מבני נתונים – תרגיל מעשי 1

מגישים: שי פוקס 313452252

Username: shayakivafux

נתן בלוך 316130707

Username: nathanbloch

תיעוד

<u>המחלקה AVLNode</u>

<u>שדות המחלקה:</u>

- AVLNode right – מצביע אל האיבר שנמצא מימין("בן ימני")
- AVLNode left – מצביע אל האיבר שנמצא משמאל("בן שמאלי")
- AVLNode parent – מצביע אל האיבר שנמצא מעליו("הורה")
- Int key – המפתח של האיבר
- Int rank – הדרגה של האיבר
- String info – המידע לגבי האיבר אותו שומרים
- Int size – הגודל של תת-העץ של האיבר(כולל את האיבר עצמו)
- Int joinedResult – משתנה עזר בפונקציה join. לשימוש פנימי בלבד.

<u>בנאי המחלקה:</u>

- בנאי ריק – ()AVLNode. מיועד לבנייה של עלים וירטואליים בעץ, כלומר של צמתים וירטואליים.
- בנאי אמיתי - AVLNode(AVLNode right, AVLNode left, AVLNode parent, int key, String info)
  זהו הבנאי אשר בונה איבר חדש במחלקה. הוא מקבל את המצביעים לאלמנטים משמאל,מימין ומעליו.
  בנוסף מקבל ערך מספרי key וכן את ה-value שלו כמחרוזת.

<u>מתודות המחלקה:</u>

- במחלקה יש פונקציות Setters/Getters עבור השדות המחלקה. מאפשרות גישה לשדות האיבר ועריכתם של שדות המחלקה AVLNode.
  כל אחת מהן פועלת בסיבוכיות O(1).
- מתודה נוספת היא ()boolean isRealNode – מחזירה true אם ורק אם האיבר(ה-Node) הוא איבר אמיתי, ולא איבר וירטואלי(כלומר עלה וירטואלי). הפונקציה פועלת בסיבוכיות O(1).

המחלקה AVLTree

שדות המחלקה:

- IAVLNode root – מצביע אל השורש של העץ.
- Int n – מספר המייצג את מספר האיברים בעץ. מתעדכן בהכנסה/מחיקה.
- IAVLNode min – מצביע אל האיבר בעץ שהוא בעל המפתח(key) הקטן ביותר(המינימלי) בעץ.
- IAVLNode max – מצביע אל האיבר בעץ שהוא בעל המפתח(key) הגדול ביותר(המקסימלי) בעץ.

בנאי המחלקה:

- הבנאי הריק – ()AVLTree. הבנאי מאתחל את העץ להיות העץ הריק. מספר האיברים בעץ זה הוא 0. סיבוכיות זמן הריצה היא O(1) שכן מדובר רק באתחול העץ.
- הבנאי (AVLTree(IAVLNode node – הבנאי מקבל Node ובונה לפי השדות שלו עץ AVL חדש. משתמשים בבנאי זה רק בפונקציית ה-split. סיבוכיות זמן היא O(1) שכן מדובר רק באתחול העץ.

מתודות המחלקה:

- ()boolean empty – המתודה מחזירה true אמ"מ העץ הוא ריק. במימוש שלנו עץ ריק הוא עץ אשר השורש שלו מצביע ל-null. סיבוכיות זמן ריצה O(1).

- (String search(int k – זו פונקציית החיפוש בעץ. עבור מפתח נתון k הפונקצייה מחזירה את הערך של value של האיבר בעץ עם מפתח k. אם אין אף איבר בעץ אם המפתח k, אז הפונקציה מחזירה null. סיבוכיות זמן הריצה היא לכל היותר O(log n).
  פונקציה זו ממומשת בעזרת פונקציית (String search(int k, IAVLNode node. פונקציה זו הינה רקורסיבית והיא בעצם מבצעת את החיפוש על גבי העץ. הפונקציה עוברת לכל היותר בצומת אחד בכל רמה, ומבצעת השוואות בסיבוכיות O(1) בכל צומת. מספר הצמתים בעץ חסום על ידי O(log n), ולכן סיבוכיות זמן הריצה של הפונקציה (search(int k הוא O(log n).

- (Int insert(int k, String i – פונקציית הכנסת איבר חדש לעץ. הפונקציה בודקת האם ניתן להכניס איבר חדש לעץ עם מפתח k. אם כבר קיים המפתח k באחד מאיברי העץ אז יוחזר 1-. במצב זה O(log n) שכן נידרש לבצע חיפוש על גבי העץ.
  אם ניתן להכניס איבר חדש, הפונקציה מכניסה איבר חדש לעץ ולאחר מכן מעדכנת את שדות min, max בזמן בO(log n) על ידי מעבר על לכל היותר O(log n) רמות בעץ כדי להגיע לאיברים הללו. לאחר מכן מתבצעת פונקציית rebalance שמאזנת את העץ. אם האיבר הוכנס לnode שהיה unary אז אין צורך לאזן את העץ. אחרת האיבר הוכנס לnode שהיה עלה ונדרשות פעולות rebalance.
  סיבוכיות זמן הריצה של הפונקצייה insert היא במקרה הגרוע O(log n) שכן גם פעולות האיזון מתבצעות במקרה הגרוע ב- O(log n). כלומר בעצם בפעולת הכנסה מתבצעות מספר פעולות כך שכל פעולה היא לכל היותר O(log n), ולכן זמן הריצה הכולל של פעולת insert הוא גם כן O(log n). המתודה מחזירה את מספר הפעולות האיזון שנדרשו – כולל בין היתר את מספר פעולות
  ה- promote, left rotate, right rotate שהתבצעו במהלך פעולות האיזון בעץ לאחר הכנסה של איבר שגרם לאי-איזון.
  בסיום כל פעולת insert נתחזק ונעדכן את שדות העץ. אם בוצעה הכנסה בפועל של איבר חדש לעץ, נגדיל את השדה n שמייצג את מספר האיברים. נבצע קריאה לפונקציות updateMin, updateMax אשר יעדכנו את השדות min, max כך שיצביעו אל האיברים המינימליים/מקסימליים הנוכחים. פעולות אלו מתבצעות בסיבוכיות זמן של O(log n) כפי שיפורט בהמשך, שכן מדובר במעבר על לכל היותר O(log n) רמות בעץ.

- (Int rebalance(IAVLNode y, int counter – זוהי פעולת האיזון של עץ לאחר הכנסה של איבר לעץ. המתודה הנ"ל מומשה לפי ההנחיות ולפי הנלמד בהרצאות. המתודה מטפלת בכל מקרה בו יש אי-איזון בעץ על ידי פעולות מסוג promote, rotate left, rotate right. כל אחת מבין הפעולות הללו לוקחת במקרה הגרוע O(1) שכן מדובר במספר סופי של פעולות שכל אחת מהן מתבצעת בזמן קבוע. לכל היותר בפעולת rebalance נעבור על כל רמה בעץ, ובכל רמה נעבור על צומת אחד בלבד ונבצע עליו פעולות מהסוגים שהוזכרו. כלומר לכל היותר נבצע O(log n) פעולות כך שכל פעולה מתבצעת בזמן קבוע של O(1) ולכן סיבוכיות זמן הריצה במקרה הגרוע של הפונקציה rebalance היא O(log n).
  המתודה הזו מונה את מספר פעולות האיזון שהתבצעו במהלכה.
  כל פעולה מסוג promote, left rotate, right rotate נספרת פעם אחת. המתודה מחזירה את מספר פעולות האיזון שביצעה.

- ()Void updateMin – פונקצייה זו היא פונקציית עידכון המינימום בעץ. המינימום(האיבר בעל המפתח הקטן ביותר) נשמר בשדה min של העץ. הפונקציה הזו עוברת משורש העץ עד לנקודה השמאלית ביותר בעץ, על ידי מעבר שמאלה בכל רמה. כידוע, המינימום בעץ חיפוש הוא האיבר השמאלי ביותר בעץ.
  לאחר המעבר והגעה לאיבר בעל המפתח המינימלי, הפונקציה מעדכנת את השדה min.
  זמן הריצה הוא כגובה העץ. בכל רמה בה נעבור מבצעים פעולה אחת שלוקחת זמן קבוע O(1) שבה עוברים שמאלה(לבן השמאלי), ולכן זמן הריצה הכולל של הפעולה updateMin הוא כגובה העץ, כמספר הרמות בו נעבור, ולכן במקרה הגרוע ביותר O(log n).

- ()Void updateMax – פונקצייה זו היא פונקציית עידכון המקסימום בעץ. המקסימום(האיבר בעל המפתח הגדול ביותר) נשמר בשדה max של העץ. הפונקציה הזו עוברת משורש העץ עד לנקודה הימנית ביותר בעץ, על ידי מעבר ימינה בכל רמה. כידוע, המקסימום בעץ חיפוש הוא האיבר הימני ביותר בעץ.
  לאחר המעבר והגעה לאיבר בעל המפתח המקסימלי, הפונקציה מעדכנת את השדה max.
  זמן הריצה הוא כגובה העץ. בכל רמה בה נעבור מבצעים פעולה אחת שלוקחת זמן קבוע O(1) שבה עוברים ימינה(לבן הימני), ולכן זמן הריצה הכולל של הפעולה updateMax הוא כגובה העץ, כמספר הרמות בו נעבור, ולכן במקרה הגרוע ביותר O(log n).

- (Int[] RankDifference(IAVLNode y – פונקצייה זו מקבלת איבר(Node) ומחשבת את הפרשי הדרגות בינו לבין הבנים שלו. פונקציה זו אינה מיועדת עבור null, ואינה מיועדת עבור virtual leaves. הפונקציה מחזירה מערך מהצורה: [left difference, right difference] שמייצגים את הפרשי הדרגות. סיבוכיות זמן הריצה היא קבועה O(1) שכן בכל קריאה לפעולה זו מתבצע מספר סופי של פעולות שכל אחת מהן O(1). ולכן במקרה הגרוע ביותר(והכללי) סיבוכיות זמן הריצה הכולל של הפעולה הוא O(1).

- (Void RightRotate(IAVLNode y – פונקציה זו היא פונקצייה אשר מבצעת גלגול של העץ ימינה סביב הצומת הניתן y. כלומר הפונקציה מסובבת את העץ ימינה וגורמת לסיבוב של y ושל הבן השמאלי שלו. סיבוכיות זמן הריצה היא O(1) שכן נדרשות מספר סופי של פעולות שכל אחת מהן היא O(1). בסה"כ מדובר במספר סופי של שינויי מצביעים ומשתנים(כגון size).
  ולכן במקרה הגרוע ביותר(ובמקרה הכללי) סיבוכיות זמן הריצה הכולל של הפעולה הוא O(1).

- Void LeftRotate(IAVLNode x) – פונקציה זו היא פונקצייה אשר מבצעת גלגול של העץ שמאלה סביב הניתן x. כלומר הפונקציה מסובבת את העץ שמאלה וגורמת לסיבוב של x ושל הבן הימני שלו.
סיבוכיות זמן הריצה היא O(1) שכן נדרשות מספר סופי של פעולות שכל אחת מהן היא O(1), שכן מדובר בסה"כ במספר סופי של שינויי מצביעים ומשתנים(כגון size).
ולכן במקרה הגרוע ביותר(ובמקרה הכללי) סיבוכיות זמן הריצה הכולל של הפעולה הוא O(1).

- Void demoteSize(IAVLNode successParent) – פונקציה זו היא פונקצית עזר במחיקה של איבר מהעץ. הפונקציה מבצעת פעולה של הורדת ב-1(demote) של השדה size של כל איבר במסלול כלפי מעלה(עד השורש, כולל) של האיבר שהתקבל כפרמטר.
במקרה הגרוע ביותר סיבוכיות זמן הריצה הכולל של הפעולה הוא O(log n), כגובה העץ. במקרה הגרוע ביותר נעבור מאיבר שנמצא ברמה התחתונה ביותר כלפי מעלה עד השורש, ובכל מעבר נבצע O(1) פעולות. ולכן זמן הריצה במקרה הגרוע ביותר הוא O(log n).

- Int delete(key k) – זוהי פונקציית המחיקה מהעץ של איבר(Node) בעל מפתח כלשהו k. הפונקציה ראשית מחפשת את האיבר בעל מפתח k. במידה ולא קיים איבר כזה, היא מחזירה 1-. אחרת, היא מוחקת אותו לפי האלגוריתם שנלמד בכיתה ולאחר מכן במידה ויש צורך באיזונים של העץ הפונקציה קוראת לפונקציית העזר rebalancedelete שבה מתבצעים בפועל האיזונים הדרושים לעץ.
סיבוכיות זמן הריצה הכולל של הפונקציה הוא O(log n), כפי שנפרט:
  - זמן הריצה הכולל מחשיב את פעולות החיפוש אחר האיבר בעץ – מתבצע חיפוש בינארי של האיבר בעץ בעל המפתח k – הפעולה היא בסיבוכיות זמן של O(log n).
  - במידה וקיים, נבצע חיפוש של המיקום בו מוחקים – O(log n).
  - המחיקה של האיבר מתבצעת לפי האלגוריתם הנלמד בכיתה. אם האיבר בעל שני ילדים – אז מתבצעת החלפה שלו עם האיבר העוקב לו(successor), ואחרת הוא בן אונארי או עלה ומוחקים אותו ישירות – גם כאן לכל היותר O(log n).
  - לאחר מחיקת האיבר נעבור לפעולות האיזונים של העץ – מתבצעות בפונקציה rebalancedelete פונקציה זו רקורסיבית וסיבוכיות זמן הריצה שלה חסום על ידי גובה העץ – כלומר O(log n) במקרה הגרוע ביותר.
  - הערך המוחזר מהפונקציה הוא מספר פעולות האיזון שהתבצעו על העץ, כלומר מספר פעולות ה-demote, promote, left rotate, right rotate, double rotations כאשר כל פעולה נספרת ומחזירה את סך הפעולות שהתבצעו. את פעולות הסיבוב הכפול סופרים פעמיים.

סה"כ כפי שפורט הפונקציה delete מתבצעת בסיבוכיות זמן כוללת במקרה הגרוע ביותר ב-O(log n).
כפי שהוסבר מדובר במספר סופי של פעולות אשר כל אחת חסומה על ידי O(log n) ולכן כך גם סיבוכיות זמן הריצה הכוללת של הפונקציה delete.

בסיום כל פעולת delete נתחזק ונעדכן את שדות העץ. נקטין ב-1 את השדה n שמייצג את מספר האיברים(אם התבצעה מחיקה בפועל). נבצע קריאה לפונקציות updateMin, updateMax אשר יעדכנו את השדות min, max כך שיצביעו אל האיברים המינימליים/מקסימליים הנוכחים. פעולות אלו מתבצעות בסיבוכיות זמן של O(log n) כפי שפורט, שכן מדובר במעבר על לכל היותר O(log n) רמות בעץ.

- Int rebalancedelete(IAVLNode z, int counter) – זוהי פונקציית האיזונים לאחר מחיקה של איבר מהעץ. זוהי פונקציה רקורסיבית שמקבל איבר בעץ Node ואת מונה הפעולות עד כה, ומבצעת על האיבר המתקבל את פעולות האיזון הנדרשות לפי הפרשי הדרגות שלו מהבנים שלו. אם יש צורך באיזונים נוספים, הפונקציה קוראת לעצמה עם השלב הבא באיזונים הדרושים.
הפונקציה הזו מחזירה counter שסופר את מספר פעולות האיזון שביצענו. הפעולות הן promotes, demotes, left-rotates, right-rotates, double-rotates כאשר כל פעולה נספרת, ופעולות הסיבוב הכפול נספרות פעמיים.
סיבוכיות זמן הריצה הכוללת היא O(log n). בכלב שלב של איזון, נידרש לבצע O(1) פעולות. בכל שלב מבצעים מספר סופי של פעולות איזון לפי הפרשי הדרגות של האיבר, אך ייתכן כי נידרש לגלגל את הבעיה כלפי מעלה מספר פעמים כגובה העץ, ולכן במקרה הגרוע סיבוכיות זמן הריצה היא O(log n).

- String min() – פונקציה זו מחזירה את ה-value של האיבר המינימלי בעץ. הפונקציה פועלת בסיבוכיות זמן של O(1) במקרה הגרוע ביותר שכן בכל שלב מתוחזק מצביע אל האיבר(ה-Node) בעל המפתח המינימלי בעץ. לכן סיבוכיות זמן הריצה היא O(1) והיא קבועה בכל מקרה.

- String max() – פונקציה זו מחזירה את ה-value של האיבר המקסימלי בעץ. הפונקציה פועלת בסיבוכיות זמן של O(1) במקרה הגרוע ביותר שכן בכל שלב מותחזק מצביע אל האיבר(ה-Node) בעל המפתח המקסימלי בעץ. ולכן סיבוכיות זמן הריצה היא O(1) והיא קבועה בכל מקרה.

- Int[] keysToArray() – פונקציה זו מחזירה מערך ממוין של כל המפתחות בעץ בסדר עולה. במידה והעץ ריק, הפונקציה מחזירה מערך ריק. הפונקציה מבצעת קריאה לפונקציה הרקורסיבית keysToArray(IAVLNode node, int[] arr, int index) שבה בעצם מתבצעת כל בניית מערך המפתחות.
הפונקציה פועלת בסיבוכיות זמן של O(n) שכן נידרש לעבור על כל איבר בעץ כדי לבנות את מערך המפתחות הממוין. הפונקציה הרקורסיבית פועלת על גבי אלגוריתם של in-order tree traversal כפי שנלמד. נשלחים לפונקציית הפרמטרים הבאים: שורש העץ, מערך בגודל n אשר בעצם מכיל רק אפסים ואינו מעודכן, וכן האינדקס 0. פרמטר האינדקס יעבור בקריאות הרקורסיביות ויעודכן בכל הכנסה של איבר לעץ. פרמטר זה ייצג את המיקום במערך שבו יכנס האיבר הבא.

  סיבוכיות זמן כוללת – O(n) כמספר האיברים בעץ

- פונקציית העזר הרקורסיבית int keysToArray(IAVLNode node, int[] arr, int index) – כפי שהוסבר, כאן בעצם באופן רקורסיבי מתבצע תהליך in-order tree traversal שבכל שלב נכניס למערך המועבר את המפתח של האיבר הנוכחי במיקום index, נגדיל את המשתנה index באחד על מנת להכניס את איבר הבא במקום הבא. לפי אלגוריתם in-order tree traversal בכל צומת ראשית נבצע את הפעולה הרצויה(כאן הכנסה של המפתח למערך) על תת-העץ השמאלי שלו, לאחר מכן נבצע את הפעולה על הצומת בעצמו, ואז נבצע את הפעולה על תת-העץ הימני של הצומת. בסה"כ בכל צומת נבקר פעם אחת בלבד, ונבצע בה פעולה אחת שעלותה O(1) – הכנסת האיבר למערך, ולכן סיבוכיות זמן הריצה של הפונקציה הרקורסיבית הזו הוא O(n) כמספר האיברים הכולל בעץ.

- String[] infoToArray() – פונקציה זו בעצם דומה לפונקציה keysToArray() רק שכאן נחזיר מערך של המידע(Strings) שנמצאים בכל איבר ולא את מערך המפתחות. הפונקציה מחזירה מערך ממוין **לפי המפתחות** של כל הערכים(info) של האיברים בעץ.
  המימוש גם כאן נעשה בעזרת פונקציית עזר רקורסיבית infoToArray(IAVLNode, String[] arr, int index) שפועלת בעצם לפי אותו האלגוריתם של in-order tree traversal, רק שכעת הפעולה שהיא מבצעת על כל צומת היא הכנסת ה-info של הצומת למערך. סיבוכיות זמן הריצה זהה – O(n) שכן נידרש לעבור על כל איבר בעץ.

- Int infoToArray(IAVLNode, String[] arr, int index) – פונקציית העזר הרקורסיבית. דומה לפונקציית העזר של בניית מערך המפתחות הממוינים רק שכעת בכל צומת נכניס למערך את הערך(info) של הצומת למערך. הפונקציה פועלת לפי אלגוריתם in-order tree traversal. בכל צומת תתבצע פעולת אחת של הכנסת הערך info של הצומת למערך במיקום index, הגדלת index ב-1, והתקדמות לצומת הבאה. סה"כ בכל צומת נבצע פעולות בסיבוכיות זמן של O(1), ולכן סיבוכיות הזמן הכוללת היא O(n) כמספר האיברים שנמצאים בעץ.

  המערך המוחזר הוא מערך של Strings המכיל את המידע של האיברים, והמערך בעצם ממוין לפי המפתחות התואמים של המידע.

- IAVLNode getRoot() – הפונקציה מחזירה את שורש העץ, כיוון שיש שדה המצביע לשורש, הפעולה עולה O(1).

- Int getSize() - הפונקציה מחזירה את מספר האיברים בעץ ע"י החזרת הערך השמור בשדה size של השורש (נעשה שימוש במתודה getSize() שכבר פורטה) הגישה לשדה עולה O(1) ולכן גם הפונקציה פועלת בכל מקרה בסיבוכיות זמן O(1).

- AVLTree[] split(int x) – נבצע חיפוש בינארי בעץ כדי למצוא את הצומת בעל המפתח x (עולה במקרה הגרוע $O(\log n)$) נסמן צומת זו ב-Y. נאתחל שני עצי AVL: אחד עבור העץ בעל המפתחות הגבוהים מ-x נסמנו ב-high (בהתחלה עץ זה יכיל את תת העץ הימני של Y) והשני עבור המפתחות הקטנים מ-x נסמנו ב-low (בהתחלה עץ זה יכיל את תת העץ השמאלי של Y). כעת לפי האלגוריתם הנראה בכיתה נעלה במסלול מ- Y עד לשורש(המסלול במקרה הגרוע הוא $O(\log n)$, כגובה העץ) ונבצע את פעולת join באופן הבא: אם Y הוא בן ימני נאחד ע"י פעולת join את תת- העץ השמאלי של אבא של Y עם low ונשמור את התוצאה ב-low(כלומר נעדכנו) אם y הוא בן שמאלי נאחד ע"י פעולת join את תת העץ הימני של אבא שלו עם high ונשמור את התוצאה ב-high(כלומר נעדכנו) כאשר בשני המקרים גם האבא של y מצטרף לאיחוד שני העצים . נמשיך עם פעולה זו כאשר בכל איטרציה אבא שלו Y משמש כ-y בלולאת ה-while. ראינו בכיתה כי סיבוכיות split הינה $O(\log n)$ כיוון שהשתמשנו בכל איטרציה בפונקציית join אשר עלותה גם כן O(log n) אך בתחילה העצים המאוחדים קטנים וכפי שנלמד סיבוכיות הפונקציה split הינה $O(\log n)$.

- int join(IAVLNode x, AVLTree t) –

הפונקציה מקבלת Node ועץ AVL ומאחדת את העץ המתקבל כארגומנט עם הצומת x ועם העץ this. הפונקציה עושה שימוש באחת המתודות joinLeft או joinRight שנפרט עליהם בהמשך. הפונקציה מחליטה לאיזה פונקציה מהשתיים לשולח את הארגומנטים לפי ה-rank וגודל המפתחות. מכיוון שסיבוכיות המקרה הגרוע בכל אחת מהפונקציות joinLeft ו- joinRight הינה $O(\log n)$ אז גם כאן סיבכויות הזמן במקרה הגרוע הינה $O(\log n)$.

- IAVLNode joinRight(AVLTree t1,IAVLNode x, AVLTree t2) -

לפונקציה זו הפונקציה join שולחת את הארגומנטים כך שמתקבל :

$$t1.keys < x.key < t2.keys \quad \&\& \quad rank(t1) \geq rank(t2)$$

כלומר עץ t1 בעל מפתחות קטנים יותר ובעל rank גדול יותר. לפיכך נרד במסלול הכי חיצוני <u>הימני</u> בעץ t1 מהשורש ועד לרגע בו מצאנו צומת בעל rank קטן או שווה מה-rank של השורש של t2 (המסלול יכול עלות לכל היותר $O(logn)$) . ולפי האלגוריתם שנלמד בכיתה : נסמן את הצומת שירדנו במורד המסלול החיצוני הימני ביותר ב-b , נהפוך אותו לבן שמאלי הצומת x ואת השורש של t2 נהפוך לבן ימני של x, ואת x עצמו נהפוך לבן ימני של אבא של b(נסמנו ב-c) בעץ t1 . כעת נבדוק האם העץ צריך איזון לאחר האיחוד הנ"ל. אם כן נשלח ל- rebalance של פעולת insert, שתוארה כבר שעלותה לכל היותר $O(logn)$. לפני השליחה לrebalance נעדכן את כל שדות ה-size מהצומת c ולכל האבות הקדמונים שלו עד השורש ע"י הפונקציה updateSizes שעלותה במקרה הגרוע $O(logn)$ (הפונקציה תפורט בהמשך). נשים לב כי עלול להיווצר מצב שהפונקציה rebalance אינה יודעת לטפל בו כאשר הצומת c בעלת הפרשי rank 2,0 כדי לפתור זאת נבצע גלגול שמאלי שעלותה O(1) ולאחר מכן נשלח את הפונקציה ל-rebalance עם אבא שלcשל לאחר הגלגול. הפונקציה מחזירה את הפרשי ה-rank ועוד 1 באופן הבא: $rank(t1) - rank(t2) + 1$. סיבוכיות הפונקציה הינה $O(logn)$ שכן כפי שפורט מתבצעות מספר סופי של פעולות שעלותן היא $O(logn)$.

- IAVLNode joinLeft(AVLTree t1,IAVLNode x, AVLTree t2) –

לפונקציה זו הפונקציה join שולחת את הארגומנטים כך שמתקבל :

$$t1.keys < x.key < t2.keys \quad \&\& \quad rank(t1) \leq rank(t2)$$

כלומר עץ t2 בעל מפתחות גדולים יותר ובעל rank גדול יותר. לפיכך נרד במסלול החיצוני ביותר <u>השמאלי</u> בעץ t2 עד לרגע בו נמצא צומת שה-rank שלה קטן או שווה ל-rank של השורש של t1(המסלול עולה לכל היותר$O(logn)$),ונפעל באופן סימטרי לחלוטין לפונקציה joinRight עם ההבדלים הבאים: נעשה גלגול ימני במקום גלגול שמאלי במידה והצומת הדרושה עם הפרשי rank 0,2 ונחזיר את הפרשי ה-rank באופן הבא: $rank(t2) - rank(t1) + 1$. גם כאן בדומה ל-joinRight סיבוכיות הפונקציה הינה $O(logn)$ כפי שפורט מתבצעות מספר סופי של פעולות שעלותן היא $O(logn)$.

- void updateSizes(IAVLNode node) –

הפונקציה מעדכנת את כל שדות ה-size של צומת ושל כל האבות הקדמונים במסלול אליו מהשורש. סיבוכויות פונקציה זו במקרה הגרוע הינה כאורך המסלול במקרה הגרוע : $O(logn)$ שכן עדכון השדות עולה O(1).

מדידות

ניסוי 1 – פעולות הכנסה ומחיקה

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	3.3768	2.4136	17	28
2	20,000	3.4374	2.4206	18	28
3	30,000	3.4126	2.4190	17	29
4	40,000	3.4242	2.4028	20	33
5	50,000	3.3948	2.4116	18	37
6	60,000	3.4052	2.4168	18	37
7	70,000	3.4160	2.4160	18	40
8	80,000	3.4227	2.4120	18	36
9	90,000	3.4211	2.4119	20	33
10	100,000	3.4268	2.4089	19	35

הסבר התוצאות שנמדדו:

לפי התיאוריה שנלמדה בכיתה, עצי AVL הינם מאוזנים וגובהם O(log n), ובנוסף ראינו כי עבור סדרה של פעולות הכנסה **בלבד** זמן הריצה ה-amortized הינה O(1) לפעולת הכנסה בודדת. לפי התוצאות שהתקבלו במדידות, ניתן להתייחס למספר פעולות האיזון הממוצע לפעולת insert, כזמן ה- amortized לפעולת insert בודדת, ואכן ניתן לראות כי זמן זה חסום על ידי קבוע(בקירוב 3.4, והקבוע נשאר יציב לפי דרישות סיבוכיות אסימפטוטיות). כלומר, לכל n(מספר האיברים בעץ) קיים קבוע כך שזמן הפעולה ה-amortized עבור פעולת insert בודדת חסום על ידי הקבוע.

עבור פעולת מחיקה, לפי התיאוריה שראינו בכיתה, לאחר שביצענו את כל פעולות ההכנסה, עבור סדרת פעולות מחיקה כלשהי, ובפרט עבור סדרת מחיקות לפי הגודל, זמן הריצה ה-amortized לפעולת delete בודדת חסום על ידי קבוע שהוא מספר פעולות האיזון הממוצעות לאחר פעולת delete. בדומה ומשיקולים דומים לפעולת insert.

מספר פעולות האיזון בכל שלב במהלך האלגוריתם(עבור insert וגם עבור delete) חסום על ידי קבוע כלשהו. לכל היותר במקרה הגרוע נבצע מספר פעולות איזון לא אחרונות כגובה העץ, כלומר מספר הפעולות האיזון הכולל(עבור insert וגם עבור delete) חסום על ידי קבוע כלשהו כפול גובה העץ. ואכן ניתן לראות זאת כאן במדידות.

לסיכום, הציפיות מהנלמד בתיאוריה של עצי AVL, התממשו במדידות.

ניסוי 2 – פעולות join ו-split

מספר סידורי	מספר פעולות	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	10,000	2.545	6	2.692	16
2	20,000	2.153	4	2.538	17
3	30,000	2.533	5	2.571	18
4	40,000	2.500	6	2.466	18
5	50,000	2.285	6	2.400	19
6	60,000	2.470	9	2.600	19
7	70,000	2.333	5	2.222	19
8	80,000	2.400	6	2.666	20
9	90,000	2.285	5	2.588	20
10	100,000	2.444	7	2.533	20

הסבר המדידות:

בניסוי הראשון, ניתן להבחין כי הסבירות להגריל איבר שמסלול פעולות ה-join הינו "מאוזן", כלומר מבצעים מספר פעולות join לעץ עם הערכים הקטנים במידה דומה למספר פעולות ה-join לעץ עם הערכים הגדולים. בהתאם, בכל שלב שני העצים יהיו דומים בגודלם, ולכן עלות ה-join בשלבים אלו הינה נמוכה למדי ולכן כאשר נגיע לשלב האחרון של האלגוריתם בו ממזגים עם תת-העץ השני של השורש(כתלות באיטרציה האחרונה), נגיע למצב זה עם שני עצים דיי דומים בדרגתם ולכן פעולת ה-join הנ"ל תהיה נמוכה בהשוואה למצב הקיצוני בניסוי 2, כמפורט בהמשך. ולכן עלות ה-join המקסימלי נמוכה בהשוואה לעלות ה-join המקסימלי בניסוי 2. ועל כן, כיוון שעלות join נמדדת כ- RankT2 – RankT1 + 1 נוכל להסיק כי עלות הגוין המקסימלית בניסוי זהה תהיה נמוכה יחסית בסבירות גבוהה.

העלות הממוצעת של פעולות ה-join הינה נמוכה יחסית למקסימלית כיוון שפעולת ה-join היקרה יחסית התבצעה פעם אחת בלבד בסוף האלגוריתם, כפי שתואר. מרבית פעולות ה-join נעשו על עצים בעלי דרגות שוות יחסית בהתאם לפרופורציה של תתי-העצים של צומת בעץ AVL תקין.

בניסוי השני, מבצעים פעולת split על האיבר המקסימלי בתת-העץ השמאלי של השורש, כלומר האיבר יהיה עלה או צומת אונארי. כיוון שהעץ הינו AVL ניתן להניח כי הצומת הנ"ל נמצא בעומק O(log n), בדומה לשאר הצמתים האוניאריים והעלים בעץ AVL תקין. איבר זה הינו האיבר המקסימלי בתת-העץ השמאלי של השורש. במהלך האלגוריתם אין איברים גדולים יותר מאשר הצומת הנ"ל ולכן בכל שלב(מלבד האחרון) נבצע פעולות join אך ורק על תתי-העצים של האיברים הקטנים מהצומת הנ"ל. בשלב האחרון, בו העץ של האיברים הגדולים מהצומת בעצם הוא עץ ריק, ונבצע פעולת join עליו ועל תת-העץ הימני של השורש בעזרת השורש. כאן נקבל בעצם כי מתבצע join על עץ אחד עם דרגה גבוהה יחסית בדומה לדרגת העץ המקורי, ועץ בעל דרגה שהינה 1- כלומר עץ ריק, ומכאן נובע גם מספר פעולות הגוין המקסימלי שקורה בשלב זה בדיוק. ועל כן, כיוון שעלות join נמדדת כ- RankT2 – RankT1 + 1 נוכל להסיק כי עלות הגוין הינה גבוהה יחסית לפעולות join סטנדרטיות.

העלות הממוצעת של פעולות ה-join הינה נמוכה יחסית למקסימלית כיוון שפעולת ה-join היקרה באופן משמעותי התבצעה פעם אחת בלבד בסוף האלגוריתם, כפי שתואר. מרבית פעולות ה-join נעשו על עצים בעלי דרגות שוות יחסית בהתאם לפרופורציה של תתי-העצים בעץ AVL תקין.

ניתן לראות כי בשני הניסויים הממוצעים דיי דומים כיוון שיש פעולת join יקרה אחת והיא קורה בסוף התהליך, לאחר מספר פעולות join שהן זולות יחסית אליה ומתבצעות פעמים רבות. על כן, הממוצע מושפע בעיקר מפעולות ה-join הזולות יחסית שקורות בתדירות גבוהה יותר.