מבני נתונים - פרויקט מספר 1 - עץ דרגות

מגישים:
שם: איתי צמח
שם משתמש: itaizemah
ת.ז: 209637453

שם:עודד כרמון
שם משתמש: ododecarmon
ת.ז: 208116517

תיעוד המחלקה:
המחלקה AVLTree מממשת עץ AVL המכיל מחרוזות ממוספרות על ידי מספרים טבעיים שונים.
צמתי העץ מוגדרים על ידי המנשק הפנימי IAVLNode שממומש על ידי המחלקה הפנימית AVLNode.

המחלקה AVLTree
למחלקה 5 שדות:

- מספר שלם size המתאר את גודל העץ, מאותחל כ-0.
- צומת EXT, אובייקט אחד משותף המייצג את כל הצמתים הוירטאולים שיושבים בעץ כבנים של העלים.
- צומת root, שורש העץ, מאותחל כ-EXT.
- צומת min, שמצביע לצומת המכילה את המפתח הקטן ביותר בעץ. מאותחלת כ EXT
- צומת max, שמצביע לצומת המכילה את המפתח הגדול ביותר בעץ, מאותחלת כ EXT.

מתודות המחלקה:

- public boolean empty()

מתודה זו מחזירה True אם ורק אם הגודל של העץ הוא 0. פועלת בסיבוכיות $O(1)$ כיוון שמבצעת פעולת השוואה אחת על שדה של המחלקה ומחזירה את התוצאה.

- public String max()-

מחזיר את ערך הinfo של האיבר הכי גדול בעץ. שמור כתכונה בעץ, ועל כן פועל בסיבוכיות $O(1)$

- public String min()-

מחזיר את ערך הinfo של האיבר הכי קטן בעץ. שמור כתכונה בעץ, ועל כן פועל בסיבוכיות

- public IAVLNode getRoot()-

מתודה זו תחזיר מצביע לשורש כעצם מטיפוס IAVLNode, ואם העץ ריק תחזיר null. סיבוכיות זמן ריצה: $O(1)$, החזרת מצביע.

- public int size()

מתודה זו תחזיר את מס' הצמתים הנוכחי בעץ, ששמור כתכונה בעץ. סיבוכיות זמן: $O(1)$, החזרת תכונה בעץ.

- public String search(int k)

מתודה זו מחזירה את המחרוזת המזוהה עם המפתח k אם הוא מופיע בעץ ואחרת מחזירה null. המתודה קוראת למתודת העזר searchNode(int k) הפועלת בסיבוכיות $O(\log n)$ ומקבלת ממנה את הצומת המתאים, לאחר מכן מפעילה על צומת זה את getKey() הפועל בסיבוכיות $O(1)$ ומלבד זאת מבצעת רק כמות קבועה של פעולות לכן גם מתודה זו פועלת בסיבוכיות $O(\log n)$.

- <u>private IAVLNode searchNode(int k)-</u>

_מימוש האלגוריתם שהוצג בכיתה, לחיפוש צומת בעץ, והחזרתו. משתמשת בהיותו של עץ AVL עץ חיפוש בינארי, אך עוצרת בהתאם כדי להמנע מהגעה לצמתים וירטואליים בעץ. סיבוכיות: $O(\log n)$. עוברים בכל רמה פעם אחת בדיוק- השוואה עם צומת האב תעדכן את הפעולה אם ללכת לכיוון שמאל או ימין.

- <u>private void rotate(IAVLNode parent, IAVLNode child)</u>-

<u>סיבוכיות זמן ריצה:</u> $O(1)$. מקבלת מצביע לאב ובנו בעץ, ועל פי המצביעים הנ"ל מבצעת רוטציה/ רוטציה כפולה לפי ההנחיות בשקף 31 במצגת על \.BST

- <u>public int insert(int k, String i)-</u>

מכניסה לעץ צומת חדשה, במיקום שבו הייתה מוכנסת בעץ הנוכחי על פי החוקים של עץ בינארי פשוט, וקוראת לפעולה rebalanceInsert שמתקנת את העץ כך שחוקי AVL יישמרו. <u>סיבוכיות זמן ריצה:</u> $O(\log n)$. מחפשים את המיקום המתאים בעץ כמו בעץ חיפוש בינארי רגיל, וקוראים ל rebalanceInsert שעולה אף היא $O(\log n)$. המתודה מחזירה לבסוף את מספר פעולות האיזון שנדרשו להכנסה לעץ תוך שמירה על תכונת הAVL, או -1 אם הצומת כבר בעץ.

- <u>private int rebalanceInsert(IAVLNode x, IAVLNode y)-</u>

מבצע את כל התיקונים להכנסה של הצומת שהוכנסה זה עתה לעץ (או בין שני צמתים כלליים בהם יש חשד להפרה), על פי פירוט המקרים במצגת WAVL, שקף 22, והמקרים הסימטריים להם. <u>סיבוכיות זמן ריצה:</u> $O(\log n), \Omega(1)$. תיקון יחיד/ רוטציה יחידה בעץ עולות זמן קבוע, ובמקרה הטוב ביותר לא נצטרך לעלות ולתקן רמות גבוהות יותר בעץ, כך שזמן הריצה של המתודה יישאר קבוע. במקרה הגרוע ביותר, נצטרך לעלות ולתקן את העץ עד שנגיע לשורש, ואז נקבל כי זמן הביצועי של המתודה הוא $O(height) = O(\log n)$. תחזיר את מספר פעולות האיזון בעבור ההכנסה.

- public int delete(int k)-

מוחק את הצומת שהמפתח שלו שווה לk, אם נמצא בעץ. המתודה תחזיר את מספר פעולות האיזון שנעשו כדי לשמור על האינווריאנטות של עץ AVL . במידה ולא נעשו פעולות תיקון, יוחזר 0, ואם המפתח לא בעץ, נחזיר -1.

- private IAVLNode deleteLeaf(IAVLNode node)-

פעולת עזר בעבור delete, מוחקת צומת מהעץ במקרה שהוא עלה. מחזירה את האב של העלה לצורכי איזון. סיבוכיות זמן ריצה: $O(1)$

- private IAVLNode deleteUnary(IAVLNode node)-

פעולת עזר בעבור delete, מוחקת צומת מהעץ במקרה שהוא צומת אונארי.. מחזירה את האב של העלה לצורכי איזון. סיבוכיות זמן ריצה: $O(1)$

- private rebalanceDelete(IAVLNode z)-

פעולת עזר בעבור delete, מבצעת את כל המקרים למחיקה בעבור delete כמפורט בשקף 38 במצגת WAVL ומחזירה את פעולות האיזון. דואגת במקביל גם לתחזוקת גדלי העצים. סיבוכיות זמן ריצה: $O(\log n)$

- private IAVLNode getSuccessor(IAVLNode node)-

מחזיר את הsuccessor של הצומת node בעץ, על פי האלגוריתם שתואר במצגת BST, אולם עם עצירה בצומת EXT ולא null. <u>סיבוכיות זמן ריצה:</u> $O(\log n)$

- <u>private IAVLNode getPredecessor(IAVLNode node)-</u>

סימטרית לsuccessor ולאלגורית מההרצה, ומחזירה את הקודם לצומת הנתון- אם קיים. <u>סיבוכיות זמן ריצה:</u> $O(\log n)$

- public String[] infoToArray()-

מחזיר מערך המכיל את כל הערכים השמורים תחת המפתחות בעץ, בסדר ממויין, על ידי שימוש בפעולת העזרת nodesToArray שהופכת את כל הצמתים בעץ למערך ממויין ע"י סריקה רקורסיבית בסדר תוכי ושימוש בגדלים של תתי עצים. סורק מערך זה ומחזיר את ערכי הinfo של כל אחד מהצמתים. <u>סיבוכיות זמן ריצה</u>: $O(n)$.כפי הסיבוכיות של nodesToArray, וסיבוכיות סריקת המערך המוחזר

- public int[] keysToArray()-

מחזיר מערך המכיל את כל המפתחות בעץ, בסדר ממויין, על ידי שימוש בפעולת העזרת nodesToArray שהופכת את כל הצמתים בעץ למערך ממויין ע"י סריקה רקורסיבית בסדר תוכי ושימוש בגדלים של תתי עצים. סורק מערך זה ומחזיר את ערכי הkey של כל אחד מהצמתים. <u>סיבוכיות זמן ריצה</u>: $O(n)$. כפי הסיבוכיות של nodesToArray, וסיבוכיות סריקת המערך המוחזר

- private IAVLNode[] nodesToArray()-

פונקציית מעטפה רקורסיבית, שקוראת לפונקציה פרטית נוספת עם מערך ריק של IAVLNode ומערך של מיקומים להכנסה ומחזירה מערך ממויין של כל הצמתים בעץ. <u>סיבוכיות זמן ריצה</u>: $O(n)$. כפי הסיבוכיות של פעולת העזר.

- private IAVLNode[] nodesToArray(IAVLNode node, IAVLNode[] arr, int[] i)-

חוצה את המערך, על ידי סריקה בסדר תוכי של הצמתים בעץ, ועל ידי סיפוק האינדקס המתאים להכנסה במערך i. בסופו של דבר, בסיום עץ הקריאות יוחזר מערך לפעולה הלא רקורסיבית שמלא בערכי העץ, ממויינים לפי ערכי המפתחות.

סיבוכיות זמן ריצה: $O(n)$. נעבור על כל צומת בעץ פעם אחת, בקריאה בסדר תוכי, ונכניס אותו למקום המתאים במערך.

- public join(IAVLNode x, AVLTree t)

מקבלת עץ AVL נוסף ומצביע לאיבר x, ומאחדת את t ואת x לתוך העץ הנוכחי. הנחת קדם חזקה: ישנו עץ אחד שכל מפתחותיו גדולים מהאחר, וx בין המפתחות של שניהם. סיבוכיות זמן ריצה: $O(1+|height(this)-height(t)|)=O(\log n)$. כפי שהוסבר בכיתה, העלות תהיה כעלות החיפוש בשרוך המתאים של צומת שדרגתו כדרגת צומת העץ הנמוך.

- protected void fixRanks(IAVLNode c, IAVLNode x)

פעולת עזר עבור join, שמטרתה להתמודד עם מקרה שלא מכוסה בהכנסה, ויש להתמודד איתו כפי שהוסבר בפורום במודל. סיבוכיות זמן ריצה: $O(\log n)$. מאחר שחיברנו שני עצים שונים, אנו בודקים אם יש צורך בשינוי בדרגות עד לשורש העץ.

- protected IAVLNode replaceBinary(IAVLNode node)

מטפל בהחלפה של צומת בעלת שני בנים עם העוקב שלה בעץ. את הצומת המקורית תחזיר במיקומה החדש, לצורך מחיקה שלה ב deleteLeaf או deleteUnary. סיבוכיות זמן ריצה: $O(\log n)$ מוצאת את העוקב- שדורשת זמן לוגריתמי במספר הצמתים בעץ, ומטפלת במצביעים כנדרש.

- public AVLTree[] split(int x)-

מפצל את העץ הנתון לפי המפתח x- יחזיר שני עצי AVL מאוזנים, שבאחד יש את כל המפתחות שקטנים מX ובאחר את כל המפתחות שגדולים ממנו. ישתמש בjoin לצורך יצירת העצים. סיבוכיות זמן ריצה: $O(\log n)$. כפי שראינו בכיתה

המחלקה AVLNode

תכונות:

Height- מייצג את דרגת הצו

מדידות:

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	10,000	3.4221	2.4035	15	29
2	20,000	3.41445	2.4027	17	31
3	30,000	3.4008667	2.4250333	17	37
4	40,000	3.404325	2.411225	17	32
5	50,000	3.41506	2.41682	18	35
6	60,000	3.4247	2.414	18	39
7	70,000	3.4026144	2.4136286	18	32
8	80,000	3.40765	2.412825	18	35
9	90,000	3.4206777	2.4145555	19	37
10	100,000	3.41414	2.4145555	19	38

התוצאות שהיינו מצפים לקבל, על סמך ההסבר התיאורטי של עצי AVL שנלמד בכיתה, היא שברצף של הכנסות וברצף של מחיקות - מספר פעולות האיזון הממוצע הוא $O(1)$, ומספר פעולות האיזון המקסימלי הוא $O(\log n)$.

התוצאות שקיבלנו בפועל תואמות את הציפיות, ניתן לראות כי כאשר מסתכלים על כמות הפעולות הממוצעות (כלומר האמורטייזד) היא קבועה ללא תלות בכמות האיברים בעץ, בעוד שהכמות המקסימלית עולה אך בקצב איטי ביותר.

משמעות המדידות שביצענו היא שהניתוח התיאורטי משקף את תפקוד מבנה הנתונים בפועל.

2.

מספר סידורי	מספר פעולות	עלות join ממוצע עבור split אקראי	עלות join מקסימלי עבור split אקראי	עלות join ממוצע עבור split של איבר מס בתת העץ השמאלי	עלות join מקסימלי עבור split של איבר מקסימלי בתת העץ השמאלי
1	10,000			12.279	15
2	20,000			13.295	16
3	30,000			13.831333333333333	17
4	40,000			14.25025	17
5	50,000			14.608	18
6	60,000			14.841833333333334	18
7	200				
8	200				
9	200				
10	200				