**מבני נתונים – תרגיל מעשי**

|  |  |
| --- | --- |
| שם: רן ערמוני  שם משתמש: ranarmoni  ת.ז - 204027312 | שם: שחף גורן  שם משתמש: shahafgoren  ת.ז - 203269931 |

**סימונים –**

n-גודל העץ

**תיעוד-**

**מחלקה 1 – WAVLTree**

השדה היחיד באובייקט מסוג זה הוא root, שהוא משתנה מסוג WALNode המפנה לשורש העץ.

מתודות –

|  |  |  |
| --- | --- | --- |
| חתימה | סיבוכיות | הסבר |
| **public** WAVLTree() | O(1) | בונה עץ עם שורש דיפולטי. |
| **public** WAVLTree(WAVLNode root) | O(1) | מקבלת WAVLNode ובונה עץ שזהו שורשו |
| **public** **boolean** empty() | O(1) | מחזירה true אם השורש הוא צומת וירטואלי (כלומר אין שורש אמיתי לעץ, לכן העץ ריק) ולהפך. |
| **public** String search(**int** k) | O(logn) | מבצעת חיפוש של צומת עם מפתח k בעץ, ע"י קריאה לפונקציה searchVal של המחלקה WAVLNode על שורש העץ. |
| **public** **int** insert(**int** k, String i) | O(logn) | מכניסה לעץ צומת עם מפתח k ומידע i. תחילה הפונקציה קוראת לפונקציה searchNode, שמחזירה את המקום בעץ אליו צריכה להתבצע ההכנסה. אם מקום זה הוא צומת אמיתי, לא מתבצעת הכנסה. אם זהו צומת וירטואלי – הפונקציה יוצרת את הNode החדש, מכניסה אותו למקומו, וקוראת לפונקציה rebalance שמבצעת איזון מחדש לעץ. Rebalance מחזירה את מספר פעולות האיזון שביצעה, וinsert מחזירה מספר זה. |
| **public** **int** delete(**int** k) | O(logn) | מחפשת את המפתח המבוקש בעזרת searchNode – O(logn) אם הצומת לא נמצא מוחזר -1.  לאחר מכן הפונקציה בודקת אם מדובר בצומת פנימי, במידה וכן היא מחליפה אותו עם הsuccessor שלו ועוברת למחוק את האיבר הנדרש (כעת זהו עלה או צומת אונרי)  הפונקציה קוראת לפונ' עזר (O(1)) בהתאם לסוג הצומת (אונרי או עלה) ולאחר מכן קוראת לreBalance ע"מ לאזן את העץ |
| **public** **int** reBalance(WAVLNode node, String state) | O(logn) | זוהי פונקציית האיזון של העץ שנקראת ממתודות ההכנסה או המחיקה. היא מקבלת כקלט את הצומת בו נעשה השינוי והאם מדובר בהכנסה או מחיקה. המתודה מאתחלת קאונטר שסופר את פעולות האיזון, ופועלת בלולאת WHILE שרצה כל עוד הצומת הנוכחי אינו השורש, ואחרי כל איטרציה האיטרציה הבאה מתבצעת על האבא של השורש בו טיפלנו (כלומר המתודה "מטפסת" עד שורש העץ). המתודה מורכבת מתנאי if שכל אחד מהם קורא למתודה הבודקת האם פעולת איזון מסוג מסוים צריכה להתרחש (כל המתודות שחתימתן מתחילה במילה needs) ובמידה ואותה מתודה מחזירה true, rebalance קוראת למתודה שמבצעת את פעולת האיזון המתאימה (למשל, rotate), מוסיפה למונה הפעולות את הכמות המתאימה (למשל – 2 עבור סיבוב כפול), ועוברת לאיטרציה הבאה על האבא של הצומת בו טיפלנו כעת. המתודה מבצעת o(1) פעולות בכל איטרציה (ראו תיעוד המתודות עצמן) , ועולה רמה בעץ בכל איטרציה, לכן בסה"כ זמן הריצה שלה הוא o(logn). בכל איטרציה גם מעודכן גודל תת העץ של אותה צומת בעזרת המתודה changeSize– הגודל מוגדל ב-1 אם מדובר באיזון אחרי הכנסה, ומוקטן ב-1 אם מדובר במחיקה. |
| **public** **void** changeSize(WAVLNode node, String state) | O(1) | משנה את גודל תת העץ - הגודל מוגדל ב-1 אם מדובר באיזון אחרי הכנסה, ומוקטן ב-1 אם מדובר במחיקה. נקראת בכל איטרציה של האיזון מחדש ומקבלת כפרמטר את הצומת הרלוונטי והאם מדובר בהכנסה או מחיקה. |
| **public** String min() | O(logn) | מוצאת את הערך המינימאלי של העץ ע"י קריאה לפונקציה minVal של המחלקה WAVLNode |
| **public** String max() | O(logn) | מוצאת את הערך המקסימאלי של העץ ע"י קריאה לפונקציה maxVal של המחלקה WAVLNode |
| **public** **int**[] keysToArray() | o(n) | מחזירה מערך של מפתחות העץ ממויינים לפי הסדר. אופן פעולה – תחילה יוצרת מערך בגודל העץ, ואז הולכת למינימום של העץ, מכניסה אותו בתחילת המערך, ואז מבצעת לולאת WHILE כשבכל איטרציה מבוצעת פעולת עוקב ואחריה מוכנס מפתח הצומת למערך. הלולאה נפסקת כאשר אין עוקב (כלומר העוקב הוא צומת וירטואלי). הסיבוכיות היא o(n) כי מציאת המינימום לוקחת o(logn) וכפי שראינו בתרגול n פעולות העוקב שיבוצעו לאחר מכן לוקחות יחד o(n), ובכל איטרציה מתבצע מספר פעולות קבוע מלבד פעולת העוקב. לכן בסה"כ יבוצעו o(logn) + o(n) =o(n). |
| **public** String[] infoToArray() | o(n) | פועלת בדיוק כמו keysToArray פרט לכך שמכניסה למערך את הערכים ולא המפתחות. |
| **public** **int** size() | O(1) | מחזירה את גודל העץ בעזרת המתודה getSubtreeSize() שמחזירה את הגודל של תת העץ של צומת מסוים. |
| **public** IWAVLNode getRoot() | O(1) | מחזירה את שורש העץ המאוחסן בשדה root. |
| **public** String select(**int** i) | O(n) | מחזירה את ערך האיבר עם המפתח הi בגודלו בעץ. אם האיבר לא קיים בעץ – מחזירה "1-". אופן פעולה – מוצאת את האיבר המינימאלי בעץ ומבצעת ממנו לולאה של i פעולות עוקב (אם במהלך אחד מהם העוקב הוא null מוחזר 1-). במקרה הגרוע ביותר i>=n ואז הסיבוכיות היא o(n) כמו במתודה KeysToArray. |
| **public** WAVLNode rotate(WAVLNode father, WAVLNode troubleMakerSon) | O(1) | המתודה מקבלת שתי צמתים המקיימים יחס של אב ובן ולפי סוג היחס (בן ימני או שמאלי) מבצעת סיבוב (שכן בעצי WAVL אנו מסובבים רק לכיוון אחד כדי לאזן את העץ לכל סוג צומת)  לאחר מכן המתודה מסדר את יחסי ההיררכיה עם שאר הצמתים שהושפעו מהסיבוב  לאחר מכן היא מעדכנת את הדרגות והגודל של הצמתי שהשתתפו בסיבוב ישירות.  המתודה פועלת מספר סופי של פעולות עם מספר סופי של צמתים, כלומר O(1) |
| **public** WAVLNode doubleRotate(WAVLNode father, WAVLNode son) | O(1) | בהתאם לכיוון הקשר בין הפרמטרים של האב והבן הניתנים למתודה, המתודה קוראת פעמיים לrotate בכיוון הנחוץ על מנת לאזן ובכך מבצעת בעצם סיבוב כפול.  לאחר מכן היא מסדרת את הדרגות והגדלים של הצמתים שהשתתפו בסיבוב הכפול. |
| **public** **void** promote(WAVLNode node) | O(1) | מעלה את הrank של הצומת ב-1. |
| **public** **void** demote(WAVLNode node) | O(1) | מוריד את הrank של הצומת ב-1. |
| **public** **void** doubleDemoteRight(WAVLNode node) | O(1) | מוריד את הrank של הצומת ושל בנה הימני ב-1. |
| **public** **void** doubleDemoteLeft(WAVLNode node) | O(1) | מוריד את הrank של הצומת ושל בנה השמאלי ב-1. |

**המחלקה WAVLNode**

המחלקה מהווה מימוש של צומת בעץ WAVL, ומממשת את הממשק IWAVLNode.

שדות המחלקה:

|  |  |
| --- | --- |
| מפתח הצומת | **public** **int** key |
| גודל תת העץ שתחת הצומת, כולל עצמה | **public** **int** treeSize |
| דרגת הצומת | **public** **int** rank |
| הערך (מידע) שהוצמת מחזיקה | **public** String val |
| הבן הימני של הצומת | **public** WAVLNode rightSon |
| הבן השמאלי של הצומת | **public** WAVLNode leftSon |
| האב של הצומת | **public** WAVLNode dad |
| האם הצומת אמתי (או וירטואלי) | **public** **boolean** isReal |

מתודות –

|  |  |  |
| --- | --- | --- |
| חתימה | סיבוכיות | הסבר |
| **public** WAVLNode() | O(1) | בנאי לעלה וירטואלי ללא אבא |
| **public** WAVLNode(WAVLNode dad) | O(1) | בנאי לעלה וירטואלי עם אבא dad |
| **public** WAVLNode(**int** key, String val) | O(1) | בנאי לצומת אמיתי עם מפתח key וערך val ללא אבא |
| **public** **boolean** isMiddleNode() | O(1) | בודק האם הצומת הוא צומת אמצעי (צומת בינארי עם שני ילדים אמיתיים) |
| **public** **void** replace(WAVLNode node) | O(1) | מתודה המחליפה בין שני צמתים אמיתיים ע"י החלפת הערכים ביניהם (ה"קופסאות" נשארות במקומן). |
| **public** **void** deleteUnary() | O(1) | המתודה מוחקת צומת אונארי בהתאם לאלגוריתם שנלמד. מבצעת מספר פעולות קבוע ונקראת מהמתודה delete במחלקה WAVLTree. |
| **public** **void** deleteLeaf() | O(1) | המתודה מוחקת עלה ומחליפה אותו בעלה וירטואלי. מבצעת מספר פעולות קבוע ונקראת מהמתודה delete במחלקה WAVLTree. |
| **private** **void** insertInPlace(WAVLNode node) | O(1) | המתודה מכניסה צומת אמיתי כעלה בעץ, במקום צומת וירטואלי. |
| **private** **boolean** isLeftSon() | O(1) | בודקת האם צומת מסוים הוא בן שמאלי של אביו. |
| **public** **void** setRightSon(WAVLNode node) | O(1) | מגדירה את node כבן ימני של הצומת this. |
| **public** **void** setLeftSon(WAVLNode node) | O(1) | מגדירה את node כבן שמאלי של הצומת this. |
| **public** **void** updateSize() | O(1) | מעדכנת את הגודל של תת העץ תחת הצומת, לפי סכימת הגדלים של בניו +1. |
| **public** **int** getKey() | O(1) | מחזירה את ערך השדה key |
| **public** String getValue() | O(1) | מחזירה את ערך השדה value |
| **public** **int** getRank() | O(1) | מחזירה את ערך השדה rank |
| **public** WAVLNode getLeft() | O(1) | מחזירה את ערך השדה leftSon |
| **public** WAVLNode getRight() | O(1) | מחזירה את ערך השדה rightSon |
| **public** WAVLNode getDad() | O(1) | מחזירה את ערך השדה dad |
| **public** **boolean** isRealNode() | O(1) | מחזירה את ערך השדה isReal |
| **public** WAVLNode successor() | o(logn) | מוצאת את העוקב של הצומת בעזרת האלגוריתם שנלמד בכיתה. עשויה לעלות או לרדת את כל העץ (תוך ביצוע מספר פעולות קבוע בכל רמה) לכן פועkת בo(logn) |
| **public** WAVLNode predeccessor() | o(logn) | מוצאת את ה"קודם" של הצומת בעזרת האלגוריתם שנלמד בכיתה. עשויה לעלות או לרדת את כל העץ (תוך ביצוע מספר פעולות קבוע בכל רמה) לכן פועלת בo(logn). (סימטרית ל"עוקב") |
| **public** **boolean** isValidDifs() | O(1) | בודקת האם הפרשי הדרגה של הצומת משני בניה חוקיים לפי הגדרת wavl. |
| **public** **boolean** needsPromote() | O(1) | בודקת האם צריך לבצע promote לדרגת הצומת (בהתאם לתנאים שנלמדו). |
| **public** **boolean** needsRightRotate() | O(1) | בודקת האם צריך לבצע סיבוב ימינה ע"מ לאזן את העץ בסביבת הצומת (בהתאם לתנאים שנלמדו). |
| **public** **boolean** needsLeftRotate() | O(1) | בודקת האם צריך לבצע סיבוב שמאלה ע"מ לאזן את העץ בסביבת הצומת (בהתאם לתנאים שנלמדו). |
| **public** **boolean** needsRotate() | O(1) | בודקת האם אחת משתי המתודות הקודמות החזירו אמת – כלומר האם צריך סיבוב כלשהו. |
| **public** **boolean** needsDoubleRotateRight() | O(1) | בודקת האם צריך לבצע סיבוב כפול ימינה ע"מ לאזן את העץ בסביבת הצומת (בהתאם לתנאים שנלמדו). |
| **public** **boolean** needsDoubleRotateLeft() | O(1) | בודקת האם צריך לבצע סיבוב כפול שמאלה ע"מ לאזן את העץ בסביבת הצומת (בהתאם לתנאים שנלמדו). |
| **public** **boolean** needsDemote() | O(1) | בודקת האם צריך לבצע demote לצומת ע"מ לאזן את העץ בסביבת הצומת (בהתאם לתנאים שנלמדו). |
| **public** **boolean** needsDoubleDemoteRight() | O(1) | בודקת האם צריך לבצע demote כפול לצומת ולבנה הימני ע"מ לאזן את העץ בסביבת הצומת (בהתאם לתנאים שנלמדו). |
| **public** **boolean** needsDoubleDemoteLeft() | O(1) | בודקת האם צריך לבצע demote כפול לצומת ולבנה השמאלי ע"מ לאזן את העץ בסביבת הצומת (בהתאם לתנאים שנלמדו). |
| **public** **int** getSubtreeSize() | O(1) | מחזירה את ערך השדה treeSize. |
| **public** WAVLNode searchNode(**int** k) | o(logn) | מחפשת צומת עם מפתח k ע"י חיפוש בינארי משורש העץ ומחזירה את הצומת עצמה. |
| **public** String searchVal(**int** k) | o(logn) | מחפשת צומת עם מפתח k ע"י חיפוש בינארי משורש העץ ומחזירה את ערכה. |
| **public** WAVLNode minNode() | o(logn) | מאתרת את הצומת המינימאלי בעץ ע"י הליכה שמאלה עד לעלה השמאלי ביותר (רקורסיבית). |
| **public** String minVal() | o(logn) | מאתרת את הצומת המינימאלי בעזרת המתודה הקודמת ברשימה ומחזירה את ערכו. |
| **public** WAVLNode maxNode() | o(logn) | מאתרת את הצומת המקסימאלי בעץ ע"י הליכה ימינה עד לעלה הימני ביותר (רקורסיבית). |
| **public** String maxVal() | o(logn) | מאתרת את הצומת המקסימאלי בעזרת המתודה הקודמת ברשימה ומחזירה את ערכו. |

**מדידות**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| מספר פעולות האיזון המקסימלי לפעולת delete | מספר פעולות האיזון המקסימלי לפעולת insert | מספר פעולות האיזון הממוצע לפעולת delete | מספר פעולות האיזון הממוצע לפעולת insert | מספר פעולות | מספר סידורי |
| 9 | 15 | 1.4373 | 2.4717 | 10,000 | 1 |
| 10 | 16 | 1.433 | 2.482 | 20,000 | 2 |
| 10 | 17 | 1.436 | 2.482 | 30,000 | 3 |
| 10 | 18 | 1.438 | 2.485 | 40,000 | 4 |
| 10 | 18 | 1.435 | 2.484 | 50,000 | 5 |
| 10 | 18 | 1.436 | 2.487 | 60,000 | 6 |
| 10 | 19 | 1.438 | 2.486 | 70,000 | 7 |
| 10 | 19 | 1.437 | 2.487 | 80,000 | 8 |
| 10 | 19 | 1.437 | 2.488 | 90,000 | 9 |
| 10 | 19 | 1.437 | 2.483 | 100,000 | 10 |

1. מספר פעולות איזון ממוצע לפעולת הכנסה ומחיקה הוא מספר קבוע עקבי בין הנסיונות. תוצאה זו מתיישבת עם הצפוי מהתאוריה, מפני שראינו שכמות פעולות האיזון לפעולת מחיקה או הכנסה היא o(1) אמורטייזד.
2. פעולות איזון ממוצעות לפעולת הכנסה – פעולת סיבוב כפול היא פעולה אחרונה, לכן הרצף שיגרום לפעילות איזון מקסימליות הוא רצף של פעולת איזון יחידה בכל רמה, המסתיים בפעולת סיבוב כפול ברמה העליונה ביותר (בשורש). כלומר – נצפה שאורך הרצף הארוך ביותר יהיה o(logn) וספציפית, לא יותר מ- . המספרים המתקבלים תואמים את הצפוי (עד כדי הבדל של פעולה יחידה בשתי התצפיות הראשונות), כפי שניתן לראות בעזרת הטבלה הבאה:

|  |  |  |
| --- | --- | --- |
| n | log(n) |  |
| 10,000 | 13.287712 | 14 |
| 20,000 | 14.287712 | 15 |
| 30,000 | 14.872675 | 15 |
| 40,000 | 15.287712 | 16 |
| 50,000 | 15.60964 | 16 |
| 60,000 | 15.872675 | 16 |
| 70,000 | 16.095067 | 17 |
| 80,000 | 16.287712 | 17 |
| 90,000 | 16.457637 | 17 |
| 100,000 | 16.60964 | 17 |

1. כמות פעולות האיזון לפעולת מחיקה קטן מכמות פעולות האיזון לפעולת הכנסה, לכן, בהמשך לסעיף הקודם, מספר פעולות האיזון בוודאי עומד בחסם o(logn) וספציפית לא גדול מ- .