מחלקת AVLNode

|  |  |  |
| --- | --- | --- |
| שדה (חבר) | ערך דיפולטי | תפקיד |
| Key | none | מפתח של הצומת |
| Value | none | ערך של הצומת |
| left | none | צומת שמאלי (none אם לא קיים כזה) |
| Right | none | צומת ימני (none אם לא קיים כזה) |
| Parent | none | צומת הורה (none אם שורש ולא קיים הורה) |
| height | -1 | גובה (מספר הקשתות מהעלה הכי נמוך). |
| size | 0 | גודל תת העץ של הצומת (כולל עצמו) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | תפקיד | ערך החזרה | תיאור הלוגיקה | סיבוכיות |
| Get\_key(self) | מחזיר את המפתח של הצומת self (none אם וירטואלי) | Self.key |  |  |
| Get\_value(self) | מחזיר את הערך של הצומת (none אם וירטואלי) | Self.value |  |  |
| Get\_left(self) | מחזיר את הצומת השמאלי (none אם לא קיים כזה) | Self.left |  |  |
| Get\_right(self) | מחזיר את הצומת הימני (none אם לא קיים כזה) | Self.right |  |  |
| Get\_parent(self) | מחזיר את ההורה (none אם לא קיים כזה) | Self.parent |  |  |
| Get\_height(self) | מחזיר את הגובה של תת העץ של הצומת (-1 אם וירטואלי) | Self.height |  |  |
| Get\_size(self) | מחזיר את הגובה של תת העץ של הצומת (0 אם וירטואלי) | Self.size |  |  |
| Set\_key(self, key) | מגדיר מפתח של צומת | self |  |  |
| Set\_value(self, value) | מגדיר ערך של צומת | Self |  |  |
| Set\_left(self, node) | מגדיר צומת כבן שמאלי | Self |  |  |
| Set\_right(self, node) | מגדיר צומת כבן ימני | Self |  |  |
| Set\_parent(self, node) | מגדיר צומת כהורה | Self |  |  |
| Set\_height(self, height) | מגדיר גובה לתת העץ של הצומת | Self |  |  |
| Set\_size(self, size) | מגדיר גודל לתת העץ של הצומת | Self |  |  |
| Is\_real\_node(self) | מחזיר האם הצומת אמיתי או וירטואלי | Self.key!=none |  |  |
| Convert\_node\_to\_real(self) | נקראת לאחר יצירת צומת חדש כדי להפוך אותו לצומת תקין שאינו וירטואלי |  | מגדירה ילדים וירטואלים (ומקשרת אותם להורה שלהם)  מגדירה גובה 0  מגדירה גודל 1 |  |

מחלקת AVLTree

|  |  |  |
| --- | --- | --- |
| שם שדה (חבר) | ערך דיפולטי | תפקיד |
| root | none | שורש העץ |

פונקציות נדרשות –

הערה: הפונקציות הנדרשות משתמשות בפונ' עזר שמתועדות בהמשך המסמך (ולכן בחלק זה נשתמש בסיבוכיות שלהן כנתון, ובחלק הבא נוכיח סיבוכיות כל פונקציה).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | תפקיד | ערך החזרה | לוגיקה | סיבוכיות |
| Search(self,key) | מחזיר מצביע לצומת המתאים למפתח | Node (none אם לא קיים כזה) | 1. אם העץ ריק נחזיר none 2. נבדוק האם שורש העץ הוא הערך הנדרש    1. אם כן נחזיר אותו    2. אם הערך הנדרש קטן יותר מהשורש, נחזור לסעיף 1 עם הבן השמאלי    3. אם הערך הנדרש גדול יותר נחזור לסעיף 1 עם הבן הימני | כאשר n גודל העץ. זו הסיבוכיות במקרה הגרוע בגלל שגובה העץ חסום על ידי  *כפי שראינו בהרצאה* |
| Insert(self, key, value) | מכניסה צומת (עם המפתח והערך) לעץ באופן ששומר על תקינות העץ AVL | מספר פעולות הגלגול שבוצעו | 1. הכנסה רגילה לעץ בעזרת פונ' עזר insert\_node\_bst 2. בדיקת ההורה של הצומת הנכנס:    1. אם לא עבריין AVL, קרא לפונ' עזר update וחזור לסעיף 2 עם ההורה    2. אם כן עבריין, בצע גלגול, ואז חזור לסעיף 2 עם ההורה. 3. החזר כמה גלגולים בוצעו (ערך שחוזר מפונ' העזר rotate) | ההכנסה היא logn עבודה.  סעיף 2 בודק את כל הצמתים עד השורש (אורך המסלול חסום על ידי logn)  לכל צומת מתבצע או גלגול או עדכון – ושניהם דורשים O(1) עבודה. |
| Delete(self, node) | מוחקת את הצומת הנתון | מספר פעולות הגלגול שבוצעו | 1. מחיקה רגילה בעזרת פונ' עזר delete\_node\_bst 2. בדיקת ההורה של הצומת הנמחק    1. אם לא עבריין AVL קרא לפונ' update וחזור לסעיף 2 עם ההורה    2. אם כן עבריין בצע גלגול ואז חזור לסעיף 2 עם ההורה 3. החזר כמה גלגולים בוצעו | המחיקה הרגילה היא logn עבודה.  סעיף 2 בודק את כל הצמתים עד השורש (אורך המסלול חסום על ידי logn)  לכל צומת מתבצע או גלגול או עדכון – ושניהם דורשים O(1) עבודה. |
| Avl\_to\_array(self) | מחזירה מערך עם איברי העץ בסריקת inorder | מערך כנדרש | 1. צור מערך ריק 2. קרא לפונ' העזר inorder\_rec שתמלא את המערך 3. החזר המערך המלא | כסיבוכיות פונ' העזר שנדרשת לעבור על כל צומת פעם אחת |
| Size(self) | מחזירה את הגודל של שורש העץ  אם אין השורש לא הוגדר (עץ ריק) מחזירה 0 | מספר הצמתים תחת השורש (כולל עצמו) |  | Size הוא שדה של הצומת והשורש הוא שדה של העץ ולכן סה"כ מתבצעות רק שליפות שדות |
| Split(self, node) | מפרידה את העץ לשני עצי AVL כך שבראשון יש רק צמתים שמפתחותיהם קטנים מnode ובשני גדולים מnode | מערך עם שני העצים | 1. הגדר את תת העץ הימני של node להיות "עץ גדולים יותר" ואת תת העץ השמאלי להיות "עץ קטנים יותר" (אם לא קיים אחד הבנים אז יווצר עץ ריק) 2. עולים במורד העץ    1. בכל פנייה שמאלה לצומת x, מאחדים בין "עץ קטנים יותר" והבן השמאלי של x כאשר x הצומת המחבר    2. בכל פנייה ימינה לצומת x מאחדים בין "עץ גדולים יותר" והבן הימני של x כאשר x הצומת המחבר 3. החזר את "עץ קטנים יותר" ואת "עץ גדולים יותר" | כל איחוד של עצים עולה עבודה של הפרש הגבהים בינהם. סך כל העבודה יוצא טור טלסקופי כפי שראינו בהרצאה שסכומו O(logn) |
| Join(self, tree, key, value) | מאחדת את tree לתוך self בעזרת צומת מחבר בעל המפתח והערך הנתונים | הפרש הגבהים בין העצים + 1 | 1. אם אחד העצים ריק, הכנס את הצומת המחבר לתוך השני (בעזרת insert) וזה העץ המאוחד 2. אם self גבוה יותר מtree    1. אם המפתחות של self גדולים יותר משל tree – הכנס את tree לתוך self בעזרת פונ' עזר join\_to\_left    2. אחרת, הכנס את tree לתוך self בעזרת join\_to\_right    3. Self הוא העץ המאוחד הנדרש 3. אחרת (tree גבוה יותר מself)    1. אם המפתחות של self גדולים יותר משל tree – אחד את self לתוך tree בעזרת join\_to\_right    2. אחרת, אחד את self לתוך tree בעזרת join\_to\_left    3. Tree הוא העץ המאוחד הנדרש, הזזת המצביע של self להיות tree | הסיבוכיות היא הפרש הגבהים כי זאת הסיבוכיות של פונקציות העזר.  מלבדן, פונקציה זו עושה מספר פעולות קבוע בלבד |
| Rank(self, node) | מחזירה את המיקום בעץ של הצומת | המיקום (לפי המפתח) | סוכמים את תת גודל תת העץ השמאלי (אם לא קיים כזה אז אפס).  עולים מהצומת לכיוון השורש  בכל פנייה שמאלה, מוסיפים לסכימה את גודל תת העץ השמאלי של הצומת שהגענו אליו (מימין) + 1 (כדי לספור גם את הצומת הנוכחי). | נבצע לכל היותר עלייה אחת במורד העץ, שגובהו חסום על ידי logn.  בכל צומת נבצע O(1) פעולות עבור הסכימה. |
| Select(self, i) | מחזירה את הצומת במיקום הi (לפי מפתח) בעץ | הצומת הנדרש | קוראת לפונ' העזר select\_rec עם העץ המלא (שמתחיל בשורש) | כי זו הסיבוכיות של פונ' העזר שמבצעת לכל היותר ירידה אחת במורד העץ |
| Get\_root(self) | מחזירה את שורש העץ | השורש |  | השורש הוא שדה של העץ |

פונקציות עזר –

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם | תפקיד | ערך החזרה | תיאור לוגיקה | סיבוכיות |
| Insert\_node\_bst(self, key, val) | הכנסה לעץ חיפוש בינארי רגיל | מצביע לצומת החדש שנוצר | 1. אם העץ ריק – הכנס את הצומת כשורש  2. אם העץ צומת יחיד- הכנס שמאלה אם קטן יותר או ימינה אם גדול יותר  3. אחרת  3.1 אם הצומת בעל מפתח קטן יותר, חזור לסעיף 2 עם תת העץ השמאלי  3.2 אם הצומת בעל מפתח גדול יותר, חזור לסעיף 2 עם תת העץ הימני | כאשר n גודל העץ. זו הסיבוכיות במקרה הגרוע בגלל שגובה העץ חסום על ידי  *כפי שראינו בהרצאה* |
| Delete\_leaf(self, node) | מחיקה של עלה | none | מגדירים להורה של node ילדים וירטואלים |  |
| Delete\_one\_child(self, node) | מחיקה של צומת שיש לו בן אחד וירטואלי ובן שני עם תוכן | none | מחברים את ההורה של הצומת לבן היחיד של הצומת |  |
| Get\_successor(self,node) | מחזירה את הצומת בעל הערך הגדול הכי קרוב לצומת. אם הצומת הוא המקסימלי בעץ – מחזירים none | AVLNode | אם יש לצומת בן ימני – הולכים ימינה ועד הסוף שמאלה  אחרת – עולים למעלה עד הפנייה הראשונה ימינה.  (זהה למימוש שהוצג בהרצאה | זו הסיבוכיות במקרה הגרוע בגלל שגובה העץ חסום על ידי  *כפי שראינו בהרצאה* |
| Delete\_node\_bst(self, node) | מחיקה מעץ חיפוש בינארי | none | אם הצומת עלה – מוחקת בעזרת הפונקציה delete\_leaf  אם הצומת בעל בן יחיד – מוחקים בעזרת delete\_one\_child  אחרת – מוצאים את הsuccessor של הצומת, מוחקים אותו (הוא בהכרח יהיה עלה או צומת בעל בן יחיד) ומציבים במקומו את התוכן של הצומת | זו הסיבוכיות במקרה הגרוע כיוון שלכל היותר נבצע פעולת successor אחת שזו העלות שלה. |
| Calculate\_bf(self, node) | חישוב BF של הצומת | BF | הפרש הגבהים בין הבן השמאלי לבן הימני | כי גובה הוא שדה של הצומת |
| Update(self, node) | מעדכנת שדות גובה וגודל של הצומת הנתון (לפי ערכי ילדיו) | none | מגדירים גודל להיות 1+ סכום גדלי הילדים  מגדירים גובה להיות 1+ מקסימום גבהי הילדים | משתמשת בשליפת שדות בלבד |
| Rotate(self, criminal\_node, criminal\_node\_bf) | מבצעת גלגול/ים מתאים מהצומת העבריין | מספר פעולות הגלגול שבוצעו | מפנה לגלגול שמאלה/ימינה/גם וגם לפי הערכים שהוצגו בהרצאה.  למשל – עבור bf=2 של הצומת וbf=1 של הבן השמאלי מבצעים גלגול ימינה | מבצעת לכל היותר שתי פעולות גלגול שגם הן בסיבוכיות הזאת. |
| Right\_rotation(self, node) | מבצעת גלגול אחד ימינה | none | שינוי מצביעים כך שהבן השמאלי של הצומת "יעלה" למעלה, הצומת יהפוך להיות בנו הימני (והילדים שלהם זזים בהתאם).  קריאה ל update עבור הצומת ובנו השמאלי (כי הם היחידים שעשויים לשנות גובה) | הזזת מצביעים (מספר פעולות קבוע) וקריאה לupdate שהוא גם |
| Left\_rotation(self, node) | מבצעת גלגול אחד שמאלה | none | סימטרי לגלגול ימינה | מאותו שיקול של גלגול ימינה |
| Inorder\_rec(self, node, avl\_list) | ממלאת את הרשימה באיברי העץ בסריקת inorder | None  אין צורך להחזיר כי הפונקציה עורכת את הרשימה | קוראים לפונקציה עם הבן השמאלי של הצומת  מכניסים לרשימה את הצומת  קוראים לפונקציה עם הבן הימני של הצומת | עוברים על כל צומת בעץ פעם אחת ומבצעים בו עבודה של |
| Select\_rec(self, node, i) | מחפשת את הצומת במיקום ה- i בתת העץ של node | AVLNode | אם node הוא הצומת הנדרש, מחזירים אותו.  אם i קטן מהמיקום של node - קוראים רקורסיבית לפונקציה שוב עם תת העץ השמאלי שלו.  אחרת – קוראים לפונקציה עם תת העץ הימני (ומחפשים ערך i מתאים) | נבצע לכל היותר ירידה אחת מלאה במורד העץ, וגובהו חסום על ידי logn |
| Join\_to\_left(self, tree, key, value) | מאחדת לתוך self עץ נמוך יותר בעל מפתחות קטנים יותר בעזרת צומת מחבר (שהמפתח שלו קטן ממפתחות self וגדול ממפתחות tree) | הפרש גבהי העצים + 1 | 1. יורדת לאורך ענף שמאל עד לצומת בגובה של tree (נסמנו b) 2. משנה מצביעים שהבן השמאלי של המחבר יהיה tree, הבן הימני שלו יהיה b וההורה שלו יהיה ההורה הקודם של b 3. מבצעת גלגולים (במידת הצורך) ועדכונים (תמיד) מהצומת המחבר עד לשורש העץ | כמות העבודה היא כהפרש הגבהים כיוון ש:  זאת כמות הצמתים שיהיה צריך לרדת במורד העץ  שינוי המצבעים הוא מספר קבוע של פעולות  גלגולים ועדכונים למעלה דורשים כמות קבועה של פעולות לכל צומת במסלול עד השורש (ומסלול זה באורך הפרש הגבהים). |
| Join\_to\_right(self, tree, key, value) | מאחדת לתוך self עץ נמוך יותר בעל מפתחות גדולים יותר בעזרת צומת מחבר (שהמפתח שלו גדול ממפתחות self וקטן ממפתחות tree) | הפרש גבהי העצים + 1 | זהה לפונקציה הקודמת (איחוד משמאל) | מאותו שיקול כמו הפונקציה הקודמת (איחוד משמאל) |