**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:**

**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.

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.

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.

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 ה info – 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.   
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)

min()

**סיבוכיות:** O(log n)

max()

**סיבוכיות:** O(log n)

keysToArray()

**סיבוכיות:** O(n)

keysToArray\_rec(AVLNode N, **int**[] arr, **int** pos)

**סיבוכיות:** O(n)

infoToArray()

**סיבוכיות:** O(n)

infoToArray\_rec(AVLNode N, **boolean**[] arr, **int** pos)

**סיבוכיות:** O(n)

size()

**סיבוכיות:** O(1)

getRoot()

**סיבוכיות:** O(1)

successor(AVLNode node)

**סיבוכיות:** O(log n)

prefixXor(**int** k)

**סיבוכיות:** O(log n)

succPrefixXor(**int** k)

**סיבוכיות:** O(log n + n)

**public** **class** AVLNode

**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.