**צומת**

פונקציות מחלקה:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| אינדקס | פונקציה | תיאור הפונקציה | פונקציות עזר בשימוש | סיבוכיות זמן ריצה | הסבר לסיבוכיות |
|  | \_\_init\_\_(self, key, value) |  |  | O(1) |  |
|  | get\_key(self) |  |  | O(1) |  |
|  | get\_value(self) |  |  | O(1) |  |
|  | get\_left(self) |  |  | O(1) |  |
|  | get\_right(self) |  |  | O(1) |  |
|  | get\_parent(self) |  |  | O(1) |  |
|  | get\_height(self) |  |  | O(1) |  |
|  | get\_size(self) |  |  | O(1) |  |
|  | set\_key(self, key) |  |  | O(1) |  |
|  | set\_value(self, value) |  |  | O(1) |  |
|  | set\_left(self, node) |  |  | O(1) |  |
|  | set\_right(self, node) |  |  | O(1) |  |
|  | set\_parent(self, node) |  |  | O(1) |  |
|  | set\_height(self, h) |  |  | O(1) |  |
|  | set\_size(self, s) |  |  | O(1) |  |

פונקציות עזר:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| אינדקס | פונקציה | תיאור הפונקציה | פונקציות עזר בשימוש | סיבוכיות זמן ריצה | הסבר לסיבוכיות |
| 1 | create\_leaf\_with\_virtual\_nodes(key, value) | יוצר עלה בעץ שבניו וירטואליים |  | O(1) |  |
| 2 | compute\_balance\_factor(self) | מחשב את הBF של הצומת |  | O(1) |  |
| 3 | is\_empty\_node(node) | בודק האם הצומת היא None או צומת וירטואלי |  | O(1) |  |
| 4 | is\_leaf(self) | בודק האם הצומת הוא עלה |  | O(1) |  |
| 5 | has\_one\_child(self) | בודק האם לצומת ילד אחד בלבד |  | O(1) |  |
| 6 | is\_left\_child\_of\_parent(node) | בודק האם הצומת היא הבן השמאלי האב שלו |  | O(1) |  |
| 7 | update\_parents\_child(self, old\_child, new\_child) | בודקת אם old\_child הוא הבן הימני או השמאלי של self, ושמה את new\_child במקומו. |  | O(1) |  |
| 8 | is\_real\_node(self) | בודק אם הצומת הוא "אמיתי" או וירטואלי |  | O(1) |  |

**עץ**

פונקציות מחלקה:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| אינדקס | פונקציה | תיאור הפונקציה | פונקציות עזר בשימוש | סיבוכיות זמן ריצה | הסבר לסיבוכיות |
|  | \_\_init\_\_(self) |  |  |  |  |

פונקציות עזר:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| אינדקס | פונקציה | תיאור הפונקציה | פונקציות עזר בשימוש | סיבוכיות זמן ריצה | הסבר לסיבוכיות |
| 9 | set\_root(self, node) |  |  | O(1) |  |
| 10 | rotate\_right(self, old\_root: AVLNode) |  | 7 | O(1) |  |
| 11 | rotate\_left(self, old\_root: AVLNode) |  | 7 | O(1) |  |
| 12 | rotate\_left\_then\_right(self, old\_root: AVLNode) |  | 10,11 |  |  |
| 13 | rotate\_right\_then\_left(self, old\_root: AVLNode) |  | 10,11 |  |  |
| 14 | find\_successor\_for\_node\_with\_two\_childs(self, node: AVLNode) |  | 3 | O(logn) כגובה העץ |  |
| 15 | find\_parent\_for\_insert(self, key) |  | 8 | O(logn) כגובה העץ |  |
| 16 | find\_parent\_with\_illegal\_balance\_factor(self, node: AVLNode) | מחפשת צומת במעלה העץ (לכיוון השורש) שיש לו balance factor לא תקין. | 2 | O(logn) כגובה העץ | ״עולים למעלה בעץ״ מהצומת הנתון ועד צומת עם balance factor לא תקין. בכל צומת מבצעים פעולה בסיבוכיות O(1). המקרה הכי גרוע הוא שהמסלול הוא בגובה העץ – O(logn) ולכן הסיבוכיות של הפונקציה היא O(logn). |
| 17 | fix\_tree\_of\_illegal\_root(self, illegal\_root) | מתקנת את תת העץ שהשורש שלו הוא צומת עם balance factor לא תקין, באמצעות גילגולים. | 2, 10, 11, 12, 13 | O(1) |  |
| 18 | fix\_tree(self, node\_with\_illegal\_balance\_factor) | מתקנת את העץ ל-AVL לאחר שבוצעו בו שינויים, מצומת שעלול להיות בעייתי ולמעלה עד לשורש העץ | 16, 17 | O(logn) כגובה העץ | ״עולים למעלה בעץ״ (ה״עלייה״ עצמה מתבצעת בפונקציה 16 שמחפשת צומת עם balance factor לא תקין), אך במהלך כל הפונקציה עוברים על כל צומת במסלול למעלה רק פעם אחת. עבור כל צומת מבצעים פעולה ב-O(1). המסלול הארוך ביותר הוא מסלול בגובה העץ – O(logn), וזוהי הסיבוכיות של הפונקציה. |
| 19 | physical\_insert(self, leaf\_for\_insert: AVLNode) |  | 15 | O(logn) כגובה העץ |  |
| 20 | physical\_delete(self, node: AVLNode) |  | 8, 6 | O(1) |  |
| 21 | replace\_node\_in\_tree(self, old\_node: AVLNode, new\_node: AVLNode) |  | 9, 7 | O(1) |  |
| 22 | avl\_to\_array\_rec(self, node: AVLNode, array) |  | 3, 22 | O(n) |  |
| 23 | create\_tree(self, root) | יוצרת עץ AVL מהתת עץ שהשורש שלו הוא הצומת הנתון. | 9 | O(1) |  |
| 24 | join\_trees\_with\_equal\_heights(self, t1, t2, x) | מאחדת שני עצים כששניהם באותו הגובה. במקרה זה אין צורך לבצע חיפוש, כל עץ הופך לבן של הצומת המקשר. |  | O(1) |  |
| 25 | join\_trees\_left\_tree\_is\_smaller(self, t1, t2, x) | איחוד עצים כשהעץ עם הערכים הקטנים הוא ״נמוך״ יותר מהעץ עם הערכים הגדולים. | 18 | O(logn) כגובה העץ | עוברים על הצמתים הכי שמאליים בעץ הגבוה, במקרה הגרוע עוברים על גובה עץ בגודל n שהוא O(logn).  בנוסף הקריאה ל-fix\_tree (18) היא בסיבוכיות O(logn). |
| 26 | join\_trees\_right\_tree\_is\_smaller(self, t1, t2, x) | איחוד עצים כשהעץ עם הערכים הגדולים הוא ״נמוך״ יותר מהעץ עם הערכים הקטנים. | 18 | O(logn) כגובה העץ | עוברים על הצמתים הכי ימניים בעץ הגבוה, במקרה הגרוע עוברים על גובה עץ בגודל n שהוא O(logn).  בנוסף הקריאה ל-fix\_tree (18) היא בסיבוכיות O(logn). |
| 27 | join\_tree\_with\_array\_of\_node\_tuples(self, array\_of\_node\_tuples) | מקבלים רשימה של זוגות – כל זוג מכיל צומת מקשר ועץ. הפונקציה מאחדת את self עם העצים ברשימה כשהצמתים המקשרים מקשרים ביניהם. | קורא פונקציה join, 23 | O(logn) כגובה העץ | יש קריאה ל-join O(logn) פעמים עם תתי עצים שיש לאחד לצורך מימוש split. מכיוון שתתי העצים אינם תתי עצים שכל אחד מהם הוא בגובה O(logn), כי הם נוצרו בתהליך split של עץ יחיד, נסיק מניתוח הסיבוכיות שעשינו בשיעור שהסיבוכיות של הפונקציה היא O(logn). |

פונקציות שלהם:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| אינדקס | פונקציה | תיאור הפונקציה | פונקציות עזר בשימוש | סיבוכיות זמן ריצה | הסבר לסיבוכיות |
| 1 | search(self, key) |  | 8 | O(logn) כגובה העץ |  |
| 2 | insert(self, key, val) |  | 1, 9, 18, 19 | O(logn) כגובה העץ |  |
| 3 | delete(self, node: AVLNode) |  | 4, 5, 20, 14, 21, 18 | O(logn) כגובה העץ |  |
| 4 | avl\_to\_array(self) |  | 22 | O(n) |  |
| 5 | size(self) |  |  | O(1) |  |
| 6 | join(self, tree, key, val) | מאחדת שני עצים עם צומת מקשר ביניהם, כאשר  - בעץ אחד כל המפתחות קטנים מהצומת המקשר  - ובעץ השני כל המפתחות גדולים מהצומת המקשר | 1, 8, 24, 25, 26 | O(logn) כגובה העץ | קוראים פעם אחת לאחת מבין הפונקציות 24, 25, 26, כאשר הסיבוכיות של 25 ו-26 היא O(logn) |
| 7 | split(self, node: AVLNode) | מפצלת עץ לשני עצים לפי הצומת הנתון:  - עץ אחד מכיל את הצמתים עם המפתחות הקטנים מהמפתח של הצומת שקיבלנו - העץ השני מכיל את הצמתים עם המפתחות הגדולים מהמפתח של הצומת שקיבלנו | 23, 6, 27 | O(logn) כגובה העץ | ״עולים למעלה״ בעץ מהצומת שקיבלנו עד השורש – לולאה זו במקרה הגרוע היא בסיבוכיות גובה העץ – O(logn).  קוראים לפונקציה 27 פעם אחת, והסיבוכיות שלה היא O(logn). |
| 8 | rank(self, node) | מחזירה את ה״מיקום״ של הצומת כשממיינים את המפתחות של העץ לפי הסדר, כלומר את מספר האיברים בעץ שקטנים מהצומת שקיבלנו + 1. |  | O(logn) כגובה העץ | מבצעים מעבר יחיד על המסלול בעץ שמוביל לצומת הנתון, לכן במקרה הגרוע נעבור על מסלול שהוא בגובה העץ – O(logn), ולכן זו הסיבוכיות של הפונקציה. |
| 9 | select(self, i) | מחזירה את הצומת שה-rank שלו הוא i. |  | O(logn) כגובה העץ | מבצעים מעבר יחיד על המסלול בעץ שמוביל לצומת עם ה-rank הנתון, לכן במקרה הגרוע נעבור על מסלול שהוא בגובה העץ – O(logn), ולכן זו הסיבוכיות של הפונקציה. |
| 10 | get\_root(self) | מחזירה את השורש של העץ |  | O(1) |  |

**חלק תאורטי**

**שאלה 1**

1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| מספר סידורי i | מספר חילופים במערך הממוין-הפוך | עלות מיון AVL עבור מערך ממוין-הפוך | מספר חילופים במערך מסודר אקראית | עלות מיון AVL עבור מערך מסודר אקראי | מספר החילופים במערך כמעט ממוין | עלות מיון AVL עבור מערך כמעט ממוין |
| 1 | 4498500 | 58825 | 2265280 | 50505 | 403650 | 38248 |
| 2 | 17997000 | 129656 | 9039491 | 107903 | 852150 | 88627 |
| 3 | 71994000 | 283319 | 35984560 | 246931 | 1749150 | 189385 |
| 4 | 287988000 | 614646 | 143828004 | 565531 | 3543150 | 390904 |
| 5 | 1151976000 | 1325301 | 577191993 | 1223481 | 7131150 | 793943 |

חישוב מספר החילופים:

חישבנו לכל איבר חדש שהוספנו לעץ עם כמה איברים הוא מתחלף מתוך האיברים שכבר הוכנסו, וסכמנו את התוצאות עבור כלל האיברים. כך ספרנו כל זוג בדיוק פעם אחת.

כדי לחשב לאיבר חדש v עם כמה איברים הוא מתחלף, השתמשנו בנוסחא הבאה:

כדי לחשב את הדרגה של האיבר, מצאנו את האיבר הראשון בענף הכי ימני של העץ עם מפתח שגדול ממנו (אם קיים כזה), וחיסרנו את גודל תת העץ הימני של צומת זה מגודל העץ כולו. משם המשכנו בחיפוש המיקום הספציפי שאליו יש להכניס את הצומת, ובדרך החסרנו מה-rank בכל פעם שפנינו שמאלה בעץ, בדומה לתהליך חישוב ה-rank מהשורש.

1. להשלים ניתוח מספר חילופים – סתיו

ניתוח תיאורטי של עלות החיפושים של ה-AVL במקרה של מערך ממוין-הפוך:

לאור ההנחיה בשאלה שעל החיפוש להתחיל מהאיבר המקסימלי, עבור מערך ממוין הפוך, בכל פעם שאנו מכניסים צומת לעץ, המפתח שלה יהיה קטן משל כל שאר הצמתים שכבר נמצאים בעץ (כמובן, עבור המקרה בו העץ הוא ריק ואז הצומת שנכנסת תהפוך להיות שורש העץ). לכן, על מנת להכניס את הצומת, נצטרך לעלות עד לשורש העץ וממנו לרדת עד לצומת העץ המחזיקה במפתח הקטן ביותר, ורק לאחר מכן נכניס את הצומת לעץ, בתור בן שמאלי של הצומת המחזיק במפתח המינימלי.

חסם תחתון :בתרגול 4 על עצי AVL ראינו שגובה של עץ AVL מושלם בעל n צמתים שבו כל הרמות מלאות פרט לרמה התחתונה הוא . בהתאמה להסבר שכתוב למעלה, כשאנחנו מכניסים את הצומת עם המפתח ה לעץ, יהיו בעץ כבר צמתים ולכל הפחות, נעלה לשורש העץ ונבצע "עבודה". נשים לב, ש כי log לא מוגדרת עבור הערך 0 ו כי אז יוצא שאנחנו מבצעים 0 "עבודה", ולכן, נניח שעבור מקרים אלו אנו מבצעים 1 "עבודה". כעת, נקבל את הנוסחא הבאה לחישוב סיבוכיות זמן הריצה:

*כאשר המעבר \* נובע מתרגול 1 בו ראינו ש .*

*לכן, החסם התחתון של* עלות החיפושים של ה-AVL במקרה של מערך ממוין-הפוך הוא *.*

חסם עליון :

בתרגיל עיוני 2, ראינו שגובהו של עץ AVL חסום על ידי הביטוי כשn הוא מספר הצמתים בעץ. כפי שציינו למעלה בכל הכנסה של צומת לעץ, אנחנו עולים עד השורש כגובה העץ, לאחר מכן, יורדים עד האיבר המינימלי שזה גם ירידה שהיא כגובה העץ, ולבסוף, מכניסים את הצומת – סך הכל סיבוכיות זמן הריצה היא O(h) כשh מוגדר להיות גובה העץ. בכל הכנסה של צומת לעץ, בעץ יהיו פחות מn צמתים. לכן, מהחסם שקיבלנו בתרגיל עיוני 2, h חסום על ידי , ולכן, נוכל לומר שבכל הכנסה לעץ אנחנו מבצעים "עבודה" בסיבוכיות זמן ריצה של O(. אנחנו חוזרים על פעולת ההכנסה של צומת n פעמים, ולכן, מבצעים n פעמים פעולות שסיבוכיות זמן הריצה שלהם היא O(. לכן, סיבוכיות זמן הריצה הכוללת של הכנסת n איברים ממערך ממוין הפוך לעץ AVL מתקבלת מהנוסחא:

*לכן, החסם העליון של* עלות החיפושים של ה-AVL במקרה של מערך ממוין-הפוך *הוא .*

*סך הכל, קיבלנו ש*עלות החיפושים של ה-AVL במקרה של מערך ממוין-הפוך הוא *.*

*2. להשלים התאמה של סעיף א' לב' וגרפים - סתיו*

**שאלה 2**

1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מספר סידורי i | עלות join ממוצע עבור split אקראי | עלות join מקסימלי עבור split אקראי | עלות join ממוצע עבור split של האיבר מקסימלי בתת העץ השמאלי | עלות join מקסימלי עבור split של איבר מקסימלי בתת העץ השמאלי |
| 1 | 2.7142857142857144 | 5 | 2.7 | 14 |
| 2 | 3.0 | 5 | 2.8181818181818183 | 15 |
| 3 | 3.0 | 5 | 2.6666666666666665 | 16 |
| 4 | 2.642857142857143 | 5 | 2.642857142857143 | 17 |
| 5 | 2.466666666666667 | 5 | 2.5714285714285716 | 19 |
| 6 | 3.142857142857143 | 6 | 2.8 | 19 |
| 7 | 2.3529411764705883 | 7 | 2.3333333333333335 | 20 |
| 8 | 2.6315789473684212 | 5 | 2.5294117647058822 | 22 |
| 9 | 2.3684210526315788 | 6 | 2.6842105263157894 | 23 |
| 10 | 2.6315789473684212 | 8 | 2.888888888888889 | 25 |

2. בביצוע הפעולה split, אנו עולים מעלה בעץ עד לשורש, החל מצומת שניתן כקלט לפונקציה, נקרא לו x, ובכל פעם שאנו עולים להורה, כלומר, בכל רמה, אנו בוחרים עץ אותו אנו מאחדים לאחד מהעצים החדשים שיתקבלו. לכן, אנו מאחדים את מבצעים join-ים כגובה העץ. מאחר שהעץ שלנו הוא עץ AVL, גובה העץ הוא O(log n), ולכן, אנו מבצעים O(log n) פעולות join. סה״כ הסיבוכיות של split היא O(log n), ולכן הסיבוכיות של כל join יחיד במהלך split היא O(1). ניתוח זה מתיישב עם התוצאות בניסוי, כי אנו רואים שהערך הממוצע של העלות של join היא קבועה (בין 2-3) ואינה תלויה ב-n.

3. במקרה של split עם האיבר המקסימלי בתת העץ השמאלי של השורש, ה-join המקסימלי יהיה join של תת העץ הימני של העלה (עץ ריק) עם תת העץ הימני של השורש (שהוא עץ בגובה העץ המקורי פחות 1), ביחס לשורש שיהיה ביניהם בסוף הריצה. לכן, במקרה זה, העלות של ה-join המקסימלי יהיה O(log n). תוצאות הניסוי מתיישבות עם הניתוח התיאורטי.