עבודה מעשית 1

חלק א

מחלקת AVLTree

שדות:
*שורש העץ - root.
*מצביע לעלה החיצוני - external_leaf.
*מצביע לצומת עם המפתח הקטן ביותר - min.
*מצביע לצומת עם המפתח הגדול ביותר - max.

מחלקת AVLNode

מח' פנימית המממשת את הממשק IAVLNode.
אובייקט/עצם במח' זו מייצג צומת בעץ.

שדות:
*מצביע לבן השמאלי - left.
*מצביע לבן הימני - right.
*מצביע לצומת ההורה - parent.
*מפתח הצומת - key.
*מידע המאוכסן בצומת - value.
*גובה/דרגת הצומת בעץ - height.
*גודל הצומת (מספר הצמתים בעץ, כולל) - size.

הבנאי של המחלקה מקבל מפתח ומידע כמובן, ומאתחל את הצומת.
שאר המתודות במחלקה הן getters, setters ברורות מאליו.

פעולות

הבנאי AVLTree (): מאתחל עץ ריק; זמן קבוע.

הבנאי AVLTree (IAVLNode head): המתודה מקבלת מצביע לצומת ויוצרת ממנו עץ AVL:
השורש של העץ הוא המצביע שקיבלנו, וכמו כן נעדכן את המינ' והמקס' הרלוונטים בעזרת Min ו-Max (פירוט בהמשך). הפעולות Min ו-Max לוקחות log n כל אחת (ההשמות הן בזמן קבוע)
לכן בסה"כ O (log n) = 2*log n.

Max: בהינתן שורש העץ, יורדים ימינה ככל האפשר, ומחזירים את הצומת השמאלי ביותר - הצומת עם המפתח המקסימאלי. בסה"כ מספר האיטרציות הוא כגובה העץ, קרי O (log n).

Min: בהינתן שורש העץ, יורדים שמאלה ככל האפשר, ומחזירים את הצומת השמאלי ביותר - הצומת עם המפתח המינמאלי. בסה"כ מספר האיטרציות הוא כגובה העץ, קרי O (log n).

getRoot: המתודה מחזירה את השורש של העץ, המתודה לא מקבלת שום ערכים. במידה והעץ ריק, הפונקציה מחזירה null. כיוון שיש לנו מצביע לשורש של העץ, פעולה זאת לוקחת סיבוכיות של O(1).

size: מחזירה את השדה size של העץ. פעולה זו כמובן לוקחת זמן קבוע.

empty: בודקת האם העץ לא ריק או ריק (ע"י השוואת השורש לnull); זמן קבוע.

max: מחזירה את השדה max של העץ. פעולה זו כמובן לוקחת זמן קבוע.

min: מחזירה את השדה min של העץ. פעולה זו כמובן לוקחת זמן קבוע.

search:
המתודה מחפשת בעץ איבר בעל מפתח k שמקבלת בארגומנט.
במידה וקיים איבר כזה, היא מחזירה את המחרוזת שלו, אחרת מחזירה null.
המתודה משתמשת במתודת העזר search rec.(יפורט בהמשך). אופן פעולת האלגוריתם:
במידה והעץ ריק נחזיר null, אחרת נגדיר משתנה string בשם info שעתיד להחזיר את המחרוזת המתאימה.
נפעיל את מתודת search_helper, בסיומה נחזיר את info שכעת מחזיק את המחרוזת הרצויה.
סיבוכיות זמן הריצה תלוי בעיקר במתודת העזר; נראה שסיבוכיות מתודת העזר הוא O(log(n)),
וזה בדיוק הסיבוכיות של מתודת Search.

search helper: מתודה שנשתמש לעיתים תכופות בקוד.
זוהי מתודת עזר למתודה search . תפקידה לחפש באמצעות רקורסיה בעץ האם קיים איבר בצומת בעל המפתח k. במידה וכן, נחזיר את האיבר/הצומת המתאים. אחרת נחזיר null. מימוש מתודת העזר זהה למימוש שראינו בהרצאה: נבדוק האם המפתח קטן/שווה/גדול מהשורש ונחלק למקרים:
אם המפתח שווה לשורש סימן שהשורש הוא האיבר הרצוי ונחזיר את המצביע שלו.
אם המפתח קטן ממש מהשורש נעבור להסתכל על הבן השמאלי של השורש
אחרת, נעבור להסתכל על הבן הימני של השורש.

אנו נבצע את האלגוריתם עד שנמצא את האיבר הרצוי או שנגיע לאיבר מדומה (שההורה של האיבר המדומה הוא עלה בעץ), במידה והגענו לצומת מדומה נובע שלא קיים איבר בעץ עם המפתח שאנו מחפשים, לכן נחזיר null.

החיפוש בעץ חסום ע"י גודל העץ h.
על פי ההרצאה h=O(Log(N)), לכן סיבוכיות זמן הריצה של מתודת העזר הוא כגודל העץ כלומר O(log(n)).

keysToArray: למעשה תפעל כפונ' מעטפת. מחזירה את ערך הפונ' keysToArray_Helper;
סיבוכיותה הוא ליניארי, שכן עיקר פעילותה היא בהתאם למתודת העזר.

keysToArray_Helper: פונ' רקורסיבית, שפותחת 2 קריאות: לתת-העץ השמאלי, ולתת העץ הימני. משום שהעץ הוא מאוזן (או כמעט מאוזן ליתר דיוק) גודל הארגומנט/תת העץ מהווה חצי מהקודם. בנוסף, מגדילים את הרץ באחד ומכניסים את present למערך (עלות קבועה).
קרי, נוסחת הנסיגה היא T(n) = θ(n/2) + 1, ופתרונה הוא ליניארי. בסה"כ: O(n).

Info to Array:
המתודה מחזירה מערך מחרוזות המכיל את כל המחרוזות בעץ, ממוינות על פי סדר המפתחות, במידה והעץ ריק הפונקציה תחזיר מערך ריק. המתודה לא מקבלת שום ערכים.המתודה משתמשת במתודת עזר orderValue (יפורט בהמשך) אופן פעולת האלגוריתם: במידה והעץ ריק נחזיר מערך ריק, אם לא ניצור שני מערכים:
המערך arrayVal- מערך string שגודלו הוא כמספר הצמתים בעץ (יש לנו שדה לעץ שבכל רגע נתון מחזיר את מספר הצמתים בעץ) שנחזיר בסיום התוכנית, בו יהיו כל המחרוזות ממוינות לפי סדר המפתחות.
המערך OrderArray- מערך int בגודל 1 שתפקידו לספור כמה מפתחות הכנסנו למערך arrayVal.
נפעיל את מתודת orderValue, בסיומה המערך arrayVal יכיל את כל המחרוזות בעץ, ונחזיר את arrayVal.

סיבוכיות זמן הריצה:
יצירת מערך בגודל N כאשר N מספר הצמתים בעץ לוקח O(N)
הפעלת מתודת העזר orderValue לוקח O(N)
לכן בסה"כ זמן הריצה של info to Array הוא O(N)+O(N)=O(N) כלומר O(N).

order Value:
מתודת עזר למתודה info to Array, תפקידה לעבור ברקורסיה על כל הצמתים בעץ, ולהכניס את כל המחרוזות של העץ לפי סדר המפתחות לתוך המערך arrayVal. מקבלת כארגומנטים:
שורש של העץ, מערך string ריק ומערך של int. המתודה מבצעת סריקת in-order בעץ, כלומר עוברים על כל הצמתים בעץ מהצומת בעל המפתח הקטן ביותר לצומת בעל המפתח הגדול ביותר.
ברגע שנגיע לצומת ללא ילדים נוסיף את הצומת למערך arrayVal במקום הOrderArray[0],
כאשר OrderArray סופר כמה מפתחות כבר הכנסנו למערך arrayVal.
בכל פעם שנכניס מפתח למערך, נקדם את OrderArray[0] ב1+.

לאחר סיום התוכנית לא נחזיר פלט, אבל המערך arrayVal יכיל את כל המחרוזות ממויינים כרצוי.
סיבוכיות הזמן של האלגוריתם: מבצעים סריקת inorder על העץ, לכן מעבר על כל הצמתים בעץ לוקח O(N) כאשר N מייצג את מספר הצמתים בעץ (בדיוק כמו בתרגול). אנו מבצעים N הכנסות למערך arrayVal. כל הכנסה למערך היא הכנסה למיקום ידוע מראש במערך, לכן כל הכנסה לוקחת O(1) פעולות. אנו מבצעים N הכנסות, לכן בסה"כ עלות כל ההכנסות הוא O(N)
אנו מקדמים בכל הכנסה למערך arrayVal את OrderArray[0], לכן O(1).

לכן בסה"כ סיבוכיות הזמן של orderValue הוא O(N)+O(N)=O(N)כלומר O(N).

left_rotation, right_rotation: בהתאם לחומר הנלמד בביתה, המתודות הללו מבצעות פעולת סיבוב על 2 צמתים בעץ הAVL, זאת באמצעות שינוי של המצביעים וריצה בזמן קבוע.

successor: מטרת הפונ' היא להחזיר את הצומת עם המפתח העוקב;
אם לצומת אין בן ימני, נעלה בעץ כל זמן שהצומת שהתקבל כארגומנט נמצא בתת העץ הימני של הצומת הבא.
במידה ויש - נחפש את הצומת המינימאלי בתת העץ הימני.
במקרה הראשון נעלה לכל היותר עד השורש, ובשני נרד בתת עץ שגובה חסום מגובה העץ "המקורי". בכל אחד מהמקרים עבור עץ הAVL נקבל O (log n).

Insert:
פעולת insert מוסיפה לעץ AVL צומת (אם הוא לא קיים), ומחזירה את מס' פעולות האיזון שנדרשו ע"מ לשמר את תכונות עץ ה AVL בעץ הנוכחי לאחר הכנסת הצומת החדש.
ראשית נאתחל את הצומת שיש להכניס (לפי הבנאי השני) - O (log n) .
נשתמש במתודת העזר insert_helper שרצה בסיבוכיות O (log n).
לסיכום, לאחר סכימת הסיבוכיויות נקבל זמן ריצה WC של O(log n) .

Insert_helper: אם העץ הינו עץ ריק נכניס את הצומת שיצרנו עם הערכי הקלט ,נחזיר 0 (כי לא בוצעו איזונים) וסיימנו. אחרת , נרצה לחפש האם הצומת עם המפתח שהתקבל כקלט קיים בעץ ,משום שאם המפתח נמצא כבר בעץ אזי אין צורך בהכנסה. נבדוק האם הצומת נמצא בעץ בעזרת search_helper, אם כן נחזיר 1- .
אחרת, נצטרך "לדחוף" את הצומת החדש (שהתקבל כארגומנט), ונחבר אותו במיקום הנדרש. בנוסף להכנסה , עדכנו גם בזמן O(1) את הצומת המינימלי, מקסימלי והשורש (אם נדרש). כמו כן תחזקנו כמובן את שדה ה size בכל הצמתים הרלוונטים (שבמעלה הצומת שהוכנס, עד השורש כולל), וכן עדכנו את הגובה. לבסוף , נשלח את העץ לפונקציה insert_rebalance הדואגת לאיזונים ומחזירה את מספר פעולות האיזון שנדרשו, כאשר פונ' זו מבוצעת בזמן O(log n) בזמן הגרוע ביותר. סיבוכיות: קראנו לsearch helper (לוגרתימי), תחזקנו את שדה ה size (log n * 1), והקריאה לפונ' הrebalance גם לוגריתמית, לכן בסה"כ קיבלנו סיבוכיות O(log n).

Insert_rebalance:
הפונקציה מאזנת את העץ לאחר הכנסה של צומת חדש. בפונקציה ישנה לולאה שרצה לכל היותר log(n) פעמים – היא מתחילה מעלה ובכל שלב עולה במעלה העץ, ולכל היותר תרוץ עד השורש. הפונ' סוקרת את כל המקרים שהוצגו בהרצאה בעת הכנסה, ומעלה את המונה בהתאם (מונה אותו היא כמובן מחזירה, שכן בסופו של דבר לאחר ההכנסה נרצה לדעת כמו פעולות איזון בוצעו). סיבוכיות: בלולאה יש מספר קבוע של

קריאות לפונקציות left_rotation, right_rotation, העלאת מונה פעולות האיזון ('counter') והשמה/עדכון הגבהים. הקריאה לפונ' האיזון, ויתר הפעולות שצוינו אחריהן, לוקחות כמובן זמן קבוע, ולכן הלולאה רצה בסה"כ ב (O(log(n) זמן.

:delete

מטרתה של פעולת ה delete היא מחיקת צומת מהעץ במידת הצורך. התוכנית מקבלת כקלט את המפתח שהמשתמש רוצה למחוק. אם העץ ריק מוחזר למשתמש -1 . אחרת ,ניעזר בפונקציית העזר search_helper אשר רצה בסיבוכיות זמן של O(log n) ומטרתה לסייע לנו למצוא צומת בעץ עם מפתח ספציפי. אם הפעולה תחזיר צומת וירטואלי , אזי המפתח שהמשתמש הזין כקלט אינו נמצא בעץ ועל כן נחזיר (1-). לאחר שווידאנו שמקרי קצה אינם מתקיימים , נתקדם הלאה בתוכנית שכן משמעות הדבר היא שהמפתח המבוקש נמצא.

נבחן הצומת שקיבלנו מהחיפוש של search_helper. נבדוק אם הוא הוא צומת שמהווה בן ימני / שמאלי או בדיוק צומת השורש (ובמקרה שהוא הצומת היחידי כמובן שלאחר מחיקה העץ יהיה ריק). בכל אחד מהמקרים נבדוק האם הצומת היא אונארית/בינארית/עלה, ובהתאם נפעל ונעדכן את המצביעים והשדות הרלוונטים בזמן קבוע; כחלק מתהליך המחיקה נעשה שימוש במתודות:

*delete_rebalance - על מנת לאזן את העץ. סיבוכיות log n.

*diminish – כדי לתחזק את שדה הsize וheight. סיבוכיות log n.

* successor (שתוארה כבר). סיבוכיות log n.

כל אחת מ-3 המתודות לעיל מופעלות לכל היותר פעם אחת כל אחת - 0(log n)

נעדכן כמובן את שדות המינ' והמקס' (0(logn. נחזיר את התוצר של delete_rebalance, קרי את מספר פעולות האיזון.

סה"כ ,קיבלנו כי סיבוכיות הזמן הכוללת של פונקציית delete תהיה O(log n).

:delete_rebalance

תבצע את האיזונים הנדרשים בעץ בהתאם לכל המאורעות שהוצגו בהרצאות. הפונ' תחזיר לנו כפלט את מס' פעולות האיזון ע"י מונה . בפונקציה ישנה לולאה שרצה לכל היותר log(n) פעמים – היא מתחילה מעלה ובכל שלב עולה במעלה העץ, לכל היותר תרוץ עד השורש. בלולאה יש מספר קבוע של קריאות לפונ' left_rotation, right_rotation וכמו כן השמת גבהים וצבירה של המונה. הפעולות האחרונות ופונקציות הסיבוב לוקחות O(1) זמן ולכן הפונ' רצה בסה"כ ב O(log n).

diminish: תחזוק שדות ה size ו ה height של העץ לאחר מחיקה (מעדכן את השדות הללו בכל הצמתים במעלה העץ, עד לשורש כמובן). מתבצעת כמובן בסיבוכיות O(log n) (עולים פעם אחת במעלה העץ).

join: המתודה מקבלת עץ AVL שמפתחותיו גדולים או קטנים מהמפתחות של העץ עליו מפעילים את הפעולה, וצומת עם מפתח הגדול ממפתחות אחד העצים וקטן ממפתחות השני .המתודה מחזירה את עלות הפעולה, אשר מוגדרת להיות הפרשי גבהיי העצים ועוד אחד. ראשית אנו בודקים את מאפייני העצים - גובה וגדלי המפתחות , זאת כדי שנדע כיצד אנו צריכים לפעול. לאחר מכן אנו פועלים בדיוק לפי האלגוריתם שנלמד בכיתה , יורדים במורד דופן העץ הגבוה (דופן שמאלי אם מפתחותיו גדולים יותר , דופן ימני אם קטנים יותר) עד שנגיע לצומת שגובה קטן-שווה לגובהו של העץ הנמוך. בהמשך, נהפוך את הצומת x שקיבלנו להיות ההורה של הצומת שהגענו אליו ושל שורש העץ הנמוך (שוב, בהתאם לגודל המפתחות), ואת ההורה של הצומת שהגענו אליו נהפוך להיות ההורה של x. בנוסף ,נדרש לעדכן את גדלי הצמתים מצומת x ועד השורש. כל הפעולות האחרונות יתבצעו ע"י מתודת העזר join_assisstance (בהמשך). בסיום נאזן את העץ על-ידי מתודת האיזון שהשתמשנו בעת הכנסה - Insert_rebalance.

במקרה הגרוע , ביצענו O(log n) בירידה מטה בעץ , עוד O(log n) בעדכון גדלי הצמתים באיזון , כל שאר הפעולות הן בזמן O(1) ולכן לסיכום המתודה רצה בזמן O(log n).

:join_assisstance

מקבל ארבעה ארגומנטים: a,x,b ,t2. המתודה מתחזקת הגובה בעץ לאחר שעידכנה את המצביעים באופן כזה ש: a (צומת של העץ הזה/הthis) הבן שמאלי של x, ו- b (צומת בt2) הבן הימני של x. זמן הריצה WC יהיה ללכת את כל גובה העץ שהוא O(log n) ועל כן סיבוכיות הזמן של פונקציה זו היא O(log n).

split:

המתודה מקבלת מפתח x שנמצא בעץ ,ומחזירה מערך של שני עצי AVL ובו עץ עם כל המפתחות הקטנים מ x ועץ עם כל המפתחות הגדולים מ x. ראשית , חיפשנו את מיקומו של צומת עם מפתח x באמצעות מתודת העזר search_helper בזמן לוגריתמי. לאחר מכן , פעלנו בדיוק לפי האלגוריתם הנלמד בכיתה – עלינו מהצומת שמצאנו עד לשורש ,ובכל איטרציה ביצענו פעולת join בין העץ עם המפתחות הגדולים , לתת-העץ הימני של הצומת שאנו נמצאים בו. אם הגענו לצומת החדש מצד ימין, ביצענו join בין העץ עם המפתחות הקטנים, לתת העץ השמאלי של הצומת שאנו נמצאים בו. המתודה כולה רצה בזמן O(log n) זאת משום שהוכחנו בכיתה באמצעות טור טלסקופי שעלות סדרת ה join שמבצעים לאורך כל ריצת לולאת ה while היא O(log n) .