**תרגיל בית מעשי – תיעוד פונקציות**

**עומר פלביץ 206840126  
אלעד שבא 207909409**

מחלקת AVLNode

getLeft: פונקציה המחזירה את הבן השמאלי, במידה ואין מחזירה None – O(1)  
get Right: פונקציה המחזירה את הבן הימני, במידה ואין מחזירה None – O(1)  
get parent: פונקציה המחזירה את ההורה, במידה ואין מחזירה None – O(1)

Get value: פונקציה המחזירה את ערך הצומת – O(1)

Get Height: פונקציה המחזירה את גובה הצומת – O(1)

Set left: פונקציה המקבלת צומת, ומכניסה אותה כבן השמאלי של הצומת הנוכחית – O(1)

Set right: פונקציה המקבלת צומת, ומכניסה אותה כבן הימני של הצומת הנוכחית – O(1)

Set parent: פונקציה המקבלת צומת, ומכניסה אותה כההורה של הצומת הנוכחית – O(1)

Set value: פונקציה המקבלת ערך, ומכניסה אותו כערך של הצומת הנוכחית – O(1)

Set parent: פונקציה המקבלת ערך מספרי, ומכניסה אותה כגובה של הצומת הנוכחית – O(1)

Is real Node: פונקציה המקבלת צומת, ומחזירה ערך שקר אם הצומת וירטוראלית, ושקר אחרת – O(1)

Get balance: פונקציה הבודקת את הפרש הגבהים של בניו של הצומת הנוכחית, מחזירה את הפרש הגבהים – O(1)

Get Size: פונקציה המקבלת צומת ומחזירה את גודלה – O(1)

Set Size: פונקציה המקבלת צומת ומשנה את ערך הגודל שלה- O(1)

Successor: פונקציה שממשת את היורש של כל צומת. מחזיר None אם אין. הפונקציה בודקת אם יש בן ימני, אם יש היא הולכת אליו ואז ממשיכה הכי שמאלה שיכולה ומחזירה. אם לא, עולה למעלה לאב. אחרי זה ממשיכה לעלות למעלה כל עוד האב הוא בן שמאלי. בשניה שהוא לא בן שמאלי מצאנו את היורש ונחזיר אותו. נבדוק בסוף אם קיבלנו שהיורש שווה לעצמו, נחזיר None. הסיבוכיות של פונקציה זאת היא כגובה העץ, O(Log n).

if\_node\_is\_left/right\_child: פונקציה שבודקת האם הצומת הוא בן שמאלי או ימני. הפונקציה בודקת אם יש לצומת אב, ואז בודקת האם הצומת הוא בן שמאלי של האב או ימני. בוליאנית – מחזירה אמת או שקר בהתאם לשם שלה. הסיבוכיות של פעולה זאת היא קבועה.

replace\_to\_virtual: פונקציה שמשנה צומת לצומת וירטואלי בעזרת שינוי הערך, הגודל והגובה. היא בעצם בונה צומת חדש עם הערכים הdefault , ומחזירה אותו. סיבוכיות קבועה.

מחלקת AVLTreeList

Empty: פונקציה המקבלת עץ, ומחזירה ערך אמת אם הינו עץ ריק ושקר אחרת, הפונקציה בודקת האם יש צומת בשורשו כדי לדעת אם הינו ריק – O(1)

getRoot: פונקציה המקבלת עץ, ומחזירה את שורשו, משתמשת בשדה המתוחזק- O(1)

Retrieve : פוקנציה שמחזירה את הערך של האיבר במקום ה-i . הפונקציה נעזרת בפונקציה TreeSelect. היא בודקת האם העץ ריק, ומחזירה None אם כן. אם לא, בודקת האם זה האיבר המינימלי או המקסימלי, אם לא משתמשת בפונקציה שתיארנו. הסיבוכיות זהה לselect שזה O(log n ).

Insert: פונקציה המקבלת ערך ומיקום, ומכניסה את האיבר למיקום ברשימה. הפונקציה מחזירה את מספר פעולות הגלגול שנדרשו כדי לממש את עץ ה AVL המממש את הרשימה.  
הפונקציה מתייחסת לארבעה מקרים – אם העץ ריק, פשוט מכניסה את הצומת כשורש העץ הריק ומחזירה 0 - סיבוכיות של O(1)  
אם ההכנסה היא לאינדקס "0" ברשימה, היא מגיעה להורה המתאים באמצעות פוינטר שמתוחזק לאיבר ה"מינימום" בעץ, מבצעת את ההכנסה ולאחר מכן מבצעת תיקון ערכים וגלגולים בהתאם לצורך.  
אם ההכנסה היא לסוף הרשימה, באופן סימטרי באמצעות פוינטר לאיבר ה"מקסימום".  
אחרת – מוצאים את המיקום הנכון להכנסה באמצעות פונקציית TreeSelect (שסיבוכיותה הוא O(log n)) מבצעים את ההכנסה, ומתחילים לעלות מהעלה באמצעות הפונקציה CheckInsertion (שסיבוכיותה הוא O(log n))אשר מתקנת את גובה וגודל כל צומת עד לשורש, סיבוכיות פעולה זו הוא סה"כ O(log n) .

help\_tree: פוקנציית עזר למחיקה. מקבלת צומת ועץ, והיא "מסדרת" את העץ לכדי עץ תקין, ומחזירה את מספר הגלגולים שלקח לצומת לעשות את זה. הסיבוכיות בהתאם לפונקציות שמסדרות, O(log n)

delete: פונקציה אשר מוחקת את האיבר ה-i ומחזירה את מספר הגלגולים שלוקח לעץ בכדי לעמוד בהגדרת עץ AVL. הפונקציה מתחילה בכך שהיא בודקת האם האיבר שרוצים למחוק הוא מינימלי או מקסימלי ומעדכנת את הערך בעץ בהתאם. לאחר מכן מחלקת הפונקציה את המקרים ל-3: מקרה ראשון הוא שהצומת הוא עלה, כך שנותר רק למחוק אותו לסדר. המקרה השני הוא שהוא עם בן ימני או שמאלי. במקרה זה נחליף את הצומת בבן שלו, "ננתק" את איפה שהיה הבן מלפני, ונסדר. המקרה השלישי הוא שלצומת יש גם בן ימני וגם בן שמאלי, כלומר הוא צומת פנימי. במקרה הזה נחפש את היורש שלנו בעזרת פונקציה שתועדה כאן, ולאחר מכן נחלק גם כן לשני מקרים אפשריים – שהיורש הוא עלה ואז פשוט ננתק את הצומת המקורית שלו ונחליף. מקרה שני זה שיש לו בן ימני, ואז נחליף בין המקום הקודם שלו לבין הבן הימני. בסוף נסדר ונחזיר את כמות הגלגולים. כל הפונקציות שהובאו פה כולן בתיעוד הן O(log n) , לכן בכל מחיקה במקרה הגרוע נצטרך למצוא יורש, להחליף אותו בבן שלו, לגלגל עד לסידור חוקי ולהחזיר. זה פעולות מקבילות של O(log n) ולכן המחיקה היא O(log n ).

First: פונקציה שמחזירה את ערך האיבר הראשון ברשימה – O(1).

Last: פונקציה שמחזירה את ערך האיבר האחרון ברשימה – O(1).

ListToArray: פונקציה שמחזירה את הרשימה שממומשת באמצעות עץ AVL כטיפוס מערך (רשימה) של פייתון, סיבוכיות הפונקציה היא O(n) מאחר והיא קוראת לפונקצית עזר inOrder אשר זו סיבוכיותה.

InOrder: פונקציה זו מקבלת מערך ריק ואת שורש העץ, ומבצעת טיול inOrder על העץ, כאשר הפעולה המתבצעת היא הוספת ערך הצמתים של העץ למערך(באמצעות append, שסיבוכיותה זמן קבוע), מכיוון שהפונקציה מבקרת בכל צומת פעם אחת (וכפי שהוכחנו בכיתה ובתרגול) סיבוכיותה הינו O(1).

Length: הפונקציה מחזירה את אורך הרשימה – O(1)

Sort: פוקנציה שממיינת את העץ מקטן לגדול. הפונקציה משתמשת בפונקציה listToArray שמתועדת פה, והופכת את העץ למערך ממוין. לאחר מכן ממיינת את המערך בעזרת Mergesort, שהמימוש שלו גם כן נמצא בתיעוד. לאחר המיון מכניסים לעץ חדש בעזרת insert לפי המערך הממוין ומחזירים את העץ. הסיבוכיות של העברת העץ למערך הוא O(n) , והמיון של mergesort הוא O(n log n) , וההכנסה היא O(log(n) . סה"כ מאחר והפוקנציות פועלות במקביל נקבל סיבוכיות של O(n log n).

Permutation: פונקציה המחזירה עץ חדש שערכיו "מעורבבים" בצורה רנדומלית, הפונקציה מקבלת את המערך ListToArray של העץ, קוראת לפונקציה ShuffleList כדי לערבב את ערכי המערך, ולאחר מכאן משתמשת בפונקציה BuildTree כדי לבנות עץ avl חוקי מהמערך, סיבוכיות כל פונקציות העזר האלו הינם O(n) כמו כן הפונקציה משתמשת בfindMaxMin בשביל למצוא את איברי המקסימום/מינימום חדשים, סיבוכיות פונקצית עזר זו הינה O(log n) ולכן סיבוכיות כוללת הינה O(n)

ShuffleList: הפונקציה מקבלת מערך, ועוברת על המערך באמצעות לולאה, כאשר בכל מיקום היא מגרילה אינדקס (הגרלת מספר באמצעות randInt ) ומחליפה בין האיבר הנוכחי לאיבר באינדקס המוגרל, עוברים על כל המערך פעם אחת ומבצעים פעולות בO(1) לכן סיבוכיות כוללת הינה O(n)

BuildTree: פונקציה המקבלת מערך, ובונה ממנו עץ AVL תקין. אנו למעשה מתייחסים למערך כמערך "ממויין" ומשתמשים באלגוריתם שראינו והוכחנו בתרגול (לקיחת חציון הרשימה בתור "שורש", בניית בן שמאלי בצורה רקורסיבית מהחצי השמאלי של הרשימה, ובן ימני מהחצי הימני לחציון של הרשימה) לבניית עץ AVL בO(n).

Concat: הפונקציה מקבלת רשימה במימוש AVL, ומאחדת אותה לסוף הרשימה הנוכחית, הפונקציה מחזירה את הפרש הגבהים בין העצים. גם כאן המימוש הוא באמצעות חלוקה למקרים:

אם הרשימה self ריקה – פשוט נהפוך את self להיות Other ונחזיר את גובה העץ (נשים לב שהדבר עובד גם אם שני הרשימות ריקות סימולטנית).

אם הרשימה Other ריקה – אין מה לחבר, פשוט נחזיר את גובה העץ הנוכחי.  
בשני המקרים הללו הסיבוכיות הינה O(1)

אם גובה אחד מהעצים הינו 1 – נתלה את הצומת הבודדת על העץ ונחזיר את הפרש הגבהים. (אם self בגובה 1, נתלה את הצומת על other באינדקס הראשון, ונהפוך את self להיות other, אם other בגובה 1, פשוט נתלה את שורשו באינדקס האחרון), נשים לב ששני המקרים הללו השימוש פשוט הינו בinsert ולכן הסיבוכיות הינה O(log n)   
אחרת – נלך לצומת הימנית ביותר של self, נשמור את הצומת ונמחק אותה מהעץ (סיבוכיות של Log n ), כעת נשווה בין הגדלים ונקרא לפונקצית עזר מתאימה joinBigger או joinSmaller עם הצומת השמורה בצד, סיבוכיות פונקציות אלה הינם O(log n) לאחר המיזוג, נעלה במעלה העץ נתקן פרמטרים וגלגולים באמצעות Check במקרה הצורך – סיבוכיות הינה O(log n), לכן סה"כ סיבוכיות הפעולה O(log n)

JoinBigger: פונקציה זו מקבלת שני עצים, וצומת כך ש self<x<other לכל ערכי העצים, נקרא לפונקציה זו במקרה שגובה Self הינו קטן מגובה Other, בפונקציה זו קוראים לפונקצית עזר findOnRIghtH (עם סיבוכיות O(log n) ) שמוצאת את האינדקס הראשון על המסלול הימני של self כך שגובה גדול או שווה לגודל שורש other, ומחזירה את העץ מחובר בצורה זהה למה שראינו בהרצאה, ולכן הסיבוכיות הינה O(log n) או בצורה הדוקה יותר O(h2-h1)

JoinSmaller: הפונקציה זהה למה שראינו בהרצאה ובתרגול ומחזירה את העץ מחובר בהתאם,היא הפעולה הסימטרית ל JoinBigger, כאשר גובה self גדול מגובה other, נמצא את הצומת הראשונה על השרשרת השמאלית של other כך שגובה מתאים באמצעות פונקצית findOnLeftH סיבוכיותה היא O(log n) לכן הסיבוכיות הכוללת הינה O(log n) כפי שהוכחנו בהרצאה.

findOnLeftH: פונקציה המקבלת גובה ועץ, ומחזירה את הצומת הראשונה בשדרה השמאלית של העץ ככה שגובה גדול מהגובה שהתקבל, במקרה הגרוע ביותר נגיע עד לשורש, וזוהי סיבוכיות O(log n)

findOnRIghtH: פונקציה זהה לעיל, פשוט מחזירה על השדרה הימנית של העץ – O(log n)

Search: פונקציה שמקבלת ערך של צומת ומחפשת אותו בעץ. אם זה נמצא בעץ נחזיר את האינדקס שלו בעץ, אם לא נחזיר -1 . הפונקציה מעבירה את העץ למערך, ואחר מכן עושה חיפוש על המערך הלא ממוין כך שהיא עוברת איבר איבר במערך ובודקת האם הערך שלו שווה לערך המבוקש. אם כן, מחזירה את האינקדס שלו במערך. הסיבוכיות של העברת העץ למערך היא O(n) ומעברעל כל האיברים במקרה הגרוע היא גם O(n) . לכן סה"כ סיבוכיות היא O(n).

PredNode: פונקציה המוצאת את הpredecessor של הצומת בעץ, הפונקציה ממומשת כפי שראינו בהרצאות ותרגולים בעלת סיבוכיות O(log n).

TreeSelectRec: פונקצית מעטפת הקוראת ל TreeSelectRec

TreeSelect: פונקציה המקבלת צומת (בהתחלה שורש) ומספר i, הפונקציה מחזירה את האיבר הi הכי קטן. כפי שראינו בהרצאות סיבוכיות של O(log n)

CheckInsertion: הפונקציה מאזנת את העץ באמצעות גלגולים, ועולה מהצומת שהוכנסה עד לשורש לטובת עדכון שדות/פרמטרים וביצוע גלגולים נדרשים. הפונקציה ממומשת באופן זהה לפונקציה לתיקון עץ לאחר delete שראינו בהרצאה, כאשר התוספת הינה קריאה לפונקצית עזר FixHS אשר מתקנת את הפרמטרים בO(1)– לכן הסיבוכיות הינה O(log n)

Left/RightRotate: הפונקציה מקבלת צומת ומבצעת לה גלגול כנדרש, הפונקציה ממומשת באופן זהה לנראה בהרצאה/תרגול, למעט קריאה לפונקצית עזר FixHS שהיא בעלת O(1) ולכן סיבוכיות הכוללת הינה O(1)

FixHS: פונקציה המקבלת צומת ומתקנת את ערך הגובה והגודל, סיבוכיות O(1).

findMaxMin: פונקציה שמוצאת את האיבר שמייצג את ה"מקסימום" או ה"מינימום" בעץ (אינדקס ראשון ואחרון ברשימה), הפונקציה הינה פונקציית עזר ל permutation, מכיוון שהפונקציה רצה מהשורש עד לעלה, ללא חזרות אחורה, הסיבוכיות הינה O(log n)

Merge+MergeSort: פוקנציות עזר לפונקציית sort , בעצם עושות Mergesort בעזרת רקורסיה כפי שנלמד בכיתה ולפי האינטרנט. סה"כ סיבוכיות O(n log n) .

מדידות וחלק תיאורטי:

|  |  |  |  |
| --- | --- | --- | --- |
| **insert** | **delete** | **both** | **n** |
| **3728** | **1121** | **1838** | **3000** |
| **7473** | **2182** | **3637** | **6000** |
| **15095** | **4483** | **7305** | **12000** |
| **29897** | **8903** | **14637** | **24000** |
| **60329** | **17831** | **29597** | **48000** |
| **119574** | **35726** | **58713** | **96000** |
| **240446** | **71975** | **117600** | **192000** |
| **481260** | **143764** | **235821** | **384000** |
| **960138** | **287322** | **470335** | **768000** |
| **1920673** | **573605** | **943564** | **1536000** |

**שאלה 1:**

ניתן לראות כי בכל שלושת המצבים קיבלנו ביטויים ליניאריים, לכן הביטוי האסימפטוטי המתאים לכל שלושת המצבים הוא O(n).

**שאלה 2:**

הערות:  
כלל הזמנים נמדדו בעזרת חישוב ממוצע על time.perf\_counter()  
בהתאם לנאמר בפורום- מימוש מערך התבצע באמצעות הרשימה המובנת של פייתון  
במימוש רשימה מקושרת (להכנסות להתחלה) השתמשנו ב add\_at\_start ולא ב add\_at(0,val)

לכלל המימושים הוכנסו אותם ערכי str

|  |  |  |  |
| --- | --- | --- | --- |
| זמן ריצה ממוצע ערך i | עץ AVL  הכנסות להתחלה | רשימה מקושרת הכנסות להתחלה | מערך (רשימה) הכנסות להתחלה |
| 1 | 2.0396396983414887e-05 | 1.747764428310827e-06 | 1.3706668590505918e-06 |
| 2 | 2.3455800954252482e-05 | 1.747764428310827e-06 | 1.3706668590505918e-06 |
| 3 | 2.2795222803122466e-05 | 1.5842642616062436e-06 | 1.3706668590505918e-06 |
| 4 | 2.13433132265453e-05 | 1.5842642616062436e-06 | 1.3706668590505918e-06 |
| 5 | 2.4086576731254658e-05 | 1.5567264236844624e-06 | 1.3706668590505918e-06 |
| 6 | 2.3092576095627415e-05 | 1.5567264236844624e-06 | 1.3706668590505918e-06 |
| 7 | 2.4143361589974826e-05 | 1.63373760989035e-06 | 1.3706668590505918e-06 |
| 8 | 2.554414787640174e-05 | 1.63373760989035e-06 | 1.3706668590505918e-06 |
| 9 | 2.515583684934037e-05 | 1.645433010757124e-06 | 1.3706668590505918e-06 |
| 10 | 2.9198008279005686e-05 | 1.7043483151786447e-06 | 1.3706668590505918e-06 |

התחלנו מרשימות ריקות, וכדי לבחור בכל פעם היכן להכניס איבר חדש,השתמשנו בrandom.randint כדי להגריל את מיקום ההכנסה  
לכלל המימושים הוכנסו אותם ערכי str

ההכנסה לעץ AVL נעשתה באמצעות מצביע לאינדקס הראשון (איבר ה"מינימום" בעץ)

|  |  |  |  |
| --- | --- | --- | --- |
| זמן ריצה ממוצע ערך i | עץ AVL  הכנסות אקראיות | רשימה מקושרת הכנסות אקראיות | מערך (רשימה) הכנסות אקראיות |
| 1 | 2.663506598522266e-05 | 1.1432731834550698e-05 | 3.9566582781189717e-07 |
| 2 | 2.4573234220345816e-05 | 1.892553192252914e-05 | 4.571542347538825e-07 |
| 3 | 2.570124508606063e-05 | 3.032899337510268e-05 | 4.928450800427121e-07 |
| 4 | 2.7095746248960494e-05 | 4.080872020373742e-05 | 5.439885381288756e-07 |
| 5 | 3.0837772910793625e-05 | 5.370405353605747e-05 | 6.364198790617557e-07 |
| 6 | 2.7832612809207703e-05 | 6.643742493664224e-05 | 7.114526401529122e-07 |
| 7 | 3.0119162274613266e-05 | 8.30496885325937e-05 | 8.450832322665434e-07 |
| 8 | 2.870787342544645e-05 | 9.768053248990327e-05 | 9.302965142077657e-07 |
| 9 | 3.077174119513344e-05 | 0.00011948675589842929 | 9.302965142077657e-07 |
| 10 | 3.077174119513344e-05 | 0.0001384650979191065 | 9.302965142077657e-07 |

התחלנו מרשימות ריקות , וכל פעם הכנסו לסוף הרשימה.  
בחרנו לממש את הרשימה המקושרת ללא מצביע לסוף הרשימה, מאחר ועם מצביע לסוף הרשימה הדבר דומה קונספטואלית להכנסה לתחילת הרשימה, ורצינו להראות את התוצאות כשצריך לעבור ממש על כל הרשימה (שיערנו כי ברשימה במימוש פייתון לא נוכל לראות תוצאות אלה טוב).  
ההכנסה לעץ AVL נעשתה באמצעות מצביע לאינדקס האחרון (איבר ה"מקסימום" בעץ)

לכלל המימושים הוכנסו אותם ערכי str

|  |  |  |  |
| --- | --- | --- | --- |
| זמן ריצה ממוצע ערך i | עץ AVL  הכנסות בסוף | רשימה מקושרת הכנסות בסוף | מערך (רשימה) הכנסות בסוף |
| 1 | 2.5476395152509212e-05 | 1.9032958263731383e-05 | 4.737991839647293e-07 |
| 2 | 2.231754067664345e-05 | 2.9510106343593386e-05 | 2.2789944584170976e-07 |
| 3 | 2.3441462467114132e-05 | 4.7164705005863765e-05 | 2.371583961778217e-07 |
| 4 | 2.5591266031066577e-05 | 6.042495320232377e-05 | 2.1831683504084747e-07 |
| 5 | 2.7160413935780527e-05 | 7.901267144952731e-05 | 2.500830839077632e-07 |
| 6 | 2.5978793027914232e-05 | 9.180866769683773e-05 | 2.191868196758959e-07 |
| 7 | 2.729517106144201e-05 | 0.00010965462306203734 | 2.5371927767992017e-07 |
| 8 | 2.664901661531379e-05 | 0.0001421856733663215 | 2.8616255925347406e-07 |
| 9 | 2.7889802913974832e-05 | 0.00015759169364089887 | 2.158158631236465e-07 |
| 10 | 2.752777300775051e-05 | 0.0001637927483844514 | 2.2011495505770047e-07 |

ניתוח תוצאות:

נשים לב כי בהכנסה בתחילת הרשימה, הרשימה במימוש מערך (של פייתון) מבצעת את ההכנסה במהירות הטובה ביותר, בפער קטן אחריה מימוש הרשימה המקושרת ולאחר מכן רק המימוש שלנו.

בהכנסות אקראיות – כמו מקודם, מימוש פייתון היה הכי מהיר באופן עקבי בכל הגדלים, בגדלים הקטנים מאוד מימוש הרשימה המקושרת ועץ הAVL היו יחסית דומים עם יתרון קטן לAVL  
אך ככל שהגדלים גדלו, עץ הAVL צבר יתרון על פני הרשימה המקושרת.

הכנסות בסוף- כמו מקודם, מימוש פייתון היה הכי מהיר באופן עקבי בכל הגדלים, כאן בשלב מוקדם מאוד כבר היה פער כך שמימוש הAVL היה מהיר יותר מהרשימות המקושרות.

באופן כללי לאורך כל סוגי ההכנסות, רשימת פייתון הייתה הכי מהירה, ובפרט בהכנסות בסוף, בהכנסות בהתחלה המהירות בין הרשימה הפייתונית לרשימה במימוש מקושר הייתה קטנה מאוד.

לאורך כל ההכנסות, מימוש הAVL היה בעל אחידות יחסית מבחינת זמן ריצה.

הרשימה המקושרת הייתה מאוד מהירה בהכנסות לראש, אך בשאר ההכנסות הייתה יחסית איטית.

התוצאות יצאו כפי שצפינו,

אומנם רשימה של פייתון אמורה לבצע הכנסות בתהחלה בO(n) וגם הכנסות רנדומליות ידרשו העתקה של איברים פיזית בזיכרון, אך פייתון ממש רשימה זו בעזרת קוד בC שמהיר מאוד יחסית לקוד שנכתב בפייתון "טהור".

הרשימה המקושרת הייתה מאוד מהירה בהכנסות לראש, מאחר והיה לה מצביע לשם (ולמעשה כמעט השתוותה למהירות הרשימה הפייתונית), אך שאר ההכנסות היו איטיות יחסית, בפרט לסוף הרשימה מכיוון שיש ממש צורך לעבור על כל איברי הרשימה המקושרת כדי להגיע לסופה.

מהירות העץ AVL הייתה יחסית אחידה, מכיוון שבכל מקרה שנכניס, גם באמצעות המצביעים לmin ו max שיצגו את האינדקס הראשון והאחרון, אנו מבצעים עליה לאורך העץ לבדוק את הפרמטרים (ולמעשה זה O(log n) בשביל התיקונים).