**AVLTree**

**תיעוד הקוד**

**מחלקת AVLTree**

מחלקה זו מממשת AVL Tree שבו כל המפתחות הינם מספרים טבעיים שונים זה מזה. העץ יכול לאחסן מחרוזות.

שדות:

* min – שדה השומר מצביע לצומת בעלת המפתח המינימלי בעץ
* max – שדה השומר מצביע לצומת בעלת המפתח המקסימלי בעץ
* root – שדה השומר מצביע לשורש העץ

מתודות:

* **empty()** – פונקציית מופע של אובייקט מסוג AVLTree. מחזירה True אם העץ ריק ומחזירה False אם העץ אינו ריק.

**אלגוריתם:** הפונקציה בודקת אם העץ ריק לפי המצביע של השורש – אם המצביע השמור בשדה root מצביע על null , הפונקציה תחזיר True.

**סיבוכיות:** הפונקציה מבצעת פעולת השוואה אחת ולכן הסיבוכיות תהיה .

* **Insert(int k, String i)**  - פונקציית מופע של אובייקט AVLTree. הפונקציה תכניס צומת בעלת מפתח k וערך i לעץ אם לא קיים צומת עם אותו המפתח. הפונקציה תחזיר את מספר פעולות האיזון שנדרשו בשלב התיקון. אם קיים איבר בעל אותו מפתח הפונקציה תחזיר -1.

**אלגוריתם:**

Case A: העץ לפני ההכנסה הוא עץ ריק - במקרה זה נכניס את הצומת כשורש של העץ, נעדכן את שדות מינ' מקס' ו size של הצומת. בנוסף נעדכן את מונה פעולות האיזון להיות 0.

Case B: קיים איבר בעל מפתח זהה בעץ - נבדוק באמצעות הפונקציה isExist(k) האם קיים מפתח בעל מפתח זהה בעץ. אם כן נצא מהפונקציה ונחזיר את הערך -1.

Case C: הכנסת איבר לעץ לא ריק ובו לא קיים איבר בעל מפתח k – נשלח את הצומת לפונקציה Tree\_Insert ונשמור את הערך שהיא מחזירה במונה פעולות האיזון.

נעדכן את שדה המקס' והמינ' של העץ ע"י שימוש בפעולות minNode,maxNode.

נחזיר את מונה פעולות האיזון.

**סיבוכיות:**

Case A: נבצע מספר קבוע של פעולות ולאחר מכן נצא מהפונקציה - .

Case B: נקרא לפונקציה isExist ולאחר מכן נצא מהפונקציה. סיבוכיות isExist היא .

Case C: נעדכן מצביעים (מספר קבוע של פעולות) – ולאחר מכן נקרא לפונקציה Tree\_Insert שסיבוכיותה היא . לאחר האיזון נקרא לפונקציות maxNode,minNode שגם כן – סה"כ נקבל .

* **Tree\_Insert(AVLNode root, AVLNode toInsert)**  - הפונקציה מקבלת שורש של עץ וצומת להכניס ומבצעת הכנסה של הצומת. אם נשתמש בפונקציה זו זה רק ב Case C: כלומר לא קיים איבר בעץ עם מפתח זהה ל toInsert. הפונקציה תחזיר את מספר פעולות האיזון הנדרשות בשלב התיקון.

**אלגוריתם:**

נחפש את הצומת בעץ הקיים ע"י קריאה לפונקציה Tree\_Position. הפונקציהTree\_Position מעדכנת את שדה ה size של כל צומת שעברנו בה ב +1 כדי לתחזק את שדה size לאחר ההכנסה. על ידי השוואת המפתחות נדע האם להכניס את הצומת החדשה כבן ימני או שמאלי. לאחר מכן נעדכן את שדות ה height, parent של הצומת החדשה ונשים לב 2 בנים וירטואלים באמצעות שימוש בפונקציה InsertVirtualChilds.

כמו כן נבדוק האם הצומת שבה היינו צריכים להכניס את הצומת החדשה (כעת היא ההורה של הצומת החדשה) הייתה עלה לפני ההכנסה ע"י קריאה לפונקציה is\_leaf לפני ההכנסה.

אם הצומת לא הייתה עלה לפני ההכנסה, זה אומר שהיא בוודאות הייתה אונארית ולכן הכנסה של צומת חדשה לא תדרוש פעילות איזון ולכן נצא מהפונקציה ונחזיר 0.

אחרת, נגדיל את הדרגה של הצומת ע"י שימוש בפונקציה Promote() ונשלח את העץ לפונקציה rebalanceTreeInsert עם פוינטר לאבא של הצומת המוכנסת כארגומנט בפונקציה. נחזיר את הערך שיוחזר ממנה.

**סיבוכיות:**

תחילה נקרא לפונקציה Tree\_Position פעם אחת וסיבוכיותה היא .

קריאה לפונקציה is\_leaf פעם אחת וסיבוכיותה היא

בדיקה האם הצומת המוכנסת תהיה בן שמאלי או ימני והכנסתה תוך עדכון השדות הרלוונטים – מספר קבוע של פעולות –

קריאה לפונקציה Promote() –

קריאה לפונקציה rebalanceTreeInsert פעם אחת -

סה"כ נקבל שסיבוכיות הפונקציה היא -

* **rebalanceTreeInsert(AVLNode node)** – הפונקציה מקבלת את הצומת שהחל ממנה צריכים לבצע את תיקון העץ. הפונקציה תתקן את העץ כך שבסיום הפונקציה העץ יהיה עץ AVL תקין המקיים את האינוורנטה של עצי AVL. הפונקציה תחזיר את מספר פעולות האיזון הנדרשות לכך. כל פעולת rotate, demote, promote תיספר כפעולה אחת.

**אלגוריתם:**

לפני הכניסה לפונקציה נקדם את דרגת הצומת parent (האבא של הצומת המוכנסת) ב 1, ולכן מונה פעולות האיזון יאותחל בערך 1. נעדכן את parent להיות אבא שלו. נרוץ בלולאה כל עוד הצומת parent היא לא צומת ריקה. בתחילת כל לולאה נבדוק אם יש בעיה בעץ לאחר שגלגלנו את הבעיה למעלה (ע"י בדיקות isValid המפורט בהמשך [בודק הפרשי דרגות]). אם הבעיה נפתרה, נצא מהלולאה. (בדיקת parent.parent.isValid נועדה עבור מקרה קצה של פונקציית join).

אחרת, נשמור את הפרש הדרגות בין הצומת לבן השמאלי (ע"י קריאה לפונקציה getLeftEdge) ואת הפרש הדרגות בין הצומת לבן הימני (ע"י קריאה לפונקציה getRightEdge).

בתוך הלולאה נחלק את הטיפול בצומת למספר מקרים בהתאם להפרשי הדרגות בין הצמתים לבנים ובין הבנים לנכדים ונטפל בכל בעיה בצורה שונה. בתיעוד זה נתייחס רק לכיוון אחד של המקרים אך בקוד יש גם את המקרה הסימטרי שמטופל באופן סימטרי לחלוטין.

**\*הפרשי הדרגות בקוד מחושבים לאחר שעשינו Promote לאבא של הצומת אותה בודקים.**

Case A: צומת 2-0 עם בן שהוא צומת 1-2 – במקרה זה נבצע סיבוב ימינה ע"י קריאה ל RRotate של הצומת, נבצע הורדה של דרגת הצומת ב 1 ע"י קריאה ל demote() ונגדיל את מונה פעולות האיזון ב 2. במקרה זה לאחר הסיבוב העץ תקין ולכן נצא מהלולאה ונחזיר את מונה פעולות האיזון.

Case B: צומת 2-0 עם בן 2-1 – במקרה זה נבצע demote לצומת ולבנה השמאלה, נבצע promote לבנה הימני נבצע כפול (שמאל ואז ימין) של הצומת ע"י קריאה לפונקציה LRRotate ונקדם את מונה פעולות האיזון ב 5. לאחר הסיבוב הכפול העץ יהיה מאוזן ולכן נצא מהלולאה ונחזיר את מונה פעולות האיזון.

Case C: קורה כאשר הצומת היא 1-0 – במקרה זה נבצע promote לצומת ונקדם את מונה פעולות האיזון ב 1.

Case D: מקרה אשר הוכנס לפונקציה זו עבור פונקצית join – יכול להתרחש רק כאשר נשתמש בה. במקרה זה בודקים האם קיים צומת 3-1 שבנה השמאלה הוא צומת 1-1. במקרה זה נבצע demote לאבא, נבצע סיבוב שמאלה לאבא של הצומת ולאחר מכן נבצע promote לצומת. פעולה זו לא תקדם את מונה פעולות האיזון מכיוון שהיא מתרחשת רק ב join ולא ב insert.

\*נציין שוב, שכל המקרים יכולים להתרחש גם במקרה הסימטרי למקרים שציינו ומטופלים בקוד, לא נציין אותם בתיעוד מכיוון שהם מטופלים באופן זהה וסימטרי למקרים הנ"ל.

בסיום הלולאה נקדם את parent להיות אבא שלו ונמשיך לרוץ בלולאה.

בסיום הלולאה, נחזיר את מונה פעולות האיזון.

**סיבוכיות:**

תחילה, מבצעים מספר קבוע של פעולות וקריאה לפונקציות שהן מסיבוכיות : isValid,getRightEdge,getLeftEdge.

בתוך הלולאה:

נחשב כעת סיבוכיות של איטרציה בודדת: תחילה,קוראים לפונקציות שהן מסיבוכיות :isValid ,getRightEdge getLeftEdge. לאחר מכן, אם נכנס למקרה A, נבצע מספר קבוע של פונקציות עם סיבוכיות : RRotate, LRrotate,demote,promote,getLeftEdge,getRidgeEdge לכן סיבוכיות מקרה זה היא . אם נכנס למקרה B, נבצע מספר קבוע של פונקציות עם סיבוכיות : RLRotate,LRRotate,demote,promote,getLeftEdge,getRightEdge ולכן סיבוכיות מקרה זה היא . באופן דומה נקבל שהסיבוכיות של מקרה C ומקרה D הם גם . סה"כ נקבל שכל איטרציה היא .

מספר איטרציות:

כל סיום איטרציה מעדכנים את הצומת להיות אבא שלה, ולכן נבצע לכל היותר איטרציות , מכיוון שהגובה חסום ע"י (תכונות AVL).

חישוב סיבוכיות של הפונקציה כולה:

סה"כ נקבל שלפני הלולאה, מבצעים ומבצעים איטרציות שכל איטרציה היא . סה"כ נקבל שהסיבוכיות של הפונקציה כולה היא .

* **Tree\_Position(AVLNode node, int k)**  - מקבלת צומת (שורש של תת עץ או של עץ) ומפתח k ומחזירה את הצומת עם המפתח k . בנוסף היא מקדמת את שדה ה size בכל צומת שעוברים בה ב 1 (משמש בפעולה insert).

**אלגוריתם:** נאתחל מצביע ל AVLNode להצביע על node שהיא מקבלת כארגומנט (השורש). בכל איטרציה נבדוק האם המפתח שווה / גדול / קטן ל k. אם שווה נחזיר את הצומת, אם גדול נעדכן את הצומת להיות בנה השמאלי ואם קטן נעדכן את הצומת להיות בנה הימני. נרוץ בלולאה כל עוד הצומת היא אמיתית (הבדיקה מתבצעת ע"י קריאה ל isRealNode). לאחר הלולאה (אם הגענו לחלק זה, זה אומר שלא קיים איבר עם מפתח k בעץ נחזיר המצביע (יצביע כעת על בן וירטואלי).

**סיבוכיות:** בכל לולאה מבצעים מספר קבוע של פעולות. נבצע לכל היותר איטרציות מכיוון שכל איטרציה יורדים מהשורש רמה אחת למטה בעץ, ומכיוון שזה עץ AVL גובהו חסום ע"י . סה"כ קיבלנו שסיבוכיות פונקציה זו היא .

* **RRotate(AVLNode node)**  - הפונקציה מקבלת צומת ומבצעת סיבוב ימני שלו.

**אלגוריתם:** נעדכן את המצביעים והשדות של הצמתים כך שיתאים לסיבוב ימינה של העץ. אם הצומת שמבצעים עליה סיבוב ימינה הייתה שורש, נעדכן את השורש להיות הצומת החדשה שעלתה דרגה בעקבות הסיבוב. כמו כן נעדכן את שדות ה size בצמתים הרלוונטיות שמושפעות מהסיבוב.

**סיבוכיות:** מתבצע מספר קבוע של פעולות בכל קריאה לפונקציה שאינו תלוי במספר הצמתים בעץ ולכן הסיבוכיות תהיה .

* **LRotate(AVLNode node)**  - הפונקציה מקבלת צומת ומבצעת סיבוב שמאלה שלו.

**אלגוריתם:** נעדכן את המצביעים והשדות של הצמתים כך שיתאים לסיבוב שמאלה של העץ. אם הצומת שמבצעים עליה סיבוב שמאלה הייתה שורש, נעדכן את השורש להיות הצומת החדשה שעלתה דרגה בעקבות הסיבוב. כמו כן נעדכן את שדות ה size בצמתים הרלוונטיות שמושפעות מהסיבוב.

**סיבוכיות:** מתבצע מספר קבוע של פעולות בכל קריאה לפונקציה שאינו תלוי במספר הצמתים בעץ ולכן הסיבוכיות תהיה .

* **LRRotate(AVLNode node)** - הפונקציה מקבלת צומת ומבצעת סיבוב כפול שלו (שמאלה ואז ימינה)

**אלגוריתם:** נבצע סיבוב שמאלה על בנה השמאלית של הצומת ולאחר מכן נבצע סיבוב ימינה על הצומת עצמה.

**סיבוכיות:** מתבצעת קריאה ל 2 פונקציות שהן סיבוכיות לכן סה"כ נקבל שסיבוכיות הפונקציה היא .

* **RLRotate(AVLNode node)** - הפונקציה מקבלת צומת ומבצעת סיבוב כפול שלו (ימינה ואז שמאלה)

**אלגוריתם:** נבצע סיבוב שמאלה על בנה השמאלית של הצומת ולאחר מכן נבצע סיבוב ימינה על הצומת עצמה.

**סיבוכיות:** מתבצעת קריאה ל 2 פונקציות שהן סיבוכיות לכן סה"כ נקבל שסיבוכיות הפונקציה היא .

* **delete(int k)**- פונקציית מופע של מחלקת AVLTree. הפונקציה מקבלת מספר שלם k המהווה מפתח בעץ, ומוחקת אותו אם הוא קיים. הפונקציה מחזירה את מספר פעולות האיזון שנדרשו סה"כ בשלב תיקון העץ. מחזירה 1- אם לא קיים מפתח כזה בעץ.

**אלגוריתם:**

Case A: נקרא לפונקציה isExist. אם המפתח k לא קיים בעץ . נחזיר -1.

Case B: לפני מחיקה , קיים איבר אחד בלבד בעץ – נעדכן את השורש, שדה המינימום ושדה המקסימום להיות null, ונחזיר 0.

אחרי Case B נחפש את האיבר המיועד למחיקה ע"י קריאה לפונקציה searchNode.

Case C: האיבר אותו רוצים למחוק הוא לא עלה ולא צומת אונארי, כלומר עלה פנימי – נמצא את הsuccessor של הצומת, שהוא בהכרח יהיה צומת אונארי או עלה, נקטין את size כל הצמתים בדרך לsuccessor, נחליף את ערכי הצומת שרצינו למחוק בערכי הsuccessor, ונשלח את הsuccessor לפונקציה Tree\_Delete. לאחר ביצוע המחיקה בפועל, נמצא את האיבר המקסימלי והמינימלי המעודכנים לאחר המחיקה ונחזיר את מס' פעולות האיזון שהיו.

Case D: האיבר אותו רוצים למחוק הינו עלה או צומת אונארי – נקטין את size כל הצמתים בדרך לאיבר המיועד למחיקה, ונשלח אותו לפונקציה Tree\_Delete. לאחר המחיקה בפועל, נחשב מחדש את max וmin של העץ ונחזיר את מס' פעולות האיזון שהיו.

**סיבוכיות:**

Case A: סיבוכיות פונקציה isExist מפורטת בהמשך (). לאחר מכן, נצא מהפונקציה או נמשיך הלאה למקרה הבא. לכן סה"כ הסיבוכיות של מקרה זה הינה .

Case B: נבצע מספר קבוע של פעולות, נעדכן מצביעים ולאחר מכן נצא מהפונקציה. לכן סה"כ סיבוכיות היא .

מציאת האיבר המיועד למחיקה ע"י הפונקציה seachNode עולה (מפורט בהמשך), לאחר מכן נמשיך למקרה הבא:

Case C: מציאת הSuccessor עולה (מפורט בהמשך), פונקציית decreasePath גם כן עולה (מפורט בהמשך(. לאחר מכן ישנן מס' פעולות קבועות וקריאה לפונקציה Tree\_Delete שהיא מסיבוכיות . לאחר ביצוע המחיקה, חיפוש האיבר המקסימלי והמינימלי בעץ עולה ולכן סה"כ הסיבוכיות של מקרה זה היא .

Case D: הקטנת size של כל האיברים בדרך לצומת ע"י הפונקציה decreasePath עולה (מפורט בהמשך). לאחר מכן, קוראים לפונקציה Tree\_Delete שהיא מסיבוכיות . לאחר מכן, מציאת האיבר המקסימלי והמינימלי החדש בעץ עולה לכן סה"כ הסיבוכיות של מקרה זה היא .

לכן, ראינו כי סה"כ סיבוכיות הפונקציה היא .

* **Tree\_Delete(AVLNode toDelete)**- פונקציית מופע של מחלקת AVLTree המקבלת צומת המיועד למחיקה שהוא בהכרח אונארי או עלה. הפונקציה מבצעת מחיקה של הצומת ומחזירה את מספר פעולות האיזון הנדרשות בשלב התיקון.

**אלגוריתם:**

Case A: הצומת אותו מוחקים הינו עלה – נבדוק אם הצומת המיועד למחיקה הינו בן ימני או בן שמאלי של אבא שלו (נזכר שטיפלנו במקרה שבעץ יש איבר 1 בפונקציה delete, לכן תמיד בשלב זה לצומת המיועד למחיקה יהיה אבא). בה"כ הצומת הינו בן ימני, אזי נשים בתור בן ימני של האבא בן של הצומת הנמחק (שזה בעצם עלה ווירטואלי), ונקבע שאבא שלו של העלה הווירטואלי יהיה האבא הנ"ל. המקרה שהצומת הינו בן שמאלי הוא סימטרי.

נבדוק אם העץ עדיין נשאר חוקי לאחר מחיקה זו (הקשתות שהשתנו נהיה 2 1 או 1 2). אם כן, נחזיר 0. אחרת, נשלח את העץ לrebalanceTreeDelete ונחזיר את מספר פעולות האיזון.

Case B: הצומת אותו מוחקים הינו צומת אונארי –   
case B.1: הצומת האונארי הינו שורש (אבא שלו null) – נבדוק אם הבן הקיים של השורש הינו ימני או שמאלי, ונעדכן את המצביעים בהתאם: השורש החדש הוא הבן הקיים, האבא של הבן הקיים הוא null, ומצביעי הmax והmin יצביעו על הבן הקיים של השורש שמחקנו.

case B.2: הצומת האונארי אינו השורש – נבדוק אם הצומת שמוחקים הינו הבן הימני או השמאלי של אבא שלו. אם הוא הבן הימני, נבדוק אם הבן הימני של הצומת שמוחקים קיים, אם כן, נגדיר את הבן הימני שלו בתור הילד הימני של האבא של הצומת שמוחקים. אם הוא לא קיים נגדיר את הבן השמאלי של הצומת שמוחקים בתור הבן הימני של אבא של הצומת שמוחקים. מקרה סימטרי עושים כאשר הצומת שמוחקים הוא הבן השמאלי של אבא שלו.

לאחר 2 הcases הנ"ל נבדוק אם העץ עדיין נשאר חוקי (הקשתות שהשתנו נהיו 2 1 או 1 2). אם כן, נחזיר 0. אחרת, נשלח את העץ לrebalanceTreeDelete ונחזיר את מספר פעולות האיזון.

**סיבוכיות:**

Case A: כלל הפעולות המבוצעות בחלק זה הינן פעולות השוואה קבועות ושינוי מצביעים, למעט הפעלת הפונקציה rebalanceTreeDelete שהנה בסיבוכיות (יפורט בהמשך).

Case B: בדומה למקרה A, כלל הפעולות המבוצעות הינן פעולות השוואה בעלות זמן ריצה קבוע, ושינויי מצביעים, למעט הפונקציה rebalanceTreeDelete שהנה בסיבוכיות .

לכן, סה"כ סיבוכיות פונקציה זו הנה .

* **decreasePath(int k)**- פונקציית מופע של מחלקת AVLTree המקבלת מספר שלם k המהווה מפתח בעץ, ומפחיתה את שדה size של כל הצמתים בדרך מהroot לצומת זה.

**אלגוריתם:** הפונקציה תחילה בודקת אם המפתח שקיבלנו הינו השורש. במידה וכן, היא מפחיתה את ערך size שלו ב1 ומחזירה אותו. אם הוא לא השורש, הפונקציה סורקת את העץ עד שמגיעה אליו, ולכל צומת שעוברת בו מקטינה את הsize ב1. לבסוף מחזירה את הצומת שהגענו אליה.

**סיבוכיות:** בפונקציה זו נסייר לכל היותר גובה העץ (כי בכל איטרציה נירד לבן השמאלי\ימני). גובה עץ AVL הינו ולכן סה"כ סיבוכיות פונקציה זו היא .

* **Successor(AVLNode node)**- פונקציה זו מקבלת צומת הנמצאת בעץ ומחזירה את הצומת העוקבת לה בעץ.

**אלגוריתם:** ראשית, נגדיר עץ זמני temp חדש. הפונקציה בודקת אם קיים בן ימני לצומת הנקלטת. אם קיים, כזה, נגדיר את שורש העץ temp להיות הבן הזה, ונחזיר את האיבר המינימלי בעץ temp (נשלח לפונקצייה minNode()). במידה ולא קיים בן ימני, נעלה "למעלה ושמאלה" כל עובד קיים אבא שאכן הצומת הנוכחית היא הבן הימני שלו. לבסוף, נחזיר את האבא של הצומת האחרונה עבורה מתקיימת הלולאה, שהוא יהיה האבא הכי נמוך, ככה שnode המקורי נמצא בתת עץ השמאלי שלו.

**סיבוכיות:** בפונקציה זו, אנו מבצעים פעולת minNode שעלותה . או שכפי שתיארנו נעלה במעלה העץ עד שנגיע לאבא הנמוך ביותר ככה שהאיבר שאנו מחפשים לו successor נמצא בתת עץ השמאלי שלו. עלייה זו תהיה לכל יותר גובה העץ, כלומר .

* **rebalanaceTreeDelete(AVLNode node)**- הפונקציה מקבלת את הצומת שהחל ממנה צריכים לבצע את תיקון העץ. הפונקציה תתקן את העץ כך שבסיום הפונקציה העץ יהיה עץ AVL תקין המקיים את האינווריאנטה של עצי AVL. הפונקציה תחזיר את מספר פעולות האיזון הנדרשות לכך. כל פעולת rotate, demote, promote תיספר כפעולה אחת.

**אלגוריתם:** ראשית, נאתחל את מונה פעילויות האיזון ל-0. נגדיר נשתנה parent תחילה בתור הnode הנוכחי, ונרוץ בלולאה כל עוד node שיעודכן בכל איטרציה להיות אבא שלו שונה מnull או עד שהגענו לאחד המקרים הסופיים. בתחילת כל איטרציה נשמור את הפרש הדרגות בין הצומת לבן השמאלי (ע"י קריאה לפונקציה getLeftEdge) ואת הפרש הדרגות בין הצומת לבן הימני (ע"י קריאה לפונקציה getRightEdge). בתוך הלולאה נחלק את הטיפול בצומת למספר מקרים בהתאם להפרשי הדרגות בין הצמתים לבנים ובין הבנים לנכדים ונטפל בכל בעיה בצורה שורה כפי שלמדנו בכיתה. בתיעוד זה נתייחס רק לכיוון אחד של המקרים אך בקוד יש גם את המקרה הסימטרי שמטופל באופן סימטרי וזהה לחלוטין.

Case A: הפרשי הדרגות בין הצומת לבין הבנים שלה הם 2-2. נוריד את הדרגה של הצומת, ונעלה ב1 את מונה פעולות האיזון. במקרה זה, או שהבעיה נפתרה, או שגלגלנו את הבעיה צעד אחד למעלה.

Case B: הפרשי הדרגות בין הצומת לבין הבנים שלה הם 1 – 3.

Case B.1 – הפרשי הדרגות בין הבן הימני לנכדים של הצומת הם 1 – 1: זהו מקרה סופי, נוריד את הדרגה של הצומת, נעלה את הדרגה של הבן הימני שלה ונבצע גלגול שמאלה מהצומת. נחזיר את הקאונטר בתוספת 3 פעולות איזון.

Case B.2 – הפרשי הדרגות בין הבן הימני לנכדים של הצומת הם 1 – 2: נוריד את דרגת הצומת ב2 ונבצע עבורה גלגול שמאלה. הבעיה או נפתרה או שגלגלנו אותה צעד אחד למעלה.

Case B.3 – הפרשי הדרגות בין הבן הימני לנכדים של צומת הם 2 – 1: נוריד את דרגת הצומת פעמיים, נוריד את דרגת הבן הימני של הצומת ונעלה את דרגת הבן השמאלי של הבן הימני של הצומת. לאחר מכן נבצע סיבוב כפול Right Rotate ואז Left Rotate ונעלה את הקאונטר ב6. במקרה זה הבעיה או נפתרה או עלתה צעד אחד למעלה.

רגע לפני סיום הלולאה נעדכן את הצומת להיות אבא שלה. בסוף הלולאה הגדולה נחזיר את מספר פעולות האיזון.

* נציין שוב, שכל המקרים יכולים להתרחש גם במקרה הסימטרי למקרים שציינו ומטופלים בקוד, לא נציין אותם בתיעוד מכיוון שהם מטופלים באופן זהה וסימטרי למקרים הנ"ל.

**סיבוכיות:** לפני הלולאה מתבצעות מספר קבוע של פעולות.   
בתוך הלולאה: נחשב סיבוכיות של איטרציה בודדת: לפני חלוקה למקרים, מתבצעות קריאות לפונקציות מסיבוכיות O(1) (getLeftEdge וgetRightEdge). בתוך כל מקרה מתבצעות מספר קבוע של פעולות עם סיבוכיות O(1) שהן: getLeftEdge \ getRightEdge, demote, promote, LRotate , RRotate. סה"כ נקבל שכל איטרציה היא מסיבוכיות O(1).   
מספר איטרציות: כל סיום איטרציה מעדכנים את הצומת להיות אבא שלה, ולכן לנבצע לכל היותר איטרציות, מכיוון שבעץ AVL הגובה חסום ע"י .

סה"כ נקבל מתבצעות איטרציות שבכל איטרציה מבצעים O(1) פעולות ולכן הסיבוכיות של הפונקציה כולה היא .

* **min()**- הפונקציה הינה פונקצית מופע על עץ. מחזירה את הערך של הצומת בעלת המפתח המינמלי בעץ. תחזיר null אם העץ ריק.

**אלגוריתם:** מחזירה את הערך של הצומת עליו שדה min של העץ מצביע. הפונקציה תחזיר null אם העץ ריק (מתבצע ע"י בדיקה של המצביע השמור ב root).

**סיבוכיות:** מתבצע מספר קבוע של פעולות לכן סיבוכיות הפונקציה תהיה .

* **max()**- הפונקציה הינה פונקצית מופע על עץ. מחזירה את הערך של הצומת בעלת המפתח המקסימליבעץ. תחזיר null אם העץ ריק.

**אלגוריתם:** מחזירה את הערך של הצומת עליו שדה max של העץ מצביע. הפונקציה תחזיר null אם העץ ריק (מתבצע ע"י בדיקה של המצביע השמור ב root).

**סיבוכיות:** מתבצע מספר קבוע של פעולות לכן סיבוכיות הפונקציה תהיה .

* **keysToArray()**- הפונקציה היא פונקציית מופיע של עץ. הפונקציה מחזירה מערך int ממויין המכיל את כל המפתחות בעץ, או מערך ריק אם העץ ריק.

**אלגוריתם:** ראשית, אנו מגדירים מערך בגודל העץ, ומערך ריק. אם העץ ריק אזי נחזיר את המערך הריק. אחרת נשלח לפונקצייה הרקורסיבית keyToArray\_rec את שורש העץ, המערך הריק המיועד להתמלאות ואת המספר 0 שישמש בתור האינדקס הראשון במערך. לאחר ביצוע הפונקציה הרקורסיבית מוחזר המערך.

**סיבוכיות:** בפונקציה יש פעולות קבועות וקריאה לפונקציה הרקורסיבית keyToArray\_rec שהסיבוכיות שלה היא , לכן סה"כ סיבוכיות הפונקציה היא .

* **keysToArray\_rec(AVLNode node, int[] arr, int index)**- הפונקציה מקבלת את שורש העץ, מערך ריק ואינדקס 0, ומעדכנת בתוך המערך הריק את כל מפתחות העץ בסדר ממוין.

**אלגוריתם:** תנאי העצירה של הפונקציה הרקורסיבית הוא הגעה לצומת שאינו אמיתי. במצב כזה, מחזירים את המשתנה המתאר אינדקס. בתוך הפונקציה עצמה, נבצע 2 קריאות רקורסיביות, קריאה ראשונה עם הבן השמאלי, וקריאה שנייה עם הבן הימני, ככה שבין 2 הקריאות נוסיף למערך את המפתח הנוכחי ונעלה את האינדקס ב1. צורת הכנסה זו הנה זהה לאלגוריתם in-order שראינו בתרגול, אך נדאג במקרה שלנו להחזיר ולתחזק את האינדקס הנוכחי לאחר כל קריאה רקורסיבית.

**סיבוכיות:** בכל קריאה רקורסיבית אנו מבצעים פעולות בעלות זמן קבועה . בכל קריאה ישנן 2 קריאות רקורסיביות נוספות לבן ימני ובן שמאלי של כל צומת, ככה שסה"כ יהיו n קריאות רקורסיביות ונעבור בכל צומת בעץ בדיוק פעם אחת. מכאן, סה"כ הסיבוכיות של פונקציה זו היא .

* **infoToArray()**- הפונקציה היא פונקציית מופע של עץ. הפונקציה מחזירה מערך string ממויין המכיל את כל הערכים של המפתחות בעץ, לפי סדר ממויין של המפתחות שלהם. או מערך ריק אם העץ ריק.

**אלגוריתם:** האלגוריתם זהה לחלוטין לאלגוריתם של keysToArray() המפורט לעיל.

**סיבוכיות:** סיבוכיות הפונקציה היא . הסבר תחת הפונקציה הזהה keysToArray לעיל.

* **infoToArray\_rec(AVLNode node, int[] arr, int index)**- הפונקציה מקבלת את שורש העץ, מערך ריק ואינדקס 0, ומעדכנת בתוך המערך הריק את כל הערכים בעץ בסדר ממויין לפי המפתחות.

**אלגוריתם:** האלגוריתם זהה לחלוטין לאלגוריתם של keysToArray\_rec המפורט לעיל. (למעט השמת node.value במערך במקום node.key)

**סיבוכיות:** סיבוכיות הפונקציה היא . הסבר תחת הפונקציה הזהה keystoArray\_rec.

* **size() –** פונקצית מופע שפועלת על אובייקט AVLTree. מחזירה את גודל העץ.

**אלגוריתם:** אם העץ ריק (מבצע ע"י בדיקה של המצביע בשדה root) הפונקציה תחזיר 0, אחרת תחזיר את ערך השדה size השמור בצומת הנמצאת בשורש העץ.

**סיבוכיות:** מתבצע מספר קבוע של פעולות לכן סיבוכיות הפונקציה תהיה .

* **getRoot()** – פונקצית מופע שפועלת על אובייקט AVLTree. מחזירה את השורש (אובייקט IAVLNode).

**אלגוריתם:** אם העץ ריק (הבדיקה מתבצעת ע"י קריאה לפונקציה empty) הפונקציה תחזיר null, אחרת תחזיר את המצביע השמור בשדה root.

**סיבוכיות:** מתבצע מספר קבוע של פעולות לכן סיבוכיות הפונקציה תהיה .

* **split(int x)**– פונקצית מופע שפועלת על אובייקט AVLTree. הפונקציה מקבלת ערך X ומחזירה מערך בגודל 2 של AVLTree שבה האיבר הראשון הוא עץ AVL עם כל המפתחות שקטנים ממש מ X והאיבר השני הוא עץ AVL עם כל המפתחות שגדולים ממש מ X.

**אלגוריתם:** נאתחל 2 עצים חדשים – עץ שישמור מפתחות גדולים מ X ועץ שישמור מפתחות קטנים מ X. לאחר מכן נאתחל מצביע לצומת עם מפתח בעל הערך X ע"י שימוש בפונקציה searchNode. כעת המצביע מצביע על צומת עם מפתח X. נוסיף את תת העץ השמאלי של צומת זו לעץ הקטן (כינוי לעץ ששומר את המפתחות הקטנים) ואת תת העץ הימני לעץ הגדול (כינוי לעץ ששומר את המפתחות הגדולים). אם X הוא עלה ולמעשה הוספנו בנים וירטואלים לעצים הללו נאתחל אותם להיות שוב עצים חדשים (כדי למנוע באגים בקוד). נבדוק כעת אם X הינו השורש, במידה וכן, נחפש את איברי המינימום והמקסימום של תת העץ הימני ותת העץ השמאלי של העץ ונחזיר את שניהם. במידה והוא לא השורש, נרוץ בלולאה כל עוד המצביע שלנו איננו Null. בכל איטרציה נבדוק האם עלינו מכיוון ימין או מכיוון שמאל. אם עלינו מכיוון ימין, זה אומר שתת העץ השמאלי של הצומת והצומת עצמה כולם קטנים מ X, לכן ננתק אותם מהעץ המקורי שלנו ונוסיף אותם באמצעות join לעץ הקטן. באופן דומה, אם עלינו מכיוון שמאל, זה אומר שהצומת הנוכחית וכל תת העץ הימני שלה גדולים מ X ולכן נצרף אותם באמצעות join לעץ הגדול. לבסוף נעדכן את שדות המינ' והמקס' של העץ הגדול והקטן ע"י קריאות לפונקציה minNode ו maxNode. לאחר מכן נגדיר את האיבר הראשון במערך להיות העץ הקטן, את האיבר השני במערך להיות העץ הגדול ונחזיר את המערך.

**סיבוכיות:** לפני הלולאה מבצעים מספר קבוע של פעולות עם סיבוכיות ופועלת חיפוש אחת שעולה . לאחר הלולאה מבצעים 4 קריאות לכל היותר לפונקציות minNode / maxNode ולכן לאחר הלולאה נבצע פעולות מסדר גודל . כעת נחשב את הסיבוכיות של הלולאה עצמה:

1. נניח ונרצה לעשות join לסדרה של עצים T1,T2,….Tk כאשר מתקיים:

– מתקיים בלולאה שלנו מכיוון שכל איטרציה אנחנו עולים צומת אחת למעלה והעצים תלויים במיקום יותר גבוה.

1. מטענה שהוסברה בכיתה, ראינו שמתקיים עבור כל i בין 2 ל k במקרה זה:

מכיוון שאנו יודעים שהסיבוכיות של join היא הפרש הדרגות בין העצים, למדנו בכיתה שנוכל להגיע לכך שכל פעולת join במהלך split תעלה לנו:

כאשר נסכום את כל ה join שביצענו במהלך split נקבל:

כאשר השוויון הראשון נובע מטענה 2, השוויון השני נובע מעצם היותו טור טלסקופי, והשוויון השלישי והאחרון נובע מכך ש rank(Tk) חסום ע"י הגובה, rank(T1) רק מוריד ו k גם כן חסום ע"י הגובה (מספר האיטרציות).

סה"כ קיבלנו שסיבוכיות הלולאה היא ולכן **סיבוכיות הפונקציה כולה היא .**

* **searchNode(AVLNode, int k)** – הפונקציה מקבלת צומת (שורש של עץ / תת עץ כלשהו) ומחזירה את הצומת בעלת מפתח k, אם לא קיים תחזיר null

**אלגוריתם:** הפונקציה הינה פונקציה רקורסיבית – אם 2 תנאי עצירה, אחד הוא שהגענו לצומת וירטואלית (מתבצע ע"י בדיקת isRealNode) והשני הוא שהגענו לשורש שערך המפתח שלה שווה ל k, במקרה זה נחזיר את השורש עצמה. אחרת, אם הגענו לצומת עם מפתח שגדול מk נקרא לפונקציה עם הבן השמאלי של הצומת, ואם הגענו לצומת שערך המפתח שלה קטן מ k נקרא לפונקציה שוב עם בנה הימני.

**סיבוכיות:**

כל קריאה מתבצעים מספר קבוע של בדיקות, נבצע סה"כ – כגובה העץ, ומכיוון שזה עץ AVL הגבוה חסום ע"י . כל קריאה או שמגיעים לתנאי עצירה או שקוראים שוב לפונקציה עם בן ימני/שמאלי לכן מספר הקריאות לפונקציה יהיה כאשר הפונקציה עצמה היא מסיבוכיות (מספר קבוע של בדיקות). סה"כ קיבלנו שסיבוכיות הפונקציה היא .

* **Join(IAVLNode node x, AVLTree t) –** פונקציית מופע על אובייקט מסוג AVLTree. הפונקציה מקבלת צומת x מסוג IAVLNode ועץ נוסף , כך שכל המפתחות בעץ אחד גדולים מX וכל המפתחות בעץ השני קטנים מX. הפונקציה מאחדת את t וx עם העץ עליו מופעלת הפונקציה ומחזירה את סיבוכיות הפעולה.

**אלגוריתם:**

Case A: 2 העצים הינם ריקים – נגדיר את שורש העץ להיות X, נגדיר לו ילדים וירטואלים באמצעות הפונקציה (insertVirutalChild(, נעדכן את הsize להיות 1, את ההורה שלו להיות null וכן נעדכן את המינימום והמקסימום בעץ. נחזיר 1 (הפרש הראנקים של העצים + 1 ).

Case B: אחד העצים בלבד הינו ריק – נמצא min וmax חדשים בהתאם לערך X ונכניס לעץ שלא ריק את האיבר X ע"י הפעלת insert. נעדכן את השורש של העץ עליו מופעלת הפונקציה (במידה והוא זה שהיה ריק), ונעדכן את שדות min ו max שלו. לאחר מכן נחזיר את הראנק של העץ שאינו ריק + 2 (הפרש הראנקים של העצים + 1).

לאחר בדיקת מקרים אלו, נשמור תחת largerTree את העץ שבו המפתחות הגדולים מX ותחת smallerTree את השני. כמו כן , נגדיר את המינימום החדש להיות האיבר המינימלי בעץ הקטן, ואת המקסימלי להיות האיבר המקסימלי בעץ הגדול.

Case C: 2 העצים לא ריקים והם באותו הגובה - נגדיר את X להיות שורש העץ, כך שsmallerTree הוא בנו השמאלי וlargerTree הוא בנו הימני. נעדכן את הsize והrank שלו בהתאם. ונעדכן את המצביעים של min ו max. נחזיר 1 (הפרש הראנקים של העצים + 1).

Case D: העץ עם המפתחות הגדולים יותר גבוה יותר – נפעיל את הפונקציה leftJoin (מפורטת בהמשך), כך שנאחד "משמאל" את העץ עם המפתחות הקטנים וX לעץ עם המפתחות הגדולים. לאחר האיחוד, נעדכן את השורש בהתאם, ונעדכן את המינימום והמקסימום החדשים ונחזיר את הפרש הראנקים של העצים + 1.

Case E: העץ עם המפתחות הקטנים גבוה יותר – נפעיל את הפונקציה rightJoin (מפורטת בהמשך), כך שנאחד "מימין" את העץ עם המפתחות הגדולים וX לעץ עם המפתחות הקטנים. לאחר האיחוד, נעדכן את השורש בהתאם, ונעדכן את המינימום והמקסימום ונחזיר את הפרש הראנקים של העצים + 1.

**סיבוכיות:**

נשים לב כי במקרים A, B וC כלל הפעולות המתבצעות הינן פעולות בעלות זמן ריצה קבוע (כולל הפונקציה insertVirtualChild שפועלת ב (, למעט קריאה לפונקציה: insert שמתבצעת במקרה הגרוע ביותר שבו עץ אחד הינו ריק, והסיבוכיות שלה הינה (הסיבוכיות מפורטת בתיעוד הרלוונטי לפונקציה).

במקרים D ו E אנו קוראים לפונקציות leftJoin או rightJoin שמתבצעות בסיבוכיות . לכן, סה"כ הסיבוכיות של פונקציה זו הינה לכל היותר . או חסם הדוק יותר: שהסבר אליו מפורט תחת הפונקציה leftJoin.

* **leftJoin(AVLNode x, AVLTree smallerTree) –** הפונקציה הינה פונקציית מופע של מחלקת AVLTree. הפונקציה נקראת בCase D בפונקציה join, כאשר העץ עם המפתחות הגדולים הינו גבוה יותר. הפונקציה מקבלת צומת X ועץ נוסף עם המפתחות הקטנים, ומאחדת אותם לעץ הגבוה יותר שעליו הופעלה הפונקציה.

**אלגוריתם:** נלך מהשורש של העץ הגבוה יותר ונטייל שמאלה עד הפעם הראשונה שנגיע לצומת שהדרגה שלו קטנה\שווה לדרגה של השורש של העץ השני. בדרך, נעלה את שדה הsize של כל צומת שנעבור בה בגודל של העץ עם המפתחות הקטנים אותו אנו עומדים להוסיף + 1 עבור הצומת X. לאחר שנגיע לצומת זו, נשמור במשתנה prevParent את האבא של צומת זו, נכניס את X עם השמת מצביעים מתאימה ונעדכן את גודל הצומת של X. אם הrank של prevParent גדול ב2 הדרגה של העץ הנמוך יותר, אזי העץ תקין וסיימנו. אחרת, נעשה לו promote ואז או שהבעיה נפתרה או התגלגלה למעלה. נשלח את העץ לrebalanceTreeInsert עם הצומת X.

**סיבוכיות:** בWC אנו מבצעים 2 טיולים – טיול משורש העץ הגבוה עד לדרגה של שורש העץ הנמוך, וחזרה עבור הrebalancing שבו נתקן את הדרגות.. הטיול מלמעלה למטה עולה לכל היותר מס' הצעדים בין שורש העץ הגבוה לצומת שבגובה העץ הנמוך, ולכן חסום ע"י הפרשי הדרגות שלהם: . במקרה הגרוע, שבו המסלול הוא מהשורש עד עלה, איחוד זה יעלה .

* **rightJoin(AVLNode x, AVLTree smallerTree) -** הפונקציה הינה פונקציית מופע של מחלקת AVLTree. הפונקציה נקראת בCase E בפונקציה join, כאשר העץ עם המפתחות הקטנים הינו גבוה יותר. הפונקציה מקבלת צומת X ועץ נוסף עם המפתחות הגדולים, ומאחדת אותם לעץ הגבוה יותר שעליו הופעלה הפונקציה.

**אלגוריתם:** האלגוריתם של איחוד זה, זהה לחלוטין לאלגוריתם של leftJoin איך הפעם נטייל "ימינה" על הדופן הימני של העץ הגבוה יותר (עם המפתחות הקטנים). לא נפרט מקרה זה כי הינו סימטרי לאלגוריתם המתואר בleftJoin.

**סיבוכיות:** בדומה לסיבוכיות leftJoin, סיבוכיות פונקציה זו הינה בWC .

* **isExist(int k) –** פונקצית מופע על אובייקט מסוג AVLTree. הפונקציה מקבלת ערך מפתח k ומחזירה True האם קיים צומת עם מפתח K בעץ, אחרת False.

**אלגוריתם:** אם העץ ריק נחזיר False, אחרת נקרא לפונקציה isExist\_rec ונחזיר את הערך שמוחזר ממנה.

**סיבוכיות:** הפונקציה מבצעת בדיקה אחת ולאחר מכן קוראת לפונקציה isExist\_rec(AVLNode node, int k) שסיבוכיותה היא .

* **isExist\_rec(AVLNode node, int k) –** הפונקציה מקבלת מצביע לצומת ומפתח. הפונקציה תחזיר True אם קיים צומת בעלת מפתח k , False אחרת.

**אלגוריתם:** הפונקציה היא פונקציה רקורסיבית. תנאי עצירה ראשון הוא שהמפתח הנוכחי הוא בעל ערך k. אם הצומת הנוכחית בעלת מפתח שגדול מ k נקרא לפונקציה שוב עם בנה השמאלי, ואם המפתח של הצומת הנוכחית קטן מ k נקרא לפונקציה שוב עם בנה הימני. אם לא נכנס לשום תנאי אז נחזיר false. הרקורסיה בטוח תעצר כי גם אם לא נכנס לשום תנאי נגיע ל return false בסוף (במקרה שלא קיימת שום צומת עם ערך k בעץ).

**סיבוכיות:** הפונקציה עצמה היא מכיוון שמתבצע מספר קבוע של פעולות. נקרא לפונקציה לכל היותר פעמים (כגובה העץ שהוא חסום ע"י מכיוון שהוא עץ AVL) מכיוון שכל קריאה לפונקציה הרקורסיבית יורדים לצומת הימנית או השמאלית.

* **minNode() –** הפונקציה הינה פונקצית מעטפת שפועלת על אובייקט מסוג AVLTree. הפונקציה תחזיר את הצומת עם המפתח המינמלי בעץ זה.

**אלגוריתם:** הפונקציה קוראת לפונקציה minNode\_rec ומחזירה את הערך שמוחזר ממנה.

**סיבוכיות:** סיבוכיות הפונקציה minNode\_rec היא ופונקציה זו רק קוראת לה (פונקצית מעטפת) ולכן סיבוכיות פונקציה זו היא .

* **minNode\_rec(AVLNode root) –** הפונקציה מקבלת מצביע לצומת AVLNode ומחזירה את האיבר עם המפתח המינמלי בתת עץ זה.

**אלגוריתם:** הפונקציה הינה פונקציה רקורסיבית. תנאי העצירה של הפונקציה הוא כאשר נגיע לצומת וירטואלית (מתבצע ע"י בדיקה של הצומת עם הפונקציה isRealNode). כל עוד לא הגענו לבן וירטואלי, נקרא שוב לפונקציה עם בנה השמאלי של הצומת.

**סיבוכיות:** הפונקציה עצמה היא מסיבוכיות מכיוון שמתבצע מספר קבוע של בדיקות. נקרא לפונקציה לכל היותר פעמים (כגובה העץ שהוא חסום ע"י מכיוון שהוא עץ AVL) מכיוון שכל קריאה לפונקציה הרקורסיבית יורדים לבן השמאלי של הצומת הנוכחית.

* **maxNode() -** הפונקציה הינה פונקצית מעטפת שפועלת על אובייקט מסוג AVLTree. הפונקציה תחזיר את הצומת עם המפתח המקסימלי בעץ זה.

**אלגוריתם:** הפונקציה קוראת לפונקציה maxNode\_rec ומחזירה את הערך שמוחזר ממנה.

**סיבוכיות:** סיבוכיות הפונקציה minNode\_rec היא ופונקציה זו רק קוראת לה (פונקצית מעטפת) ולכן סיבוכיות פונקציה זו היא .

* **maxNode\_rec(AVLNode root) -** הפונקציה מקבלת מצביע לצומת AVLNode ומחזירה את האיבר עם המפתח המקסימלי בתת עץ זה.

**אלגוריתם:** הפונקציה הינה פונקציה רקורסיבית. תנאי העצירה של הפונקציה הוא כאשר נגיע לצומת וירטואלית (מתבצע ע"י בדיקה של הצומת עם הפונקציה isRealNode). כל עוד לא הגענו לבן וירטואלי, נקרא שוב לפונקציה עם בנה הימני של הצומת.

**סיבוכיות:** הפונקציה עצמה היא מסיבוכיות מכיוון שמתבצע מספר קבוע של בדיקות. נקרא לפונקציה לכל היותר פעמים (כגובה העץ שהוא חסום ע"י מכיוון שהוא עץ AVL) מכיוון שכל קריאה לפונקציה הרקורסיבית יורדים לבן הימני של הצומת הנוכחית.

**מחלקת AVLNode**

מחלקה זו ממשת את המנשק IAVLNode. מחלקה זו נועדה לממש צומת בעץ AVL.

שדות:

* key – שדה השומר את ערך המפתח
* value -שדה השומר מחרוזת עם הערך שאנחנו רוצים לשמור בצומת
* left – שדה השומר מצביע לבן שמאלי
* right – שדה השומר מצביע לבן ימני
* parent – שדה השומר מצביע להורה, אם הצומת היא שורש יצביע ל null
* rank – שדה השומר את דרגת/גובה העץ (בעץ AVL מתקיים ש rank = height)
* size – שדה השומר כמה איברים יש בתת העץ של צומת זה (כולל הצומת עצמה)

פונקציות: (לא כולל פונקציות get ו set שהמימוש שלהם טריוויאלי)

* **AVLNode(int key, String i)–** פונקציה זו היא בנאי המחלקה. מקבלת מפתח ומחרוזת.

**אלגוריתם:** הפונקציה תאתחל את המפתח של הצומת להיות k, את שדה value להיות i ואת שדה size להיות מאותחל ב 1. שאר השדות יאותחלו לערכים הדיפולטים שלהם.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **isRealNode()–** פונקצית מופע על אובייקט מסוג AVLNode. הפונקציה תחזיר True אם הצומת אינה צומת וירטואלית, אחרת False.

**אלגוריתם:** אם הדרגה של הצומת היא לא -1 הפונקציה תחזיר True , אחרת False.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **isVirtual()–** פונקצית מופע על אובייקט מסוג AVLNode. הפונקציה מקבלת צומת ריקה ומאתחלת את הצומת להיות צומת וירטואלית.

**אלגוריתם:** הפונקציה תאתחל את שדה rank להיות -1, שדה key להיות -1 ואת שדה size להיות 0.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **InsertVirtualChilds() -** פונקצית מופע על אובייקט מסוג AVLNode. הפונקציה תכניס 2 בנים וירטואלים לצומת שהפונקציה מופעלת עליה.

**אלגוריתם:** נבנה 2 צמתים וירטואלים חדשים ונכניס אותם כבן ימני ושמאלי של הצומת.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **getLeftEdge()-** פונקצית מופע על אובייקט מסוג AVLNode. מחזירה את הפרש הדרגות בין הצומת הנוכחית לבן השמאלית שלו.

**אלגוריתם:** מחזירה את הפרש הדרגות בין הצומת הנוכחית לבן השמאלי שלה.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **getRightEdge()-** פונקצית מופע על אובייקט מסוג AVLNode. מחזירה את הפרש הדרגות בין הצומת הנוכחית לבן הימני שלו.

**אלגוריתם:** מחזירה את הפרש הדרגות בין הצומת הנוכחית לבן הימני שלה.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **isLeaf() -** פונקצית מופע על אובייקט מסוג AVLNode. מחזירה את True אם הצומת היא עלה, אחרת False.

**אלגוריתם:** מחזירה True אם 2 הבנים של הצומת הם בנים וירטואליים, אחרת False.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **isUnary() -** פונקצית מופע על אובייקט מסוג AVLNode. מחזירה את True אם הצומת היא אונארית, אחרת False.

**אלגוריתם:** מחזירה True אם אחד מהבנים הוא צומת אמיתית והשני וירטואלי, False אחרת.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **promote() -** פונקצית מופע על אובייקט מסוג AVLNode. מגדילה את הדרגה של הצומת ב 1

**אלגוריתם:** מגדילה את שדה ה rank של הצומת שעליה מפעילים את הפונקציה ב 1.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **demote() -** פונקצית מופע על אובייקט מסוג AVLNode. מקטינה את הדרגה של הצומת ב 1

**אלגוריתם:** מקטינה את שדה ה rank של הצומת שעליה מפעילים את הפונקציה ב 1.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

* **IsValid()-** פונקצית מופע על אובייקט מסוג AVLNode. משמשת בפונקצית Insert. תחזיר True אם הצומת היא צומת 1,1 או 2-1 או 1-2. אחרת, False

**אלגוריתם:** נחשב את הפרשי הדרגות בין הצומת הנוכחית לבן השמאלי והימני שלה וכך נוכל לבדוק האם היא צומת 1-1 2-1 או 1-2 ונחזיר True אם כן, אחרת False.

**סיבוכיות:** הפונקציה מבצעת מספר קבוע של פעולות ולכן הסיבוכיות היא .

**מדידות**

**שאלה 1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| מספר סידורי | מספר פעולות | מספר פעולות האיזון הממוצע לפעולת insert | מספר פעולות האיזון הממוצע לפעולת delete | מספר פעולות האיזון המקסימלי לפעולת insert | מספר פעולות האיזון המקסימלי לפעולת delete |
| 1 | 10000 | 3.401 | 2.415 | 15 | 26 |
| 2 | 20000 | 3.411 | 2.410 | 17 | 29 |
| 3 | 30000 | 3.418 | 2.412 | 19 | 30 |
| 4 | 40000 | 3.424 | 2.411 | 19 | 31 |
| 5 | 50000 | 3.412 | 2.412 | 19 | 35 |
| 6 | 60000 | 3.422 | 2.414 | 20 | 36 |
| 7 | 70000 | 3.423 | 2.416 | 20 | 37 |
| 8 | 80000 | 3.410 | 2.416 | 21 | 39 |
| 9 | 90000 | 3.409 | 2.412 | 21 | 39 |
| 10 | 100000 | 3.421 | 2.414 | 23 | 39 |

התוצאות שציפינו לקבל בטבלה על סמך ההסבר התאורטי של עצי AVL שנלמד בכיתה הן :

**מספר פעולות איזון ממוצע -** כאשר דנים במספר פעולות איזון ממוצע, צריך לבחון את עלות הamortized של הפעולה. בכיתה למדנו שעלות amortized של רצף פעולות insertions לעץ AVL היא (כלומר, מס' פעולות קבוע ללא תלות בn). כמו כן, למדנו שעלות amortized של רצף פעולות deletions בעץ נתון לאחר שהוכנסו אליו איברים, הינה גם כן , לכן נצפה לראות מס' פעולות איזון קבוע ללא תלות במס' האיברים גם בinsertions וגם בdeletions.

**מספר פעולות איזון מקסימלי** – כאשר דנים במספר פעולות איזון מקסימלי, צריך לבחון את עלות הWorst Case של הפעולה. כפי שהראנו הWC של insert ו delete הינו (גובה העץ), לכן נצפה לראות את מספר פעולות האיזון המקסימלי גדל באופן לוגריתמי גם בdelete וגם בinsert.

התוצאות שקיבלנו בפועל תואמות את הציפיות, ואכן מס' פעולות האיזון הממוצע של insert ו delete נשאר קבוע ללא תלות בערך הנוכחי של n. כמו כן, מס' הפעולות המקסימלי שקיבלנו לכל פעולה אכן גדל באופן לוגריתמי ולא לינארי כפי שניתן לראות בטבלה.

משמעות המדידות שביצענו היא שהמימוש שלנו אכן תואם את הסיבוכיות המצופה מהפעולות הנ"ל.

**שאלה 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מספר סידורי | עלות join ממוצע עבור split אקראי | עלות join מקסימלי עבור split אקראי | עלות join ממוצע עבור split של איבר מקס בתת עץ השמאלי | עלות join מקסימלי עבור split של איבר מקס בתת עץ השמאלי |
| 1 | 2.797 | 6 | 2.769 | 16 |
| 2 | 2.642 | 9 | 2.615 | 17 |
| 3 | 2.714 | 6 | 2.571 | 18 |
| 4 | 2.525 | 9 | 2.642 | 18 |
| 5 | 2.647 | 9 | 3 | 19 |
| 6 | 2.589 | 6 | 2.562 | 19 |
| 7 | 2.785 | 9 | 2.866 | 19 |
| 8 | 2.597 | 9 | 2.933 | 19 |
| 9 | 2.583 | 7 | 3 | 20 |
| 10 | 2.636 | 9 | 2.75 | 20 |

**הסבר תוצאות הטבלה** –

התוצאות מתיישבות עם הניתוח התאורטי של סיבוכיות הזמן שראינו בכיתה, לפיה בניתוח סיבוכיות WC, העלות של פעולת Join תהיה הפרש הגבהים בין 2 העצים שמאחדים. כאשר עושים split לאיבר המקסימלי בתת העץ השמאלי (נסמן "המפתח"), כל האיברים בתת העץ השמאלי יהיו קטנים ממנו ולכן ייכנסו לעץ שמחזיק את האיברים הקטנים מהמפתח. כאשר נגיע לבן השמאלי של השורש, נצטרך לבצע join בין העץ המחזיק את המפתחות הגדולים מהמפתח (שהוא כרגע ריק) , לבין השורש ולבין כל תת העץ הימני של השורש שגובהו בשלב זה הוא עדיין . לכן כאשר נבצע את join זה, הפרש הגבהים בין 2 העצים שמאחדים יהיה מקסימלי וערכו כגובה העץ. לכן כאשר נגדיל את מספר האיברים בעץ, גם העלות של פעולת ה join היקרה ביותר (איחוד תת העץ הימני, עם השורש והעץ שמחזיק את המפתחות הגדולים) תגדל לוגריתמית בהתאם, כפי שניתן לראות בטבלה.

בניגוד לניסוי בו עשינו split לאיבר המקס' בתת העץ השמאלי שבו "אילצנו" את Join להגיע לזמן הריצה ה WC שלה, בניסוי בו עושים split לאיבר אקראי לא בהכרח נגיע ל WC של הפעולה ולכן העלות המקסימלית לא בהכרח תעלה בקצב לוגריתמי אך עדיין תהיה לכל היותר .

לגבי העלות הממוצעת, בניתוח הסיבוכיות של הפונקציה Split בחלק התיעוד, הראנו שלמרות שיש לכל היותר קריאות לפונקציה join, סך הפעולות יתכנסו לכדי סיבוכיות מכיוון שהשתמשנו באלגוריתם של join חכם (העיקרון המנחה הוא שהפרשי הגבהים בין העצים יהיו לכל להיותר קבוע מסוים לאורך כל פעולת split). כאשר נחשב את העלות הממוצעת של הפונקציה join, נחלק את העלות הכוללת של כל הקריאות לפונקצית join () ונחלק אותם במספר הפעמים שקראנו ל join ( – כגובה העץ, מכיוון שלאחר לכל קריאה ל join עולים רמה בעץ). לכן, ללא תלות ב n, נקבל עד כדי קבוע, עלות זמן ריצה ממוצעתO(1) כפי שניתן לראות בטבלה.

בונוס

נשים לב, שהעלות המקסימלית של join תתרחש במצבים בהם נבצע split על האיבר המקסימלי בתת העץ השמאלי או על האיבר המינמלי בתת העץ הימני (כפי שהסברנו בסעיף קודם). ככל שנתרחק מעלים אלו (גם מרחק אופקי באותה רמה וגם מרחק מבחינת גבהים בעץ), הפרשי הגבהים המקסימלים שיכולים להתרחש לאורך split יקטנו (מכיוון שהעץ שמכיל את האיברים הקטנים והעץ שמכיל את האיברים הגדולים לא יהיו ריקים). כאשר נגדיל את מספר האיברים בעץ, אומנם נגדיל את גובה העץ, ובהתאם את הפוטנציאל לקבל מקרה WC בעלות יותר גבוהה, אבל מצד שני נקטין את הסיכוי "להגריל" צומת שתגרום למצב WC של פעולת Join ולכן בתוחלת נקבל שהעלות של הפעולה בsplit אקראי לא בהכרח תעלה כפי שניתן לראות בטבלה. מבחינת העלות הממוצעת של פעולת join עבור split אקראי, מכיוון שעץ AVL הוא עץ מלא, בתוחלת, נניח ונבחר בה"כ צומת מתת העץ השמאלי של השורש, הצומת תקיים את זה שהיא תהיה במקום יחסית מאוזן מבחינת הדירוג שלה בתוך תת העץ, כלומר לא תהיה קטנה מרוב האיברים או גדולה מרוב האיברים בתת העץ השמאלי. לכן, כאשר נבצע את אלגוריתם ה Join החכם, ההפרש בין תת העץ שנאחד עם העץ ששומר על המפתחות הקטנים או העץ ששומר על המפתחות הגדולים יהיה לכל היותר קבוע מסוים. קבוע זה הוא העלות של פעולת join בודדת, ולכן בממוצע נקבל שללא תלות בגודל n, נקבל שהפרש זה קבוע.