<u>**תוכן עניינים**</u>

## חלק מעשי

## תיאור המחלקה שמומשה - AVLTree

המחלקה יוצרת אוביקט של עץ AVL

**קבועים**

1. External leaf - מאותחל על ידי קריאה לבנאי ריק של המחלקה AVLNode. כל השדות שלו מוגדרים להיות null, מלבד הגובה = -1, גודל = 0, מפתח =-1. (אין משמעות לערך המפתח).
2. מקרים חוקיים בעץ AVL:
   a. 1,1
   b. 2,1
   c. 1,2
3. מקרים לא חוקיים, שבהם צריך לטפל:
   a. מקרים לאחר הכנסה:
      i. 0,1 והמקרה הסימטרי 1,0
      ii. 0,2 עם בן 1,2 והמקרה הסימטרי 2,0 עם בן 2,1
      iii. 0,2 עם בן 2,1 והמקרה הסימטרי 2,0 עם בן 1,2
   b. מקרה לאחר join:
      i. 2,0 עם בן 1,1 והמקרה הסימטרי 0,2 עם בן 1,1
   c. מקרים לאחר מחיקה:
      i. 2,2
      ii. 1,3 עם בן 1,1 והמקרה הסימטרי 3,1 עם בן 1,1
      iii. 1,3 עם בן 1,2 והמקרה הסימטרי 3,1 עם בן 2,1
      iv. 1,3 עם בן 2,1 והמקרה הסימטרי 3,1 עם בן 1,2

**שדות**

1. שורש
2. צומת עם מפתח מינימום
3. צומת עם מפתח מקסימום

**בנאי**

1. בנאי ריק: מאתחל את כל השדות להיות EXTERNAL LEAF
2. בנאי שמקבל צומת: מאתחל את כל השדות להיות הצומת הזו, ואת ההורה של הצומת =null.

**מתודות שנדרשנו לממש**

**empty()**

מה היא עושה: מחזירה TRUE אם ורק אם העץ ריק.

כיצד היא פועלת: אם השורש = null מחזירה true, אם השורש הוא לא עלה אמיתי מחזירה true - עם קריאה לפונקציית עזר המוגדרת במחלקה AVLNode, isRealNode(), פירוט בהמשך.

סיבוכיות זמן ריצה: O(1).

**search(int k, String s)**

מה היא עושה: הפונקציה מחפשת איבר בעל מפתח k. אם קיים איבר כזה, היא מחזירה את הערך השמור עבורו, אחרת היא מחזירה null.

כיצד היא פועלת:

1. אם העץ ריק – בדיקה באמצעות הפונקציה empty() – מחזירה null.
2. אחרת – מגדירה משתנה x שהוא שורש העץ. כל עוד x הוא צומת אמיתי (ולא עלה חיצוני), ואינו null, הפונקציה בודקת:
   a. אם המפתח של הצומת = k, מחזירה את הערך השמור בצומת.
   b. אם המפתח של הצומת גדול מ-k, אז נעבור לחפש בתת העץ השמאלי של הצומת (שמכיל ערכים קטנים יותר).
   c. אם המפתח של הצומת קטן מ-k, אז נעבור לחפש בתת העץ הימני של הצומת (שמכיל ערכים גדולים יותר).
   d. אם סיימנו את לולאת ה-while מבלי להחזיר שום ערך, המשמעות היא ש-k לא קיים בעץ, ונחזיר null.

סיבוכיות זמן ריצה: במקרה הגרוע, נצטרך לטייל מהשורש עד לעלה, כלומר טיול בגובה העץ, שחסום על ידי O(logn) ולכן הסיבוכיות היא O(logn).

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

מה היא עושה: הכנסת איבר בעל ערך i ומפתח k לעץ, אם המפתח לא קיים. הפונקציה מחזירה את מספר פעולות האיזון שנדרשו בסה"כ בשלב תיקון העץ על מנת להשלים את הפעולה (גלגולי LR ו RL -נחשבים ל-2 פעולות איזון). אם קיים איבר בעל מפתח k בעץ הפונקציה מחזירה -1 ולא מתבצעת הכנסה.

כיצד היא פועלת:

1. יוצרת צומת חדשה לפי ערכי המפתח והערך שהוכנסו.

2. מאתחלת את מספר פעולות האיזון ל-0.
3. אם העץ ריק – מכניסה את הצומת לעץ כצומת היחיד, ומחזירה את מס' פעולות האיזון שהוא 0.
4. מאתחלים מערך ריק של IAVLNode, באורך גובה העץ (גובה השורש) + 1.
5. קוראים לפונקציית העזר positionAndTrack (פירוט בהמשך), עם המפתח k והמערך הריק. הפונקציה מחזירה את הצומת שבו עצרנו בחיפוש אחר הצומת עם המפתח k. נשמור אותה כ-x. בנוסף הפונקציה מעדכנת את המערך, ושומרת בו את כל הצמתים בהם עברנו בדרך מהשורש ל-x, כולל.
   a. אם הצומת שבו עצרנו, היא צומת עם המפתח k, לא עושים כלום ומחזירים 1-.
   b. אם המפתח של הצומת שבו עצרנו גדול מ-k – נכניס את הצומת החדשה (עם המפתח k), כבן שמאלי שלה. נעשה זאת עם קריאה לפונקציית העזר setChildAndParent עם הפרמטרים המתאימים.
      לאחר מכן נעדכן את size של כל הצמתים בדרך מהשורש ל-x: נוסיף להם 1.
   c. אם המפתח של הצומת שבו עצרנו קטן מ-k, נכניס את הצומת החדשה כבן ימני שלה, ונעדכן את size של הצמתים, כאמור לעיל.
6. פעולות איזון: נקרא לפונקציית העזר InsertionBalance עם הפרמטר x. הפונקציה מבצעת את האיזונים בעץ ומחזירה את מספר פעולות האיזון.
7. נעדכן את המיני' והמקסי' של העץ.
8. נחזיר את מס' פעולות האיזון שבוצעו.

סיבוכיות: הפונקציה מבצעת פעולות בזמן קבוע, קוראת לפונקציה positionAndTrack , שסיבוכיות זמן הריצה שלה היא O(log(n)), וקוראת לפונקציה InsertionBalance שסיבוכיות זמן הריצה שלה היא גם O(log(n)). סה"כ O(log(n)).

**delete(int k)**

מה היא עושה: מוחקת איבר עם מפתח k, אם קיים. מחזירה את מספר פעולות האיזון שנדרשו בשלב תיקון העץ. אם לא קיים איבר בעל מפתח k, מחזירה 1-.

כיצד היא פועלת:

1. אם העץ ריק, מחזירה 1-.
2. מאתחלת את מספר האיזונים ל-0.
3. יוצרת מערך בגודל גובה העץ + 1.
4. מחפשת את האיבר עם המפתח k, תוך שמירת כל הצמתים בדרך מהשורש אליו, באמצעות קריאה לפונקציה positionAndTrack.
5. שומרת את התוצאה של הקריאה לפונקציה ב-x. התוצאה היא או צומת עם המפתח k, במידה שקיים כזה בעץ, או הצומת בו עצרנו בחיפוש.
   a. אם x, הוא לא צומת עם המפתח k: אין בעץ צומת עם המפתח k, אז אין מה למחוק. הפונקציה תחזיר 1- ותסיים לרוץ.
   b. אם xהוא כן עם מפתח k:
      i. שומרים את ההורה של x, כדי לעשות את האיזונים בהמשך.
      ii. **אם x הוא עלה** (=גם הבן הימני שלו וגם הבן השמאלי שלו הם עלים חיצוניים):

1. אם x הוא השורש, אז העץ מורכב מצומת אחד בלבד, שהוא x. אז נמחק אותו, ונהפוך את העץ לעץ ריק. מספר האיזונים נשאר 0.
2. אם x הוא לא השורש, כלומר לא הצומת היחיד בעץ: אם x הוא הבן הימני של ההורה שלו, נשנה את הבן הימני של ההורה להיות עלה חיצוני (באמצעות קריאה לפונקציה SetChildAndParent עם הפרמטרים המתאימים). אם x הוא הבן השמאלי של ההורה שלו, נעשה את אותו הדבר באופן סימטרי. לאחר מכן, נעבור על כל הצמתים שעברנו בהם בדרך ל-x, ונעדכן את הגודל שלהם: נוריד מ-size שלהם 1. (זה כולל את ההורה של x).

iii. **אם x הוא צומת אונארי, עם בן שמאלי:**

1. אם x הוא השורש: נעדכן את השורש החדש להיות הבן השמאלי, וכעת הוא הצומת היחיד בעץ.
2. אם x הוא לא השורש: אם x הוא הבן הימני של ההורה שלו, נחבר בין ההורה שלו, לבין הבן השמאלי של x. (באמצעות קריאה לפונקציה SetChildAndParent עם הפרמטרים המתאימים). אם x הוא הבן השמאלי של ההורה שלו, נעשה את אותו הדבר באופן סימטרי. לאחר מכן נעדכן את כל הגדלים של הצמתים שעברנו בדרך.

**iv. אם x הוא צומת אונארי, עם בן ימני:** אותו הדבר, באופן סימטרי.

**v. אם x הוא צומת עם שני בנים:**

1. קוראים לפונקציה successor על x, ושומרים את העוקב שלו.
2. קוראים לפונקציה delete, עם המפתח של העוקב שמצאנו. כלומר, מוחקים את העוקב.
3. זו לא רקורסיה אינסופית, כי בסופו של דבר נגיע לצומת שהוא עלה או צומת אונארי.
4. מוסיפים למספר פעולות האיזון, את מספר פעולות האיזון שנדרשו במחיקת העוקב.
5. מגדירים את הבנים החדשים של העוקב, להיות הבן השמאלי של x והבן הימני של x, בהתאמה.
6. הבנים הקודמים של העוקב, אם היו, טופלו בעת המחיקה שלו.

7. מעדכנים את הגובה של העוקב ואת הגודל של העוקב, להיות אלה של x.
8. אם ההורה של x הוא null או עלה חיצוני, זה אומר ש-x היה שורש העץ. אז מגדירים את שורש העץ להיות העוקב.
9. אחרת, אם ל-x יש הורה, מגדירים את ההורה שלו להיות ההורה של העוקב (בן ימני או שמאלי, לפי מה ש-x היה).

vi. מבצעים פעולות איזון עם קריאה לפונקציה deletionBalancing, עם ההורה של x שמחקנו.
vii. מעדכנים מיני' ומקס' של העץ במקרה הצורך.

סיבוכיות זמן ריצה: הפונקציה עושה פעולות בזמן קבוע, קריאה לפונקציה positionAndTrack – O(log(n)), מעבר על הצמתים במערך בגודל גובה העץ - O(log(n)), קריאה לפונקציה deletionBalancing - O(log(n)). סה"כ O(log(n)).

**min()**

מה היא עושה: מחזירה את הערך של האיבר בעץ בעל המפתח המינימלי, או null בעץ ריק.

כיצד היא פועלת: אם העץ ריק מחזירה null, אחרת קוראת לפונקציה getValue על השדה MIN (שמצביע לאיבר המינימלי).

סיבוכיות זמן ריצה: O(1).

**max()**

מה היא עושה: מחזירה את הערך של האיבר בעץ בעל המפתח המקסימלי, או null בעץ ריק.

כיצד היא פועלת: אם העץ ריק מחזירה null, אחרת קוראת לפונקציה getValue על השדה MAX (שמצביע לאיבר המקסימלי).

סיבוכיות זמן ריצה: O(1).

**keysToArray()**

מה היא עושה: מחזירה מערך ממוין המכיל את כל המפתחות בעץ, או מערך ריק אם העץ ריק.

כיצד היא פועלת: קוראת לפונקציה inOrderWalk שמחזירה מערך ממוין של כל איברי העץ, ושומרת את המערך. מאתחלת מערך ריק בגודל של size של העץ. רצה בלולאה על כל האיברים במערך הממוין, ושומרת כל מפתח. מחזירה את המערך.

אם העץ ריק, יוחזר מערך ריק.

סיבוכיות זמן ריצה: קריאה לפונקציה inOrderWalk – O(n). לולאה על כל האיברים בעץ – O(n). ועוד פעולות בזמן קבוע. סה"כ O(n).

**infoToArray()**

מה היא עושה: מחזירה מערך מחרוזות המכיל את כל המחרוזות בעץ, ממוינות על פי סדר המפתחות. או מחזירה מערך ריק אם העץ ריק.

כיצד היא פועלת: קוראת לפונקציה inOrderWalk שמחזירה מערך ממוין של כל איברי העץ, ושומרת את המערך. מאתחלת מערך ריק, , בגודל של size של העץ. רצה בלולאה על כל האיברים במערך הממוין, ושומרת כל ערך של צומת. מחזירה את המערך.

אם העץ ריק, יוחזר מערך ריק.

סיבוכיות זמן ריצה: קריאה לפונקציה inOrderWalk – O(n). לולאה על כל האיברים בעץ – O(n). ועוד פעולות בזמן קבוע. סה״כ O(n).

**size()**

מה היא עושה: מחזירה את מספר האיברים בעץ.

כיצד היא פועלת: קוראת לפונקציה getSize על שורש העץ, שמחזירה את ערך השדה size של השורש. כפי שניתן לראות בניתוח הפונקציות האחרות, השדה size מתוחזק במהלך שינויים בעץ.

סיבוכיות זמן ריצה: O(1).

**split(int x)**

מה היא עושה: מקבלת מפתח שנמצא בעץ. מפרידה את העץ לשני עצי AVL, אחד שכל המפתחות שלו קטנים מהמפתח שהתקבל, ואחד שכל המפתחות שלו גדולים מהמפתח שהתקבל. מחזירה מערך של שני העצים, העץ הראשון הוא בעל המפתחות הקטנים.

כיצד היא פועלת:

1. שומרת את הגובה של העץ.
2. מאתחלת מערך ריק של צמתים, בגודל גובה העץ + 1.
3. מוצאת את הצומת שהמפתח שלו x, באמצעות קריאה לפונקציה positionAndTrack, שגם שומרת במערך את כל הצמתים עד לצומת שממנו מפצלים את העץ.
4. שומרת את הבן השמאלי של הצומת שממנו מפצלים.
   a. אם הבן הוא עלה אמיתי (ולא עלה חיצוני): יוצרת עץ חדש, עם הבן השמאלי. מפרידה את הצומת שממנו מפרידים מהבן: הבן השמאלי של הצומת מוגדר עכשיו להיות עלה חיצוני, וההורה של הבן מוגדר להיות null.
   b. אם לא, נשמר עץ ריק.
5. אותו התהליך לבן הימני.
6. מגדירה את t1 להיות תת העץ שנוצר מהבן השמאלי, ואת t2 להיות תת העץ שנוצר מהבן הימני.
7. רצים על הצמתים שנשמרו במערך, מהצומת הקרוב ביותר ל-x, עד לשורש. עבור כל צומת, אם המפתח שלו גדול מ-x, צריך להוסיף אותו ל-t2. נעשה זאת בסדרת פעולות:
   a. המפתח גדול מ-x, זה אומר ש-x (או אחד האבות/האבות הקדמונים של x) הוא בן שמאלי של הצומת הנוכחי. לכן ניצור תת עץ, מהבן הימני של הצומת. ונשמור אותו כ- subTree.

b. נגדיר את ההורה של הבן הימני להיות null.
c. נגדיר את ההורה של הבן השמאלי להיות null.
d. נגדיר את שני הבנים של הצומת הנוכחי, להיות עלים חיצוניים.
e. נגדיר את ההורה של הצומת הנוכחי להיות null.
f. כעת לאחר שניתקנו את הצומת הנוכחי מהעץ, נחבר את ה-subTree לעץ t2, ביחד עם הצומת הנוכחי, באמצעות קריאה לפונקציה join.

8. (אין בעיה לנתק את הצומת הנוכחי מההורה שלו, כיוון שהצומת "הבא בתור", שמור לנו במערך, ולא צריך את הצומת הנוכחי כדי להגיע אליו).
9. אם המפתח של הצומת קטן מ-x, נוסיף אותו לעץ t1, באמצעות פעולות סימטריות.
10. נעדכן את המינימום והמקסימום של t1 ושל t2.
11. נחזיר את המערך של שני העצים שקיבלנו.

סיבוכיות זמן ריצה: כפי שראינו בכיתה, מימוש split באמצעות קריאות לפונקציה join עולה O(log(n)).

## join(IAVLNode x, AVLTree t)

מה היא עושה: מקבלת x ועץ t שכל המפתחות שלהם קטנים ממפתחות העץ הנוכחי, או שכל המפתחות שלהם גדולים ממפתחות העץ הנוכחי. הפונקציה מאחדת את העץ הנוכחי, עם t ועם x. מחזירה את העלות של פעולת ה-join = את הפרש גבהי העצים + 1.

כיצד היא פועלת:

1. מחשבת את הפרשי הגבהים כדי להחזיר אותם.
2. יוצרת צומת חדש, s.
3. אם המפתח של x גדול מהמפתח המקסימלי של העץ הנוכחי,וקטן מהמפתח המינימלי של העץ t, אז העץ הנוכחי הוא עם המפתחות הקטנים. נקרא לפונקציה JoinHelper עם הפרמטרים המתאימים, ונשמור את מה שהיא מחזירה בצומת s. s הוא ההורה של x בעץ החדש שנוצר.
4. אחרת, העץ הנוכחי הוא העץ עם המפתחות הגדולים. נקרא לפונקציה JoinHelper עם הפרמטרים המתאימים, ונשמור את מה שהיא מחזירה בצומת s.
5. נקרא לפונקציה InsertionBalance, שעושה את האיזונים, על הצומת s.
6. נחזיר את הפרשי הגבהים שחישבנו קודם.

סיבוכיות זמן ריצה: פעולות בזמן קבוע וקריאה לפונקציה שעושה את החיבור עצמו, JoinHelper, שעולה O(log(n)) (n = מספר הצמתים בעץ הגבוה יותר) ובפרט O( |T1.height - T2.height| +1 ). סה"כ O(log(n)).

## getRoot()

מה היא עושה: מחזירה את השורש של העץ.

כיצד היא פועלת: מחזירה את השדה ROOT.

סיבוכיות זמן ריצה: O(1).

**מתודות עזר**

**positionAndTrack(int k, IAVLNode[] TrackArray)**

מה היא עושה: מחזירה את הצומת עם המפתח k אם הוא קיים, ומחזירה את הצומת אליו הגענו בחיפוש, אם לא קיים צומת עם מפתח k. בנוסף, שומרת במערך הצמתים, את הצמתים שעברנו בדרך, מהשורש עד לצומת אליה הגענו כולל.

כיצד היא פועלת:

1. מאתחלת משתנה i ל-0
2. אם העץ ריק מחזירה null
3. אחרת, מאתחלת צומת x להיות שורש העץ. כל עוד x הוא עלה אמיתי (לא עלה חיצוני):
   a. נוסיף את הצומת למערך, במקום i, ונקדם את i ב-1.
   b. אם המפתח של הצומת = k, נחזיר את הצומת
   c. אם המפתח של הצומת גדול מ-k, נעבור לחפש בתת העץ השמאלי על ידי x=x.getLeft()
   d. אם המפתח של הצומת קטן מ-k, נעבור לחפש בתת העץ הימני על ידי x=x.geRight()
4. בסיום הלולאה, אם לא החזרנו צומת שהמפתח שלו k, אנחנו יוצאים מהלולאה כש-x הוא עלה חיצוני. לכן, נחזיר את ההורה שלו.

סיבוכיות זמן ריצה: בדומה ל-search, O(log(n)). תוספת הצמתים למערך עולה זמן קבוע ולא משנה את הסיבוכיות.

**setChildAndParent (IAVLNode child, IAVLNode parent, boolean inRight)**

מה היא עושה: הופכת את הצומת שהוכנס ראשון לבן של הצומת שהוכנס שני בהתאם לערך הבוליאני

כיצד היא פועלת:

1. אם ההורה אינו null:
   a. אם הערך הבוליאני הוא TRUE - הבן הוא בן ימני – מעדכנת את הבן הימני של ההורה להיות הצומת.
   b. אם הערך הבוליאני הוא FALSE – הבן הוא בן שמאלי – מעדכנת את הבן השמאלי של ההורה להיות הצומת.
2. אם הבן אינו null ולא עלה חיצוני:
   מעדכנת את ההורה שלו.

סיבוכיות זמן ריצה: O(1).

**InsertionBalance(IAVLNode s)**

מה היא עושה: איזונים במקרים הנדרשים לאחר הכנסה של צומת חדשה או לאחר join. מקבלת צומת s ומאזנת ממנו. מחזירה את מספר הפעולות שנעשו לאיזון.

<u>לגבי הכנסה:</u>

צומת s הוא הצומת שבו עצרנו בחיפוש אחר צומת עם המפתח k, שאותו רצינו להכניס. לפני ההכנסה צומת s היה או עלה, או צומת אונארי.

אם הוא היה עלה, אחרי ההכנסה המקרה שצריך לאזן הוא **<u>המקרה הראשון:</u>** מקרה של 1,0. או המקרה הסימטרי 0,1.

אם הוא היה צומת אונארי, יש שתי אפשרויות:

1. הצומת החדש נכנס באותו הכיוון של הבן שלו (כלומר אם הבן הוא בן שמאלי, הצומת נכנס כבן שמאלי. ואם ימני נכנס כבן ימני). ואז המקרה שצריך לאזן הוא **<u>המקרה השני:</u>** s הוא במצב 2,0 והבן הוא במצב 2,1. או המקרה הסימטרי: s הוא במצב 0,2 והבן הוא במצב 1,2.
2. הצומת הנכנס, נכנס בכיוון אחר מהבן (כלומר אם הבן הוא בן שמאלי, הצומת נכנס כבן ימני, ולהפך). ואז המקרה שצריך לאזן הוא **<u>המקרה השלישי</u>**: s הוא במצב 2,0 והבן הוא במצב 1,2. או המקרה הסימטרי: s הוא במצב 0,2 והבן הוא במצב 2,1.

במקרה הראשון אנחנו "מעלים" את הבעיה צומת אחד למעלה. במקרים השני והשלישי, הבעיה נפתרת, כי הגובה של תת העץ ששינינו לא השתנה.

<u>לגבי join:</u>

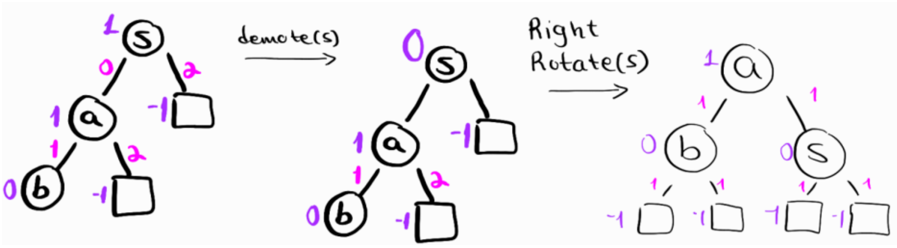
לאחר join, יכולים להיווצר כל שלושת המקרים הנל, ויש גם **<u>מקרה רביעי:</u>** s הוא 0,2 עם בן שמאלי 1,1. או המקרה הסימטרי: s הוא 2,0 עם בן שמאלי 1,1.

במקרה כזה, הגובה של תת העץ ששינינו גדל ב-1, ולכן הבעיה "עולה" צומת אחד למעלה, וממשיכים לבדוק.

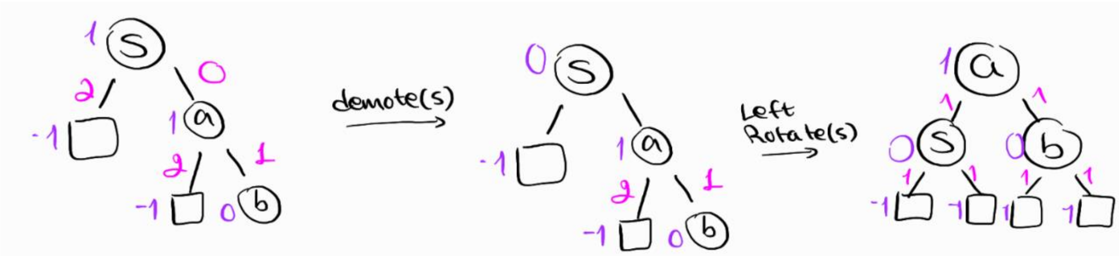
כיצד הפונקציה פועלת:

* השוואות למקרים השונים נעשות עם פונקציית העזר EQUAL

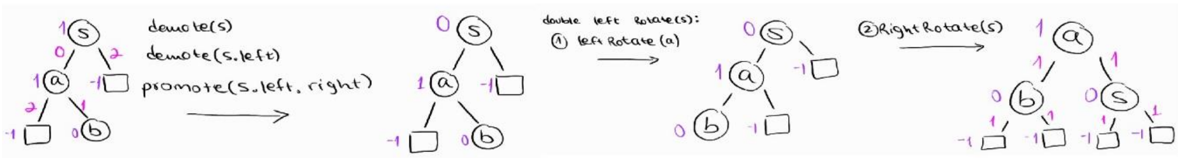
1. מאתחלת את מספר הפעולות ל-0.
2. מאתחלת מערך של שני int, שהוא המצב של הצומת, ל- [0,0].
3. אם הצומת s לא ריק, קוראת לפונקציה BalanceDiff על הצומת s, שמחשבת את המצב עבור הצומת ושומרת אותו במערך.
4. לולאה רצה כל עוד המצב של s לא חוקי = כלומר שונה משלושת המצבים החוקיים, וגם s לא null ולא עלה חיצוני. בתוך הלולאה:
   a. שומרים את ההורה של s, למקרה שנצטרך "לעלות" לצומת זה. זה קורה אם המקרה הוא **המקרה הראשון**.
   b. אם המצב של s הוא **המקרה הראשון** (אחד משני המקרים הסימטריים) קוראים לפונקציה promote על s.
   c. אם המצב של s הוא **המקרה השני**, הראשון, כלומר s הוא 0,2 והבן השמאלי שלו הוא 1,2: נוריד את הגובה/דרגה של s (עם הפונקציה demote) ונעשה סיבוב אחד ימינה (עם קריאה לפונקציה RightRotate על s).



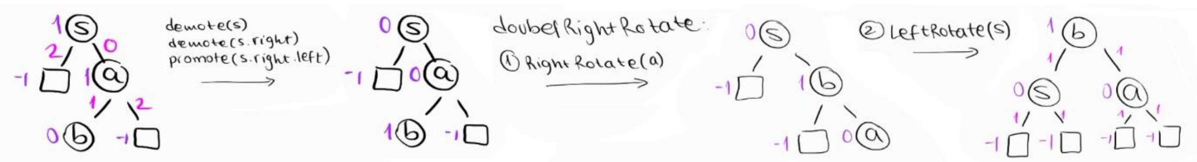

d. אם המצב הוא **המקרה השני**, במקרה הסימטרי: s הוא 2,0 והבן הימני שלו הוא 2,1:
נוריד את הגובה/דרגה של s ונעשה סיבוב אחד שמאלה, באופן דומה.



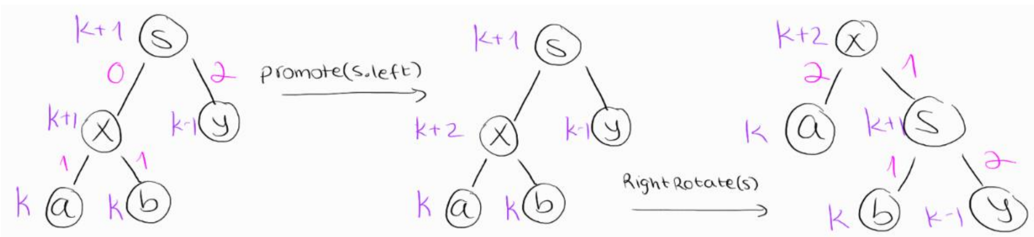

e. אם המצב הוא **המקרה השלישי**, הראשון: s הוא 0,2 והבן השמאלי שלו הוא 2,1:
מתאימים את הדרגות ומבצעים סיבוב כפול – סיבוב אחד שמאלה ואז סיבוב אחד ימינה.



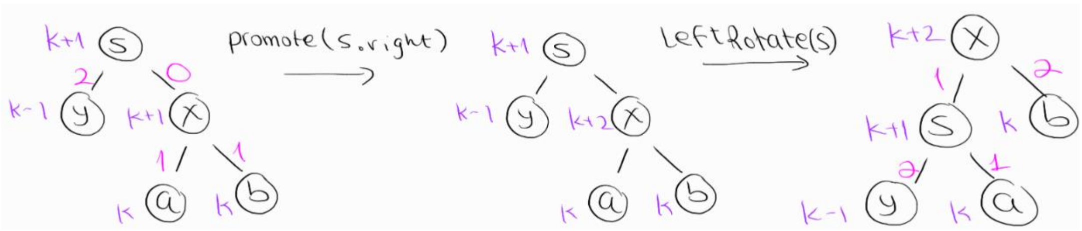

f. אם המצב הוא **המקרה השלישי**, הסימטרי: s הוא 2,0 והבן השמאלי שלו הוא 2,1:
מתאימים את הדרגות ומבצעים סיבוב כפול – סיבוב אחד ימינה ואז סיבוב אחד שמאלה.




g. **מקרה רביעי**: s הוא 0,2 עם בן שמאלי 1,1. במקרה כזה נקדם את הדרגה של הבן השמאלי של s ב-1, ונבצע סיבוב אחד ימינה.




h. **מקרה רביעי** סימטרי: s הוא 2,0 עם בן ימני 1,1. במקרה כזה נקדם את הדרגה של הבן הימני של s ב-1, ונבצע סיבוב אחד שמאלה.



* בכל אחד מהמקרים, מעדכנים את מספר האיזונים.

i. מקדמים את s להיות ההורה שלו. אם s לא null, מחשבים שוב את המצב שלו, וחוזרים לתחילת הלולאה.

5. מחזירים את מספר האיזונים שנעשו.

סיבוכיות זמן ריצה: במקרה הגרוע הלולאה מתבצעת שוב ושוב עד שמגיעים לשורש העץ. התחלנו מהצומת s, והמרחק ממנו לשורש חסום ע״י log(n). ובפרט המרחק הוא גובה השורש פחות הגובה של s, כלומר: log(n) – s.heigth.

לכן הלולאה תרוץ לכל היותר log(n) פעמים. בכל ריצה של הלולאה, מתבצעות פעולות בזמן קבוע. לכן סה״כ הסיבוכיות היא O(log(n)).

**DeletionBalncing(IAVLNode s)**

מה היא עושה: איזונים במקרים הנדרשים לאחר מחיקת צומת. מקבלת צומת current ומאזנת ממנו. מחזירה את מספר הפעולות שנעשו לאיזון.

כיצד היא פועלת: הצומת current היא ההורה של הצומת, אותו מחקנו.

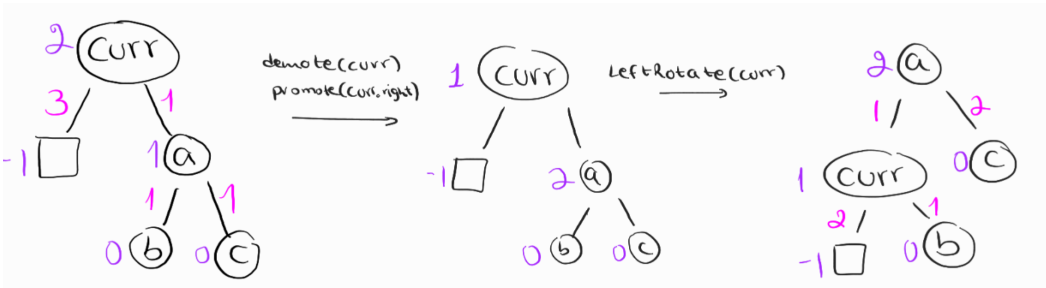
בדומה לפונקציה InsertionBalance, הפונקציה רצה בלולאה מהצומת שקיבלה, עד השורש, ובודקת על כל הצמתים אם הם חוקיים. אם לא, עושה את הגלגולים הנדרשים, וסופרת את פעולות האיזון.

המקרים שיכולים להיות, שדורשים איזון:

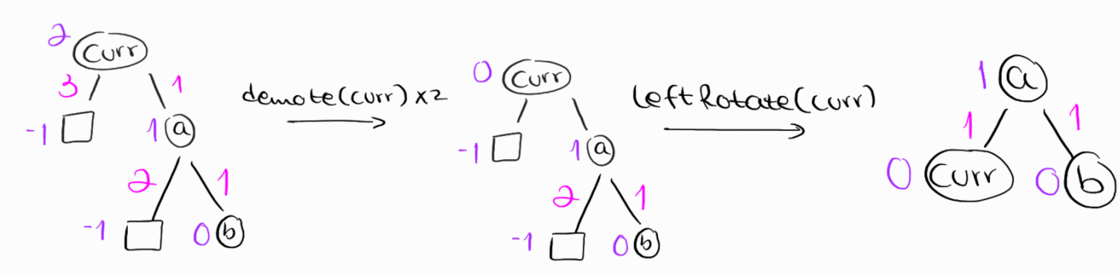
הצומת שמחקנו, היה או עלה או צומת אונארי. (אם הצומת שמחקנו היה צומת בינארי, אז קראנו לפונקציה delete שוב, עד שבסופו של דבר אנחנו מגיעים למחוק עלה או צומת אונארי).

יש סה״כ 7 מקרים שיכולים להיווצר – המקרה 2,2 ועוד 3 מקרים והסימטרי שלהם.

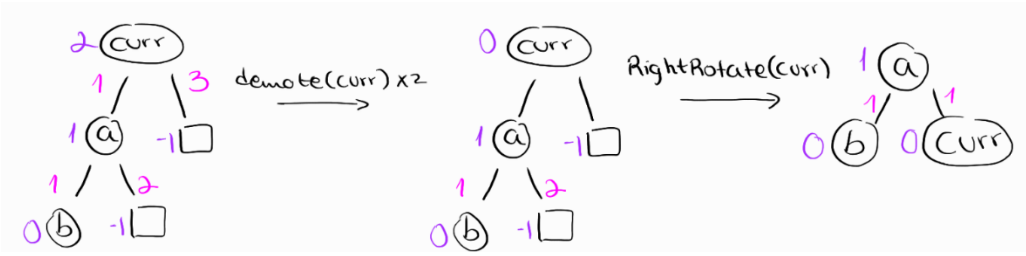
1. **במקרה של 2,2:** עושים demote לצומת. ייתכן שהבעיה עלתה צומת אחד למעלה ולא נפתרה, לכן ממשיכים לבדוק.
2. **במקרה של 3,1, עם בן ימני 1,1:** מורידים את הדרגה של current ב-1, ומקדמים את הדרגה של הבן הימני שלו ב-1. עושים סיבוב אחד שמאלה. הבעיה נפתרה, כי סך הכל הגובה של תת העץ לא השתנה.



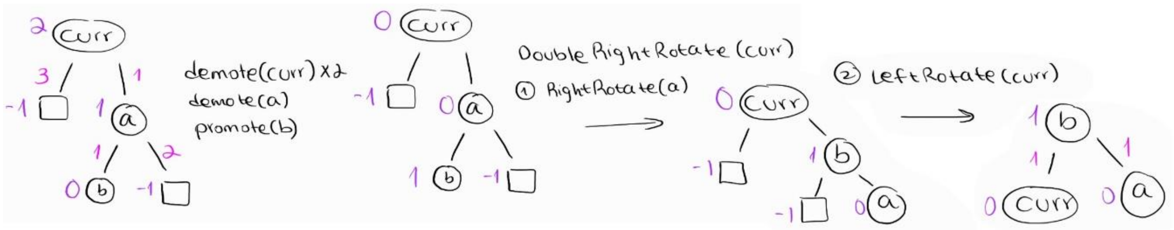

3. **במקרה הסימטרי - במקרה של 1,3 עם בן שמאלי 1,1:** אותו פתרון, באופן סימטרי.
4. **במקרה של 3,1 עם בן ימני 2,1:** נוריד את הדרגה של current ב-1 פעמיים, ונעשה סיבוב אחד שמאלה. הבעיה אולי עלתה למעלה, כי הגובה של תת העץ קטן ב-1.




5. **במקרה הסימטרי – 1,3 עם בן שמאלי 1,2 :** אותו דבר, באופן סימטרי.




6. **במקרה של 3,1 עם בן ימני 1,2 :** מתאימים את הדרגות, ועושים סיבוב כפול – סיבוב אחד ימינה, וסיבוב אחד שמאלה.




7. **במקרה הסימטרי – 1,3 עם בן שמאלי 2,1 :** אותו דבר באופן סימטרי.

סיבוכיות זמן ריצה : בכל צומת אנחנו מבצעים סיבוב ו/או שינויים בדרגה - מספר קבוע של פעולות. במקרה הגרוע, אנחנו עוברים על כל הצמתים, מהצומת שהוא ההורה של הצומת שמחקנו ועד השורש. המרחק הזה חסום על ידי גובה העץ, שחסום ע״י log(n). לכן הסיבוכיות היא O(log(n)).

**RightRotate(IALNode y)**

מה הפונקציה עושה : סיבוב ימינה, כמו שראינו בכיתה.

כיצד הפונקציה פועלת :

1. מקבלים צומת y.
2. שומרים את ההורה שלו ב-grandparent .
3. שומרים את הבן השמאלי שלו ב-x, ואת הבן הימני של x ב-b.
4. שומרים את ה-size שיהיה ל-y אחרי השינוי : אחרי השינוי, הבן הימני של y נשאר אותו דבר. אבל הבן השמאלי שלו משתנה, מ-x, ל-b. כלומר מ-x ל-רק הבן הימני של x. אז צריך להוריד מהגודל את הבן השמאלי של x. כמו כן מורידים עוד 1 עבור x עצמו.

<u>טיפול בהורה של y</u> :

5. אם ה-grandparent לא null והוא לא עלה חיצוני :

a. אם y הוא בן שמאלי של הסבא, הופכים את x לבן שמאלי של הסבא, במקום y: קוראים לפונקציה SetChildAndParent עם הפרמטרים המתאימים.
b. אם y הוא בן ימני של הסבא, הופכים את x לבן ימני של הסבא, באותו אופן.

6. אחרת – אם הסבא הוא כן null – המשמעות היא ש-y הוא השורש. אז נגדיר את שורש העץ להיות x ואת ההורה של x להיות null.

* הגודל של כל תת העץ (שקודם שורשו היה y ועכשיו הוא x) לא משתנה, לכן לא צריך לשנות את הגודל של צמתים עליונים.

* כפי שראינו בכיתה גם הגובה של תת העץ לא משתנה.

<u>עדכון הגדלים של x ושל y</u>:

7. הגודל של x הופך להיות הגודל הקודם של y (כי עכשיו x הוא שורש תת העץ, שקודם y היה השורש שלו).
8. הגודל של y מקבל את הגודל שחישבנו בסעיף 4.

<u>טיפול ב-x,y,b</u>:

9. הופכים את y לבן הימני של x, באמצעות קריאה לפונקציה SetChildAndParent עם הפרמטרים המתאימים.
10. הופכים את b לבן השמאלי של y, באופן דומה.

סיבוכיות זמן ריצה: יש מספר קבוע של פעולות שצריך לעשות, אז הזמן הוא O(1).

**<u>LeftRotate(IALNode y)</u>**

מה היא עושה: סיבוב שמאלה, כמו שראינו בכיתה.

כיצד היא פועלת: באופן סימטרי לסיבוב ימינה.

סיבוכיות זמן ריצה: כמו בסיבוב ימינה, O(1).

**<u>DoubleRightRotate(IAVLNode parent)</u>**

מה היא עושה: סיבוב ימינה ואז סיבוב שמאלה.

כיצד היא פועלת: מקבלת צומת. קוראת לסיבוב ימינה, על הבן הימני של הצומת. לאחר מכן, סיבוב שמאלה על הצומת.

סיבוכיות זמן ריצה: לכל אחד מהסיבובים זמן קבוע של פעולות, אז הזמן הוא O(1).

**<u>DoubleLeftRotate(IAVLNode parent)</u>**

מה היא עושה: סיבוב שמאלה ואז ימינה.

כיצד היא פועלת: מקבלת צומת. קוראת לסיבוב שמאלה, על הבן השמאלי של הצומת. לאחר מכן, סיבוב ימינה על הצומת.

סיבוכיות זמן ריצה: לכל אחד מהסיבובים זמן קבוע של פעולות, אז הזמן הוא O(1).

**JoinHelper(AVLTree T1, IAVLNode x, AVLTree T2)**

מה היא עושה: מקבלת שני עצים וצומת x, כל המפתחות ב-T1 קטנים מהמפתח של x, וכל המפתחות ב-T2 גדולים מהמפתח של x. מחברת את שני העצים והצומת, לעץ אחד שהשורש שלו הוא השורש של העץ הגבוה יותר.

אם שני העצים לא ריקים, מחזירה את הצומת שעכשיו הוא ההורה של x. אם אחד העצים ריק או שניהם ריקים, מחזירה null.

כיצד היא פועלת:

1. שומרת את הגבהים והגדלים של העצים. אם אחד העצים ריק, הגובה שלו הוא -1.
2. שומרת את המינימום של העץ עם המפתחות הקטנים, והמקסימום של העץ עם המפתחות הגדולים.
3. יוצרת צומת ריק.
4. אם שני העצים ריקים: ניצור עץ חדש, שמכיל רק את הצומת x. נחזיר null.
5. אם T1 הוא עץ ריק: מוסיפה את x ל- T2 עם הפונקציה insert (שגם מאזנת את העץ במקרה הצורך), ומתאימה את השדות בהתאם. מחזירה null.
6. אם T2 הוא עץ ריק: מוסיפה את x ל-T1, באופן דומה. מחזירה null.
7. אם שני העצים לא ריקים:
    a. מאתחל מערך של צמתים, ריק.
    b. מאתחל מצביע לצומת נוכחי.
    c. אם T1 גבוה יותר:
        i. נאתחל מערך צמתים בגודל של הפרשי הגבהים + 1.
        ii. נתחיל עם מצביע לשורש של T1.
        iii. נתקדם ב-T1, על הצלע הימנית שלו, עד שנגיע לצומת שהגובה שלה קטן שווה לגובה של T2. נסמן את הצומת הזאת curr, והיא הולכת להיות הבן השמאלי של x. במהלך ההתקדמות אנחנו שומרים את כל הצמתים שעברנו בדרך, במערך.
        iv. נעדכן את הגודל של x להיות הגודל של T2 + הגודל של curr.
        v. נעדכן את הגובה של x להיות הגובה של T2 + 1.
        vi. נעדכן את parent להיות ההורה של x.
        vii. נעבור על כל הצמתים ששמרנו במערך, ולכל צומת נעדכן את הגודל שלה: נוסיף לה את הגודל של T2+1 .
        viii. נבצע את החיבורים בין הצמתים עם קריאות לפונקציה SetChildAndParent: נחבר את x כבן הימני ל-parent. נחבר את השורש של T2 כבן ימני של x. ונחבר את curr כבן שמאלי של x.
        ix. נגדיר את השורש להיות השורש של T1.
    d. אם T2 גבוה יותר:
        i. נעשה את אותן הפעולות, באופן סימטרי.
        ii. הפעם נלך על הצלע השמאלית של T2, עד שנגיע לצומת שהגובה שלה קטן שווה לגובה של T1.

e. אם הגבהים שווים:

i. נעדכן את הגובה והגודל של x.

ii. נגדיר את השורש של T1 להיות הבן השמאלי של x, ואת השורש של T2 להיות הבן הימני של x.

8. נעדכן את המינימום והמקסימום אם צריך

9. נחזיר את ההורה של x (נגיע לכאן כאמור רק אם שני העצים לא ריקים).

סיבוכיות זמן ריצה: בנוסף לפעולות הקבועות, יש לולאה שרצה מהשורש של העץ הגבוה יותר, עד לצומת בגובה של העץ הנמוך יותר. אם בעץ הגבוה יותר יש n צמתים, הגובה שלו חסום ע״י log(n) ולכן הסיבוכיות היא O(log(n)). ובפרט ( O( |T1.height - T2.height| +1.

**EQUAL(int[] a, int[] b)**

מה היא עושה: מקבלת שני מערכים של שני מספרים ומחזירה TRUE אמ״מ המערכים שווים.

סיבוכיות זמן ריצה: O(1).

**Promote(IAVLNode node)**

מה היא עושה: מקבלת צומת, ומגדילה את הגובה שלו ב-1.

סיבוכיות זמן ריצה: O(1).

**Demote(IAVLNode node)**

מה היא עושה: מקבלת צומת, ומקטינה את הגובה שלו ב-1.

סיבוכיות זמן ריצה: O(1).

**balanceDiff((IAVLNode parent)**

מה היא עושה: מקבלת צומת, ומחזירה מערך של שני מספרים שהם הפרשי הגבהים בינו לבין הבנים שלו.

כיצד היא פועלת: המספר הראשון הוא הגובה של הצומת פחות הגובה של הבן השמאלי שלו.

המספר השני במערך הוא הגובה של הצומת פחות הגובה של הבן הימני שלו.

(אם אין לו בן באחד הצדדים או בשניהם, הבן מוגדר להיות עלה חיצוני, עם גובה 1-).

סיבוכיות זמן ריצה: O(1).

**InOrderWalk()**

מה היא עושה: מחזירה מערך של איברי העץ, ממוינים לפי המפתחות (בסדר עולה)

כיצד היא פועלת: מאתחלת מצביע לעץ להיות שורש העץ, ומצביע למערך להיות 0. מאתחלת מערך ריק של צמתים בעץ, בגודל של size העץ (מספר צמתים בעץ). רצה בלולאה על המערך. בכל איטרציה:

1. מוסיפה למערך את הצומת הנוכחי.

2. מקדמת את המצביע לצומת, להיות מצביע ל-succesor של הצומת (קריאה לפונקציה successor).
3. מקדמת את המצביע למערך.

סיבוכיות זמן ריצה: הפונקציה successor נקראת n-1 פעמים. ראינו בתרגול, שזמן הריצה של k פעולות successor חסום ע"י: k + גובה העץ. במקרה שלנו זה O(n) = O(n-1 + log(n)).

**Successor(IAVLNode node)**

מה היא עושה: מקבלת צומת, ומחזירה את הצומת שהוא עם המפתח העוקב למפתח של הצומת שקיבלנו.

כיצד היא פועלת:

1. אם הצומת שקיבלנו הוא עלה חיצוני, מחזירה null.
2. אם לצומת שקיבלנו יש בן ימני שהוא אמיתי (ולא עלה חיצוני), מחזירה את המינימום של תת העץ הימני, באמצעות קריאה לפונקציה NodeMin על הבן הימני.
3. אם אין בן ימני:
   a. שומרים את ההורה של הצומת, כ-parent.
   b. כל עוד ההורה הוא לא null, וכל עוד ההורה הוא הורה משמאל (=קטן יותר), נמשיך לעלות למעלה ולחפש  את ההורה הבא.
   c. כשהגענו להורה שהוא הורה מימין (=גדול יותר), נחזיר אותו.

סיבוכיות זמן ריצה: במקרה הראשון, שלצומת יש בן ימני, אז במקרה הגרוע נרד מהשורש עד לעלה, כלומר טיול בגובה העץ, שחסום ע"י log(n). במקרה השני, שלצומת אין בן ימני, אז במקרה הגרוע נעלה מעלה עד לשורש, ושוב זה טיול בגובה העץ שחסום ע"י log(n). אז סה"כ הסיבוכיות היא O(log(n)).

**NodeMIN(IAVLNode node)**

מה היא עושה: מקבלת צומת, ומחזירה צומת המקושר אליו, שהוא בעל המפתח המינימלי.

כיצד היא פועלת: אם הצומת הוא עלה חיצוני, מחזירה null. אם לא, הולכת בלולאה כל פעם לבן השמאלי ביותר (כל עוד הבן השמאלי הוא לא עלה חיצוני). מחזירה את העלה השמאלי ביותר.

סיבוכיות זמן ריצה: במקרה הגרוע, נרוץ מהשורש עד לעלה – O(log(n)).

**NodeMAX(IAVLNode node)**

מה היא עושה: מה היא עושה: מקבלת צומת, ומחזירה צומת המקושר אליו, שהוא בעל המפתח המקסימלי.

כיצד היא פועלת: כמו הפונקציה NodeMin, רק שהיא הולכת לעלה הימני ביותר, באופן סימטרי.

סיבוכיות זמן ריצה: O(log(n)).

**המנשק IAVLNode**

מנשק לצומת בעץ.

**המחלקה AVLNode**

מימוש המנשק.

שני בנאים: בנאי ריק שיוצר "עלה חיצוני", ובנאי שמקבל מפתח וערך ויוצר צומת "רגילי".

כל הפונקציות הן פונקציות שנותנות גישה לראות או לשנות את שדות האוביקט.

**isRealNode()**

מחזירה TRUE אמ"מ הצומת הוא לא null ולא עלה חיצוני, כלומר אם הגובה שלו שונה מ- 1-.

## חלק ניסויי

### שאלה 1

#### סעיף א'

| מספר סידורי i | מספר חילופים במערך ממוין - הפוך | עלות החיפושים במיון AVL עבור מערך ממוין-הפוך | מספר חילופים במערך מסודר אקראית | עלות החיפושים במיון AVL עבור מערך מסודר אקראי |
|---|---|---|---|---|
| 1 | 1999000 | 36884 | 1021160 | 30871 |
| 2 | 7998000 | 81764 | 4156277 | 71112 |
| 3 | 31996000 | 179524 | 16765092 | 158724 |
| 4 | 127992000 | 391044 | 66887000 | 350831 |
| 5 | 511984000 | 846084 | 266906578 | 784116 |

#### סעיף ב'

**מספר החילופים** : במערך ממוין הפוך, כל זוג אינדקסים מהווה חילוף. כמות הזוגות היא $\binom{n}{2}$ ולכן זה מספר החילופים.

**עלות החיפושים** :

חסם מלמעלה :

בכל הכנסה, האיבר שנכניס במקום ה-i, הוא האיבר המינימלי בעץ (כי המערך ממוין הפוך). המרחק שלו מהאיבר המקסימלי, שבו מתחילים את החיפוש, הוא לכל היותר פעמיים גובה העץ. זה עץ AVL אז גובה העץ הוא לכל היותר logi. אז המרחק הוא לכל היותר 2logi.

עבור n הכנסות נקבל :

$$\sum_{i=1}^{n} 2\log(i) \leq 2\sum_{i=1}^{n} \log(n) = 2nlog(n) = O(n\log(n))$$

חסם מלמטה :

בכל הכנסה, המרחק בין האיבר המינימלי, i, לבין האיבר המקסימלי הוא לפחות גובה העץ, כי במסלול ביניהם נצטרך לעבור דרך השורש

$$\sum_{i=1}^{n} \log(i) \geq \sum_{i=n/2}^{n} \log(i) \geq \sum_{i=n/2}^{n} \log\left(\frac{n}{2}\right) = \frac{n}{2} log\left(\frac{n}{2}\right) = \Omega(n\log(n))$$

לכן סה"כ קיבלנו $\theta(n\log(n))$.

**<u>סעיף ג'</u>**

מספר החילופים :

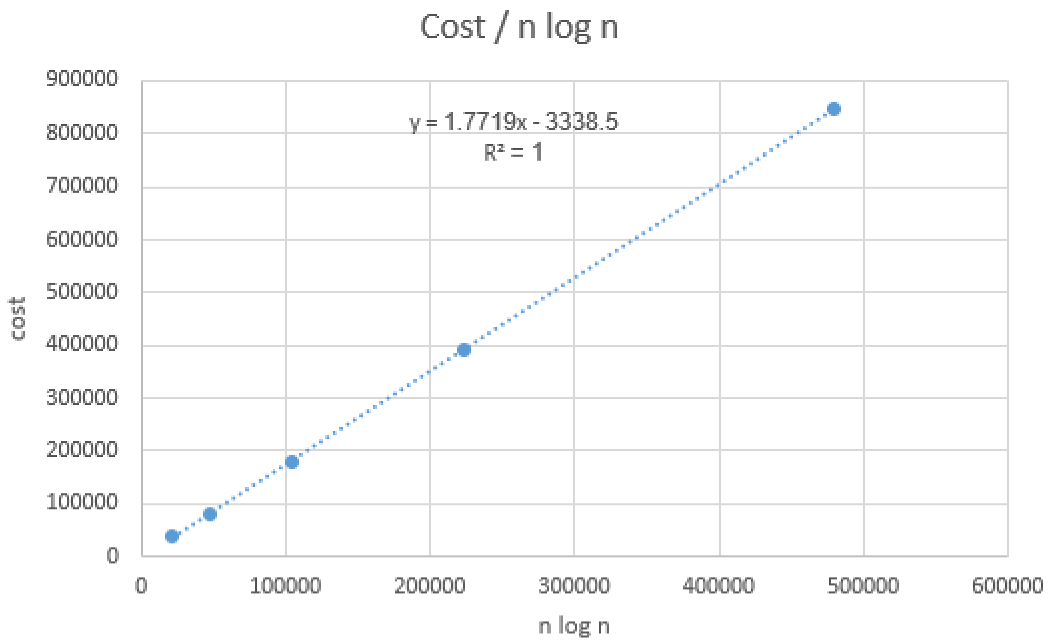
המספר יצא בדיוק כמו המספר שקיבלנו בניסוי בסעיף א'.

| מספר סידורי i | גודל n | מספר חילופים במערך ממוין - הפוך - בניסוי | מספר חילופים צפוי $\binom{n}{2}$ |
|---|---|---|---|
| 1 | 2000 | 1999000 | 1999000 |
| 2 | 4000 | 7998000 | 7998000 |
| 3 | 8000 | 31996000 | 31996000 |
| 4 | 16,000 | 127992000 | 127992000 |
| 5 | 32,000 | 511984000 | 511984000 |

עלות החיפושים :

ציפינו שעלות החיפוש היא $\theta(n\log(n))$.

אז נצפה שכאשר נשים את הערכים שקיבלנו בניסוי בסעיף א', בציר y, ואת ערכי nlog(n) בציר x, נראה תלות לינארית וזה אכן מה שקיבלנו. קיבלנו תלות לינארית עם איכות קירוב גבוהה, $R^2 = 1$.




<u>סעיף ד'</u>

נסמן ב-$h_i$ את מספר החילופים של איבר i עם האיברים שהאינדקס שלהם קטן מ-i, כלומר עם אינדקסים j שמקיימים $a_j > a_i$ וכן $j < i$. (i, j הם אינדקסים במערך הלא ממוין).

נקבל שמספר החילופים הכולל הוא :

$$h_1 + h_2 + \cdots + h_n = h$$

ניקח את האיבר ה-i במערך לפי סדר ההכנסה. הוכנסו לע i-1 האיברים הקודמים ומספר האיברים שהוכנסו לעץ וגדולים מ- $a_i$ הינו $h_i$ .

מתכונות עץ חיפוש finger-tree סיבוכיות זמן ההגעה לאיבר ה-$k$ הכי גדול יהיה $O(\log(k))$ (*Select(k)*) כלומר המרחק בין האיבר המקסימלי לבין אותו איבר חסום מלעיל ע"י *c log(k)* כאשר *c* הינו קבוע כלשהו.

האיבר ה-*i* שנכניס הינו ה-$h_i + 1$ הכי גדול, מכאן:

סיבוכיות זמן חיפוש (מרחק מהאיבר המקסימלי) עבור האיבר *i* שנכניס חסומה מלעיל ע"י:

$$\log(h_i + 1) + 1$$

(תוספת זמן קבוע עבור זמן חיפוש קבוע כאשר $h_i = 0$)

מכאן סיבוכיות הזמן הכוללת חסומה מלעיל ע"י:

$$\sum_{i=1}^{n} c\log(h_i + 1) + 1 = n + c \cdot \sum_{i=1}^{n} \log(h_i + 1)$$

$$\sum_{i=1}^{n} \log(h_i + 1) = \log\left(\prod_{i=1}^{n} h_i + 1\right) = n \cdot \frac{1}{n} \cdot \log\left(\prod_{i=1}^{n} h_i + 1\right) =$$

$$= n \cdot \log\left(\left(\prod_{i=1}^{n} h_i + 1\right)^{\frac{1}{n}}\right) \leq_* n \cdot \log\left(\frac{\sum_{i=1}^{n} \mathrm{h_i} + 1}{n}\right) = n \cdot \log\left(\frac{h+n}{n}\right) \leq$$

(*) אי שוויון ממוצעים.

$$\leq\ n \cdot \log(h + 1)$$

מכאן:

$$\sum_{i=1}^{n} c\log(h_i + 1) + 1 \leq n + c \cdot n \cdot \log(h + 1) = O(n + n\log(h + 1))$$

מכאן מצאנו חסם עליון $O(n + n\log(h + 1))$.

## **שאלה 2**

סעיף א'

| מספר סידורי i | עלות join **ממוצע** עבור split**אקראי** | עלות join **מקסימלי** עבור split **אקראי** | עלות join **ממוצע** עבור splitשל **האיבר** מקסימלי בתת העץ השמאלי | עלות join **מקסימלי** עבור split של **איבר** מקסימלי בתת העץ השמאלי |
|---|---|---|---|---|
| 1 | 2.9 | 5 | 3 | 13 |
| 2 | 2.25 | 4 | 2.818 | 14 |
| 3 | 2.75 | 5 | 2.538 | 16 |
| 4 | 2.846 | 5 | 2.571 | 17 |
| 5 | 2.714 | 6 | 2.571 | 18 |
| 6 | 2.375 | 6 | 2.8 | 20 |
| 7 | 2.312 | 6 | 2.933 | 20 |
| 8 | 2.368 | 6 | 2.687 | 21 |
| 9 | 2.473 | 10 | 2.823 | 23 |
| 10 | 2.631 | 5 | 2.6 | 24 |

סעיף ב'

נוכיח **למה:**

**נוכיח כי עבור $T_1, \dots, T_n$ כך ש: $height(T_1) \le \dots \le height(T_n)$ מתקיים כי:**

$$\forall i \;\; \left|height(T_i) - height\big(join(T_1, \dots, T_{i-1})\big)\right| \le height(T_i) - height(T_{i-1}) + c$$

**כאשר c הינו קבוע.**

אם $height(T_i) \ge height\big(join(T_1, \dots, T_{i-1})\big)$:

$$\left|height(T_i) - height\big(join(T_1, \dots, T_{i-1})\big)\right| = height(T_i) - height\big(join(T_1, \dots, T_{i-1})\big) \le$$

$$\le_* height(T_i) - height(T_{i-1}) \le height(T_i) - height(T_{i-1}) + c$$

(*) $height(T_{i-1}) \le height\big(join(T_1, \dots, T_{i-1})\big)$.

אם $height(T_i) < height\big(join(T_1, \dots, T_{i-1})\big)$ אז:

$$\left|height(T_i) - height\big(join(T_1, \dots, T_{i-1})\big)\right| = height\big(join(T_1, \dots, T_{i-1})\big) - height(T_i) \le$$

$$\le_{(1)} height(T_{i-1}) + c - height(T_i) \le_{(2)} height(T_i) - height(T_{i-1}) + c$$

**(1)** נראה באינדוקציה $height\left(join(T_1, \dots, T_j)\right) \le height\left(parent(T_j)\right)$

מקרה בסיס *j=2*:

$$height\big(join(T_1, T_2)\big) \le \max\{height(T_1), height(T_2)\} + 1 = height(T_2) + 1 \le height\big(parent(T_2)\big)$$

נניח נכונות עבור *j-1* ונוכיח עבור *j*:

$$height\left(join(T_1, \dots, T_j)\right) \le$$

$$\le \max\left\{height\left(join(T_1, \dots, T_{j-1})\right), height(T_j)\right\} + 1 \le$$

$$\le_* \max\left\{height\left(parent(T_{j-1})\right) - 1, height(T_j)\right\} + 1 \le$$

(*) הנחת אינדוקציה.

$$\le_* \max\left\{height\left(parent(T_{j-1})\right) - 1, height\left(parent(T_j)\right) - 1\right\} + 1 =$$

(*) גובה עץ שווה לגובה שורש העץ, שקטן לפחות באחד מהגובה של ההורה שלו.

$$= \max\left\{height\left(parent(T_{j-1})\right), height\left(parent(T_j)\right)\right\} = height\left(parent(T_j)\right)$$

ומכאן מתקיים והרי הפרש הגבהים בין בן ואב חסום בעץ *AVL* מאוזן. אז מכאן:

$$height\left(join(T_1, \dots, T_j)\right) - height(T_j) \le height\left(parent(T_j)\right) - height(T_j) \le c$$

$$height\left(join(T_1, \dots, T_j)\right) \le height(T_j) + c$$

**(2)** $height(T_{i-1}) - height(T_i) \le 0 \le height(T_i) - height(T_{i-1}) + c$

ומכאן הוכחנו את הלמה.

**עלות join ממוצע בתרחיש של split על איבר מסוים שבחרנו – על האיבר המקסימלי בתת העץ השמאלי:**

נסמן את עומק האיבר הנבחר ב-$d=\theta(\log(n))$.

כמות ה-*join* שנבצע תהיה *d* כאשר כל *d-1* ה-*join* הראשונים שנבצע יהיו לצד שמאל.

נשים לב כי גובה של *join* לעץ בגובה $h_1$ עם עץ בגובה $h_2$ הינו לכל היותר $\max\{h_1, h_2\}+1$

נשים כי מהאיבר שאותו מבחרנו עד לבן השמאלי של השורש נבצע הכנסות של תתי עצים שההפרשים בין הגבהים ביניהם יהיו חסומים ע״י קבוע כלשהו ***t*** (העץ מאוזן לכן הפרשי גובה בין שני בנים לאותו אב לא יהיה גדול מ-1.) נראה זאת באינדוקציה.

לכל *node* במסלול מהעלה לשורש שבחרנו נסמן את גובה העלה ה-*i* במסלול (העלה שמתקבל ע״י תנועה מעלה *i* פעמים) בתור $h_i$.

מתקיים כי $h_i \geq h_j,\ \forall j < i$ מכיוון שאנו עולים בעץ וגדלי תתי העצים גדלים ככל שמתקדמים מעלה מכך העץ מאוזן. מה**למה:**

$$h_i - h_{join(i-1)} \leq h_i - h_{i-1} + c \leq t + c, \qquad c \text{ קבוע}$$

לאחר שביצענו את *d-1* ה-*join* לעץ בעל הערכים הקטנים נוסיף עוד תת עץ אחרון שגובהו הינו גובה העץ פחות 1 לעץ ריק מכאן ממוצע הפרשי הגבהים יהיה חסום מלעיל ע״י

$$\frac{(t+c)(d-1)+h-1}{d} \leq_* \frac{(t+c)\cdot k\cdot\log(n)+m\log(n)}{l\log(n)} \leq \frac{(t+c)\cdot k+m}{l} = O(1)$$

$$l\log(n) \leq d \leq k\log(n), \qquad h \leq m\log(n), \qquad h,d=\theta(\log(n))$$

מכיוון שחסמנו מלמעלה את ההפרש ע״י קבוע את הוא מתפקד גם כחסם תחתון. וממוצע הפרשי הגבהים הינו $\theta(1)$.

**עלות join ממוצע בתרחיש של ל- split על איבר אקראי:**

נבחר איבר אקראי כלשהוא בעץ. תחילה ניצור תתי-עץ מהבנים השמאליים והימניים שלו, הפרשי הגבהים בין עצים אלו חסום ע״י 1 מכיוון שהעץ מאוזן, נניח אם כך כי ההפרש ביניהם שווה ל-0 וגובהם $h$.

כעת נסמן את האיברים במסלול מהאיבר לשורש אחר המפתח שלהם גדול מן האיבר שבחרנו ב-:

$$a_1, a_2, \dots, a_k$$

וגובה תתי העץ של הבנים שלהם ב- $H_1, H_2, \dots, H_k$ בהתאמה.

כעת ונסמן את האיברים במסלול מהאיבר לשורש אחר המפתח שלהם גדול מן האיבר שבחרנו ב-:

$$b_1, b_2, \dots, b_l$$

וגובה תתי העץ של הבנים שלהם ב- $h_1, h_2, \dots, h_l$ בהתאמה.

$$k + l = d$$

כאשר d הינו עומק העלה.

נשים לב כי $H_1, H_2, \dots, H_k$ וגם $h_1, h_2, \dots, h_l$ סדרות עולות . ואנו מוספים את תתי העצים המתאימים פי הסדר לכן:

**מהלמה** הפרש הגבהים בין העצים יהיה חסום ע״י:

$$h_i - h_{join(i-1)} \leq h_i - h_{i-1} + c$$

כאשר c הינו קבוע. מכאן סכום הפרשי הגבהים חסום ע״י:

$$c \cdot (l + k) + h_l - h_{l-1} + +h_{l-1} - h_{l-2} + \cdots + h_1 - h + H_k - H_{k-1} + H_{k-1} - H_{k-2} + \cdots + H_1 - h$$
$$= h_l + H_k - 2h + c \cdot (l + k) \leq (2 + c)d$$

מכאן נעשה ממוצע:

$$AVG \leq \frac{(2 + c)d}{k + l} = \frac{(2 + c)d}{d} \leq 2 + c = O(1)$$

מכיוון שחסמנו מלמעלה את ההפרש ע״י קבוע את הוא מתפקד גם כחסם תחתון. וממוצע הפרשי הגבהים הינו $\theta(1)$.

נשים לב כי תוצאות הניסויים עם ניתוח התיאורטי כי ניתן לראות כי גם בניסוי אין מגמת עלייה ככל שעולים במספר האיברים והממוצע נאשר באותו טווח בין 2 ל-3 ולכן זה מתיישב עם הניתוח תיאורטי שבו מצאנו כי הממוצע חסום ע״י קבוע.

סעיף ג׳

**עלות join מקסימלי בתרחיש של split על איבר מסוים שבחרנו – על האיבר המקסימלי בתת העץ השמאלי:**

נשים לב כי עלות ה-join המקסימלית תתקבל ב-join האחרון בו נוסיף את תת העץ הימני של השורש ב-join לעץ ריק.

עבור כל המקרים האחרים הראנו בסעיף ב׳ כי הינם חסומים ע״י קבוע כלשהו נסמן c.

גובה תת העץ הימני יהיה h-1 או h-2 כאשר h הינו גובה העץ, $h = \theta(\log(n))$. ואז הפרש הגבהים המקסימלי יהיה לכל היותר h-1 ולכל הפחות h-2. בנוסף קיימים קבועים כך ש:

$$k\log(n) \leq h \leq l\log(n)$$

ואז נקבל כי ההפרש (נסמן ב-diff):

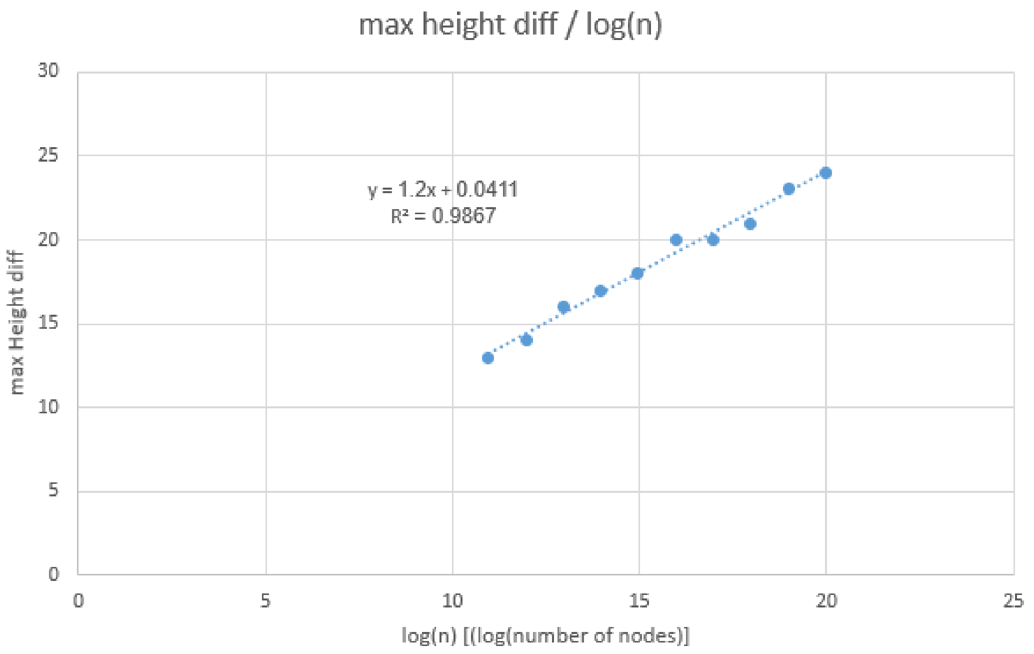
$$\Omega(\log(n)) \leq k\log(n) - 2 \leq diff \leq \max\{l\,log(n) - 1, c\}$$

נשים לב כי קיים N כך שלכל n>N מתקיים $l\,log(n) > c + 1$ (מכיוון ש-$c$ קבוע ולכן:

$$\max\{l\,log(n) - 1, c\} = O(\log(n))$$

מכאן הפרש הגבהים המקסימלי הינו $\theta(\log(n))$.

אם נבחן את ההפרשים המקסימליים בתוצאות הניסוי ביחס ל-log(n) (ציר y הפרש גבהים מקסימלי וציר log x של מספר האיברים) נצפה לראות תלות לינארית, ואכן תוצאות הניסוי מתיישבות עם הניתוח התיאורטי:




ניתן לראות כי התלות קרובה להיות לינארית ובקירוב לא רע של $R^2 = 0.9867$.