**תיעוד + סיבוכיות זמן:**

**המחלקה AVLTree :**

כללי:

לכל אובייקט במחלקה נתחזק את השדות הבאים:

* Root – מצביע לשורש של העץ
* Min – מצביע לצומת בעל המפתח המינימלי בעץ
* Max – מצביע לצומת בעל המפתח המקסימלי בעץ

שדות נוספים שיש בהם צורך כמו size או height נתחזק דרך הצמתים עצמם.

לאורך כל הפונקציות שמשנות את מצב העץ נתחזק את כלל השדות הנ"ל.

פונקציות:

* AVLTree

בנאי ריק למחלקה – מאתחל את השדות באופן דיפולטי

סיבוכיות: O(1).

* AVLTree(root)

בנאי שמקבל שורש של עץ ומכייל את השדות min max ע"י קריאה לפונקציות myMin \ myMax שמחפשות מינימום \ מקסימום בתת העץ של הצומת הקלט – פועלות בלכל היותר O(logn).

סיבוכיות: O(logn) - נרד בחיפוש מינימום \ מקסימום עד למטה.

* Insert:

הפונקציה תקבל key, value ותיצור IAVLnode.

תחילה הפונקצייה בודקת אם העץ ריק- אם כן מעדכנת את כל השדות (מינימום ומקסימום להיות הנקודה ומחזירה 0- לא היו סיבובים ועדכונים.

אחרת, נחפש את הצומת שאליה נכניס את הצומת החדש/ אם הצומת קיימת נחזיר אותה באמצעותplace to insert .

נחבר את הצומת החדש למקום המתאים ונבצע עדכוני מינימום ומקסימום.

נקרא לrebalance ולבסוף לupdate till root.

סיבוכיות: כפי שמפורט למטה כל אחת מהקריאות לפונקציות העזר מתרחשות ב- O(log(n)) שאר הבדיקות בפונקציה מתקיימות בזמן קבוע ולכן סה"כ נקבל O(log(n)).

* Place\_to\_insert:

טיול ימינה כל עוד המפתח גדול מהצומת החדש ושמאלה אם ההפך.

זאת נבצע כל עוד האיבר שאליו אנו רוצים ללכת אינו צומת פיקטיבית אם כן נחזיר את הצומת האחרונה שאינה פיקטיבית וגם אם הגענו למפתח של הצומת החדש (כבר קיים בעץ) נחזיר אותו.

סיבוכיות: במקרה הגרוע תרד משורש העץ אל עבר העלה הרחוק ביותר – כגודל גובה העץ-

* Rebalance:

פונקצייה רקורסיבית שמאזנת את העץ לאחר הכנסה בהינתן צומת שממנו צריך להתחיל\ להימשך תהליך האיזון.

בהינתן קלט של צומת מסוים בעץ ומס' השינויים שנעשו בתהליך האיזון הנוכחי עד כה, הפונקציה בודקת את getrankright, getrankleft- הפרשי הדרגות (הגבהים) של הצומת הנתון ביחס לבניו ולפי זה מסווגת לאיזה מקרה איזון מחדש צריך לקרוא או האם הצומת מאוזן וישר לסיים את הריצה הרקורסיבית.

בחלק ממקרה הסיווג יש צורך הבחנה לתתי מקרים באופן הבא:

הפרש דרגות (0,1)/(1,0)- נבצע promotion ונקרא שוב לrebalace (בתנאי שזה לא הroot ואז אין צורך).

הפרש דרגות (0,2)+ הפרש דרגות של בן שמאלי (1,2) – סיבוב ימינה. נקרא לright rotation. ראינו כי פעולה זאת מביאה למצב תקין ומאוזן בעץ ולכן נסיים ונחזיר את מספר הפעולות שאספנו עד כה.

הפרש דרגות (0,2)+ הפרש דרגות של בן שמאלי (2,1) – סיבוב כפול – תחילה שמאלה של הבן השמאלי ואז ימינה של הצומת העליונה והבן השמאלי.

\*המקרים הסימטרים של (2,0) מטופלים באופן סימטרי בתת תנאי נוסף בפונקציה.

בסיום כל פעולה נקרא לupdateNode עבור הצמתים שיש לעדכן מהנמוך יותר בעץ אשר תלוי בצמתים שלא השתנו ולגבוה יותר- לכל היותר 3 עדכונים.

הפונקציה אוספת באופן רקורסיבי בעזרת משתנה מוחזר מטיפוס int אשר אוסף את מספר הפעולות -עבור כל rotation, promotion, demotion (promotion, demotion ייספרו באמצעות updateNode).

סיבוכיות: כפי שהוכחנו בכיתה promotion עשוי לגרור את הבעיה מעלה ואילו סיבוב/ סיבוב כפול יובילו לreturn - כלומר ליציאה מהרקורסיה.

לאחר פעולת הpromote יש קריאה לrebalance עם parent- כלומר לכל היותר קריאות כגובה העץ.

בrotation\_right/left מבצעים ניתוקים וחיבורים קבועים- לכל היותר 6 ולכן סיבוכיות הזמן קבועה - .

בעדכון הצמתים נבצע טיפוס בעץ כגודל גובהו סיבוכיות O(logn).

* Join:

פונקציית מעטפת אשר מקבלת 2 עצים וקודקוד וממפה למצבים:

\*אם 2 העצים ריקים נקבע את x להיות השורש ונעדכן את שדות המינימום והמקסימום.

\*אם אחד העצים ריק ניצור קודקוד פיקטיבי ונשלח לפונקצייה הפנימית בהתאם לסדר האיברים (smaller, x, bigger).

נעדכן את שדות המינימום והמקסימום בהתאם.

\*2 העצים לא ריקים- נבדוק באיזה סדר לשלוח לפונקצייה הפנימית בהתאם ל(smaller, x, bigger).

נעדכן את שדות המינימום והמקסימום בהתאם.

בכל אחד מהשלבים נחזיר את הפרש הגבהים פלוס אחד כסיבוכיות זמן הריצה של join.

נשלח בכל פעם לjoin\_in את שלשת הקודקודים וjoin\_in מחזירה את הקודקוד המיועד להיות שורש העץ המעודכן. לכן נעדכן את שורש העץ להיות join\_in(..).

סיבוכיות: הפונקציה מבצעת פעולות אריתמטיות ועדכוני שדה ב עבודה ולאחר מכן קוראת ל join\_in שרצה ב - O(|hight(t1)-hight(t2)|+1) ולכן סה"כ נקבל

O(|hight(t1)-hight(t2)|+1)

* Join\_in:

הפונקציה מחברת בין קודקודים בהתאם לאלגוריתם join שלמדנו בכיתה, תוך שימוש בrebalance ובעדכון שדות הsize והhight של הקודקודים בקריאה לפונקצייה updateTillRoot . בנוסף, הפונקצייה תחזיר את הקודקוד המיועד להיות שורש העץ המעודכן.

נחלק למקרים:

אם העצים ריקים נחזיר את X.

אם גבהי העצים שווה, נחבר את איקס לשני הקודקודים בהתאם למיקום (קטן, גדול) ונחזיר את X.

אחרת נבדוק מי מבין הגבהים גדול יותר ונטייל עד לקודקודים שגובהו שווה לגובה העץ הקטן. נחבר את איקס אליו ונעדכן כלפי מעלה.

לבסוף, נקרא לפונקצייה find\_root אשר תעלה במעלה העץ עד לקודקוד שההורה שלו הוא null- זה יהיה שורש העץ החדש.

סיבוכיות:

\*ירידה בעץ בעל הגובה הגדול יותר עד לגובה העץ הקטן- |hight(t1)-hight(t2)|.

\*חיבורי X לקודקודים – פעולות קבועות בO(1).

\*הקריאה לrebalance מהאב של X – ראינו כי הפונקצייה הינה בסיבוכיות logn כאשר – כגובה העץ. כעת המעבר הינו על גובה של |hight(t1)-hight(t2)|.

עדכוני שדות הקודקודים מX ועד לשורש העץ- מעבר על |hight(t1)-hight(t2)| קודקודים.

סה"כ נקבל סיבוכיות .(|hight(t1)-hight(t2)|+1

* Empty:

מחזירה true אמ"מ השורש (root) מצביע על null או שהינו צומת דיגיטלי זה קורה אמ"מ העץ ריק

סיבוכיות:O(1) .

getRoot:

מחזירה את השורש של העץ, אם העץ ריק (בדיקה ע"י קריאה ל empty ) נחזיר null.

סיבוכיות: O(1).

* Search :

מחפשת את האיבר בעל המפתח K, אופן פעולה קוראת לפונקציית העזר find על הערך K אם לא חזר null מחזירה את הערך של הצומת שחזר אחרת מחזירה null.

סיבוכיות: פונקציית העזר find רצה ב-log(n) (פירוט בהמשך) ושאר הפונקציה הינו פעולות קבועות כלומר O(1) סה"כ נקבל O(logn).

* פונקציית העזר find:

מקבלת מפתח K ובודקת אם יש צומת בעץ בעל המפתח אם כן מחזירה את הצומת אם לא מחזירה null. אופן פעולה: אופן באופן זהה לפונקציית חיפוש שראינו בהרצאה, נטייל במורד העץ ובכל צומת נבדוק אם הוא הצומת המתאים אם לא - האם עלינו ללכת שמאלה או ימינה בהתאם להשוואה בין המפתחות אם הגענו לצומת וירטואלי נפסיק את החיפוש ונחזיר null. סיבוכיות: במקרה הגורע O(logn) עומק העץ.

* rotate\_left או rotate\_right:

מבצעת סיבוב שמאלה על 2 צמתים קשר הקשת שלהם היא ציר הסיבוב. מבצעים החלפה בחיבורים של הקשתות המתאימות ובודקים מקרה קצה במידה ומדובר בעדכון של השורש. מעדכנים מצביעים לאמא של כל צומת.

סיבוכיות: כל הפעולות בקריאה לפונק' קבועות ולכן O(1).

* Delete:

אם קיים צומת בעץ בעל מפתח K מוחקת אותו ומחזירה את סך פעולות האיזון באיזון העץ לאחר המחיקה.

אופן הפעולה:

קוראת לפונקציה find שבודקת אם קיים כזה צומת סיבוכיות O(logn)

אם לא קיים מחזירה -1, אם קיים מבצעים את רצץ הפעולות הבא:

בודקים אם מדובר במקרה קצה של עדכון מינימום או מקסימום, מעדכנים ע"י קריאה לפונק' successor או predecessor בהתאמה (סיבוכיות O(logn) ) לאחר מכן מקטינים את הגודל של העץ ב-1. ואז קוראים לפונקציה deleteRetrieve שמוחקת את העץ ע"י אפיון מקרה המחיקה ומחיקה בצורה המתאימה ומחזירה את הצומת שצריך לאזן. אם בוודאות אין כזה היא מחזירה null. אם הערך null מחזירים 0, אחרת קוראים לפונקציית האיזונים reBalanceDelete עם הצומת שצריך לעבור איזון, הפונקצייה עובדת באופן רקורסיבי ומאזנת את העץ ע"פ המקרים שנלמדו בהרצאה וסופרת את מס' השינויים שנעשים בעץ. לאחר שנסיים לאזן את העץ נמשיך לעלות במעלה העץ ונעדכן את השדה size לכל צומת עד השורש ע"י קריאה לפונקציה updateTillRoot. נשים לב שכל אחת מהפונקציות מתקיימת בזמן קבוע וסך האיטרציות חסום בגובה העץ.

סיבוכיות: סה"כ הסיבוכיות ברצף הקריאות הינו O(logn)\*3 = O(logn). שכן בכל קריאה לכל פונקציית עזר אנו עולים \ יורדים במקרה הגרוע ביותר את כל גובה העץ

* reBalanceDelete:

פונקצייה רקורסיבית שמאזנת את העץ לאחר מחיקה בהינתן צומת שממנו צריך להתחיל\ להימשך תהליך האיזון. שיטת עבודה: בהינתן קלט של צומת מסוים בעץ ומס' השינויים שנעשו בתהליך האיזון הנוכחי עד כה, הפונקציה בודקת את הפרשי הדרגות (הגבהים) של הצומת הנתון ביחס לבניו (ע"י קריאה ל- getRankLeft, getRankRight O(1) ). ולפי זה מסווגת לאיזה מקרה איזון מחדש צריך לקרוא או האם הצומת מאוזן וישר לסיים את הריצה הרקורסיבית. בחלק ממקרה הסיווג יש צורך הבחנה לתתי מקרים באופן הבא:

(3,1) 🡪 (1,1) / (2,1) / (1,2), (1,3) 🡪 (1,1) / (2,1) / (1,2), (2,2) , מצב מאוזן חוקי

לאחר סיווג המקרה ותת המקרה בעת הצורך הפונקציה קוראת לאחת מ-4 פונקציות האיזון המתאימות כאשר 3 מבצעות סוג תיקון מסוים ואת התיקון הסימטרי המקביל הפונקציות הללו יודעות איזה מקרה מדובר ע"י משתנה בוליאני.

בהגעה לצומת שהינו מאוזן, נמשיך ונעלה במעלה העץ ונעדכן את שדה ה-size ע"י קריאה לפונק' הרקורסיבית updateTillRoot.

סיבוכיות: נשים לב שכל הבדיקות והקריאות במהלך ריצת הפונקציה בכל קריאה קבועות ולכן הפונקציה רצה ב- O(1), (נקראת לכל היותר O(logn) לאחר מחיקה בודדת).

* updateTillRoot:

מקבלת צומת ומעדכנת את כל הצמתים מצומת זה ועד השורש ע"י קריאה לפונקציה – updateNode

שמעדכנת את הגובה וגודל של צומת לפי בניו.

סיבוכיות: לכל היותר נעלה את כל גובה העץ ולכן הסיבוכיות הינה O(logn).

* reBalanceCase22:

פונקציית האיזון לאחר מחיקה של המקרה (2,2), מבצעת דימוט לצומת המבוקש ואז אם הגענו לשורש מחזירה 1, אחרת קוראת ל reBalanceDelete על צומת האבא ועדכון המונה שקיבלנו עד כה +1.

סיבוכיות: כל הפעולות קבועות ולכן O(1) .

* reBalanceCase3112:

פונקציית האיזון לאחר מחיקה של המקרה (3,1) - (1,2)והמקרה הסימטרי (1,3) -(2,1). מבצעים גלגול ימני ואז גלגול שמאלה לפי הצמתים המתאימים או אם אנחנו עושים את המקרה הסימטרי באופן הפוך (יודעים באיזה מקרה מדובר ע"י משתנה בוליאני בקלט) לאחר מכן מעדכנים את הדרגות של כל הצמתים הרלוונטיים. ואז אם הגענו לשורש מחזירים 6 סך השינויים שעשינו, אחרת קוראת ל reBalanceDelete על צומת האבא ועדכון המונה שקיבלנו עד כה +6.

סיבוכיות: כל הפעולות קבועות ולכן O(1).

* reBalanceCase3121:

פונקציית האיזון לאחר מחיקה של המקרה (1,3) - (1,2)והמקרה הסימטרי (3,1) -(2,1). מבצעים גלגול שמאלה לפי הצמתים המתאימים או אם אנחנו עושים את המקרה הסימטרי באופן הפוך (יודעים באיזה מקרה מדובר ע"י משתנה בוליאני בקלט) לאחר מכן מעדכנים את הדרגות של כל הצמתים הרלוונטיים. ואז אם הגענו לשורש מחזירה 3 סך השינויים שעשינו, אחרת קוראת ל reBalanceDelete על צומת האבא ועדכון המונה שקיבלנו עד כה +3.

סיבוכיות: כל הפעולות קבועות ולכן O(1).

* reBalanceCase3111:

פונקציית האיזון לאחר מחיקה של המקרה (1,3) - (1,1)והמקרה הסימטרי (3,1) -(1,1). מבצעים גלגול שמאלה לפי הצמתים המתאימים או אם אנחנו עושים את המקרה הסימטרי באופן הפוך (יודעים באיזה מקרה מדובר ע"י משתנה בוליאני בקלט) לאחר מכן מעדכנים את הדרגות של כל הצמתים הרלוונטיים. זאת פעולת איזון סופית לכן מחזירים 3 סך עלות איזון של הפעולה.

סיבוכיות: כל הפעולות קבועות ולכן O(1).

* deleteRetrieve:

הפונקציה מקבלת צומת ובודקת כיצד עלינו למחוק אותו ע"י סיווג המצב שלו – עלה \ צומת אונארי שמאלי או ימני \ או שיש לו משפחה (איזה חמודים 😊) לאחר הסיווג הפונקציה קוראת לפונקציית המחיקה המתאימה, שתמחק ותחזיר את הצומת שצריך להתחיל את תהליך איזון העץ ממנו שאותו נחזיר לקורא).

סיבוכיות: כל הבדיקות והקריאות הינם קבועות ולכן O(1).

* deleteRetrieveFamily:

מוחקת צומת מהעץ במקרה שמדובר בצומת עם 2 בנות. המחיקה מתבצעת באופן שנראה בהרצאה – ניקח את ה- successor של הצומת הנמחק – נמחק אותו (נשמור במשתנה זמני את הצומת שצריך לעבור איזון בעקבות המחיקה) מהעץ ונחליף אותו אם הצומת המבוקש (קיומו של ה-successor מובטח כי יש לצומת המבוקש 2 בנים) ולבסוף נחזיר את הצומת שצריך איזון (אם מדובר במקרה קצה של שורש נעדכן את המצביע).

סיבוכיות: כל הבדיקות מתרחשות בזמן קבוע ולכן O(1).

* deleteRetrieveRight או deleteRetrieveLeft:

מבצעת מחיקה לצומת אונארי שמאלי או ימני כפי שנראה בהרצאה ע"י 'דילוג' על הצומת, במקרה קצה של שורש נעדכן את השורש, נחזיר את צומת האב שצריך לעבור איזון.

סיבוכיות: כל הבדיקות מתרחשות בזמן קבוע ולכן O(1).

* deleteRetrieveLeaf:

מבצעת מחיקה לצומת עלה כפי שנראה בהרצאה ע"י 'דילוג' על הצומת וחיבור האב עם הצומת הוירטואלי, במקרה קצה של שורש נעדכן את השורש, נחזיר את צומת האב שצריך לעבור איזון.

סיבוכיות: כל הבדיקות מתרחשות בזמן קבוע ולכן O(1).

* Successor:

בהינתן צומת הפונקציה מחפשת את ה- successor באופן דומה כפי שנראה בהרצאה: אם יש לצומת בן ימני אז successor הוא המינימום בתת העץ הימני ולכן נמצא אותו ע"י קריאה לפונק' myMin על תת העץ הימני פונקציה זו מחזירה את הצומת המינימלי בתת עץ. אם אין נעלה במעלה העץ עד שנגיע לצומת שהגענו אליו משמאל ואותו נחזיר, אם אין והגענו לnull נחזיר null .

סיבוכיות: סה"כ סיבוכיות במקרה הגרוע נעלה מהמקסימום את כל גובה העץ log(n) או נרד כמעט את כל גובה העץ ע"י myMin.

* myMax / myMin:

הפונקציה מקבלת צומת ומחזירה את המינימום \ מקסימום בתת העץ שלו ע"י ירידה שמאלה \ ימינה בתת העץ כל עוד אפשר.

סיבוכיות: במקרה הגרוע נרד את כל גודל העץ log(n).

* Split:

נחפש את המיקום של המפתח שקיבלנו נסמן מיקום זה ב-X, נשמור מצביע לתת העץ הימני , נסמנו ב- R והשמאלי שלו, נסמנו ב- L ולאחר מכן נגרום לתתי עץ אלו להדמות לעץ ע"י שנהפוך את המצביע לאב שלהם ל – null כמו שורש. נשמור מצביע לאב של X נסמנו PARENT

לאחר מכן נתחיל לטפס במעלה העץ שבכל איטרציה נעבוד באופן הבא:

תחילה נשמור מצביע לאב של PARENT ולאחר מכן נשנה את האב של PARENT ל- null.

נבדוק אם X הוא בן שמאלי או ימני של PARENT ולפי זה ננתק את הקשר של PARENT עם הבן הימני או השמאלי בהתאמה. נעדכן את PARENT לפי השינויים שנוצרו לו בתת העץ ונקרא ל פונקציית העזר של join – join\_in שמחברת את parent עם הבנים החדשים שלו ומחזירה את השורש של העץ החדש נשמור משתנה זה במצביע R או L בהתאמה.

לאחר שינויים אלו נעדכן את PARENT לאב שלו ואת X ל- PARENT.

כשסיימנו לעלות במעלה העץ ניצור מהשורשים שיש לנו אובייקט עץ חדש ע"י קריאה לבנאי המתאים שמקבל צומת – בנאי זה מאפס גם את המינימום והמקסימום של העץ החדש ע"י חיפושם. – נוודא שהשורשים מעודכנים ונחזיר מערך של 2 תתי העצים בסדר המתאים.

סיבוכיות: כפי שראינו בהרצאה O(logn), הפונקצייה מטפסת במעלה העץ ומבצעת O(logn) איטרציות, הקריאות לפונקציה join\_in חסומות ע"י O(logn) ואיפוס השדות min \ max בסוף של כל עץ חדש חסומות גם כן ע"י O(logn).

ולכן סה"כ נקבל O(logn).

המחלקה AVLnode:

כללי:

במחלקה זו נתחזק לכל אובייקט של המחלקה את השדות הבאים:

* + Key – המפתח של הצומת
  + Info – המידע ששמור בצומת
  + Left – הבן השמאלי
  + Right – הבן הימני
  + Height – גובה הצומת
  + Parent – ההורה של הצומת
  + Size – גודל תת העץ של הצומת

VirtualNode – שדה סטטי, יהיה הצומת הדיגיטלי שיהווה "פקק" לכל העלים וצמתים האונאריים בעץ

כל הפונקציות במחלקה פועלות בצורה קבועה שאינה תלויה בקלט כלומר כולן פעולות בסיבוכיות O(1).

**פונקציות:**

* AVLnode:

בנאי ריק למחלקה שמאתחל אך ורק צומת דיגיטלי – המפתח יהיה -1, הגודל שלו 0, הגובה -1 ושאר השדות הינם null.

* AVLnode(key,info):

בנאי למחלקה שמאתחל אך ורק צמתים (עלים) שאינם דיגיטליים – מאתחל את השדות בהתאם לקלט.

* isRealNode:

בודקת אם צומת אינו וירטואלי ע"י בדיקה אם המפתח שלו שונה מ -1.

* getKey:

מחזירה את המפתח של צומת

* getValue:

מחזירה את הערך של צומת

* setLeft \ setRight:

מעדכנת את הבן השמאלי \ הימני בהתאם לקלט

* getLeft \ getRight:

מחזירה את הבן שמאלי \ הימני של צומת בהתאמה

* setParent:

מעדכנת את ההורה של צומת בהתאם לקלט

* getParent:

מחזירה את ההורה של צומת

* setHeight:

מעדכנת את הגובה של צומת בהתאם לקלט

* getHeight:

מחזירה את הגובה של צומת

* getRankLeft או getRankRight:

מחזירה את הפרש הגבהים של צומת ביחס לבן השמאלי \ ימני שלו

* setHeightAlone:

מעדכנת את הגובה של הצומת בהתאם לבנים שלו, ע"י בדיקת הגבהים שלהם, לקיחת המקסימום והוספת 1.

אם מדובר בצומת דיגיטלי – לא נעשה דבר

* setSizeAlone:

מעדכנת את גודל הצומת בהתאם לגודל בניו +1.

אם מדובר בצומת דיגיטלי – לא נעשה דבר

* updateNode:

מעדכנת את הגובה והגודל של צומת ע"י קריאה לפונקציות setHeightAlone, setSizeAlone.

**חלק תיאורטי – ניסויי:**

שאלה 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מספר סידורי | מספר חילופים במערך ממוין הפוך | עלות החיפושים במיון AVL עבור מערך ממוין הפוך | מספר חילופים במערך מסודר אקראית | עלות החיפושים במיון AVL עבור מערך מסודר אקראית |
| 1 | 1999000 | 38884 | 978828 | 31674 |
| 2 | 7998000 | 85764 | 4004900 | 73717 |
| 3 | 31996000 | 187524 | 15960205 | 161301 |
| 4 | 127992000 | 407044 | 64038517 | 361877 |
| 5 | 511984000 | 878084 | 255146028 | 762637 |

א.

ב.

נוכיח כי מספר החילופים:

במערך ממוין הפוך לאיבר הראשון יש n-1 חילופים, לאיבר השני n-2 וכן הלאה. לאיבר האחרון אין חילופים כלל ולכן זוהי סדרה חשבונית עם הפרש 1 כפי שהצגנו.

עלות החיפושים:

נוכיח כי עלות החיפושים הינה

נחשב באופן מדוייק את עלות החיפוש של האיבר הi-:

בשלב זה יש בעץ i-1 איברים ולכן גובה העץ הינו . האיבר שנכניס הינו האיבר מינימלי בעץ לאחר הכנסתו ולכן בהינתן מצביע למקסימום נעלה במעלה העץ עד השורש ונרד עד הבן השמאלי ביותר. ולכן עלות החיפוש תהיה לכל היותר פעמיים גובה העץ .

נחסום מלמעלה:

נסכום את עלות כל החיפושים כפי שתיארנו למעלה ונקבל:

כעת, נחסום מלמטה:

נתבונן בתת סדרה קשה – הכנסת n/2 האיברים האחרונים. באופן דומה, נסכום ונקבל שעלות ההכנסה היא תמיד לפחות:

בסה"כ קיבלנו שעלות סך החיפושים היא .

ג. נציג את התוצאות:

מספר חילופים במערך ממוין הפוך:

בסעיף ב מצאנו כי מספר החילופים הינו ריבועי ב-n. ואכן, ניתן לראות בגרף את הקשר הריבועי. בנוסף נבצע טרנספורמציית שורש לציר הY (מספר החיפושים) ונקבל קשר ליניארי.

Chart, line chart

Description automatically generated

the correlation R^2 is: 0.9491598820436215

Chart, line chart

Description automatically generatedובטרנספורמציית שורש של ציר הY:

the correlation R^2 is: 0.9999999999999996

כלומר קו הריבועים הפחותים (בכחול) שמסמל את הקשר הליניארי בין ערכי ציר הX לערכי ציר הY, ואת מקדם המתאם – R^2 שמעיד על "איכות" קו הריבועים הפחותים, כלומר כמה הקשר "קרוב" לליניארי, ככל שמקדם המתאם קרוב יותר ל1 הקשר יותר הדוק ואכן שואף ל-1.

עלות החיפושים במיון AVL עבור מערך ממוין הפוך:

בסעיף ב קיבלנו כי עלות החיפושים היא nlog(n). בנוסף, ראינו כי שיפוע קו הריבועים הפחותים (המייצג את הקשר הליניארי) הינו 27. נשים לב כי הפונקציה 27n הינה קירוב טוב ל2nlogn ואכן ניתן לראות זאת בגרף.

Chart, line chart

Description automatically generatedthe correlation R^2 is: 0.9989568182473707

ד. בהינתן רשימה עם h חילופים ניתן לפרק אותם לסכום החילופים של כל איבר ואיבר.

כעת, נתבונן באיבר ה-i בעל חילופים:

ישנם איברים אשר גדולים ממנו ולכן עלינו לעלות מהאיבר המקסימלי עד שנגיע לצומת שתת העץ שלה מכיל איברים- תת עץ זה הוא עץ AVL ולכן הגובה של הינו log(). איבר זה ימוקם כאיבר השמאלי ביותר בתת עץ זה (המינימלי). לכן, עלות החיפש עבורו תהיה 2log().

כלומר, בסה"כ עלות החיפושים תהיה:

כאשר אי השיוויון הראשון נובע מאי-שיוויון הממוצעים.

נשים לב שכאשר h<n נקבל למעשה שכן עלות החיפוש היא לפחות n שכן חיפוש מינימילי לכל איבר עולה לנו לפחות 1.

ה. נתבונן בהכנסה של מערך ממויין בסדר עולה אז כל חיפוש עולה לנו ולכן סה"כ נקבל

כלומר לא ניתן לתת חסם הדוק מכיוון שכפי שהראינו בסעיף הקודם ייתכן כי ואז נקבל כי עלות החיפוש היא . ולכן לא ניתן לתת חסם הדוק.

שאלה 2:

א. טבלה!

ב. נוכיח כי עלות join ממוצע לשני התרחישים הינו amortized O(1).

נוכיח בשיטת הבנק:

תהי צומת במרחק i מהשורש.

נזכיר כי במהלך פעולת הjoin אנו מטפסים במעלה העץ עד השורש (i טיפוסים) .

בכל טיפוס, אנו מבצעים פעולת join של האיברים הקטנים (השמאליים) או הגדולים (הימניים).

נפעל באופן הבא: נצבור עבור כל טיפוס 2 אסימונים בבנק. אם עשינו join של האיברים בקטנים (טיפסנו שמאלה) אז האסימון שהפקדנו ישמש לjoin עתידי של האיברים הגדולים ולהיפך.

נוכיח כי עבור פעולת join יקרה יהיו מספיק אסימונים בבנק:

תהי פעולת join בעלות K. בה"כ פעולת join של איברים גדולים (ימנית).

על מנת להגיע להפרש גבהים k עלינו לעלות במעלה העץ k פעמים מבלי לעשות join ימני, כלומר לעלות k פעמים שמאלה. ולכן יש k אסימונים בבנק.

כמו כן, עבור כל פעולה שמאלה כזו שילמנו לכל היותר 2- הפרש גבהים של אב ובן.

לכן סה"כ שילמנו בכל פעולה לכל היותר 4 אסימונים, i פעמים וביצענו i פעמים join ולכן סה"כ:

באופן דומה, עבור האיבר המקסימלי בתת העץ השמאלי:

בתרגול הוכחנו שמרחק עלה מהשורש הוא לכל הפחות  *ולכל היותר*

ניתן לראות כי בתרחיש זה, נטפס שמאלה O(logn) פעמים ולכל טיפוס נצבור אסימון.

לבסוף יהיו בבנק לפחות אסימונים עבור פעולת הjoin היקרה של תת העץ הימני הריק של האיבר עם תת העץ הימני של השורש שגובהו O(logn). לכן גם במקרה זה גם נקבל זמן O(1).

ניתן להבחין כי הניתוח התיאורטי תואם לתוצאות שקיבלנו בסעיף א, שאכן הjoin הממוצע עלה בשני התרחישים בין 1-2. כלומר .

ג. ננתח את עלות ה-join המקסימלי עבור split של האיבר המקסימלי בתת העץ השמאלי:

באופן דומה להסבר בסעיף ב': במהלך פעולת ה-split אנו מטפסים במעלה העץ ונבצע פעולות join, מכיוון שאנו מבצעים split על האיבר המקסימלי בתת העץ השמאלי אז עד שנגיע לשורש אנו מבצעים פעולת join של האיברים הקטנים (השמאליים) בלבד ולכן בהגיענו לשורש נבצע פעולת join בין תת העץ הימני של העץ והעץ הריק שהינו הבן הימני של הצומת שעליו אנו עושים את ה-split. מכיוון שתת העץ הימני של העץ הינו עץ AVL הוא בגובה של O(logn) ולכן join זה יעלה לנו O(logn) תמיד.

ואכן ניתוח תיאורטי זה עולה בקנה אחד עם התוצאות מסעיף א', ניתן לראות כי עלות ה-join המקסימלי עלה