<u>תיעוד הפרויקט המעשי עץ AVL</u>

Ruben Wolhandler rubenw 342674983
Daniel Malash danielmalash 208059113

מחלקת AVLNode

<u>שדות:</u>

`int key` – המפתח של הצומת
`String value` – הערך של הצומת
`IAVLNode left` – מצביע לבן השמאלי של הצומת
`IAVLNode right` - מצביע לבן הימני של הצומת
`IAVLNode parent` - מצביע לאבא של הצומת
`int height` – גובה הצומת (כלומר גובה תת העץ שמתחיל בצומת הנ"ל)

<u>בנאי:</u>

`AVLNode(int key , String value)` - יוצר צומת עם מפתח key וערך value.
אם המפתח שונה מ(-1), מתבצעת השמה של צמתים וירטואליים, כלומר עם מפתח (-1), להיות הבנים השמאלי והימני של הצומת. הגובה מוגדר להיות 0.
אם המפתח של הצומת הוא (-1), שני המצביעים לבנים הם null. הגובה מוגדר להיות -1.

<u>מתודות:</u>

`getKey()` - מחזירה את שדה המפתח.
`getValue()` - מחזירה את שדה הערך.
`setLeft(IAVLNode node)` - הופכת את node להיות הבן השמאלי של הצומת (this).
`getLeft()` - מחזירה את שדה הבן שמאלי של הצומת.
`setRight(IAVLNode node)` - הופכת את node להיות הבן הימני של הצומת (this).
`getRight()` - מחזירה את שדה הבן הימני של הצומת.
`setParent(IAVLNode node)` - הופכת את node להיות האבא של הצומת (this).
`getParent()` - מחזירה את שדה ההורה של הצומת.
`isRealNode()` - מחזירה האם הצומת הינה צומת וירטואלית או אמיתית.
`setHeight(int height)` - מבצעת השמה לשדה הגובה את הערך height.
`getHeight()` - מחזירה את שדה הגובה.

הסיבוכיות של כל המתודות האלה היא O(1).

מחלקת AVLTree

<u>שדות:</u>

`IAVLNode root` – צומת שהיא השורש של העץ
`int balance_counter` – כמות פעולות האיזון שעשינו לאחר פעולת הכנסה או מחיקה (מאפסים את השדה לפני כל פעולה כזו, ובסופה מחזירים את השדה).
`IAVLNode min` – הצומת המינימלי בעץ
`IAVLNode max` – הצומת המקסימלי בעץ

<u>בנאים:</u>

`AVLTree()` - יוצר עץ, שהשורש שלו הוא null.
הbalance_counter הוא אפס, הצמתים המינימלי והמקסימלי מאותחלים לnull.

`AVLTree(IAVLNode root)` - יוצר עץ, שהשורש שלו הוא root אם הצומת root אמיתי. אם לא, אז יוצר עץ שהשורש שלו הוא null.

מעדכן את שדות הmin, max ע"י קריאה ל `findMin(), findMax()` וזאת על מנת למצוא את הצמתים המינימלי והמקסימלי בתת העץ שמתחיל בroot. [נשתמש בבנאי זה עבור split].
עקב השימוש במתודות מציאת מינ' ומקס' נקבל **סיבוכיות:** O(log(n))

<u>מתודות:</u>

`empty()` – **פלט:** מחזירה אמת אם העץ ריק, ושקר אחרת. (בודקת האם השורש הוא null).
סיבוכיות: O(1).

`search(int k)` – מחפשת את המפתח k בעץ.
פלט: אם מופיעה צומת עם מפתח k תחזיר את הערך שלו, אחרת תחזיר null.
הסבר מפורט: עוברת על כל הצמתים בעץ, כאשר ההתקדמות היא שמאלה או ימינה בהתאם לערכי המפתחות. כלומר אם המפתח שאנו מחפשים קטן מהמפתח של הצומת הנוכחי נלך שמאלה, אם גדול נלך ימינה, ואם שווה נחזיר את ערכו.
סיבוכיות: O(log(n))

`insert(int k, String i)` – מכניסה צומת חדשה לעץ שהמפתח שלה שווה לk והערך שלה שווה לi. לאחר ההכנסה מבצעת איזון של העץ.
פלט: מחזירה את מס' פעולות האיזון שנדרשו על מנת לאזן את העץ לאחר ההכנסה.
הסבר בפירוט: אם העץ ריק, תשים את הצומת כשורש של העץ. אחרת, בודקת אם המפתח קיים בעץ בעזרת `search(int k)`, אם כן מחזירה -1. אם לא, בודקת איפה הצומת שלאחריה צריכה להופיעה הצומת החדשה (נעזרת בפונקציה `searchForInsert`) לאחר מכן, נעשת ההכנסה בעזרת `insertNode(IAVLNode node,IAVLNode nodeInsert )` ואז נעשה איזון העץ ע"י `rebalancing.`
סיבוכיות: O(log(n))

`insertNode(IAVLNode node,IAVLNode nodeInsert )` – מקבלת שני צמתים: את הצומת שלאחריו נעשה הכנסה, ואת הצומת שאותו נכניס. המתודה מבצעת את ההכנסה לעץ. מעדכנת את השדות min וmax.
הסבר בפירוט: אם מפתח הצומת המוכנס קטן מהמפתח של הקודם נעשה לצומת המוכנס השמה להיות הבן השמאלי של הקודם, ואם גדול אז השמה להיות הבן הימני של הקודם (עבור שווה לא מתבצעת הכנסה). בנוסף, בודקת האם הצומת המוכנס קטן מהמינימום הנוכחי, אם כן מעדכנת אותו. אם המפתח גדול מהמקסימום הנוכחי, אז מעדכנת את המקסימום.
סיבוכיות: O(1)

`rebalancingAfterInsert(IAVLNode node,IAVLNode nodeInsert)` – מקבלת שני צמתים: את הצומת שלאחריו עשינו הכנסה, ואת הצומת שאותו הכנסנו. ועושה איזון של העץ בהתאם. (נקראת לאחר פעולת הכנסה, כלומר האיזון שנעשה הוא ביחס לצומת שהוכנס).
הסבר בפירוט: בודקים כל פעם איזה מקרה אנו נמצאים, ואז מבצעים את הפעולות הדרושות בכל אחד מהמקרים כפי שלמדנו בכיתה. **סיבוכיות:** במקרה הגרוע O(log(n))

`promoteForInsert(IAVLNode node)` – מקבלים את הצומת עליה צריך לעשות promote, כלומר להעלות את הדרגה שלה ב1. ועושים העלאה של הגובה שלה ב1. הפונקציה תשתמש ב`promote` בנוסף, מתבצעת הוספה של 1 לכמות פעולות האיזון שעשינו.
סיבוכיות: O(1)

`rotationLeftforInsert(IAVLNode node)` – מקבלת את צומת האב, ומבצעת סיבוב שמאלה.
הסבר בפירוט: מורידה את גובה האב ב1, וקוראת ל`rotationLeft` על מנת לבצע את הסיבוב.
סיבוכיות: O(1)

`rotationRightforInsert(IAVLNode node)` – מקבלת את צומת האב, ומבצעת סיבוב ימינה.
הסבר בפירוט: מורידה את גובה האב ב1, וקוראת ל`rotationRight` על מנת לבצע את הסיבוב.
סיבוכיות: O(1)

`LRforInsert(IAVLNode node,IAVLNode nodeInsert)` – מקבלת שני צמתים: את הצומת שלאחריו עשינו הכנסה, ואת הצומת שאותו הכנסנו. מבצעת סיבוב שמאלה ואז ימינה.
הסבר בפירוט: מבצעת פעמיים demote, פעם אחת promote ולאחר מכן קוראת ל `rotationLeft` ו `rotationRight`. (קוראים לה רק מתוך rebalancing והיא מיועדת למקרה ספציפי). **סיבוכיות:** O(1)

`RLforInsert(IAVLNode node,IAVLNode nodeInsert)` – מקבלת שני צמתים: את הצומת שלאחריו עשינו הכנסה, ואת הצומת שאותו הכנסנו. מבצעת סיבוב ימינה ואז שמאלה.
הסבר בפירוט: מבצעת פעמיים demote, פעם אחת promote ולאחר מכן קוראת ל `rotationLeft` ו `rotationRight`. (קוראים לה רק מתוך rebalancing והיא מיועדת למקרה ספציפי). **סיבוכיות:** O(1)

`searchForInsert(int k)` – מקבלת את המפתח שאותו רוצים להכניס לעץ, ומחזירה את הצומת שלאחריו צריך להכניס את הצומת החדש. **הסבר בפירוט:** מחפשים החל מהשורש ועד שמגיעים למקום בו צריך להכניס את הצומת החדש (זזים שמאלה וימינה בהתאם לערכי המפתחות) ולבסוף מחזירים את האבא של המקום הזה, כי זו הצומת שלאחריה נרצה להכניס. **סיבוכיות:** O(log(n))

`promote(IAVLNode y)` –מקבלים צומת ועושים ועוד 1 לשדה הגובה שלו. **סיבוכיות:** O(1)

`demote(IAVLNode y)` – מקבלים צומת ועושים פחות 1 לשדה הגובה שלו. **סיבוכיות:** O(1)

`rotationRight(IAVLNode y)` – מקבלת את צומת האב של תת העץ אותו אנו רוצים לסובב ומסובבת אותו ימינה. (שינינו את כל המצביעים בהתאם, ככה שיתבצע הסיבוב) **סיבוכיות:** O(1)

`rotationLeft(IAVLNode y)` – מקבלת את צומת האב של תת העץ אותו אנו רוצים לסובב ומסובבת אותו שמאלה. (שינינו את כל המצביעים בהתאם, ככה שיתבצע הסיבוב) **סיבוכיות:** O(1)

`isLeaf(IAVLNode x)` – מקבלת צומת ובודקת האם הצומת הזה הוא עלה בעץ. (האם שני הבנים שלו הם צמתים וירטואליים). **סיבוכיות:** O(1)

`isUnaryNode(IAVLNode x)` – מקבלת צומת ובודקת האם הצומת הזה הוא צומת אונארי בעץ. (האם יש לו רק בן אחד שהוא אמיתי, ובן אחד שהוא וירטואלי). **סיבוכיות:** O(1)

`searchNode(int k)` – מקבלת מפתח ומחזירה את הצומת בעל המפתח הזה אם הוא קיים בעץ, אחרת מחזירה null. **הסבר בפירוט:** עוברת מהשורש עד שהיא מגיעה לצומת המבוקש כאשר זזים שמאלה וימינה בהתאם לערכי המפתחות. **סיבוכיות:** O(log(n))

`delete(int k)` – מקבלת מפתח ובמידה והצומת בעל המפתח הזה קיים, מוחקת אותו.
הסבר בפירוט: תחילה נקרא ל`searchNode` ובמידה והצומת קיימת, פיצלנו למקרים בהן הצומת היא עלה או צומת אונארית וטיפלנו בהתאם. כאשר הצומת היא בעלת שני בנים, נשתמש ב`successor` על מנת להחליף את הצומת הנמחק בצומת העוקב לו. בנוסף, במידה והצומת הנמחק הוא או הצומת המינימלי או המקסימלי, נעדכן את השדה בהתאם ע"י קריאה ל`findMin()` או `findMax()` בהתאם. כל אחת ממתודות העזר היא בסיבוכיות של O(log(n)), ולכן נקבל
סיבוכיות: O(log(n))

`diffRight(IAVLNode parent)` –מקבלת צומת (שהיא צומת אמיתית) ומחזירה את הפרש הגבהים בינה לבין הבן הימני שלה. **סיבוכיות:** O(1)

`diffLeft(IAVLNode parent)` – מקבלת צומת (שהיא צומת אמיתית) ומחזירה את הפרש הגבהים בינה לבין הבן השמאלי שלה. **סיבוכיות:** O(1)

`rebalancingAfterDelete(IAVLNode parent)` – מקבלת צומת אב של הצומת אותו מחקנו מהעץ. מבצעת איזון של העץ לאחר פעולת המחיקה. (נקראת רק מתוך delete).
הסבר בפירוט: פיצלנו למקרים כאשר הצומת הנמחק הוא בן ימני או בן שמאלי, ולכל אחד מהם טיפלנו בכל המקרים האפשריים לחוסר איזון בעץ לאחר מחיקה, כפי שנלמד בכיתה. נשים לב שיש מקרים בהם בעיית האיזון עולה במעלה העץ, ולכן עבור מקרים אלה הפעלנו את המתודה שוב על האב הבא במעלה העץ. אם הצומת הנמחק הוא עלה, אז לכל היותר כמות הפעמים שנקרא למתודה היא כגובה העץ. ולכן במקרה הגרוע **סיבוכיות:** O(log(n))

`min()`– מחזירה את הערך של הצומת בעלת המפתח הקטן ביותר בעץ, אם העץ ריק תחזיר null.
הסבר בפירוט: בודקת אם השדה min הוא null , אם לא מחזיר את הערך של צומת זה.
סיבוכיות: O(1)

`max()`– מחזירה את הערך של הצומת בעלת המפתח הגדול ביותר בעץ, אם העץ ריק תחזיר null.
הסבר בפירוט: בודקת אם השדה max הוא null, אם לא מחזיר את הערך של צומת זה.
סיבוכיות: O(1)

`keysToArray()`– מחזירה מערך ממויין של המפתחות בעץ. או מערך ריק אם העץ ריק.
הסבר בפירוט: הפונקציה עוברת על כל המפתחות בעץ החל מהמינימום ועד למקסימום כאשר המעבר בין הצמתים הוא באמצעות קריאה ל`successor` , ומכניסה למערך את המפתח של הצומת. אנו עוברים על כל האיברים בעץ ולכן נקבל **סיבוכיות:** O(n)

`infoToArray()`– מחזירה מערך המכיל את ערכי כל הצמתים בעץ, כאשר המערך ממויין לפי גודל המפתחות בעץ. אם העץ ריק תחזיר מערך ריק.
הסבר בפירוט: הפונקציה עוברת על כל המפתחות בעץ החל מהמינימום ועד למקסימום כאשר המעבר בין הצמתים הוא באמצעות קריאה ל`successor` , ומכניסה למערך את הערך של הצומת. אנו עוברים על כל האיברים בעץ ולכן נקבל **סיבוכיות:** O(n)

`successor(IAVLNode node)` – מקבלת צומת ומחזירה את הצומת העוקבת לה בעץ.
הסבר בפירוט: אם לצומת יש בן ימני, אז ה-successor הוא הצומת המינימלי של תת העץ הימני (כלומר נלך עד הסוף שמאלה בתת העץ הימני של הצומת, לכל היותר גובה העץ).
אחרת, נרצה להגיע לצומת שנמצאת מעל הצומת שלנו, מצד ימין לה ("פניה ראשונה ימינה"). אם אין כזו, נחזיר null. **סיבוכיות:** O(log(n))

`findMin()`– מחזירה את הצומת המינימלי בעץ. (כל עוד הבן השמאלי הוא צומת אמיתי, נעבור אליו).
סיבוכיות: O(log(n))

`findMax()`– מחזירה את הצומת המקסימלי בעץ. (כל עוד הבן הימני הוא צומת אמיתי, נעבור אליו).
סיבוכיות: O(log(n))

`SubTreeSize(IAVLNode root)` – מחזירה את כמות הצמתים בתת העץ שמתחיל בצומת שקיבלנו.
הסבר בפירוט: אם הצומת אינו אמיתי או null נחזיר 0. המתודה תקרא לעצמה ברקורסיה כאשר בכל שלב היא קוראת לעצמה עם הבן השמאלי של הצומת ועוד עצמה עם הבן הימני של הצומת ועוד 1 (עבור הצומת עצמו). לכל היותר כמות השלבים היא גובה העץ. **סיבוכיות:** O(log(n))

`size()`– מחזירה את מספר הצמתים בעץ.
הסבר בפירוט: תקרא ל `SubTreeSize` עם שורש העץ, וע"י כך תספור את כמות הצמתים בעץ.
<u>הערה:</u> לא שמרנו size לכל node כי ראינו שבמנשק IAVLNode כתוב שאי אפשר לשנות אותו.
סיבוכיות: O(log(n))

`getRoot()`– מחזירה את שדה שורש העץ. **סיבוכיות:** O(1)

`split(int x)` – מפצלת את העץ לשני עצים, כך שהעץ הראשון מכיל את כל המפתחות הקטנים מx והעץ השני מכיל את כל המפתחות הגדולים מx. **תנאי קדם:** קיימת צומת בעץ עם המפתח x.
פלט: תחזיר מערך בגודל 2 מטיפוס עץ AVL כך שבמקום הראשון יהיה העץ בעל המפתחות הקטנים ובמקום השני העץ בעל המפתחות הגדולים.
הסבר בפירוט: תחילה תמצא את הצומת שהמפתח שלה הוא x ע"י `searchNode`, לאחר מכן תיצור רשימה של צמתים שהם אבות הצומת הנ"ל (כלומר, כל האבות החל מאב הצומת עד לשורש העץ). נוצרים שני עצי עזר עבור התהליך big וlittle, כאשר השורש של big הוא הבן הימני של x, והשורש של little הוא הבן השמאלי של x. כעת, עוברים על רשימת צמתי האבות, אם המפתח קטן מx עושים איחוד בעזרת המתודה `join` של תת העץ השמאלי של המפתח עם העץ little.
אם המפתח גדול מx עושים איחוד של תת העץ הימני שלו עם העץ big. כאשר נסיים לעבור על כל הרשימה (לכל היותר בגודל גובה העץ) נקבל את שני העצים המבוקשים (כמו שראינו בשיעור).
סיבוכיות: O(log(n))

`join(IAVLNode x, AVLTree t)` – מאחדת את העץ הנוכחי עם העץ t בעזרת הצומת x.
תנאי קדם: המפתחות של x ו t הם או גדולים מהמפתחות של העץ הנוכחי (this) או קטנים ממנו.
פלט: מחזירה את סיבוכיות הפעולה.
הסבר בפירוט: שדה שורש העץ יצביע ל`join(this, x, t)` שמתוארת למטה, בה יתבצע האיחוד. שדות המינימום והמקסימום יעודכנו בהתאם לעץ החדש שנוצר.
בסוף תחזיר את הפרש גובה העצים ועוד 1.
סיבוכיות: O(1) + O(log(n)) = O(log(n))

`join(AVLTree t1,IAVLNode x, AVLTree t2)`–מקבלת שני עצים וצומת כך שמתקיים שהמפתחות בעץ t1 ובצומת x או קטנים או גדולים מהמפתחות בעץ t2. (מתקיים עקב תנאי הקדם במתודה שקוראת למתודה הזו).
פלט: עץ AVL שהוא איחוד של שני העצים והצומת בקלט.
הסבר בפירוט: אם שני העצים ריקים, תחזיר עץ שהשורש שלו הוא x.
אם אחד מהעצים ריק, תכניס את x לעץ הלא ריק, תאזן אותו ואז תחזיר את העץ הזה.

אם שני העצים לא ריקים נבדוק למי יש את המפתחות הגדולים, אם ל- t1 יש את המפתחות הגדולים אז תקרא הפונקציה join(t2,x,t1) , אחרת האיחוד יתבצע כך (נחליף את המצביעים בהתאם למה שראינו בכיתה עבור פעולת האיחוד):
אם הגובה של הצומת של t2 ממש גדול מ-2 או יותר מה שורש של t1 ,"נרד" לשמאל עד שנגיע לצומת שהגובה שלו גדול ב-2 או 1 מהשורש של t1, צומת זה מסומן nodeT2. ואז nodeT2.getLeft() יצביע ל-x , ו-x.getRight יצביע לבן השמאלי המקורי של nodeT2.
x.getLeft() יצביע ל- שורש של t1. נבצע rebalancingforInsert כדי לאזן את העץ שקיבלנו.

אם הגובה של t1 ממש גדול מ-2 או יותר מה שורש של t2 , "נרד" לימין עד שנגיע לצומת שגובה שלו גדול מ-2 או 1 מהשורש של t2, נקרא לו nodeT1. ואז nodeT1.getRight() יצביע ל-x , ו- x.getLeft() יצביע לבן הימני המקורי של nodeT1. x.getRight() יצביע ל- שורש של t2. נבצע rebalancingforInsert כדי לאזן את העץ שקיבלנו.

אם ההבדל של גובהם של השורשים קטן או שווה ל-1 אז השורש יהיה x, תת עץ השמאלי של x יהיה העץ t1, תת עץ הימני יהיה t2.

סיבוכיות: O(log(n)) במקרה הגרוע, כשנצטרך לעשות את פעולת האיזון הכי יקרה.

<u>מדידות</u>

מספר סידורי	מספר פעולות	מספר פעולות האיזון הממוצע לפעולתinsert	מספר פעולות האיזון הממוצע לפעולתdelete	מספר פעולות האיזון המקסימלי לפעולתinsert	מספר פעולות האיזון המקסימלי לפעולת delete
1	10,000	2.4793	1.1628	15	14
2	20,000	2.4706	1.16485	16	16
3	30,000	2.4761333333333333	1.1638666666666666	17	16
4	40,000	2.4883	1.1681	17	16
5	50,000	2.47812	1.16428	18	18
6	60,000	2.4817833333333335	1.1675166666666668	18	17
7	70,000	2.4804	1.1679428571428572	18	17
8	80,000	2.4858	1.1673375	19	18
9	90,000	2.4824	1.1674666666666667	19	19
10	100,000	2.48674	1.16555	19	18

<u>מספר פעולות האיזון הממוצע לפעולת **insert**:</u>

נשים לב שמספר פעולות האיזון הממוצע אינו תלוי בגובה של העץ והוא שווה בערך ל-2.5, מספר קבוע. התוצאות שקיבלנו בפועל תואמות את הציפיות, כי ראינו בכיתה שמספר פעולות האיזון לאחר insert ב-amortized הוא O(1).

מספר פעולות האיזון הממוצע לפעולת **delete**:

נשים לב כי גם במחיקה מספר פעולות האיזון הנדרש בממוצע הינו קבוע, אינו תלוי בגודל העץ. כצפוי קיבלנו גם שמספר פעולות האיזון הממוצע לפעולת delete הוא בערך 1 **ואינו תלוי** בגובה עץ. (ב-O(1) amortized).

מספר פעולות האיזון המקסימלי לפעולת **insert**:

ראינו בהרצאה שבמקרה הגרוע נצטרך לעשות O(log(n)) **פעולות איזון** ב insert.
התוצאות שקיבלנו בפועל תואמות את הציפיות, כי מקבלים שמספר פעולות האיזון המקסימלי תלוי בגובה העץ, **ושווה** בערך ל- log(10,000 * i).

מספר פעולות האיזון המקסימלי לפעולת **delete**:

ראינו בהרצאה שבמקרה הגרוע נצטרך לעשות O(log(n)) **פעולות איזון** בdelete. התוצאות שקיבלנו בפועל תואמות את הציפיות, כי מקבלים שמספר פעולות האיזון המקסימלי תלוי בגובה העץ, **ושווה** בערך ל- log(10,000 * i).

2)

מספר סידורי	עלות join ממוצע עבור split אקראי	עלות join מקסימלי עבור split אקראי	עלות join ממוצע עבור split של איבר מקס בתת העץ השמאלי	עלות join מקסימלי עבור split של איבר מקס בתת העץ השמאלי
1	2.076923076923077	3	2.2	14
2	1.1875	3	2.3	14
3	1.2142857142857142	3	2.1	15
4	1.1538461538461537	2	2.4	15
5	1.3571428571428572	3	2.3	16
6	1.5333333333333334	2	2.6	16
7	1.3076923076923077	2	2.5	17
8	1.2	2	2.2	17
9	1.3333333333333333	2	2.7	17
10	1.3846153846153846	2	2.5	18

<u>עלות join ממוצע עבור split אקראי:</u>

הצומת שעושים עליה split, יש לה סיכוי מאוד גדול להיות באמצע העץ (לא להיות בקצה של העץ) ולכן מצופה שכל פעם שעושים join, עושים join על שני עצים שיש להם הבדל קבוע בגובה, ולכן מצופה שעלות הjoin ממוצע עבור split אקראי יהיה מספר קבוע, וזה מה שקיבלנו בפועל, קיבלנו ש join עולה בין 1 ל 2 פעולות.

<u>עלות join מקסימלי עבור split אקראי:</u>

הצומת שעושים עליו split, יש לו סיכוי מאוד גדול להיות באמצע העץ ואז סדרת האיחודים שעושים, יש לה סיכוי גבוה להיראות כך בממוצע: פעם עושים join על העץ שבו מפתחות גדולים (כשעושים upright) ואז בפעם הבאה עושים join על העץ שבו מפתחות קטנים (כשעושים upleft) ואז כל פעם שעושים join ההבדל בין העצים ביחס לגובה הוא בממוצע בין 2 ל 4.
כלומר הסיכוי שעושים רצף של רק "up left" או "up right" קטן מאוד.
וזה מה שקיבלנו בפועל, אכן קיבלנו ש עלות join מקסימלי עבור split אקראי הוא 3.
(נשים לב שהתוצאה חסומה ע"י O(log(n))).

<u>עלות join ממוצע עבור split של איבר מקס בתת העץ השמאלי:</u>

מצופה לקבל בין 2 ל 3 בערך כי עבור כל הפעולות join פרט לאחרון, ההבדל בין העצים ביחס לגובה הוא בין 0 ל-2.
עבור ה join האחרון מצופה שהסיבוכיות תהיה O(log(n)) כי מאחדים עץ ריק עם עץ בגובה O(log(n)).
ולכן מצופה לקבל בממוצע (log(n)+log(n))/log(n) = 2 .
כמו כן קיבלנו בפועל תוצאות קרובות ל 2.

<u>עלות join מקסימלי עבור split של איבר מקס בתת העץ השמאלי:</u>

מצופה לקבל תוצאות כגובה עץ: כי עבור כל הפעולות join פרט לאחרון, ההבדל בין העצים ביחס לגובה הוא בין 0 ל-2. בעץ האחרון אנחנו מבצעים join של עץ ריק עם תת-עץ הימני שגובה עץ שלו הוא O(log(n)).
ופעולה זו תעלה O(log(n)) (הפרש של הגבהים).
כמו כן קיבלנו בפעול תוצאות תואמות השוות בערך ל log(n).

<u>בונוס:</u>
נסמן בM את התוחלת של הממוצע עבור Split אקראי:

```
M = (  2.076923076923077 +
       1.1875 +
       1.2142857142857142 +
       1.1538461538461537 +
       1.3571428571428572 +
       1.5333333333333334 +
       1.3076923076923077 +
```

```
1.2 +
1.3333333333333333 +
1.3846153846153846) / 10 =

1.374


Dispersion D = sqrt(((1.374 – 2.076)^2 +
                    (1.374 – 1.1875)^2 +
                    (1.374 – 1.2142)^2 +
                    (1.374 – 1.153)^2 +
                    (1.374 – 1.357)^2 +
                    (1.374 – 1.533)^2 +
                    (1.374 – 1.307)^2 +
                    (1.374 – 1.2)^2 +
                    (1.374 – 1.333)^2 +
                    (1.374 – 1.384)^2 ) / 10) =

0.256
```

פיזור הסיבוכיות זניח, עבור בחירה של צומת רנדומלית.
כלומר הסיבוכיות של Split עבור איבר אקראי אינה תלויה בכמות הצמתים, והיא שווה לO(1).