**Eyalgrinberg\_207129792 אייל גרינברג**

**Matanshvide\_201414315 מתן שביד**

**מחלקת AVLTree**

**empty() - O(1) -** ניגשת ישירות לשדה root של העץ ומחזירה ערך בוליאני בהתאם לתוכנו.

**search - O(log(n)) -** מבצעת חיפוש בינארי בעץ. כל קריאה רקורסיבית נוספת לפונקציה מורידה אותנו רמה בעץ ולכן יהיו לכל היותר log(n) קריאות, כגובה העץ. לפני כל קריאה אנחנו מאתחלים עץ חדש במספר קבוע של פעולות. בכל קריאה לפונקציה ישנו מספר פעולות בוליאניות מקסימאלי קבוע עד לקריאה הבאה.  
\*במידה והמפתח לא נמצא, אנחנו מעדכנים את הנקודה האחרונה אליה הגענו. זאת לטובת פעולת הכנסה עתידית אשר משתמשת בחיפוש. במידה ולא קיים ערך הזהה לערך שאנחנו רוצים להכניס (כלומר פעולת החיפוש השיבה ריקם), אנו נעדכן בעץ המקורי את המיקום הרלוונטי אליו אמור להיכנס הערך החדש.

**insert - O(log(n)) –** מבצעת הכנסה של צומת לעץ בהינתן ערכו ומפתחו.  
הפונקציה מחזירה את מס' פעולות האיזון שנדרשו על מנת לאזן את העץ לאחר ההכנסה.  
אם העץ ריק היא מבצעת הכנסה ראשונה של הצומת בתור שורש העץ. אחרת היא קוראת לפונקציה search כדי ראשית לבדוק אם האיבר שרוצים להכניס כבר קיים בעץ, אם קיים היא תחזיר 1-. ושנית, במהלך הsearch שמרנו מצביע (searchNodePointer) למקום ההכנסה הרלוונטי במידה והמפתח שמחפשים לא נמצא בעץ (ראו הערה באנליזת search).  
ההכנסה מתבצעת ומגדילים את שדה הsize של העץ ב1. לאחר מכן יש מספר בדיקות בוליאניות על מנת לסווג למקרים השונים שלמדנו שיכולים להיווצר בעת הכנסת צומת לעץ, ובהתאם לכך אנו מטפלים בכל מקרה לסוגו מבחינת ניתוק ויצירת קשרים בין צמתים ועדכון דרגות בהתאם כמובן. הפונקציה מבצעת בכל אחד מהמקרים קריאה לפונקציית העזר heightCheckInsert על מנת שזו תבדוק אם יש צורך בגלגולים או העלאות בדרגה, וגם כדי שהיא תעדכן את המונה של פעולות האיזון הנדרשות שinsert צריכה להחזיר. במידת הצורך הפונקציה עולה במעלה העץ כדי לתקן דרגות של צמתים שדרגות בניהם שונו ובכך הופרה תקינותם.  
הקריאה לsearch מתבצעת בסיבוכיות O(logn) אבל זו קריאה בודדת שמחברים לסיבוכיות הכוללת.  
בנוסף יש מספר פעולות בוליאניות כדי לסווג למקרים השונים המתבצעות בזמן קבוע ומס' קבוע של פעולות עדכון שדות הצמתים הרלוונטיים להכנסה, וקריאה ל heightCheckInsert הפועלת בזמן קבוע גם כן. לבסוף הפונקציה עולה במעלה העץ רק על המסלול מן הצומת שהוכנס עד השורש כדי לעדכן דרגות ולבדק תקינות צמתים, מסלול זה באורך log(n) כגובה העץ.  
סה"כ נקבל שסיבוכיות זמן הריצה היא O(log(n)).

**delete - O(log(n)) –** בהינתן צומת עם מפתח k הפונקציה קוראת לsearch ומוחקת צומת זה מהעץ אם קיים (שוב בעזרת שימוש במצביע searchNodePointer), אם לא קיים היא תחזיר 1-. בנוסף הפונקציה מחזירה את מספר פעולות האיזון שנדרשו על מנת לאזן את העץ לאחר מחיקת הצומת ממנו.  
הפונקציה מבצעת מס' פעולות בוליאניות על מנת לסווג למקרים השונים (אם הצומת אונארי או עלה, או אם הצומת הוא צומת פנימי) שלמדנו שיכולים להיווצר בעץ בעת מחיקת צומת, בהתאם מתבצע טיפול בכל מקרה ע"י יצירה וניתוק קשתות בין הצמתים הרלוונטיים ועדכון דרגותיהם.  
בכל אחד מהמקרים מתבצעת קריאה ל heightCheckDelete כדי שהיא תבדוק אם יש צורך בגלגולים או הורדות בדרגה וגם לצורך עדכון מונה פעולות האיזון שמתבצעות.  
לאחר מכן הפונקציה עולה במעלה העץ ומתקנת דרגות של צמתים שתקינותם הופרה.  
במקרה שמחקנו צומת פנימי מתבצעת החלפה שלו עם האיבר הקודם לו בעזרת קריאה לפונקציה getPredecessor. כמובן שגם לאחר המחיקה שדה הsize קטן ב1.  
לבסוף הפונקציה מחזירה את המונה שספר את מספר פעולות האיזון שנדרשו.  
מבחינת סיבוכיות מתבצעת קריאה אחת לsearch שמתבצעת בסיבוכיות O(logn), מתבצעות מס' פעולות בוליאניות בזמן קבוע ומס' קבוע של פעולות עדכון שדות של הצמתים הרלוונטיים למחיקה, קריאה ל getPredecessor בסיבוכיות O(logn) כדי למצוא את האיבר הקודם לצומת שאנו רוצים למחוק, ובנוסף לכל אלה הפונקציה עולה במעלה העץ ומבצעת קריאות ל heightCheckDelete הפועלת בזמן קבוע כדי לבדוק אם יש צורך בפעולות איזון ואם כן אז הפונקציה מבצעת פעולות אלו. במידה ויש פעולות איזון עד השורש, נעלה בעץ מרחק שגודלו כגובה העץ log(n) .   
סה"כ נקבל שיש מספר פעולות בלתי תלויות שכל אחת בסיבוכיות O(logn) ולכן הסיבוכיות הכוללת תהא O(log(n)).

**min - O(log(n)) –** יורדת עד לקצה השמאלי של העץ עד למציאת האיבר המינימלי ומחזירה את הערך (info) שלו. מבצעת מספר קריאות רקורסיביות כגובה העץ ובכל קריאה מבצעת מספר קבוע של פעולות. כפי שנלמד בכיתה, גובהו של עץ AVL חסום ע"י log(n) ולכן ישנן לכל היותר log(n) קריאות לפונקציה.

**Max - O(log(n)) –** ראו אנליזה עבור פונקציית **min**. באופן דומה כאן יורדים עד לקצה הימני של העץ למציאת האיבר המקסימלי.

**keysToArray – O(n) –** מחזירה מערך ממוין לפי המפתחות של הצמתים בעץ.   
הפונקציה יורדת לפינה השמאלית בעץ עד לאיבר המינימלי, לאחר מכן קוראת ל getSuccessor n פעמים, ולבסוף מכניסה את מפתחות הצמתים למערך לפי סדר ממוין (כתוצאה מהקריאה ל getSuccessor).  
נשים לב כי אנו עוברים על כל קשת בעץ לכל היותר פעמיים במהלך ביצוע פעולות אלו לצורך המיון בעזרת getSuccessor ולכן סך הפעולות יהיה .

**infoToArray - O(n) –** פועלת באופן דומה לkeysToArray כאשר השוני היחיד הוא בכך שהיא מחזירה במערך את הערך (info) של הצמתים.

**size - O(1) –** ניגשת ישירות לשדה size של העץ ומחזירה אותו (מס' האיברים בעץ). בכל פעולת הכנסה/מחיקת צומת מהעץ דאגנו לעדכן את השדה ולכן מדובר בסיבוכיות O(1).

**getRoot - O(1) –** ניגשת ישירות לשדה root של העץ ומחזירה אותו.

**split - O(log(n)) –** בהינתן צומת הנמצא בעץ בעל מפתח x הפונקציה מפצלת את העץ לשני תתי עצים כאשר האחד הוא בעל המפתחות הקטנים מx והשני בעל מפתחות הגדולים מ x.   
ראשית הפונקציה מבצעת קריאה לsearch כדי למצוא את x . לאחר מכן היא יוצרת משני בניו (שורשים) שני תתי עצים ומנתקת את הקשרים של x תוך שמירה על אבא שלו. לאחר מכן מתבצע תהליך הספליט על פי מה שנלמד בהרצאה, בכל שלב מתבצעת בדיקה אם הצומת המוצבע קטן מבנו או גדול ממנו ובהתאם מבצעים join עם תת העץ הרלוונטי. לבסוף מוודאים שלשני השורשים של תתי העצים שנוצרו אין הורים ושני העצים מוחזרים.  
מבחינת סיבוכיות מתבצע חיפוש בעלות O(logn), לאחר מכן מתבצע תהליך הספליט שכמו שראינו בהרצאה עלותו גם O(logn), אמנם מתבצעות פעולות join במהלך התהליך אך כפי שראינו בבהרצאה מכיוון שעלות כל join היא הפרש גבהי תתי העצים שמחברים, מתקבל בסה"כ שעלות ביצוע הספליט היא O(logn). מעבר לכך שינוי שדות על מנת לבצע חיבור וניתוק קשתות מתבצע בזמן קבוע ויחד עם ביצוע הsearch יוצא שהסיבוכיות הכוללת היא O(logn).

**Join - O(log(n)) –** בהינתן צומת x ועץ t שכל מפתחותיו קטנים או גדולים מכל המפתחות בעת הנוכחי שלנו, הפונקציה מחברת אותם לכדי עץ אחד.  
ראשית מתבצעות מס' בדיקות של מקרי קצה בהם אחד מהעצים ריק. אם שניהם ריקים מחזירה 0 ואם רק אחד ריק מתבצע insert של x לעץ שאינו ריק ומוחזר גובה העץ הלא ריק + 1 .   
לאחר מכן מתבצעות בדיקות בוליאניות כדי לגלות מי מהעצים הוא הגבוה מבין השניים ומי מהעצים הוא בעל המפתחות היותר גדולים. במידה ולשני העצים אותו גובה פשוט מחברים את שניהם לצומת x . לאחר מכן מתבצעת לולאה שמטרתה לרדת בעץ הגבוה עד לגובה של העץ הנמוך. בפעולת הjoin עצמה מתבצעים חיבור וניתוק קשתות בין הצומת x לצמתים הרלוונטיים בעץ הגבוה בהתאם למה שלמדנו בהרצאה.  
לאחר מכן מעדכנים דרגות של צמתים ושדות size . לבסוף יש לולאה שמטרתה לעלות במעלה העץ הגבוה ולקרוא ל heightCheckInsert על מנת לבדוק אם יש צורך באיזונים לעץ החדש שנוצר.  
לבסוף הפונקציה מחזירה את הפרש גבהי עצים + 1 .  
סה"כ מתבצעות מס' פעולות בוליאניות המתבצעות בזמן קבוע, מתבצעות פעולות ניתוק וחיבור קשתות בין צמתים ועדכון שדות בזמן קבוע. הקריאות ל heightCheckInsertמתבצעות בזמן קבוע גם כן ולכן סה"כ הסיבוכיות הכוללת תהא עלות הפעולות בירידה בעץ הגבוה ועוד העלייה בחזרה עד למעלה במידת הצורך אם צריך איזונים, ולכן הסיבוכיות הכוללת תהא הפרש גבהי העצים + 1. במידה ועץ אחד בעל n צמתים והעץ השני ריק הסיבוכיות הכוללת תהיה O(logn).

**מחלקת AVLNode**

**getKey – O(1)** -ניגשת לשדה key של צומת ומחזירה את ערכו.

**getValue – O(1)** - ניגשת לשדה value של צומת ומחזירה את ערכו.

**setLeft – O(1)** - ניגשת לשדה left של צומת ומעדכנת אותו.

**getLeft – O(1)** - ניגשת לשדה left של צומת ומחזירה את ערכו.

**setRight – O(1)** - ניגשת לשדה right של צומת ומעדכנת אותו.

**getRight – O(1)** - ניגשת לשדה right של צומת ומחזירה את ערכו.

**setParent – O(1)** - ניגשת לשדה parent של צומת ומעדכנת אותו.

**getParent – O(1)** - ניגשת לשדה parent של צומת ומחזירה את ערכו.

**setRealNode – O(1)** - מעדכנת צומת כצומת "אמיתי" ומעדכנת את השדות שלו.

**isRealNode – O(1)** - ניגשת לשדה rank של צומת ומחזירה ערך בוליאני בהתאם לערכו.

**setHeight – O(1)** - ניגשת לשדה rank של צומת ומעדכנת אותו (בעץ AVL rank=height).

**getHeight – O(1)** - ניגשת לשדה rank של צומת ומחזירה את ערכו (בעץ AVL rank=height).

**heightCheckInsert – O(1)** – הפונקציה בודקת האם יש צורך בגלגול או בהעלאה בדרגה של צמתים בעקבות הכנסת צומת לעץ. עבור הצומת שהוכנס היא בודקת תנאים בוליאניים לגבי ה balance factor שלו ושל בניו על מנת לסווג למקרים השונים של חוסר איזון שעלולים להיווצר כתוצאה מההכנסה.  
בהתאם לבדיקות היא מבצעת גלגול אחד או שניים (ראינו שלא ייתכנו יותר משני גלגולים בהכנסה) או שהיא מעלה בדרגה את הצמתים הרלוונטיים. במידת הצורך מתבצע מס' קבוע של קריאות רקורסיביות במעלה העץ עד ביצוע הגלגול, למקרה שיש לבצע העלאות בדרגה לצמתים שהופרה תקינותם בעקבות ההעלאה בדרגה של בנם.  
סיבוכיות הפונקציה O(1) כיוון שרק לעיתים רחוקות נצטרך לעלות במעלה העץ עד השורש (זה יקרה במצב בו מבצעים הכנסה לעץ מאוזן מלא) כדי לתקן דרגות של צמתים שבנם הועלה בדרגה.   
\* בנוסף קיים מונה שתפקידו לספור את מס' הגלגולים וההעלאות בדרגה שמתבצעים על מנת שנוכל להשתמש בכך בפונקציית insert.

**heightCheckDelete – O(1) –** פועלת בדומה ל heightCheckInsert אך עבור המקרה של מחיקת צומת מהעץ. גם כאן מתבצעות בדיקות balance factor לצורך ביצוע גלגולים או הורדה בדרגה, ותיקון של צמתים במעלה העץ.  
\*גם כאן קיים מונה שסופר גלגולים והורדות בדרגה לצורך שימוש עתידי בdelete.

**getBF – O(1)** - ניגשת לשדות rank של בניו של צומת ומחזירה את ההפרש בין הדרגות של הבן השמאלי ושל הבן הימני. עבור צומת וירטואלי מחזירה -3 (ערך שיהיה לא רלוונטי לבדיקות האיזון).

**rotateRight – O(1)** - מבצעת גלגול ימינה כפי שנלמד בהרצאה.  
הפונקציה שומרת את ה"סבא" של הצומת הרלוונטי כדי לא לנתק קשר עם שאר העץ, לאחר מכן מבצעת את כל הקישורים הרלוונטיים לצורך הגלגול, ולבסוף היא מעדכנת את שדות הrank והsize בהתאם.  
מדובר בפעולות קבועות ולכן הסיבוכיות הינה O(1).

**rotateLeft - O(1)** – מבצעת גלגול שמאלה באופן אנלוגי לחלוטין לrotateRight.

**getPredecessor – O(logn) ­–** הפונקציה מחזירה את האיבר הקודם של הצומת שהיא פועלת עליו בהתאם לדרך שלמדנו בהרצאה. אם הצומת הוא בן שמאלי, האיבר הקודם נמצא כל הדרך למעלה עד שיש הורה "שמאלי" (שהצומת הנוכחי הוא בן ימני שלו), זהו הקודם. אחרת לצומת יש בן שמאלי, ואז הפונקציה יורדת לבנו השמאלי ואז כל הדרך ימינה עד למציאת הקודם.  
מדובר במעבר על צומת אחד בכל רמה בעץ עד למציאת הצומת המבוקש ולכן הסיבוכיות של פונקציה זו היא כגובה העץ, log(n).

**getSuccesor – O(logn) ­–** הפונקציה מחזירה את האיבר העוקב של הצומת שהיא פועלת עליו בהתאם לדרך שלמדנו בהרצאה. אם הצומת הוא בן ימני, האיבר העוקב נמצא כל הדרך למעלה עד שיש הורה "ימני" (הורה שהצומת הנוכחי הוא בן שמאלי שלו), זהו העוקב. אחרת לצומת יש בן ימני, ואז הפונקציה יורדת לבנו הימני ואז כל הדרך שמאלה עד למציאת העוקב.  
מדובר במעבר על צומת אחד בכל רמה בעץ עד למציאת הצומת המבוקש ולכן הסיבוכיות של פונקציה זו היא כגובה העץ, log(n).

**setSize – O(1) –** ניגשת לשדות הsize של בניו של צומת ומחזירה את סכומם +1.

**חלק תיאורטי**

**שאלה 1**

**א.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מס' סידורי i | מס' חילופים במערך ממוין הפוך | עלות החיפושים במיון AVL עבור מערך ממוין הפוך | מס' חילופים במערך מסודר אקראית | עלות החיפושים במיון AVL עבור מערך מסודר אקראית |
| 1 | 1,999,000 | 36,885 |  |  |
| 2 | 7,998,000 | 81,765 |  |  |
| 3 | 31,996,000 | 179,525 |  |  |
| 4 | 127,992,000 | 391,045 |  |  |
| 5 | 511,984,000 | 846,085 |  |  |

**הסבר סעיף א'**

**מס' חילופים במערך ממוין הפוך:**

במערך ממוין הפוך מדובר במס' חילופים מקסימלי שכן לאיבר הראשון קיימים n-1 איברים באינדקסים גדולים ממנו שקטנים ממנו, לאיבר השני n-2 איברים כאלו וכך הלאה.  
מדובר בסכום סדרה חשבונית מ1 עד n-1 שזה גם מס' האפשרויות לבחירת זוג איברים מתוך n איברים:

**עלות החיפושים במיון AVL עבור מערך ממוין הפוך:**

כתבנו תוכנית שמתחילה את ספירת פעולות החיפוש מהאיבר המקסימלי, ומחזירה את סך פעולות החיפוש המופיע בעמודה זו בטבלה בהתאם ל n.  
\* נציין כי הפעולות אשר ספרנו הן מספר הקשתות שטיילנו על פניהן ועוד הקשת האחרונה שנוצרה כתוצאה מההכנסה עצמה.

**עלות החיפושים במיון AVL עבור מערך מסודר אקראית:**

כתבנו תוכנית שבאותו אופן מתחילה את הספירה מהאיבר המקסימלי, עבור עץ שהוכנסו לו צמתים בסדר אקראי.  
בסיום 100 הרצות הכנסנו לטבלה את הערך הממוצע כאשר כל הערכים שהתקבלו נעו בטווח של 95%-105% מהערך שהכנסנו לטבלה.

**מס' חילופים במערך מסודר אקראית:**

כתבנו תוכנית שמסדרת את הצמתים במערך לפי סדר הכנסתם לעץ, כאשר התוכנית ספרה את מס' החילופים, כלומר את מס' הזוגות .  
בטבלה הכנסנו את הערך הממוצע לאחר 100 הרצות.

**ב.**

**מס' חילופים במערך ממוין הפוך – – סיבוכיות**

מכיוון שהמערך ממוין בסדר הפוך, כל זוג שנבחר יענה לקריטריון הנ"ל ולכן התוצאה שתתקבל היא מס' הזוגות האפשריים מתוך n איברים.   
\* בהסבר לסעיף א' ציינו את העובדה שמדובר בסכום סדרה חשבונית עד n-1, דבר שגם מצדיק את התשובה הנ"ל.

**עלות החיפושים במיון AVL עבור מערך ממוין הפוך -**

חסם עליון - עבור הרמה הסופית של העץ אנחנו נכניס לכל היותר איברים. כאשר הפעולה היקרה ביותר תעלה לנו (עלייה בקצה ימני תחתון של העץ עד לשורש, וירידה לקצה שמאלי תחתון של העץ). עבור רמה מעל נכניס לכל היותר איברים, כאשר ההכנסה היקרה ביותר עבורן תהא .  
ועבור כל הרמות מעל נגדיר עבור הרמה שנכניס איברים בעלות כלומר עלות נמוכה מ . ולכן יש לנו n הכנסות בעלות של לכל היותר ומכאן החסם העליון הינו .

חסם תחתון – עבור הרמה נכניס איברים בעלות ולכן נקבל רק עבור רמה זאת ולכן רק עבור כמות פעולות חלקית זו נקבל בעקבות חלוקה ב נקבל קבוע כלשהו גדול מ0, כלומר .

\*נציין כי עלות הפעולה היא בדיוק של +- 1 , ומכיוון שהחסמים גדולים מn אנחנו יודעים שלהוספה או החסרה של +-n פעולות, אין השפעה על החסם האסימפטוטי.

**ג.**

**כן.**

עבור מס' החילופים במערך ממוין הפוך, קיבלנו את התוצאה המדויקת ע"פ הנוסחה שהזנו .  
עבור עלות החיפושים במיון AVL עבור מערך ממוין הפוך, ביצענו את הטרנספורמציה הבאה על עלות החיפושים -

ויצרנו גרף בתוכנת הסטטיסטיקה R כאשר ציר הX הינו גודל הקלט וציר הY הינו הביטוי הנ"ל לאחר הטרנספורמציה.  
הוספנו גם את הגרף ללא הטרנספורמציה.  
כמו כן, קיבלנו ערך ללא הטרנפורמציה וערך עם הטרנספורמציה.  
דבר אשר מעיד על קשר לינארי חזק לאחר הטרנספורמציה, כלומר לאחר שחילקנו את עלות החיפושים ב קיבלנו קשר לינארי מובהק, ולכן הקשר המקורי בין גודל הקלט n לבין סך עלות החיפושים הוא  *, כמו שצפינו.*

*מצורף בסוף השאלה פלט התוכנית ב R כולל הגרפים עם ובלי הטרנספורמציה הנ"ל.*

***ד.***

*בעת הכנסת איבר* i *לעץ, מס' ההחלפות שנגזרות מהכנסתו נקבע ע"H כמות האיברים הגדולים ממנו שכבר נמצאים בעץ (כך אנו מוודאים שאנחנו לא סופרים אף החלפה פעמיים). מכאן, שה של כל איבר שווה לדירוג שלו ביחס לאיברים שהוכנסו עד אליו, כלומר שאם יש לו החלפות, אזי הוא האיבר ה . בהינתן שאנו מכניסים מצביע למקסימום ומבצעים את פעולות החיפוש לפני הכנסה מהמקסימום, אנו מבינים שפעולת החיפוש היא בעצם פעולת* select(tree, *, שראינו בכיתה שסיבוכיותה היא .  
מכאן, עבור* n *הכנסות ע"פ התנאים שהוגדרו יהיו שוות בסיבוכיותן ל*n *פעולות* select *כפי שתואר.  
נסכום פעולות אלו לקבלת: וזהו החסם העליון הנדרש.  
נפרט את החישובים:*

* *המקדם המקסימלי מבין כל פעולות ה* select.
* *סכום לוגים=לוג מכפלות.*
* *העלאה בחזקה והוצאת שורש מנטרלנות אחת את השנייה. בוצעו על מנת להשתמש באי שיוויון הממוצעים.*
* *הפיכת חזקת* n *למקדם* n *ע"פ חוקי לוג.*
* *מעבר אחרון אי שיוויון הממוצעים.*

AVLTree project – Q1.c

input <- c(2000,4000,8000,16000,32000)  
numOfOps <- c(36885,81765,179525,391045,846085)  
numOfOpsWithTransformation <- c(36885/log2(2000),81765/log2(4000),179525/log2(8000),391045/log2(16000),846085/log2(32000))  
noTransformation <- lm(numOfOps~input)  
withTransformation <- lm(numOfOpsWithTransformation~input)  
#summary(noTransformation)  
#summary(withTransformation)  
plot(input,numOfOps, main="no transformation")  
abline(noTransformation)

Chart

Description automatically generated

plot(input,numOfOpsWithTransformation, main="with transformation")  
abline(withTransformation)

Chart

Description automatically generated

summary(noTransformation)$r.squared

## [1] 0.9989568

summary(withTransformation)$r.squared

## [1] 0.9999858

**שאלה 2**

**א.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מס' סידורי i | עלות join ממוצע עבור split אקראי | עלות join מקסימלי עבור split אקראי | עלות join ממוצע עבור split של האיבר המקסימלי בתת העץ השמאלי | עלות join מקסימלי עבור split של האיבר המקסימלי בתת העץ השמאלי |
| 1 |  |  |  | 13=12-0+1 |
| 2 |  |  |  | 14=13-0+1 |
| 3 |  |  |  | 15=14-0+1 |
| 4 |  |  |  | 16=15-0+1 |
| 5 |  |  |  | 17=16-0+1 |
| 6 |  |  |  | 18=17-0+1 |
| 7 |  |  |  | 19=18-0+1 |
| 8 |  |  |  | 20=19-0+1 |
| 9 |  |  |  | 21=20-0+1 |
| 10 |  |  |  | 22=21-0+1 |

הסבר לטבלה:

\*\*\* בסוגריים מופיעים ערכי ממוצע לאחר כ-20 הרצות.  
מה שלא בסוגריים התקבל לאחר הרצה בודדת.

**עלות join ממוצע עבור split אקראי -**  
כתבנו תוכנית אשר מייצרת עץ אקראי ומפצלת אותו ע"פ צומת אקראי, והדפסנו את עלויות פעולות הjoin כולל הממוצע והמקסימום.

**עלות join מקסימלי עבור split אקראי -**   
נשים לב כי באופן תיאורטי נוכל לקבל גם את המספרים שהתקבלו בעמודה הכי שמאלית. מדובר בjoin המקסימלי שיתכן במצב בו מבצעים את הsplit על המפתח המקסימלי בתת העץ השמאלי של השורש.  
בפועל ברוב המוחלט של המקרים יתקבלו ערכים נמוכים בהרבה.

**עלות join ממוצע עבור split של האיבר המקסימלי בתת העץ השמאלי -**כתבנו תוכנית אשר מייצרת עץ אקראי ומפצלת אותו ע"פ הנדרש, והדפסנו את עלויות פעולות הjoin כולל הממוצע והמקסימום.

**עלות join מקסימלי עבור split של האיבר המקסימלי בתת העץ השמאלי -** .  
ראו פירוט תשובה לסעיף ג'.  
\* במידה ואנו מתייחסים לחיבור בין תת עץ רגיל לתת עץ ריק כך שדרגתו של תת העץ הריק היא 1- כפי שהונחנו לעשות, יש להוסיף 1+ לכל העמודה הנ"ל.

**ב.**

**עלות join ממוצע עבור split של האיבר המקסימלי בתת העץ השמאלי -**עבור כל פעולות הjoin למעט האחרונה אנו נחבר בין עצים שהפרשי הגבהים שלהם הם בין 0 עד 2, היות ואנחנו מתחילים מחיבור עץ ריק לעת בגובה לכל היותר 2. ומכיוון שתת העץ השמאלי (אשר בתוכו מתבצעים כל החיבורים הראשונים) אנחנו לא נקבל 2 כל הזמן, אלא נקבל תוצאות של 1 ו2 בהתפלגות קרובה לנורמלית. כלומר, עלות פעולה ממוצעת היא בערך 1.5. פעולת החיבור האחרונה תהיה בין עץ בגובה 0 לעץ בגובה log(n)-1 וכשנסכום את כל הפעולות האלה ונחלק בlog(n) (+-1 עקב כך ששתי הרמות האחרונות לא בהכרח מלאות) פעולות join , נקבל ערך שהוא כפי שנראה למטה וזה אכן תואם את התוצאות שהתקבלו בממוצע לאחר הרצות רבות, ודומה להרצה הבודדת שתיעדנו.

**עלות join ממוצע עבור split אקראי -**בעבור פיצול מצומת אקראי, רוב פעולות החיבור יתבצעו בין עצים שהפרש הגבהים ביניהם הוא 1-2, למעט במקרים בהם ביצענו מספר פניות רצופות ימינה או שמאלה כדי להגיע לצומת בעת החיפוש. לסכום הזה יתווספו עלות הפעולות היקרות אשר תלויות בגובה הצומת בו התחלנו את הפיצולים. ככל שצומת זה נמוך יותר, פעולות אלה יצטברו לסכום גבוה יותר אך לא גבוה יותר מlogn וברוב המקרים לפחות ממנו (שכן רק חצי מהצמתים בערך הם בתחתית העץ, כלומר רק עבורם נצבור פעולות יקרות בעלות logn ועבור שאר הצמתים אם נתחיל בהם נצבור פחות). מכאן שיש לנו שוב סכימה של 1.5 בערך כפול מספר פעולות החיבור ועוד סכום פעולות יקרות בעלות של log של העומק בו התחלנו. מכאן אנו צופים לקבל ערכים נמוכים יותר מהתרחיש הקיצוני שכן עלותו המצטברת מתרחשת רק בחצי מהצמתים ובשאר המקרים נקבל סכום קטן יותר ככל שהצומת ממוקם גבוה יותר בעץ. בניסוי קיבלנו ערכים מתאימים לכך שקרובים יותר ל2.5 מאשר בניסוי הקיצוני. אנו מעריכים שזה קשור לעומק בו נמצא הצומת איתו התחלנו ובסכום עלויות הפעולות היקרות שנגזרה מכך.

**ג.**

**עלות join מקסימלי עבור split של האיבר המקסימלי בתת העץ השמאלי -** .  
עץ בעל n איברים בו הרמה הלפני אחרונה לא מלאה יכול תיאורטית להיות בעל גובה (ייתכן וזה יהיה ממש log(n) ), אנו מחסרים ממנו 0 היות וגובה העץ הנמוך שאנו מחברים לתת העץ הימני הוא תמיד 0 כי הוא עץ ריק, ולכך אנו מוסיפים 1 ע"פ נוסחת הפרש הגבהים +1.  
\* במידה ואנו מתייחסים לחיבור בין תת עץ רגיל לתת עץ ריק כך שדרגתו של תת העץ הריק היא 1- כפי שהונחנו לעשות, יש להוסיף 1+ לכל העמודה הנ"ל.

**ד.**

במבט על עץ AVL ניתן לראות כי רק2 מתוךn צמתים יביאו לערך הקיצוני ביותר (קיצוני ימין בתת עץ שמאל וקיצוני שמאל בתת עץ ימין). 2 בחזקת 2 צמתים מהרמה התחתונה ועוד 2 צמתים מהרמה הלפני תחתונה יביאו לערך הקיצוני פחות אחד. 2 בחזקת שלוש צמתים ברמה התחתונה ועוד 2 בחזקת 2 צמתים מהרמה הלפני תחתונה ועוד 2 צמתים מהרמה השניים לפני תחתונה יביאו לערך הקיצוני פחות שתיים וכן הלאה. זאת על פי ההיגיון של לקחת קיצוניים נגדיים לתתי עצים בגובה מסוים.  
על פי חישוב זה, אנו מבינים שהסיכוי לקבל את הערך הקיצוני הוא 2\n והסיכוי לקבל את הערך הבא אחריו הוא 6\n והסיכוי לקבל את הערך הקיצוני הבא אחריו הוא 10/n וכן הלאה. כלומר כמות העלים שעבורם נקבל עלות max-i או גבוה ממנו שווה ל2 בחזקת i+1 . כדי לקבל ערך גבוה מ1 max-(max-1)=1 יש לנו רק 2 בחזקת logn-1 צמתים אשר יאפשרו לנו את זה. כלומר רק חצי מהצמתים יאפשרו עלות מקסימלית מעל 2(הפרש רמות +1), ורק רבע יאפשרו עלות מקסימלית מעל 3 ורק שמינית יאפשרו עלות מעל 4.

מכיוון שחצי מהצמתים שמאפשרים עלות שתיים הם צמתים שמאפשרים עלות 3 ומתוכם חצי הם צמתים שמאפשרים 4 וכן הלאה בהתאם להסבר הנ"ל. אנו נקבל את הטור

כלומר, מכיוון שטור זה מתכנס ניתן לראות שאין תלות בגודל הקלט n ולכן נגיד שהחסם שלנו הוא O(1).