תיעוד הקוד

מחלקת AVLNode

מחלקה המממשת את המנשק IAVLNode.

מכילה את השדות:

1. key – מספר שלם המייצג את מפתח הצומת.
2. value – מחרוזת המהווה את "תוכן" הצומת.
3. left – מצביע מסוג IAVLNode אל הבן השמאלי של הצומת.
4. right – מצביע מסוג IAVLNode אל הבן הימני של הצומת.
5. parent – מצביע מסוג IAVLNode אל אביו של הצומת.
6. size – מספר שלם המייצג את כמות הצמתים בתת העץ של צומת זה, כולל הצומת עצמו.
7. height – מספר שלם המייצג את גובהו של הצומת בעץ.

למחלקה זו שני בנאים:

1. **public** AVLNode(IAVLNode parent) – בונה צומת וירטואלי.
מאתחל את מפתח הצומת ואת גובהו להיות 1-.
מאתחל את ערך הצומת, את בנו השמאלי ואת בנו הימני ל-null/
מאתחל את ה-size שלו ל-0 (הרי הוא לא צומת אמיתי ולכן גם הוא עצמו לא נכלל בספירה תת העץ שהוא יוצר).
מעדכן את שדה ה-parent להצביע על צומת ההורה שמתקבל כקלט בבנאי זה.
כל זאת מתבצע בסיבוכיות של O(1) משום שיש פה רק התעסקות בקבועים ובעדכון מצביעים מספר סופי של פעמים.

2. **public** AVLNode(**int** i, String val) – בונה צומת אמיתי.
מכניס לשדה המפתח את מספר המפתח המתקבל כקלט לפונקציה זו, ומכניס לערך הצומת את המחרוזת המתקבלת כקלט לפונקציה זו.
מאתחל את גובה הצומת ל-0, ואת ה-size שלו ל-1 (הוא עצמו נמצא בתת העץ שהוא יוצר ברגע הבנייה).
מאתחל את בניו להיות צמתים וירטואליים, ואת אביו ל-null.
כל זאת מתבצע בסיבוכיות של O(1) משום שיש פה רק התעסקות בקבועים ובעדכון מצביעים מספר סופי של פעמים.

3. פונקציות get ו-set: עבור כל שדה הוגדרו פונקציות get ו-set על מנת שנוכל לעדכן אותם ולקבל את ערכם במידת הצורך. פונקציות אלו פועלת ב- O(1) משום שמדובר רק בעדכון תוכן של שדה או החזרה של ערך אליו יש לנו גישה ב-O(1) (שדה).

מחלקת AVLTree

מכילה את השדות:

1. root – מחזיק מצביע מסוג IAVLNode לשורש העץ.
2. minimum - מחזיק מצביע מסוג IAVLNode לצומת בעל המפתח המינימלי בעץ.
3. maximum - מחזיק מצביע מסוג IAVLNode לצומת בעל המפתח המקסימלי בעץ.

פונקציות המחלקה:

public AVLTree() – בנאי המחלקה. מאתחל את שורש העץ כצומת וירטואלי שאביו null.
קוראת לבנאי הצומת הוירטואלי שבמחלקה AVLNode. סיבוכיות O(1), נעשית פה רק בניה של שורש העץ באמצעות בנאי שסיבוכיותיו כפי שכתבנו קודם – O(1).

Public boolean empty() – מחזירה אמת אם העץ ריק, שקר אחרת.
קוראת לפונקציה isRealNode() אשר ממומשת ב-O(1) כפי שנפרט בהמשך. סיבוכיותה O(1) משום שמכילה רק תנאי אחד וקריאה לפונקציה בסיבוכיות זהה.

public IAVLNode searchForParent(int k, int diff) – מחזירה מצביע לאביו של צומת עם מפתח k.
תנאי מקדים – צומת עם מפתח k אכן נמצא בעץ.
סיבוכיות O(log(n)) משום שבמקרה הגרוע ביותר הצומת עם מפתח k הוא העלה העמוק ביותר בעץ ואז נצטרך לעבור על פני מסלול שאורכו כגובה העץ, כלומר log(n).
כל פעם נעשית פנייה ימינה או שמאלה בעץ ללא חזרה מעלה, ולכן לא נעבור על פני יותר מ-log(n) צמתים בעץ.

public IAVLNode searchFor(IAVLNode curr, int k)
מחזירה מצביע לצומת עם מפתח k. אם לא קיים בעץ צומת עם מפתח K, מחזירה צומת וירטואלי.
סיבוכיות O(log(n)) משום שבמקרה הגרוע ביותר הצומת המתקבל כקלט בפונקציה (curr, ממנו מתחילים את החיפוש אחר הצומת המבוקש) הוא שורש העץ, אך הצומת המבוקש כלל לא מופיע בעץ ולכן נצטרך לעבור על פני O(log(n)) צמתים במקרה הגרוע בתהליך החיפוש.
בנוסף, קוראת לפונקציה isRealNode() אשר פועלת בסיבוכיות O(1) לכן קריאה זו לא משפיעה על סיבוכיות פונקציה זו.

public String search(int k)

מחזירה את הערך של צומת עם מפתח k. אם לא קיים בעץ צומת עם מפתח זה, מחזירה null.
קוראת לפונקציה searchFor אשר פרטנו קודם כי פועלת בסיבוכיות של O(log(n)) ולכן סיבוכיות פונקציה זו היא גם O(log(n)).

public int BFCalc(IAVLNode node)

מחזירה את ה-balance factor של הצומת.
לצורך החישוב מתבצעת גישה לשדות הגבהים של בניו של הצומת, וגישה זו מתבצעת ב-O(1) (סה"כ מדובר במצביעים ובהחזרת ערך שדה מפעולת get שפרטנו קודם כי פועלת בסיבוכיות של O(1). פונקציה זו פועלת בסיבוכיות של O(1).

private int sizeCalc(IAVLNode node)

מחשבת ומחזירה את גודל תת העץ שנוצר מהצומת המתקבל כקלט לפונקציה. מחזירה 0 עבור צומת וירטואלי.
פועלת בסיבוכיות של O(1) משום שרק נעשית גישה באמצעות פעולות get לשדות ה-size של בניו של הצומת המתקבל כקלט. פרטנו קודם כי פעולת get רצה בסיבוכיות O(1), וגם הקריאה לפונקציה isRealNode לא גורעת מסיבוכיות הפונקציה משום שגם היא רצה בסיבוכיות של O(1).

private void SizesUpdate(IAVLNode parent)

מבצעת ריצה על העץ, החל מהצומת המתקבל כקלט ועד השורש, ומעדכנת את הגדלים של הצמתים בהתאם למצב החדש (לאחר הכנסה, לאחר מיזוג וכו').
סיבוכיות : $O(logn)$.

private int HeightCalc(IAVLNode node)

מחשבת ומחזירה את גובהו של הצומת המתקבל כקלט. עבור צומת וירטואלי מחזירה 1-.
קוראת לפונקציות isRealNode ו-Math.max שרצות בסיבוכיות O(1) ולפונקציות get של בניו של הצומת ושל הגבהים שלהם. כפי שפרטנו קודם, גם פונקציות אלה פועלות בסיבוכיות של O(1), לכן סה"כ סיבוכיות פונקציה זו היא O(1).

private int FixAVLtree(IAVLNode node)

מחליטה איזה תיקון נדרש לעץ על מנת לשמור על משתמר ה-AVL, ושולחת את העץ לפונקציות הרלוונטיות לתיקון.
מחזירה 1 עבור פעולת איזון אחת בגלגולי LL או RR, ו-2 עבור שתי פעולות איזון בגלגולי RL או LR. אם היא מקבלת צומת וירטואלי, מחזירה 0 משום שאין צורך לתקן עץ עבור צומת שאינו אמיתי. קוראת לפונקציות isRealNode, BFCalc, RRRotation, RLRotation, LLRotation, LRRotation אשר כולן פועלות ב-O(1) כפי שכבר פירטנו או כפי שנפרט בהמשך.
מעבר לקריאות אלו מתבצעת גישה לשדות בניו של הצומת המתקבל כקלט לפונקציה, אשר נעשית ב-O(1), לכן סיבוכיותה הכוללת של פונקציה זו היא O(1).

private void LLRotation(IAVLNode node)

מבצעת גלגול LL. קוראת לפונקציות set ו-get הפועלות בסיבוכיות O(1), ועל כן סיבוכיותה הכוללת היא O(1) (יש פה רק עדכוני מצביעים גבהים וגדלים אשר כולם נעשים ב-O(1)).

private void RRRotation(IAVLNode node)

מבצעת גלגול RR. קוראת לפונקציות set ו-get הפועלות בסיבוכיות O(1), ועל כן סיבוכיותה הכוללת היא O(1) (יש פה רק עדכוני מצביעים גבהים וגדלים אשר כולם נעשים ב-O(1)).

private void RLRotation(IAVLNode node)

מבצעת גלגול RL על ידי קריאה לפונקציות LLRotation ו-RRRotation הפועלות ב-O(1) כפי שהוסבר קודם לכן. מעבר לכך מתבצעת קריאה לפונקציית get שגם היא פועלת בסיבוכיות של O(1), ולכן בסה"כ סיבוכיותה הכוללת של הפונקציה היא O(1).

private void LRRotation(IAVLNode node)

מבצעת גלגול LR על ידי קריאה לפונקציות LLRotation ו-RRRotation הפועלות ב-O(1) כפי שהוסבר קודם לכן. מעבר לכך מתבצעת קריאה לפונקציית get שגם היא פועלת בסיבוכיות של O(1), ולכן בסה"כ סיבוכיותה הכוללת של הפונקציה היא O(1).

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

מכניסה צומת עם מפתח k וערך i לעץ. מחזירה את מספר פעולות האיזון שהתבצעו בעץ בעקבות ההכנסה.
מעדכנת את שדות המינימום והמקסימום בעץ במידת הצורך (אם הצומת שנוסף כעת אמור להיות המינימלי/המקסימלי).
קוראת באופן בלתי תלוי לפונקציות searchFor, searchForParent ו-HieghtsUpdating אשר כולן פועלות בסיבוכיות O(log(n)) כפי שפורט קודם ויפורט בהמשך. בנוסף קוראת לפונקציות empty ו-get ו-Set למיניהן, אשר כולן רצות בסיבוכיות O(1). מעבר לכך לא מתבצעת שום קריאה רקורסיבית או לולאה, ולכן סיבוכיות הפונקציה הכוללת היא O(log(n)).

private int HieghtsUpdating(IAVLNode node)

מעדכנת את הגובה ואת הגודל של כל צומת בו נגענו במהלך תיקון העץ.
מחזירה את מספר פעולות האיזון שהתבצעו בעץ בעקבות ההכנסה.
קוראת לפונקציות HeightCalc, get, set ו-BFCalc אשר פועלות בסיבוכיות של O(1).
הלולאה רצה במקרה הגרוע log(n) פעמים, אם השינוי שביצענו הוא בצומת העמוק ביותר בעץ.
במקרה הגרוע בכל כניסה ללולאה נמצא צומת שלא מקיים את משתמר ה-AVL ולכן נצטרך לקרוא לפונקציה FixAVLtree הפועלת בסיבוכיות של O(1) כפי שפורט קודם.
על כן סה"כ סיבוכיות פונקציה זו היא O(log(n)).

public int delete(**int** k)

מוחקת צומת מהעץ וקוראת לפונקציה שמסדרת את העץ לאחר המחיקה לפי הצורך.
מחזירה את מספר פעולות האיזון שהתבצעו בתהליך המחיקה. אם הצומת אותו רצינו למחוק לא נמצא בעץ, מחזירה 1-.
קוראת באופן בלתי תלוי לפונקציות searchFor, findPredecessor, findSuccessor, HieghtsUpdating שרצות בסיבוכיות O(log(n)) כפי שפורט קודם ויפורט בהמשך.
בנוסף קוראת לפונקציות isRealNode, set ו- get שרצות בסיבוכיות O(1).
מעבר לכך אין קריאות רקורסיביות או לולאות בפונקציה, ולכן סה"כ סיבוכיות פונקציה זו היא O(log(n)).

private IAVLNode findSuccessor(IAVLNode node)

מחזירה את הצומת העוקב לצומת המתקבל כקלט לפונקציה זו. עבור הצומת המקסימלי בעץ, מחזירה null.
קוראת לפונקציות get הממומשות ב-O(1).
לפי תנאי קוראת לפונקציה minPointer הפועלת בסיבוכיות O(log(n)) כפי שיפורט בהמשך.
אחרת, רצה לולאה במקרה הגרוע ב-log(n) פעמים, ולכן בכל מקרה סיבוכיות פונקציה זו היא O(log(n)).

private IAVLNode minPointer(IAVLNode node)

מחזירה את הצומת המינימלי בתת העץ של הצומת המתקבל כקלט לפונקציה זו.
קוראת לפונקציה isRealNode ו-get (הפועלות בסיבוכיות O(1) כפי שפורט קודם) במקרה הגרוע log(n) פעמים כחלק מלולאת while הממומשת בפונקציה. במקרה הגרוע הצומת המתקבל כקלט לפונקציה זו הוא שורש העץ ולכן נעבור על פני מסלול באורך גובה העץ (log(n)) עד למציאת הצומת המינימלי, ולכן סה"כ סיבוכיות פונקציה זו היא O(log(n)).

private IAVLNode findPredecessor(IAVLNode node)

מחזירה את הצומת הקודם לצומת המתקבל כקלט לפונקציה זו. עבור הצומת המינימלי בעץ, מחזירה null.
קוראת לפונקציות get ו-isRealNode הממומשות ב-O(1) כפי שפורט קודם.
לפי תנאי קוראת לפונקציה maxPointer הפועלת בסיבוכיות O(log(n)) כפי שיפורט בהמשך.
אחרת, רצה לולאה במקרה הגרוע ב-log(n) פעמים, ולכן בכל מקרה סיבוכיות פונקציה זו היא O(log(n)).

private IAVLNode maxPointer(IAVLNode node)

מחזירה את הצומת המקסימלי בתת העץ של הצומת המתקבל כקלט לפונקציה זו.
קוראת לפונקציות isRealNode ו-get (הפועלות בסיבוכיות O(1) כפי שפורט קודם) במקרה הגרוע log(n) פעמים כחלק מלולאת while הממומשת בפונקציה. במקרה הגרוע הצומת המתקבל כקלט לפונקציה זו הוא שורש העץ ולכן נעבור על פני מסלול באורך גובה העץ (log(n)) עד למציאת הצומת המקסימלי, ולכן סה"כ סיבוכיות פונקציה זו היא O(log(n)).

public String min()

מחזירה את ערכו של הצומת בעל המפתח המינימלי בעץ. עבור עץ ריק מחזירה null.
קוראת לפונקציות empty ו-getValue הממומשת בסיבוכיות של O(1) כפי שפורט קודם, ובזכות שמירת המצביע לצומת המינימלי בעץ בשדה minimum, מחזירה ישירות את ערכו של הצומת המינימלי, ללא חיפושו בעץ. על כן סה"כ סיבוכיות פונקציה זו היא O(1).

public String max()

מחזירה את ערכו של הצומת בעל המפתח המקסימלי בעץ. עבור עץ ריק מחזירה null.
קוראת לפונקציות empty ו-getValue הממומשת בסיבוכיות של O(1), ובזכות שמירת המצביע לצומת המקסימלי בעץ בשדה maximum, מחזירה ישירות את ערכו של הצומת המקסימלי, ללא חיפושו בעץ. על כן סה"כ סיבוכיות פונקציה זו היא O(1).

public int size()

מחזירה את מספר הצמתים בעץ. קוראת לפונקציה get הממומשת ב-O(1) כפי שכבר צוין, ולכן סה"כ סיבוכיות פונקציה זו היא O(1).

public IAVLNode getRoot()

מחזירה את שורש העץ. פועלת בסיבוכיות של O(1) משום רק מתבצעת גישה לשדה השורש של העץ, וזאת ב-O(1) משום ששורש העץ שמור כשדה במחלקה.

public int[] keysToArray()

הפונקציה מחזירה מערך ממוין המכיל את כל המפתחות בעץ (מהקטן לגדול), או מערך ריק אם העץ ריק.
הפונקציה מהווה פונקציית מעטפת עבור פונקציה רקורסיבית המתוארת בהמשך
nodesToArrayRec – בסיבוכיות של $O(n)$.
בנוסף, הפונקציה רצה על מערך של צמתי העץ שנוצר בפונקציה הרקורסיבית, ומאתחל מערך של מפתחות בהתאמה – זמן לינארי (ריצה על מערך בגודל n, ופעולה של השמה בזמן קבוע).
סה"כ: $O(n) + O(n) = O(n)$.

public String[] infoToArray()

הפונקציה מחזירה מערך מחרוזות המכיל את כל המחרוזות בעץ, ממוינות על פי סדר המפתחות.
כלומר הערך ה j במערך הוא המחרוזת המתאימה למפתח שיופיע במיקום ה j במערך הפלט של הפונקציה keysToArray(). גם הפונקציה הזאת מחזירה מערך ריק אם העץ ריק.
הסיבוכיות ושיטת העבודה של הפונקציה, באופן דומה לחלוטין לפונקציית keysToArray – O(n).

private int nodesToArrayRec(IAVLNode node, IAVLNode[] arr, int index)

פונקציית עזר רקורסיבית, שיוצרת מערך מסוג AVLNode בגודל n של כלל הצמתים בעץ – מהמינימלי ועד המקסימלי.
הפונקציה רצה מהשורש ועוברת על כל צומת בעץ – לכן סיבוכיות הפעולה היא כמספר הצמתים בעץ $O(n)$.

public AVLTree[] split(int x)

הפונקציה מקבלת מפתח x שנמצא בעץ. הפונקציה את העץ ל-2 עצי AVL כאשר המפתחות של האחד גדולים מ-x ושל השני קטנים מ-x.
הפונקציה מבצעת חיפוש של הצומת עם מפתח x ומייצרת מצביע אליו – חיפוש דרך פונקציית searchFor בסיבוכיות של O(log(n)).
הפונקציה קוראת לפונקציית עזר splitLoop שמבצעת את הפיצול עצמו – כפי שיתואר בהמשך, בסיבוכיות של O(log(n)).
לכן, בסה"כ כפי שראינו בהרצאה O(log(n)).

private AVLTree[] splitLoop(IAVLNode nodeX, AVLTree [] arr)

הפונקציה רצה על הצמתים במסלול מהצומת שמכילה את המפתח x ומעלה עד השורש – כלומר מספר האיטרציות בלולאת ה- while בפונקציה הוא כל היותר כגובה העץ.
עבודה בכל איטרציה:
בכל איטרציה, מתבצעות פעולות השמה, בדיקה והגדרת מצביעים שכולן ב- O(1).
בנוסף, מתבצע join בין 2 עצים – שראינו שסיבוכיות של join הוא O(log(n)).
לכאורה, הסיבוכיות המצרפית היא $O(log^2 n)$.
עם זאת, ראינו בהרצאה הוכחה, שחסם הדוק יותר על הסיבוכיות של split הוא O(log(n)).
נראה לכך הוכחה גם בתוצאות הניסוי.
לכן נסכם בכך שהפונקציה פועלת בסיבוכיות הנדרשת O(log(n)).

public int join(IAVLNode x, AVLTree t)

פונקציית מעטפת, עבור חלוקה למקרים שבהן מטפלות פונקציות עזר.
הפונקציה join עצמה מכילה פעולות בדיקה והשמה שמתבצעות בזמן קבוע של O(1), ולפי תוצאת הבדיקות מפנה לפונקציית join נקודתית באופן בלתי תלוי אחת בשנייה:

- joinWithRoot - O(1) כפי שיפורט בהמשך.
- joinFirstCase - keys() < keys(x,t) – O(log(n)) כפי שיפורט בהמשך.
- joinSecondCase - keys() > keys(x,t) – O(log(n)) כפי שיפורט בהמשך.

לכן סה"כ סיבוכיות פונקציה זו במקרה הגרוע היא O(logn(n)).

private static void joinWithRoot(AVLTree t1, IAVLNode x, AVLTree t2)

פונקציה שמטפלת במקרה בו הפרש הגבהים בין 2 העצים בערך מוחלט הוא לכל היותר 1 – כלומר העצים "מוכנים" למיזוג (בניגוד למקרים האחרים, בהם יש "לרדת" בעץ עד לנקודה של יחסית שוויון גבהים).
הפונקציה מבחינה בין יחסי הגדלים של העצים ביחס ל-x ומטפלת בכל אפשרות בנפרד.
מתבצעות פעולות והשמות ב- O(1) ולכן זה המקרה הפשוט בו מיזוג עצים נעשה בזמן קבוע.
על כן סה"כ הסיבוכיות היא O(1).

```
private int joinFirstCase(AVLTree t1, IAVLNode x, AVLTree t2)
```

לולאת while של הפונקציה רצה על אחד העצים (בהתאם למקרה – הגבוה ביותר), עד לצומת שהגובה שלה קטן/שווה לגובה של העץ השני. בלי הגבלת הכלליות, אם t2 גבוה מt1 אז הסיבוכיות של הלולאה תהיה O(rank(t2)- rank(t1)+1) כפי שראינו בכיתה – הפרש הגבהים בין העצים.
בהמשך הפונקציה, מתבצעות השמות (הגדרות חדשות של מצביעים) על מנת לסכם את המיזוג בין שני העצים, יחד עם צומת x.
כעת, יכול להיות שקיבלנו עץ לא חוקי מבחינת חוקי AVL, לכן נרצה לתקן. נשים לב, שיכולים להיווצר מצבים בהם לאחר תיקון נקודתי, נצטרך לגלגל את הבעיה למעלה ולבדוק האם צריך תיקון נוסף (כלומר, לא בהכרח התיקון מסתיים לאחר פעולת רוטציה – כמו ב-insert).

נשתמש בפונקציה HieghtsUpdating שכפי שצוין קודם פועלת ב – O(logn) – על מנת לתקן את העץ במקרה בו העץ אינו AVL חוקי. פונקציה זו כאמור, גם מתקנת את הגבהים של העץ החדש.
עדכון גדלי הצומת עם פונקציית SizedUpdate באופן דומה בסיבוכיות O(log(n)).

בסה"כ : במקרה הגרוע בו נאלצנו לתקן את העץ החדש הסיבוכיות היא O(log(n)).

```
private int joinSecnodCase(AVLTree t1, IAVLNode x, AVLTree t2)
```

ניתוח זהה לזה של הפונקציה joinFirstCase.

```
private void setRoot(IAVLNode x)
```

פונקציה שמקבלת צומת, ומגדירה אותה להיות הצומת של העץ עליו הופעלה הפונקציה - O(1).