this הוא העץ הקטן) מגדירים את

השורש של this להיות x. מבצעים rebalanceInsertion אשר מאזנת את העץ החדש ומתבצעת ב(O(log(n)).

50. **private void joinBiggerTreeIsRight(AVLTree smaller,IAVLNode x,AVLTree bigger,boolean myTreeIsSmaller )**

בעזרת searchRankNodeRightJoin אשר מתבצעת ב(O(log(n)), מוצאים את הבן הימני החדש של x בעץ bigger. מבצעים את החיבורים כך ש newRightSonהוא הבן השמאלי של x והשורש של smaller הוא הבן הימני של x. אם צריך (this הוא העץ הקטן) מגדירים את השורש של this להיות x. מבצעים rebalanceInsertion אשר מאזנת את העץ החדש ומתבצעת ב(O(log(n)).

51. **private IAVLNode searchRankNodeRightJoin(IAVLNode node, int rank, int newSize)**

מחפשת את הצומת בגובה rank לאורך הצד השמאלי של העץ שמתחיל בnode. לכל צומת שעוברים בה, מעדכנת לsize החדש שלו. מחזירה את צומת שיהיה הבן החדש בחיבור של join. סה"כ עוברת לאורך גובה העץ לכן (O(log(n)).

52. **private IAVLNode searchRankNodeLeftJoin(IAVLNode node, int rank, int newSize)**

מחפשת את הצומת בגובה rank לאורך הצד הימני של העץ שמתחיל בnode. לכל צומת שעוברים בה, מעדכנת לsize החדש שלו. מחזירה את צומת שיהיה הבן החדש בחיבור של join. סה"כ עוברת לאורך גובה העץ לכן (O(log(n)).

## <u>מחלקה AVLNode</u>

שדות המחלקה:

1. private int key– מחזיק את מפתח הצומת.
2. private String value– מחזיק את הערך לצומת.
3. private int rank – גובה הצומת בעץ.
4. private AVLNode left – מצביע לבן השמאלי.
5. private AVLNode right - מצביע לבן הימני.
6. private AVLNode parent - מצביע לאבא.
7. private int size – מחזיר את גודל תת העץ שהצומת הוא השורש שלו, כולל הוא עצמו.

פונקציות המחלקה :

1. **public AVLNode(int newKey, String newValue)**

בנאי - יוצר צומת אמיתי חדש ומעדכן את שדות הצומת - הערך והמפתח לפי הקלט , הדרגה 0, size הוא 1- , left , right,parent, כולם null. מעדכנים מספר קבוע של מצביעים ולכן הסיבוכיות היא O(1) .

2. **public AVLNode()**

יצירת צומת לא אמיתי – מעדכנים את המפתח שלו והדרגה ל 1- , הערך הוא 0. סיבוכיות – עדכון מספר קבוע של מצביעים – O(1) .

3. **public int getKey()**

מחזיר את המפתח של הצומת . סיבוכיות O(1) .

4. **public void setLeft(IAVLNode node)**

מעדכן את הבן השמאלי של הצומת. לצומת הנתון בקלט. סיבוכיות O(1) .

5. **public IAVLNode getLeft()**

מחזיר את הבן השמאלי של הצומת. סיבוכיות O(1) .

6. **public void setRight(IAVLNode node)**

מעדכן את הבן הימני של הצומת להיות הצומת הנתון בקלט. סיבוכיות O(1) .

7. **public IAVLNode getRight()**

מחזיר את הבן הימני של הצומת. סיבוכיות O(1) .

8. **public void setParent(IAVLNode node)**
מגדירים את האבא של הצומת להיות הצומת הנתון בקלט. סיבוכיות O(1) .
9. **public IAVLNode getParent()**
מחזיר את האבא של הצומת. סיבוכיות O(1) .
10. **public boolean isRealNode()**
בודק אם צומת הוא אמיתי או לא. סיבוכיות O(1) .
11. **public void setHeight(int height)**
מעדכן את הדרגה של צומת להיות הגובה הנתון בקלט, שכן בעץ AVL הגובה של כל צומת זהה לדרגתו. סיבוכיות O(1) .
12. **public int getHeight()**
מחזיר את הדרגה של צומת , שכן הדרגה זהה לגובה בעץ AVL . סיבוכיות O(1) .
13. **public void setSize(int newSize)**
מעדכן את שדה ה size של צומת להיות השדה הנתון. סיבוכיות O(1) .
14. **public int getSize()**
מחזיר את הגודל של תת העץ שהצומת הוא השורש שלו. סיבוכיות O(1) .

## מדדים insert , delete

| מספר סידורי | מספר פעולות | מספר פעולות האיזון הממוצע לפעולת insert | מספר פעולות האיזון הממוצע לפעולת delete | מספר פעולות האיזון המקסימלי לפעולת insert | מספר פעולות האיזון המקסימלי לפעולת delete |
|---|---|---|---|---|---|
| 1 | 10,000 | 3.434967 | 2.403400 | 17 | 36 |
| 2 | 20,000 | 3.410167 | 2.415500 | 17 | 34 |
| 3 | 30,000 | 3.401211 | 2.414178 | 17 | 35 |
| 4 | 40,000 | 3.414408 | 2.412992 | 18 | 38 |
| 5 | 50,000 | 3.409347 | 2.412587 | 21 | 38 |
| 6 | 60,000 | 3.417489 | 2.413706, | 18 | 37 |
| 7 | 70,000 | 3.416819 | 2.413919 | 19 | 40 |
| 8 | 80,000 | 3.418717 | 2.415992 | 20 | 42 |
| 9 | 90,000 | 3.415411 | 2.414685 | 20 | 42 |
| 10 | 100,000 | 3.415247 | 2.414200 | 22 | 41 |

ראינו בכיתה כי זמן האומרטייז של פעולת insert ו delete כאשר מבצעים רק הכנסות ולאחר מכן רק מחיקות הוא O(1) . לכן ציפינו לקבל כי בממוצע מספר פעולות האיזון לאחר insert ולאחר מכן delete , יהיה קבוע ולא תלוי בגודל הקלט. בנוסף ב delete ישנם יותר מצבים שבהם בעיית חוסר האיזון בעץ מתגלגת כלפי מעלה ולא נפתרת – ולכן מספר פעולות איזון המקסימלי עבור delete יהיה גדול יותר מאשר insert. התוצאות תואמות את הציפיות שכן גם פעולת האיזון הממוצעות של insert וגם של delete הן קבועות ולא תלויות בגודל הקלט , כמו כן פעולת ה delete המקסימלית גדולה יותר מפעולת ה insert.

## מדדים split, join

| מספר סידורי | עלות join ממוצע עבור split אקראי | עלות join מקסימלי עבור split אקראי | עלות join ממוצע עבור split של איבר מקס בתת העץ השמאלי | עלות join מקסימלי עבור split של איבר מקס בתת העץ השמאלי |
|---|---|---|---|---|
| 1 | 2.322727 | 8 | 2.636364 | 15 |
| 2 | 2.462759 | 6 | 2.571429 | 17 |
| 3 | 2.532158 | 6 | 2.571429 | 17 |
| 4 | 2.668864 | 9 | 2.666667 | 18 |
| 5 | 2.646642 | 8 | 2.923077 | 18 |
| 6 | 2.712698 | 7 | 2.857143 | 19 |
| 7 | 2.786724 | 7 | 2.312500 | 19 |
| 8 | 2.544526 | 7 | 2.705882 | 19 |
| 9 | 2.632804 | 8 | 3.000000 | 20 |
| 10 | 2.619048 | 8 | 2.750000 | 19 |

ראינו בכיתה כי הסיבוכיות של פעולת ה join הוא הפרש הגבהים ועוד 1. בעץ AVL הפרש הגבהים בין כל צומת לבן שלו , הוא לכל היותר 2, לכן היינו מצפים שהפרש הגבהים בממוצע בין כל העצים שעליהם מבצעים join יהיה 2. בנוסף כאשר מבצעים את ה ספליט על האיבר המקסימלי בתת העץ השמאלי, לאיבר זה אין בן ימני, ולכן בהתחלה תת העץ הימני שלו ריק ולכן תתקבל פעולת join יקרה יותר מאשר פעולת ספליט על איבר אקראי בעץ שסביר שתת העץ הימני והשמאלי שלו לא יהיו ריקים ולכן פעולות ה join יתבצעו על עצים עם הפרשי גבהים נמוכים יותר. ניתן לראות שהתוצאות אכן תואמות את הציפיות שכן עלות join ממצועת על איבר אקראי וגם על האיבר המקסימלי בתת העץ השמאלי היא בין 2 ל 3 , כמו כן העלות של פעולת ה joinשל האיבר המקסימלי בתת העץ השמאלי היא גדולה יותר מאשר ספליט אקראי.