**תיעוד וסיבוכיות המתודות – תרגיל מעשי 1**

**מגישים:**

|  |  |  |
| --- | --- | --- |
|  | **מגיש א'** | **מגיש ב'** |
| **שם פרטי** | אדם | אסף |
| **שם משפחה** | טובי | מיכאלוביץ' |
| **תעודת זהות** | 215334822 | 200637270 |
| **שם משתמש אוניברסיטאי** | adamtuby | michaelovits |

**סיבוכיות ותיעוד:**

**public** **class** AVLTree

**private** **final** AVLNode virtual

משתנה זה הוא משתנה קבוע, שמאותחל להיות צומת וירטואלי בבנאי המחלקה AVLTree. כל פעם שנרצה להשתמש בנוכחותו של צומת וירטואלי לשם שלמות הפונקציונליות, נשתמש במשתנה virtual כמצביע לצומת וירטואלי שכזה. משתנה זה נועד בכדי לחסוך בזיכרון ובאיתחול של מלא צמתים וירטואלים חדשים.

**public** AVLNode root

משתנה זה יחזיק את המצביע לשורש של העץ. בבנאי המחלקה AVLTree, משתנה זה מאותחל להיות virtual, שכן יצירת עץ חדש תיצור עץ ריק, ובחרנו להציג צמתים ריקים כצמתים וירטואלים.

**public** AVLTree()

**סיבוכיות:** O(1)

זה הוא בנאי המחלקה AVLTree.   
הוא מאתחל את המשתנה virtual להיות צומת וירטואלי, ומאתחל את השורש root להיות virtual.

**public** **boolean** empty()

**סיבוכיות:** O(1)

פונקציה זו בודקת האם העץ ריק, היא עושה זאת באמצעות הבדיקה אם השורש של העץ הוא צומת וירטואלי או לא. בדיקה זו לוקחת אכן O(1) והיא מתבצעת על ידי מתודת העזר   
**public** **boolean** isRealNode() השייכת למחלקה AVLNode. פירוט של מתודה זו יהיה בהמשך המסמך. ובכן, שיטה זו נכונה מכך שכאשר אנו יוצרים מופע חדש של המחלקה AVLTree (כלומר יוצרים עץ), הוא מייצר שורש שהינו עץ וירטואלי. וכאשר אנו נכנסים איברים לעץ, צמתים אמיתיים יתפסו את מקומם של הצמתים הוירטואלים, ולכן השורש של העץ אמור להיות צומת אמיתי לאחר פעולת ההכנסה הראשונה לעץ. מכך שאם השורש הינו עץ וירטואלי אז העץ ריק.

**public** Boolean search(**int** k)

**סיבוכיות:** O(log n)

נשים לב שמתודה זו לא מבצעת כמעט כלום אלא רק משתמשת בערך המוחזר שמחזירה מתודת העזר plain\_search(**int** k) שנתעד ממש עוד רגע. מתודת עזר זו מחזירה את הצומת בעץ שהמפתח שלו הוא k, ובמידה ולא קיים צומת כזה בעץ, היא תחזיר null. אז במידה ומתודת עזר זו מחזירה אכן צומת, נחזיר את ה info של אותו צומת. במידה והצומת null, נחזיר null. סיבוכיות מתודה זו היא כסיבוכיות מתודת העזר. סיבוכיות זאת היא אכן הסיבוכיות היעילה ביותר עבור חיפוש בעץ AVL. ראינו זאת בהרצאה.

**private** AVLNode plain\_search(**int** k)

**סיבוכיות:** O(log n)

מתודה זו היא מתודת עזר בשביל Search, הנועדה למנוע שכפולי קוד של המתודה Search, שכן מתודת Search מחזירה את הערך בוליאני שהצומת עם המפתח k מחזיק, וכן plain\_search מחזירה את המצביע לצומת המחזיק את המפתח k.   
מתודה זו מבצעת חיפוש נורמטיבי בעץ בינארי כמו שראינו בכיתה. ובכן, כיוון שמדובר בעץ AVL, אז גובה העץ הוא O(log n), ולכן נבצע לכל היותר O(log n) איטראציות בשיטת החיפוש שלמדנו בכיתה. כל איטראציה בחיפוש זה מכילה מספר קבוע של השוואות (לכל היותר 2). במידה ומצאנו צומת המקיימת את הנדרש בעץ, נחזיר את הצומת. במידה ולא נמצא הצומת, יוחזר null. על כן סיבוכיות מתודה זו היא כנ"ל.

**private** AVLNode rightRotation(AVLNode N)

**סיבוכיות:** O(1)

מתודה זו והמתודה הבאה נועדו לבצע סיבובים בעץ מסויים בהינתן הצומת שעליה צריך לבצע את הסיבוב. המתודות insert ו delete ישתמשו בה בתהליך תיקון עברייני הAVL. היא מבצעת גלגול ימני יחיד, בדיוק כמו שראינו בכיתה. כלומר גלגול כאשר ה balance factor הינו גדול ממש מ-1.   
נעשה זאת באמצעות שינויי פויינטרים בסיסיים של הבנים של הצומת הנתון.  
סיבוכיות מתודה זו היא בבירור O(1), שכן נעשית כמות קבועה של שינויי פויינטרים ושינוי מבנה העץ ששורשו נמצא בצומת הנתונה כארגומנט, N.

פירוט רחב יותר של הגילגול הימני:  
- יהי L הבן השמאלי של N.  
- יהי subT2 תת העץ הימני של L.  
1. נגדיר את הבן הימני החדש של L להיות N.  
2. נגדיר את הבן השמאלי החדש של L להיות subT2.  
3. נעדכן את שאר הפרטים של הצמתים N, L לפי הבנים החדשים שלהם, לפי הצורך. כלומר נעדכן את הגבהים שלהם, את הגודל שלהם (גודל העץ ששורשו הוא הצומת), ופרטים נוספים כגון ההורה שלהם, או המשתנה XOR החדש שהגדרנו במחלקה AVLNode, עליו נפרט בהמשך.

**private** AVLNode leftRotation(AVLNode N)

**סיבוכיות:** O(1)   
מתודה זו עושה תהליך זהה לחלוטין כמו המתודה הקודמת rightRotation, אך גלגול שמאלי ולא ימני

insert(**int** k, **boolean** i)

**סיבוכיות:** O(log n)

מתודה זו תכניס צומת חדש עם המפתח k ועם הערך הבוליאני i. היא בנוסף תחזיר את מספר הגילגולים ושינויי הגובה שנעשו בתהליך תיקון עברייני ה AVL לאחר ההכנסה. נשים לב כי מתודה זו כמו search אינה המתודה העיקרית, אלא רק מעטפת למתודת העזר הרקורסיבית insert\_rec שנתעד ממש עוד רגע. במידה וקיים כבר צומת בעץ עם מפתח k, היא מחזירה -1 ולא עושה כלום (ניתן לבדוק אם קיים כבר הצומת בעץ באמצעות plain\_search). במידה וזה לא נמצא בעץ, היא תיצור מערך changes[] שמחזיק ערך integer אחד בלבד שיכיל את מספר הגלגולים ועדכוני הגבהים במהלך תיקון עברייני ה AVL.  
כעת נעדכן את שורש העץ להיות הצומת החדש שתחזיר קריאת המתודה:  
insert\_rec(**this**.getRoot(), **new** AVLNode(k,i,0), changes)

ונחזיר כעת את changes[0].   
סיבוכיות המתודה היא כמובן כסיבוכיות הקריאה של מתודת העזר עם הפרמטרים הנ"ל. סיבוכיות זאת היא אכן היעילה ביותר עבור הכנסה בעץ AVL. ראינו זאת בהרצאה.

insert\_rec(AVLNode node, AVLNode new\_node, **int**[] change\_info)

**סיבוכיות:** O(log n)

מתודה זו היא מתודת העזר והמתודה הרקורסיבית שמשומשת על ידי המתודה insert.   
נתחיל בלהסביר מה הם הפרמטרים/ארגומנטים;  
**node** – זו יהיה הצומת הנוכחי שעליו אנו מתמקדים. כלומר, במידה וקראנו למתודה זו כאשר node הוא צומת כלשהו בצומת לאחר ההכנסה, אז קריאה זו תדאג לבדוק אם הוא עבריין AVL לאחר ההכנסה (שכן תהליך ההכנסה מתבצע לפני תהליך עדכון השדות והגלגולים), ובמידה וכן, תבצע גלגולים כנדרש באמצעות מתודות העזר rightRotation(AVLNode N) ובאמצעות leftRotation(AVLNode N) שנתעד ממש עוד רגע. אם לא היה עבריין AVL, תבדוק אם גובהו השתנה. במידה וכן, תעלה את change\_info[0] באחד. במידה וכן היה עבריין AVL, תעלה את change\_info[0] באחד. בנוסף לזאת המתודה תעדכן את פרטיו, כגון גובהו, גודל העץ ששורשו נמצא בצומת המדובר, ושאר השדות שצומת AVLNode מחזיר בהם (כגון XOR). תהליך העדכון ייתבצע באמצעות מתודת עזר של המחלקה AVLNode, הנקראת update\_info() שתתועד בהמשך.

**new\_node** – זה יהיה מצביע לצומת החדש שאנו רוצים להכניס לעץ. צומת זה נוצר בקריאה הראשונה למתודת עזר זו מתוך המתודה insert, על ידי **new** AVLNode(k,i,0).

**change\_info** – ארגומנט זה יהיה מצביע למערך שמכיל בתוכו את מספר הצמתים שדרשו גלגול או עדכון גובה. נדרש לשלוח ארגומנט זה בכל קריאת רקורסיה, שכן אני לא מספיק בהכרה עם משתנים גלובליים בjava.

מתודה זו מחזירה את הצומת node, לאחר שעבר עדכון שדות ותיקון במידה והיה עבריין AVL לאחר ההכנסה.   
**ניגש לסיבוכיות:** מבוצעות O(log n) קריאות רקורסיה למתודה זו, שכן תהליך ההכנסה מבקר ב O(log n) צמתים בלבד, וכן אנחנו קוראים לקריאת רקורסיה חדשה אך ורק כאשר אנחנו מבקרים בצומת חדש בתהליך ההכנסה (ביקור משמע המעבר בצומת בתהליך החיפוש של המיקום הטריוויאלי עבור הצומת החדש).  
מתודה זו תבדוק אם node הוא עבריין AVL באמצעות מתודת העזר של המחלקה הפנימית AVLNode, הנקראת **public** **int** BalanceFactor() מתודה זו מחזירה את ה Balance Factor של הצומת, ומבצעת גלגולים לפי הערך שלו ולפי דרישות שונות ועמוקות יותר של גלגולים.

delete(**int** k)

**סיבוכיות:** O(log n)

מתודה זו פועלת בתהליך די דומה למתודת insert. מטרת מתודה זו היא למחוק את הצומת בעץ שמחזיק את המפתח k. במידה ולא קיים צומת כזה, תחזיר את הערך -1.  
זאת מתודת המעטפת למתודה הרקורסיבית הבאה delete\_rec, העושה את 'רוב' העבודה.  
ובכן, זאתי מתודה שמטרתה היא למחוק את הצומת בעץ שהמפתח שלו הוא k. במידה ולא קיים כזה צומת, תחזיר את הערך -1.   
ובכן, נסביר איך היא עושה זאת:  
ראשית היא תחפש אם המפתח אכן נמצא בעץ על ידי המתודה plain\_search(k), שאמורה להחזיר ערך שונה מ null אם המפתח k נמצא בעץ. במידה והערך הוא null, תחזיר -1, אחרת, תקרא למתודה   
delete\_rec(**this**.getRoot(), k, changes) שתחזיר משתנה מסוג AVLNode, אותו נעדכן להיות השורש של העץ.   
נשים לב שאנחנו מתחזקים מערך מגודל 1 שנקרא changes שמטרתו לתחזק פויינטר לערך integer. את מערך זה נשלח כארגומנט לכל קריאה רקורסיבית של delete\_rec. בסוף ריצת הפונקצייה delete, נחזיר את changes[0], שיכיל את מספר הצמתים שדרשו עדכון גובה או רוטציה מכל סוג שהיא.

delete\_rec(AVLNode node, **int** key, **int**[] change\_info)

**סיבוכיות:** O(log n)

מתודה זו היא מתודת העזר והמתודה הרקורסיבית אותה delete עוטפת.  
נתחיל בלהסביר מה הם הפרמטרים/הארגומנטים;  
**node** – זו יהיה הצומת הנוכחי שעליו אנו מתמקדים. כלומר, במידה וקראנו למתודה זו כאשר node הוא צומת כלשהו בצומת לאחר ההכנסה, אז קריאה זו תדאג לבדוק אם הוא עבריין AVL לאחר המחיקה (שכן תהליך המחיקה מתבצע לפני תהליך עדכון השדות והגלגולים), ובמידה וכן, תבצע גלגולים כנדרש באמצעות מתודות העזר rightRotation(AVLNode N) ובאמצעות leftRotation(AVLNode N) שנתעד ממש עוד רגע. אם לא היה עבריין AVL, תבדוק אם גובהו השתנה. במידה וכן, תעלה את change\_info[0] באחד. במידה וכן היה עבריין AVL, תעלה את change\_info[0] באחד. בנוסף לזאת המתודה תעדכן את פרטיו, כגון גובהו, גודל העץ ששורשו נמצא בצומת המדובר, ושאר השדות שצומת AVLNode מחזיר בהם (כגון XOR). תהליך העדכון ייתבצע באמצעות מתודת עזר של המחלקה AVLNode, הנקראת update\_info() שתתועד בהמשך.

**key** – ארגומנט נשאר זהה לאורך כל קריאות הרקורסיה. החלק הראשון של תהליך הרקורסיה הוא למחוק את הצומת שמחזיק את המפתח key. זה מבוצע על ידי תהליך הדומה למתודת החיפוש שמימשנו (plain\_search). לכן לאורך כל קריאות הרקורסיה נסחוב את ארגומנט זה וברגע שנגיע לצומת בה node מחזיק מפתח השווה לkey, נבצע תהליך מחיקה כנדרש. תהליך המחיקה יוסבר בהמשך.

**change\_info** – ארגומנט זה יהיה מצביע למערך שמכיל בתוכו את מספר הצמתים שדרשו גילגול או עדכון גובה. נדרש לשלוח ארגומנט זה בכל קריאת רקורסיה, שכן אני לא מספיק בהכרה עם משתנים גלובליים בjava.

מתודה זו מחזירה את הצומת node, לאחר שעבר עדכון שדות ותיקון במידה והיה עבריין AVL לאחר המחיקה.

**תהליך המחיקה:**  
במידה והגענו לצומת node שמחזיק מפתח השווה לkey, נצטרך למחוק אותו מהעץ. נבצע זאת באופן הבא:   
נרצה לבדוק אם בתת העצים שמחזיק node, יש תוכן שאינו משתנים וירטואלים. כלומר תוכן שצריך להשתמר בעץ. ובכן:  
- אם **אין** ל node בנים (בנים שאינם צמתים וירטואלים), אז נחליף את node בצומת וירטואלי (כלומר node = virtual). ונמשיך הלאה.  
- אם לnode יש **בן אחד בלבד**, נחליף את node בבן הזה ונמשיך הלאה.  
(כלומר node = node.left או node =node.right)  
- אם לnode יש **שני בנים**, נמצא את הsuccessor של node. הוא יהיה הצומת המינימלי בתת העץ הימני של node, שכן תת העץ הימני אינו ריק, ונחליף את node עם ה successor שלו   
(כלומר node = successor(node)), ונמשיך הלאה.

ההחלפות המדוברות כנ"ל, הם מבוצעות על ידי המתודה replace(AVLNode node, AVLNode tmp)  
שתתועד ממש עוד רגע.  
  
**ניגש לסיבוכיות:** מבוצעות O(log n) קריאות רקורסיה למתודה זו, שכן תהליך המחיקה מבקר ב O(log n) צמתים בלבד, וכן אנחנו קוראים לקריאת רקורסיה חדשה אך ורק כאשר אנחנו מבקרים בצומת חדש בתהליך המחיקה (ביקור משמע המעבר בצומת בתהליך החיפוש של המיקום הטריוויאלי עבור הצומת החדש).  
מתודה זו תבדוק אם node הוא עבריין AVL באמצעות מתודת העזר של המחלקה הפנימית AVLNode, הנקראת **public** **int** BalanceFactor() מתודה זו מחזירה את ה Balance Factor של הצומת, ומבצעת גלגולים ותיקונים לפי הערך שלו (הBalance Factor) ולפי דרישות שונות ועמוקות יותר של גלגולים.

replace(AVLNode node, AVLNode tmp)

**סיבוכיות:** O(1)

מתודה זו היא מתודת עזר עבור המתודה delete\_rec. תפקידה היא להעתיק את הנתונים של הצומת tmp אל הצומת node, כאשר כביכול מוחקת את הצומת node.  
ובכן, היא תיקח את הצומת node, תשמור את כל הנתונים והבנים וההורה של node מראש, ואז תיישם אל node מצביע חדש לגמרי לצומת חדש לגמרי שיחזיק את המפתח ואת הערך הבוליאני של הצומת tmp, ותעדכן את כל הנתונים של הצומת חדש כגון: בנים, הורה, גובה, XOR, וכו' (העדכון ייתבצע בעזרת update\_info()).  
ובכן, כל מה שמתודה זו עושה זה מספר קבוע של שינויי פויינטרים ומספר (קבוע של) פעולות נוספות, על כן סיבוכיות מתודה זו היא O(1).

min()

**סיבוכיות:** O(log n)

מתודה זו מחזירה את הערך הבוליאני אותו מחזיק הצומת עם המפתח הכי קטן בעץ. במידה והעץ ריק, היא תחזיר null. היא עושה זאת על ידי לולאה איטראטיבית, המתחילה בשורש של העץ, ובכל איטראציה ממשיכה לבן השמאלי של הצומת עליו אנחנו נמצאים. ברגע שמגיע לצומת שהבן השמאלי שלו לא קיים, תעצור ותחזיר את הערך הבוליאני שמחזיק הצומת. סיבוכיות מתודה זו היא כעומק העץ, ולכן הסיבוכיות היא O(log n), שכן מדובר בעצי AVL.

max()

**סיבוכיות:** O(log n)

מתודה זו מחזירה את הערך הבוליאני אותו מחזיק הצומת עם המפתח הכי גדול בעץ. במידה והעץ ריק, היא תחזיר null. היא עושה זאת על ידי לולאה איטראטיבית, המתחילה בשורש של העץ, ובכל איטראציה ממשיכה לבן הימני של הצומת עליו אנחנו נמצאים. ברגע שמגיע לצומת שהבן הימני שלו לא קיים, תעצור ותחזיר את הערך הבוליאני שמחזיק הצומת. סיבוכיות מתודה זו היא כעומק העץ, ולכן הסיבוכיות היא O(log n), שכן מדובר בעצי AVL.

keysToArray()

**סיבוכיות:** O(n)

מתודה זו תפקידה היא להחזיר מערך המכיל את המפתחות הנמצאות בעץ בסדר ממויין. היא עושה זאת באמצעות היותה המתודה העוטפת למתודה הרקורסיבית שעושה את רוב העבודה: nodesToArray(**this**.getRoot(), arr, 0) , כאשר arr הוא מערך של צמתים שיצרנו מראש לפני הקריאה, שגודלו הוא **this**.size() . ובכן, אם גודל המערך הוא כמספר הצמתים בעץ, אז המערך יוכל להכיל דיוק את מספר הצמתים בעץ. הארגומנט הראשון נשלח להיות השורש של העץ, שכן מתודת העזר הנ"ל מחזירה מערך המכיל את כל הצמתים שנמצאים בעץ ששורשו הוא הארגומנט הראשון, בסדר ממויין לפי המפתחות של הצמתים. בנוסף לזאת, נשים לב ששלחנו את הארגומנט השלישי להיות 0. הארגומנט השלישי מחזיק את מספר המקומות התפוסים במערך, ולכן צריך להיות 0 בקריאה הראשונה. בסוף הריצה של הקריאה, arr עבר מניפולציות בתוך הקריאות הרקורסיביות כך שהוא מחזיר את הרצוי, ולכן נחזיר את arr.   
בסוף הריצה של מתודת העזר, נאתחל מערך חדש array כגודל העץ של **integers**, ובאופן איטראטיבי, נעבור על כל האינדקסים i בין 0 לגודל העץ, כאשר בכל איטראציה, נדאג שarray[i] יהיה שווה למפתח של הצומת arr[i]. ואז נחזיר את arr בסוף ריצת המתודה.  
**סיבוכיות מתודה** זו היא כסיבוכיות המתודה nodesToArray(**this**.getRoot(), arr, 0), אותה נתעד עוד רגע, ולכן, סיבוכיות מתודה זו, היא O(n).

infoToArray()

**סיבוכיות:** O(n)

מתודה זו תפקידה היא להחזיר מערך המכיל את הערכים הבוליאנים שמחזיקים הצמתים בעץ, בסדר ממוין לפי מפתחות הצמתים. היא עושה זאת באמצעות היותה המתודה העוטפת למתודה הרקורסיבית שעושה את רוב העבודה: nodesToArray(**this**.getRoot(), arr, 0) , כאשר arr הוא מערך של צמתים שיצרנו מראש לפני הקריאה, שגודלו הוא **this**.size() . התהליך הינו זהה לחלוטין כמו המתודה keysToArray().  
בסוף הריצה של מתודת העזר, נאתחל מערך חדש array כגודל העץ של **boolean**, ובאופן איטראטיבי, נעבור על כל האינדקסים i בין 0 לגודל העץ, כאשר בכל איטראציה, נדאג שarray[i] יהיה שווה למפתח של הצומת arr[i]. ואז נחזיר את arr בסוף ריצת המתודה.  
**סיבוכיות מתודה** זו היא כסיבוכיות המתודה nodesToArray(**this**.getRoot(), arr, 0), אותה נתעד עוד רגע, ולכן, סיבוכיות מתודה זו, היא O(n).

**private** **int** nodesToArray(AVLNode N, AVLNode[] arr, **int** pos)

**סיבוכיות:** O(n)

מתודה זו היא המתודה הרקורסיבית אותה keysToArray()ו-infoToArray() עוטפות.  
תפקידה היא להעביר את arr מניפולציות כך שכל הצמתים שנמצאים בעץ ששורשו N יהיו בarr בסדר ממויין לפי מפתחותיהם. מתודה זו מחזירה את pos (ובכן, מתודה זו בעצם מחזירה את כמות המקומות שכבר תפוסים במערך). ובכן, נראה איך היא עושה זאת:  
במידה ול-N יש בן שמאלי, נקרא ל keysToArray\_rec(N.getLeft(), arr, pos), ונשנה את pos לערך שקריאה זו תחזיר. במידה והבן השמאלי ריק, היא תמשיך הלאה מבלי לעשות כלום.  
ובכן, אחרי שלב זה, מה שנעשה הוא arr[pos++] = N, זה בעצם דואג ש arr[pos] = N, וזה גם מקדם את pos ב-1.   
אחרי זה נמשיך הלאה ונעשה אותו הדבר כמו שעשינו עבור הבן השמאלי של N, אך לבן הימני במקום. כלומר pos = nodesToArray(N.getRight(), arr ,pos); ואז נחזיר את pos.   
נשים לב שניתן להוכיח באינדוקציה את נכונות המתודה.

**סיבוכיות המתודה:**  
סיבוכיות המתודה היא במספר הצמתים הקיימים בעץ, שכן מספר קריאות הרקורסיה המבוצעות, הן כמספר הצמתים בעץ, באשר כל קריאת רקורסיה בעלת סיבוכיות O(1) קבועה אם לא מחשיבים את קריאות ההמשך שלה. על כן, סיבוכיות מתודה זו היא O(n).

**public** **int** size()

**סיבוכיות:** O(1)

מתודה זו מחזירה את גודל העץ. היא עושה זאת באמצעות פנייה לשורש של העץ, והחזרת השדה הפרטי size של שורש העץ. זה הוא שדה של המחלקה AVLNode המתוחזק לאורך כל המתודות הממומשות במחלקה זו במחלקה הפנימית AVLNode. עבור צומת N השדה הפרטי size מחזיק את גודל העץ ששורשו הוא N, כנדרש.

**public** AVLNode getRoot()

**סיבוכיות:** O(1)

מתודה זו מחזירה את השורש של העץ. במידה והשורש לא קיים (כלומר צומת וירטואלי), תחזיר null.  
היא ניגשת לשורש באמצעות השדה root של המחלקה הנתונה AVLTree, המכיל את שורש העץ.

**public** **boolean** prefixXor(**int** k)

**סיבוכיות:** O(log n)

מטרת מתודה זו, היא להחזיר את הXOR של כל הערכים הבוליאנים של כל הצמתים בעלי המפתחות הקטנים **או שווים** מ-k. היא עושה זאת באופן הבא:

ובכן, כל צומת מחזיק בשדה הנקרא XOR, המתוחזק על ידי כל מתודות המחלקה והמחלקה AVLNode. עבור צומת N, שדה זה מחזיק את ה XOR של כל הערכים הבוליאנים של כל הצמתים הנמצאים בעץ ששורשו הוא N. ולכן, נרצה להשתמש בשדה זה במהלך תהליך המתודה:  
- נאתחל משתנה curr מסוג AVLNode להיות השורש של העץ, ו out להיות ערך בוליאני שיהיה הערך המוחזר על ידי המתודה. נעדכן את out בהתאם לצרכי המתודה.  
- נבצע לולאת while, עם תנאי עצירה הבאים: אם curr הינו null, נעצור ונחזיר את out.  
אם curr הוא צומת בעל מפתח השווה ל k, נבצע XOR בין out לבין כל הערכים הבוליאנים של הצמתים בעץ ששורשו curr שבעלי מפתחות שקטנים או שווים ל k, ונעדכן את out להיות הערך שהוחזר ב XOR הנ"ל (כלומר, נעשה XOR בין תת העץ השמאלי של curr יחד עם out, ואז עוד XOR יחד עם הערך הבוליאני שcurr מחזיק(.  
בכל איטראציה, במידה המפתח של curr שונה מ-k:  
אם curr בעל מפתח הגדול מ-k:  
נעדכן את curr להיות הבן השמאלי שלו, ונמשיך הלאה לאיטראציה הבאה.  
אם curr בעל מפתח הקטן מ-k:  
נעשה XOR בין out לבין כל הערכים הבוליאנים של כל הצמתים שנמצאים בתת העץ השמאלי של curr, ואז עוד XOR יחד עם הערך הבוליאני של curr, ואז התוצאה שמתקבלת מהXOR הארוך הזה, תהיה הערך החדש של out. לאחר זאת, נעדכן את curr להיות הבן הימני של curr, ונמשיך לאיטראציה הבאה.  
  
בפשטות, מה שעשינו הוא תהליך חיפוש של curr בעץ, ובכל פעם שעברנו רמה למטה, בדקנו איזה אבות של curr בעלי מפתחות הקטנים מ-k. כל הורה שהיה בעל מפתח הקטן מ-k, גם תת העץ השמאלי שלו מלא בצמתים עם מפתחות הקטנים מ-k, ולכן הוספנו ל 'XOR הכולל' את הערך הבוליאני של ההורה, ואת ערך ה XOR של תת העץ השמאלי.  
**סיבוכיות המתודה:**סיבוכיות הלולאה של המתודה הוא כמספר האיטראציות של הלולאה, שכן בכל איטראציה אנו מבצעים מספר פעולות קבוע של פעולות מסיבוכויות O(1). מספר האיטראציות שביצענו הוא לכל היותר כעומק העץ, משמע, O(log n).

**public** AVLNode successor(AVLNode node)

**סיבוכיות:** O(log n)

מטרת מתודה זו היא להחזיר את הצומת עם המפתח '**הכי**' עוקב למפתח של הצומת הנתון node.  
נעשה זאת באופן הבא:  
- אם יש ל node בן ימני:  
ניגש לבן הימני שלו ונמצא את הצומת עם המפתח הקטן ביותר בעץ ששורשו הוא הבן הימני. ובכן, בכדי למצוא זאת, פשוט נבצע לולאת while שמתחילה בבן הימני של node, ועוברת לבן השמאלי של הבן הימני של node, עד שאין כזה. ברגע שאין כזה, תחזיר את הצומת עם המפתח המינימלי כאמור בתת העץ הימני של node. ובכן, צומת זה בעל מפתח קטן יותר מכל הצמתים עם מפתחות גדולים יותר משל node, שכן צמתים עם מפתחות גדולים יותר משל node, יהיו אבות של node, כך שnode נמצא בתת העץ השמאלי שלהם, ולכן אם מצאנו צומת עוקב לnode שהוא בן של node, אז המפתח של הצומת העוקב הזה יהיה תמיד קטן מהאבות העוקבים של node.   
- אם אין לnode בן ימני:  
נעלה מעלה העץ מהצומת node, עד שנגיע לאב כלשהו של node שבעל מפתח שגדול מ node. נעשה זאת פשוט בלולאת while, שתנאי העצירה שלה יהיה כאשר הגענו לאיזשהו אב של node עם מפתח שגדול מ node. כל עוד הלולאה לא נעצרה, נעלה מעלה צומת אחד (כלומר רמה אחת).

אם לא מצאנו איבר עוקב בשני המקרים, נחזיר null.

**סיבוכיות המתודה:**  
ובכן, ראינו בכיתה כי סיבוכיות מתודה זו היא O(log n), וגם לא קשה לראות שזה אכן החסם העליון של סיבוכיות זמן הריצה של המתודה, שכן אנחנו עולים/יורדים לכל היותר log n פעמים במורד/מעלה העץ, שכן עומק העץ הוא .

**public** **boolean** succPrefixXor(**int** k)

**סיבוכיות:** O(log n + n)

מטרת מתודה זו, היא להחזיר את הXOR של כל הערכים הבוליאנים של כל הצמתים בעלי המפתחות הקטנים **או שווים** מ-k. היא עושה זאת באופן הבא:

באופן איטרטיבי, היא מגיעה לאיבר עם המפתח המינימלי של העץ (ראינו זאת במתודות קודמות). האיבר יסומן למשל ב curr.  
לאחר שהיא הגיעה לצומת עם המפתח המינימלי בעץ, תבצע בלולאת while, כמות לא ידוע של פעולות successor ל curr. (כלומר curr = successor(curr)). כאשר תנאי העצירה הוא שהsuccessor הינו null **או** שהמפתח של ה successor הינו גדול מ-k.  
בכל איטראציה של הלולאה, תעדכן את הערך הבוליאני שאמורה להחזיר, output, שמאותחל לfalse בתחילת המתודה. מובן המילה 'תעדכן' כוונתו היא שאנחנו עושים פעולת XOR עם output ועם הערך הבוליאני של curr, והערך המוחזר בפעולת ה XOR, יהיה הערך החדש של output.  
בסוף ריצת הלולאה, נחזיר את curr.  
  
**סיבוכיות המתודה:**נשים לב שראשית מצאנו את הצומת המינימלי בעץ – זה עלה לנו O(log n).  
שנית, ביצענו לכל היותר k פעולות successor מהצומת המינימלי בעץ עד לצומת שמפתחו k.  
ראינו בכיתה כי k פעולות successor בעלות סיבוכיות של O(log n + k), ולכן סיבוכיות מתודה זו היא O(log n + k). (ניתן היה לצמצם את סיבוכיות המתודה ל O(log n + q) כאשר q הוא מספר הצמתים בעלי מפתחות שקטנים או שווים מ-k, אך החסם הכי הדוק שיש ל -q שידוע לנו הוא k)

**public** **class** AVLNode

**private** **int** key **:** מפתח הצומת.  
  
**private** **boolean** info**:** הערך הבוליאני של הצומת  
  
**private** **int** height**:** גובה הצומת (כלומר גובה העץ ששורשו הוא הצומת)  
 **private** AVLNode leftChild**:** הבן השמאלי של הצומת  
 **private** AVLNode rightChild**:** הבן הימני של הצומת  
 **private** AVLNode parent**:** ההורה של הצומת  
 **private** **int** size**:** גודל העץ ששורשו הוא הצומת (כלומר מספר הצמתים בעץ)  
 **private** **boolean** XOR = **false:** הערך XOR של צומת. כלומר ה XOR של כל הערכים הבוליאנים של כל הצמתים שנמצאים בעץ ששורשו הוא הצומת.

**public** AVLNode()

בנאי המחלקה הדיפולטי. בנאי מחלקה זה דואג שהמפתח של הצומת יהיה -1, כך שנדע שהמפתח הינו וירטואלי, וטרם מומש.

**public** AVLNode(**int** key, **boolean** info, **int** height)

בנאי המחלקה לצומת לא וירטואלי. בנאי מחלקה זה דואג שהמפתח של הצומת יהיה key, שהערך הבוליאני של הצומת יהיה info, ושהגובה של הצומת יהיה height.   
הוא בנוסף דואג שהבן הימני, והשמאלי של הצומת יאותחל להיות הצומת הוירטואלי virtual.  
כיוון שטרם הבנים של הצומת קיימים/אותחלו להיות לא וירטואלים, ערך ה XOR של הצומת צריך להיות כערך הבוליאני של הצומת בלבד, לכן נאתחל את XOR להיות info.

**public** **boolean** isRealNode()

מתודה זו בודקת אם הצומת ממנו קראנו למתודה זו, הינו צומת אמיתי או וירטואלי.   
במידה והצומת וירטואלי, כלומר המפתח של הצומת הוא -1, נחזיר **false**, אחרת, הצומת האמיתי ונחזיר **true.**

**public** **int** BalanceFactor()

מתודה זו מחזירה את ה balance factor של העץ ששורשו הוא הצומת ממנו קראנו למתודה הזאת. במידה והצומת הינו וירטואלי, תחזיר 0. במידה והצומת לא וירטואלי, נחזיר את ההפרש בין הגובה של תת העץ השמאלי לבין גובה תת העץ הימני.

**Updater functions of class** AVLNode:

**private** **boolean** updateHeight()

מטרת מתודה זו היא לעדכן את השדה height, כדי שיהיה רלוונטי למצב הנוכחי (למשל אם שינינו את הבנים של הצומת, דבר זה דורש אישור שהשדה נכון למצב הרלוונטי.  
במידה והגובה של הצומת השתנה, נחזיר **true** אחרת נחזיר **false** . אנו מחזירים ערך בוליאני כדי שנוכל לדעת בתהליך ההכנסה והמחיקה לעץ, איזה צמתים עברו שינוי גובה במהלך תיקון עברייני ה AVL, שכן אנו רוצים להחזיר את מספר הצמתים שאכן עברו רוטציות או שינוי גובה במתודות ההכנסה והמחיקה.   
ובכן, עדכון הגובה יהיה:  
ניקח את הגובה המקסימלי מבין גבהי הבנים של הצומת, ונוסיף לגובה המקסימלי ביניהם את הערך 1. וזה יהיה גובה העץ החדש (הוספנו 1 שכן הצומת הינו רמה אחת מעל הבנים שלו).

**private** **void** updateSize()

מטרת מתודה זו, בדומה למתודה הקודמת updateHeight(), היא לעדכן את השדה size בהתאם למצב הנוכחי. ניקח את השדות size של הבנים של הצומת, נחבר ביניהם, ונוסיף לזה 1. וזה יהיה גודל הצומת החדש (הוספנו 1 שכן בנוסף לגדלי הבנים של הצומת, צריך לספור גם את הצומת עצמו).

**private** **void** updateXOR()

מטרת מתודה זו, בדומה למתודה הקודמת updateSize(), היא לעדכן את השדה XOR בהתאם למצב הנוכחי. ניקח את השדות XOR של הבנים של הצומת, נבצע ביניהם XOR, ואז נעשה עם זה עוד XOR עם הערך הבוליאני של הצומת, וזה יהיה הערך של השדה XOR החדש.

**getters and setters of class** AVLNode**:**

**public** **int** getKey()

ה getter של השדה הפרטי הפרטי key של הצומת.

**public** Boolean getValue()

ה getter של השדה הפרטי הפרטי info של הצומת. במידה והצומת וירטואלי, נחזיר null.

**public** **boolean** getXOR()

הgetter של השדה הפרטי XOR. מתודה זו מחזירה את השדה הפרטי XOR של הצומת.

**public** **void** setXOR(**boolean** out)

הsetter של השדה הפרטי XOR. מתודה זאת מעדכנת את השדה הפרטי XOR להיות out.

**public** **int** getHeight()

הgetter של השדה הפרטי height. מתודה זו מחזירה את ערך השדה הפרטי height של הצומת. במידה והצומת וירטואלי, תחזיר -1.

**public** **void** setHeight(**int** height)

הsetter של השדה הפרטי height. מתודה זו מעדכנת את השדה הפרטי height של הצומת להיות הערך ששלחנו למתודה (כלומר, **int** height).

**public** **int** getSize()

הgetter של השדה הפרטי size. מתודה זו מחזירה את גודל העץ ששורשו הוא הצומת ממנו קראנו למתודה זו. במידה והצומת וירטואלי, נחזיר 0.

**public** AVLNode getParent()

הgetter של השדה הפרטי parent. מתודה זו מחזירה את ההורה של הצומת.

**public** **void** setParent(AVLNode node)

הsetter של השדה הפרטי parent. מתודה זו מעדכנת את ההורה של הצומת להיות node.

**public** **void** setLeft(AVLNode node)

הsetter של הבן הימני של הצומת. בנוסף לכך, הוא דואג שהשדה הפרטי parent (ההורה) של הצומת node (הצומת שהופכת להיות הבן השמאלי של הצומת), יתחדש לצומת שאנחנו מעדכנים לו את הבן השמאלי. (נעשה באמצעות ה setter של parent).  
נוסף על כך, כיוון שעדכנו את הבן השמאלי של הצומת, שדות כמו size או כמו XOR עלולים להשתנות, על כן נקרא למתודה **this**.update\_info() שמעדכנת את השדות הללו כצורך.

**public** **void** setRight(AVLNode node)

זהה לחלוטין כמו המתודה setLeft, אך שפה אנחנו מעדכנים את הבן הימני ולא את הבן השמאלי.

**public** AVLNode getLeft()

מחזיר את הבן השמאלי של הצומת. במידה והבן השמאלי הינו צומת וירטואלי, נחזיר null.

**public** AVLNode getRight()

מחזיר את הבן הימני של הצומת. במידה והבן הימני הינו צומת וירטואלי, נחזיר null

**מדידות**

**1)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מספר סידורי | עלות prefixXor ממוצעת (כל הקריאות) | עלות succPrefixXor ממוצעת  (כל הקריאות) | עלות prefixXor ממוצעת  (100 קריאות ראשונות) | עלות succPrefixXor ממוצעת  (100 קריאות ראשונות) |
| 1 | 127 nanoseconds | 2187 nanoseconds | 103 nanoseconds | 349 nanoseconds |
| 2 | 110 nanoseconds | 4056 nanoseconds | 110 nanoseconds | 335 nanoseconds |
| 3 | 133 nanoseconds | 5852 nanoseconds | 112 nanoseconds | 338 nanoseconds |
| 4 | 122 nanoseconds | 9880 nanoseconds | 115 nanoseconds | 341 nanoseconds |
| 5 | 140 nanoseconds | 13805 nanoseconds | 119 nanoseconds | 339 nanoseconds |

הממוצע של 100 הקריאות הראשונות:

נשים לב כי הממוצע של 100 הקריאות הראשונות בשתי השיטות נשאר די זהה, אך הממוצע של ה100 קריאות הראשונות של succPrefixXor הרבה יותר איטי. אין לנו אבחנה מדוייקת ללמה זה קורה, אבל אם היינו צריכים לנחש, היינו אומרים שזה בגלל שאנחנו קוראים למתודה successor בכל איטראציה שרצה ב succPrefixXor, כאשר יש בממוצע 50 קריאות successor בכל קריאה של succPrefixXor ב100 האיברים הראשונים. בעוד ב prefixXor אנחנו מבצעים עלייה וירידה בכל העץ, כלומר נבצע בערך איטראציות בכל המתודה, וזה יהיה הרבה יותר מהיר, שכן מדובר **בלכל** **היותר** 25 איטראציות **בכל** איבר מתוך 100 האיברים הראשונים. זה גם הסיבה למה הממוצע של ה100 קריאות הראשונות בprefixXor עולה באופן הדרגתי (שכן הוא תלוי ב-n), באשר הממוצע של ה100 קריאות הראשונות של succPrefixXor נשאר בערך אותו דבר, שכן החלק המשמעותי של המתודה מבחינת זמני ריצה זה הקריאות ל successor, שהוא תמיד בממוצע 50.

הממוצע הכולל של הקריאות:

כמו כן, ניתן לראות בבירור שprefixXor יעיל **משמעותית** מsuccPrefixXor, שכן הוא מבצע את התהליך בזמן לוגריתמי בגודל העץ, בעוד שsuccPrefixXor מבצע את התהליך בזמן לינארי בגודל העץ ובשימוש כבד של מתודת העזר successor.   
ובכן, ניתן לראות כי זמן הריצה הממוצע של כל קריאה של succPrefixXor גדל בזמן לינארי ב-i (שכן גודל העץ תלוי באופן לינארי ב-i). ובכן זמן הריצה הממוצע עבור הi=2, גדול בערך פי 2 מזמן הריצה הממוצע עבור i=1. וזמן הריצה הממוצע עבור i=3 גדול בערך פי 3 מזמן הריצה הממוצע עבור i=1, כך גם עבור i=4 וi=5.   
בעוד זמן הריצה הממוצע של כל קריאה של prefixXor גדל באופן זניח שלא ניתן לנתח, וזאת כיוון שהריצה אינה משתמש במתודות עזר ורצה בזמן לוגריתמי בגודל העץ.  
הערה:  
נוסף על כך, ניתן לראות שהממוצע של 100 הקריאות הראשונות די זהה לממוצע של כל הקריאות כשמשתמשים בprefixXor, שכן עבור כל צומת בעץ זמן הריצה של הקריאה יהיה תלוי אך ורק בצומת העץ, בעוד שעבור succPrefixXor, זמן הריצה של הקריאה יהיה תלוי גם בגודל המפתח שניתן. ככל שהמפתח יותר גדול, ייתבצעו יותר קריאות לsuccessor וכך גם זמן הריצה יעלה. וזה למה הממוצע של ה100 הקריאות הראשונות הרבה יותר קטן מהממוצע הכולל של כל הקריאות, שכן הקריאות מבוצעות מהמפתח הקטן ביותר לגדול ביותר.

**2)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| עלות הכנסה ממוצעת  מספר סידורי | עץ AVL  סדרה חשבונית | עץ ללא מנגנון איזון  סדרה חשבונית | עץ AVL  סדרה מאוזנת | עץ ללא מנגנון איזון  סדרה מאוזנת | עץ AVL  סדרה אקראית | עץ ללא מנגנון איזון  סדרה אקראית |
| 1 | 279 nanoseconds | 325 nanoseconds | 791 nanoseconds | 710 nanoseconds | 356 nanoseconds | 322 nanoseconds |
| 2 | 372 nanoseconds | 316 nanoseconds | 788 nanoseconds | 812 nanoseconds | 413 nanoseconds | 369 nanoseconds |
| 3 | 361 nanoseconds | 374 nanoseconds | 830 nanoseconds | 801 nanoseconds | 417 nanoseconds | 374 nanoseconds |
| 4 | 357 nanoseconds | 373 nanoseconds | 970 nanoseconds | 813 nanoseconds | 428 nanoseconds | 402 nanoseconds |
| 5 | 367 nanoseconds | 466 nanoseconds | 966 nanoseconds | 908 nanoseconds | 520 nanoseconds | 637 nanoseconds |

**הערה:**בדיקת זמני הריצה של הסדרה המאוזנת נעשו בימים ובזמנים שונים מהסדרה החשבונית והסדרה האקראית, על כן, המחשב היה תחת עומסים שונים במדידות הנ"ל, וזאת הסיבה שזמני הריצה של סדרה מאוזנת היו דרסטית יותר איטיים מהסדרה החשבונית והסדרה האקראית.

**הציפיות והמציאות:**

סדרה חשבונית:

ניתן לשים לב כי שכשאנחנו מכניסים סדרה חשבונית לעץ בלי מנגנון איזון, יווצר לנו מן זנב ארוך, ועומק העץ יהיה כגודל הסדרה, בעוד בעץ AVL, העץ יהיה מאוזן ועומק העץ יהיה לוגריתמי בגודל הסדרה, ולכן, ניתן לשים לב שכיוון שהכנסה מבצעת מן תהליך חיפוש לפני שהיא באמת מכניסה את המפתח, אז רוב זמן הריצה מבוצע בחיפוש המפתח ולא בביצוע ההכנסה עצמו. ובכן, סיבוכיות זמן החיפוש קטנה משמעותית בעץ מאוזן מאשר בעץ שעומקו כגודל האיברים בעץ, שכן יבוצעו יותר השוואות בעץ לא מאוזן לעומת עץ מאוזן. עם זאת, זמני החיפוש גדלים ככל שמספר האיברים בעץ גדל, שכן זמן החיפוש תלוי בגודל העץ בשני המקרים.  
בכל זאת, הטענה שטענו כרגע, באה לידי ביטוי במדידות. ובכן, ניתן לראות שממוצע זמן ההכנסה של מפתח בסדרה חשבונית בעץ ללא מנגנון איזון, יהיה גדול יותר ממוצע זמן ההכנסה של מפתח בסדרה חשבונית בעץ AVL.

סדרה מאוזנת:

כיוון שהסדרה מאוזנת לחלוטין ותגרום לבניית עץ מאוזן גם בעץ ללא מנגנון איזון, אמור להיות ממוצע זמן ריצה די זהה להכנסה בעץ AVL ולהכנסה בעץ ללא מנגנון איזון. עם זאת, בעץ AVL, כיוון שבכל צומת בה עברנו בדרך ההכנסה, אנחנו מבצעים בדיקה אם צריך גלגול, ניתן לראות שאכן ממוצע זמן הריצה של הכנסה בעץ AVL, יהיה בדרך כלל, אך לא משמעותית, יותר גדול מממוצע זמן ההכנסה בעץ ללא מנגנון איזון.

סדרה אקראית:

ראינו בכיתה, שהעומק הממוצע של עץ חיפוש בינארי ללא מנגנון איזון, כאשר מכניסים מפתחות אקראיים, יהיה לוגריתמי במספר המפתחות שהוכנסו, על כן, בממוצע (כלומר, בדרך כלל), ההבדל בין עץ AVL לבין עץ ללא מנגנון איזון יהיה זניח, שכן הם מתנהגים דומה תחת מפתחות אקראיים. וכך גם ניתן לראות במדידות. ההבדל הוא זניח.