מבנה נתונים – תרגיל בית מעשי 1:

תיעוד:

public class RBTree	
הסבר	מימוש עץ אדום-שחור עם מפתחות אי-שליליים ייחודיים וערכים מתאימים
שדות	`private RBNode nil` צומת עלה NIL
	`private RBNode root` שורש העץ (באם העץ ריק – מצביע על NIL)
	`private int size` מספר האיברים בעץ
	`private RBNode minNode` מצביע על הצומת בעל האינדקס המינימלי בעץ (באם העץ ריק – מצביע על NIL)
	`private RBNode maxNode` מצביע על הצומת בעל האינדקס המקסימלי בעץ (באם העץ ריק – מצביע על NIL)
פונקציות	`public RBTree()` תיאור: בנאי המחלקה – מייצר עצם חדש מסוג עץ אדום-שחור. העץ הנוצר הינו ריק. אופן הפעולה: המתודה מייצרת את הצומת NIL שישמש כייצוג של כלל עלי העץ (צומת בצבע שחור, בעל מפתח 1- ערך null, הורה וילדים null). מפנה את המצביעים של השורש, הצומת המקסימלי (maxNode) והצומת המינימלי (minNode) אל NIL. מאפסת את גודל העץ. סיבוכיות: כל הפעולות בעלות זמן ריצה קבוע ולכן - $O(1)$
	`public boolean empty()` תיאור: המתודה מחזירה TRUE אם ורק אם העץ ריק. אופן הפעולה: בודקת באם השדה size הינו אפס. סיבוכיות: פעולת השוואה בודדת - $O(1)$
	`public String search(int k)` תיאור: המתודה מחפשת איבר בעל המפתח k. אם קיים איבר כזה, היא מחזירה את הערך השמור עבורו, אחרת היא מחזירה null. אופן הפעולה: המתודה קוראת לפונקציית העזר findRBNode על מנת לקבל את הצומת בעל המפתח k, ואז מחזירה את ערכו (אם לא קיים, מחזירה את ערך הצומת NIL, כלומר null). סיבוכיות: עיון בערך של צומת הינו פעולה קבועה, ולכן זמן הריצה כשל findRBNode – $O(\log n)$
	`private RBNode findRBNode(int k)` תיאור: הפונקציה מחפשת צומת בעל מפתח k. אם קיים צומת שכזה היא מחזירה אותו, אחרת מחזירה את הצומת NIL. אופן הפעולה: המתודה מתחילה מהשורש ומשווה את המפתח של כל צומת אליו היא ניגשת למפתח k. באם המפתח הנוכחי שווה ל-k, מחזירה את הצומת. באם קטן מ-k, המתודה תעבור לילדו השמאלי של הצומת. אחרת, תעבור לילדו הימני. אם מגיעה לעלה (הצומת NIL), מחזירה אותו. הפונקציה ממומשת בצורה איטרטיבית. סיבוכיות: המתודה יורדת רמה בעץ בכל שלב, ולכן סיבוכיות זמן הריצה הינה $O(\log n)$
	`public int insert(int k, String v)` תיאור: הכנסת איבר בעל ערך v ומפתח k לעץ, אם הוא לא קיים. הפונקציה מחזירה את מספר החלפות הצבע (מאדום לשחור ולהיפך) שנדרשו בסה"כ בשלב תיקון העץ על מנת להשלים את הפעולה. אם קיים איבר בעל מפתח k בעץ, הפונקציה מחזירה (1-) ולא מתבצעת הכנסה. אופן הפעולה: המתודה פועלת באופן זהה לזו שתוארה בכיתה (מציאת מקום ההכנסה, הכנסה במקום עלה כצומת אדום, ותיקון חוקי העץ באמצעות insertFixup). בנוסף, מבצעת בדיקה האם האיבר שהוכנס גדול מהמקסימום או קטן מהמינימום (ע"י השוואה למפתח של המצביעים maxNode ו- minNode), ואם אכן כך, מעדכנת את המצביע הרלוונטי. כמו כן, במידה ולא היה קיים איבר בעץ עם מפתח זהה, מעדכנת את השדה size. לבסוף, מחזירה את מספר חילופי הצבעים - הערך שמתקבל מהמתודה insertFixup לה היא קוראת בסוף פעולתה (אלא אם היה קיים איבר בעץ בעל מפתח זהה, ואז מחזירה 1-). סיבוכיות: השינויים למתודה שנלמדה בכיתה כוללים פעולות הרצות בזמן קבוע בלבד (עדכון מצביע המקסימום/מינימום ועדכון שדה ה- size), ולכן הסיבוכיות נשארת $O(\log n)$ כפי שנלמד בכיתה.

```
private int insertFixup(RBNode node)
```

תיאור: המתודה מתקנת בעיות אפשריות שנוצרו עקב הכנסת צומת אדום חדש לעץ ע"י רוטציות ושינוי צבעים ומחזירה את מספר שינויי הצבעים שבוצעו.

אופן הפעולה: המתודה פועלת באופן זהה לזו שתוארה בכיתה, מלבד העובדה שהיא סופרת כל חילוף צבע שהיא מבצעת, ולבסוף מחזירה את סה"כ שינויי הצבע שבוצעו.

סיבוכיות: השינויים למתודה שנלמדה בכיתה כוללים פעולות הרצות בזמן קבוע בלבד (ספירת שינויי הצבע), ולכן הסיבוכיות הינה עדיין $O(\log n)$, כפי שנלמד בכיתה.

```
public int delete(int k)
```

תיאור: מחיקת איבר בעל המפתח k בעץ, אם הוא קיים. הפונקציה מחזירה את מספר החלפות הצבע שנדרשו בסה"כ בשלב תיקון העץ על מנת להשלים את הפעולה. אם לא קיים איבר בעל המפתח k בעץ, הפונקציה מחזירה -1.

אופן הפעולה: המתודה פועלת כמו המתודה אשר הוצגה בכיתה (מציאת הצומת באמצעות findRBNode, מחיקתו לפי המקרים שתוארו בכיתה, ותיקון חוקי העץ באמצעות deleteFixup). בנוסף, בודקת האם האיבר שמוסר מן העץ הינו המקסימום או המינימום שלו (ע"י השוואה למצביעים maxNode ו – minNode): אם האיבר המקסימלי מוסר, המתודה תמצא את האיבר המקסימלי החדש ע"י קריאה למתודה treeMaximum לאחר המחיקה ותיקון העץ. אם האיבר שנמחק היה המינימום, המתודה תמצא את המינימום החדש ע"י קריאה למתודה successor. במידה ואכן היה קיים איבר בעל מפתח k, מעדכנת את השדה size, אחרת מחזירה (-1). לבסוף – מחזירה את מספר חילופי הצבע שנעשו.

סיבוכיות: מציאת המינימום או המקסימום החדשים במקרה הצורך לוקח $O(\log n)$, עדכון שדה ה-size היא פעולה בזמן קבוע, ולכן סיבוכיות זמן הריצה נשארת $O(\log n)$ כפי שנלמד בכיתה.

```
private int deleteFixup(RBNode node)
```

תיאור: המתודה מתקנת הפרות של חוקי העץ אדום-שחור שנוצרו בעץ כתוצאה ממחיקת צומת ממנו, ומחזירה את מספר חילופי הצבעים שהיא נאלצה לבצע על מנת להשלים את משימתה.

אופן הפעולה: המתודה פועלת באופן זהה לזו שתוארה בשיעור, מלבד העובדה שהיא סופרת כל חילוף צבע שהיא מבצעת, ולבסוף מחזירה את סה"כ שינויי הצבע שבוצעו.

השינויים למתודה שנלמדה בכיתה כוללים פעולות הרצות בזמן קבוע בלבד (ספירת שינויי הצבע), ולכן הסיבוכיות הינה עדיין $O(\log n)$, כפי שנלמד בכיתה.

```
private void rotateLeft(RBNode x)
```

תיאור: מבצעת סיבוב שמאלה של הצומת x כפי שתואר בשיעור.

אופן הפעולה: הופכת את הבן הימני של x להורה שלו, ואת הבן השמאלי של אותו הורה לבנו הימני של x.

סיבוכיות: מספר פעולות בעלות זמן ריצה קבוע – $O(1)$

```
private void rotateRight(RBNode x)
```

תיאור: מבצעת סיבוב ימינה של הצומת x כפי שתואר בשיעור.

אופן הפעולה: הופכת את הבן השמאלי של x להורה שלו, ואת הבן הימני של אותו הורה לבנו השמאלי של x.

סיבוכיות: מספר פעולות בעלות זמן ריצה קבוע – $O(1)$

```
private RBNode successor(RBNode node)
```

תיאור: המתודה מוצאת את האיבר הבא בגודלו בעץ, באם הוא קיים. אחרת, המתודה מחזירה null.

אופן הפעולה: כפי שתואר בשיעור: אם קיים בן ימני – מחזירה את המינימום של תת-העץ הימני (ע"י קריאה לפונקציה subtreeMinNode), אחרת עולה קומות בעץ עד פנייה ראשונה ימינה.

סיבוכיות: כפי שתואר בשיעור: מציאת מינימום בתת העץ הימני – כסיבוכיות subtreeMinNode – שהיא $O(\log n)$, או טיפוס במעלה העץ עד פנייה ימינה – במקרה הגרוע גם $O(\log n)$. לכן סה"כ $O(\log n)$.

```
private void transplant(RBNode x, RBNode y)
```

תיאור: המתודה מחליפה את הצומת x (ביחד עם תת העץ שהוא שורשו) בצומת y (ביחד עם תת העץ שהוא שורשו).

אופן הפעולה: כפי שתואר בשיעור: החלפה פשוטה של הבן x עם הבן y, ועדכון שדה ה-parent של y.

סיבוכיות: מספר פעולות בעלות זמן ריצה קבוע – $O(1)$

```
public String min()
```

תיאור: מחזירה את ערכו של האיבר בעץ בעל המפתח המינימלי, או null אם העץ ריק.

אופן הפעולה: המתודה מחזירה את הערך של שדה המינימום minNode בעץ.

סיבוכיות: $O(1)$

```
private RBNode subtreeMinNode(RBNode node)
```

תיאור: המתודה מחזירה את הצומת בעל המפתח הנמוך ביותר בעץ ששורשו הוא node.

אופן הפעולה: באם שורש העץ node הוא עלה NIL, או שאין לו בן שמאלי, תחזיר את node עצמו. אחרת, המתודה יורדת בעץ לכיוון שמאל (באופן רקורסיבי) עד לקבלת צומת אשר אין לו בן שמאלי ומחזירה צומת זה.

סיבוכיות: מאחר והמתודה יורדת רמה בעץ בכל קריאה של הפונקציה, סיבוכיות זמן הריצה הינה כגובה תת-העץ - $O(\log k)$, כאשר k מספר הצמתים בתת-העץ ששורשו node. עבור חסם עם גודל העץ, נקבל $O(\log n)$.

```
public String max()
```

תיאור: מחזירה את ערכו של האיבר בעץ בעל המפתח המקסימלי, או null אם העץ ריק.

אופן הפעולה: המתודה מחזירה את הערך של שדה המקסימום maxNode בעץ.

סיבוכיות: $O(1)$

```
private RBNode treeMaximum()
```

תיאור: המתודה מחזירה את הצומת בעץ בעל המפתח הגבוה ביותר.

אופן הפעולה: קוראת למתודה subtreeMaxNode על שורש העץ, ומחזירה את הערך המוחזר ממנה.

סיבוכיות: זהה לזו של subtreeMaxNode עבור העץ כולו – $O(\log n)$.

```
private RBNode subtreeMaxNode(RBNode node)
```

תיאור: המתודה מחזירה את הצומת בעל המפתח הגבוה ביותר בעץ ששורשו הוא node.

אופן הפעולה: באם שורש העץ node הוא עלה NIL, או שאין לו בן ימני, תחזיר את node עצמו. אחרת, המתודה יורדת בעץ לכיוון ימין (באופן רקורסיבי) עד לקבלת צומת אשר אין לו בן ימני ומחזירה צומת זה.

סיבוכיות: מאחר והמתודה יורדת רמה בעץ בכל קריאה של הפונקציה, סיבוכיות זמן הריצה הינה כגובה תת-העץ - $O(\log k)$, כאשר k מספר הצמתים בתת-העץ ששורשו node. עבור חסם עם גודל העץ המקורי כולו, נקבל $O(\log n)$.

```
public int size()
```

תיאור: הפונקציה מחזירה את מספר האיברים בעץ.

אופן הפעולה: מחזירה את השדה size.

סיבוכיות: החזרת שדה - $O(1)$

```
public int[] keysToArray()
```

תיאור: הפונקציה מחזירה מערך ממוין המכיל את כל המפתחות בעץ, או מערך ריק אם העץ ריק.

אופן הפעולה: המתודה קוראת למתודה createInorderNodesArray על מנת ליצור מערך ממוין (עפ"י מפתחות) של הצמתים בעץ, ובעזרתו יוצרת מערך המכיל רק את המפתחות באופן ממוין.

סיבוכיות: המתודה createInorderNodesArray בעלת סיבוכיות זמן ריצה $O(n)$, יצירת המערך החדש ע"י מעבר על כל צמתי העץ שבמערך המחוזר - $O(n)$, ולכן הסיבוכיות הכוללת הינה $O(n)$ גם כן.

```
public String[] valuesToArray()
```

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

אופן הפעולה: המתודה קוראת למתודה createInorderNodesArray על מנת ליצור מערך ממוין (עפ"י מפתחות) של הצמתים בעץ, ובעזרתו יוצרת מערך המכיל רק את הערכים באופן ממוין.

סיבוכיות: המתודה createInorderNodesArray בעלת סיבוכיות זמן ריצה $O(n)$, יצירת המערך החדש ע"י מעבר על כל צמתי העץ שבמערך המחוזר - $O(n)$, ולכן הסיבוכיות הכוללת הינה $O(n)$ גם כן.

```
private RBNode[] createInorderNodesArray()
```

תיאור: המתודה מייצרת מערך ממוין (לפי מפתחות) של הצמתים בעץ.

אופן הפעולה: המתודה יוצרת מערך ריק בגודל העץ ושולחת אותו למתודת העזר inorderArrayHelper ביחד עם שורש העץ והמספר 0, אשר ממלאת את המערך כנדרש. לבסוף מחזירה את המערך.

סיבוכיות: זהה לזו של inorderArrayHelper עבור העץ כולו – $O(n)$.

```
private int inorderArrayHelper (RBNode[] nodesArray, RBNode currentNode,
int index)
```

תיאור: המתודה ממלאת את המערך nodesArray, מהאינדקס index, בצמתי תת-העץ ששורשו currentNode, באופן ממוין לפי מפתחות. הפונקציה מחזירה את האינדקס הבא שאינו מלא.

אופן הפעולה: בדומה לסריקת in-order שלמדנו בכיתה: נמלא באופן רקורסיבי את המערך ע"י קריאה לפונקציה עם הבן השמאלי של currentNode (עם האינדקס index, והמערך nodesArray), מילוי המערך nodesArray, באינדקס המוחזר מהקריאה, עם הצומת הנוכחי currentNode, וקריאה נוספת לפונקציה עם הבן הימני של currentNode (ואינדקס הגדול ב-1, והמערך nodesArray).

סיבוכיות: המתודה עוברת באופן רקורסיבי על כל הצמתים בתת-העץ ששורשו currentNode ומבצעת פעולת השמה פשוטה לכל צומת, ולכן זמן הריצה הוא כמספר הצמתים בתת-העץ ששורשו currentNode. באם נסמן את גודל תת-העץ ב- k, נקבל סיבוכיות $O(k)$. עבור חסם עם גודל העץ המקורי כולו, נקבל $O(n)$.

public class RBNode	
הסבר	מימוש צומת בודד בעץ אדום-שחור
משתנים	`private RBNode parent` מצביע לאבא של הצומת
	`private RBNode left` מצביע לבן השמאלי של הצומת
	`private RBNode right` מצביע לבן הימני של הצומת
	`private String color` צבע הצומת (מכיל "BLACK" או "RED")
	`private final int key` מפתח הצומת
	`private final String value` ערך הצומת
פונקציות	`public RBNode(String color, int key, String value)` <u>תיאור</u>: הבנאי הראשון של מחלקת הצמתים <u>אופן הפעולה</u>: המתודה יוצרת צומת חדש בעל צבע color, מפתח key, ערך value והורה, ילד שמאלי וילד ימני null (ע"י קריאה לבנאי השני של המחלקה). <u>סיבוכיות</u>: $O(1)$
	`public RBNode(String color, int key, String value, RBNode parent, RBNode leftChild, RBNode rightChild)` <u>תיאור</u>: הבנאי השני של מחלקת הצמתים <u>אופן הפעולה</u>: המתודה יוצרת צומת חדש בעל צבע color, מפתח, ערך value, הורה parent, ילד שמאלי leftChild וילד ימני rightChild <u>סיבוכיות</u>: $O(1)$

מדידות:

מספר סידורי	מספר פעולות	מספר החלפות צבע ממוצע לפעולת insert	מספר החלפות צבע ממוצע לפעולת delete
1	10,000	2.3089	2.4501
2	20,000	2.3206	2.4627
3	30,000	2.3130	2.4581
4	40,000	2.3115	2.4655
5	50,000	2.3180	2.4662
6	60,000	2.3233	2.4677
7	70,000	2.3188	2.4672
8	80,000	2.3075	2.4639
9	90,000	2.3151	2.4650
10	100,000	2.3163	2.4651

פעולת ה-insert מכניסה צומת חדש אדום במקום עלה. על כן – יש לבצע תיקון לעץ במידה והפרנו את הכלל האדום (או את כלל השורש – שצריך תמיד להיות שחור). ראינו בכיתה 3 מקרים אפשריים: מקרה ראשון שיכול לקרות כמה פעמים ו"לטפס" עד שורש העץ, ומקרה שני ושלישי שבאם קורים – מסיימים את התיקון. במקרה הראשון ישנם 3 שינויי צבע ובמקרה השלישי 2 (במקרה השני אין שינויי צבע). משום שרק המקרה הראשון יכול לקרות מספר פעמים ברצף בכל הכנסה, היינו מצפים כי מספר החלפות הצבע הממוצע יטה לכיוון ה-3 החלפות צבע או יותר. אך ראינו בכיתה כי עבור רצף של n הכנסות, המקרה הראשון יחול <u>לכל היותר</u> n פעמים (וזו גם הסיבה שפעולת ה – insertFixup היא בעלת סיבוכיות זמן ריצה amortized $O(1)$). על כן, נצפה כי הערך הממוצע של החלפות הצבע עבור כל הכנסה יהיה בסביבות ה-3.

פעולת התיקון לאחר Delete מחולקת לארבעה מקרים (אותם ראינו בכיתה). במקרה הראשון מתבצעות 2 החלפות צבע, במקרה השני בין 2 ל-1 החלפות, במקרה השלישי 3 החלפות ובמקרה הרביעי או החלפה 1 או 3. רק המקרה השני יכול לקרות כמה פעמים ברצף באותו תיקון, אך משום שהוא מעלה את מספר הצמתים האדומים בעץ – מספר הפעמים שיתבצע מוגבל. כמו כן, לאחר חלק מהמקרים מתבצע מקרה אחר בנוסף. לכן, נצפה כי במקרה הממוצע יתקבלו בין 2 ל-3 החלפות צבע (בשל ההשפעה הגדולה יותר של המקרה השני).

בפועל, ניתן לראות כי גם עבור הכנסה וגם עבור הוצאה התקבלו תוצאות בערך זהות – גם לאחר הגדלת מספר הפעולות שנעשות. זאת משום שמדובר על מספר ממוצע של החלפות צבע לפעולה – כלומר, מספר amortized של החלפות הצבע עבור insert ועבור delete. על כן, גם באם נגדיל את מספר האיברים המוכנסים/הנמחקים, מספר ההחלפות הממוצע לא אמור להשתנות – כפי שניתן לראות בטבלה. כמו כן, לפי חלוקת המקרים, קיבלנו בערך את התוצאה המצופה. עבור insert, קיבלנו בסביבות 2.3 החלפות צבע לפעולה – מה שמרמז כי בפועל המקרה הראשון (שעלול להתבצע ברצף כמה פעמים) קורה כנראה פחות מפעם אחת בממוצע להכנסה. דבר זה מתיישב עם מה שראינו בכיתה כי לכל היותר יתבצע n פעמים, ובפועל כנראה מתבצע אפילו פחות. עבור delete קיבלנו בסביבות ה-2.4 החלפות צבע לפעולה – מה שמתיישב עם ההשערה הראשונית שלנו.

מטרת החוקים של עץ אדום-שחור היא שמירה על העץ מאוזן, מה שגורם לפעולות על העץ (כמו חיפוש, הכנסה, מחיקה וכו') ליעילות יותר, שכן כולן תלויות בגובה העץ. עלות התיקונים לשמירה על החוקים עלולה לגרוע מיעילות העץ, באם קורים המון תיקונים בתדירות גבוהה. משמעות התוצאות היא שיעילות העץ איננה נפגעת כתוצאה מהתיקונים, שכן בממוצע אנו מקבלים מספר תיקונים קבוע לפעולה, כלומר זמן amortized $O(1)$ לתיקון – בדיוק כפי שראינו בכיתה.