מבני נתונים – תרגיל מעשי 1

מגיש 1: עמית רוקח

תעודת זהות 1: 322853813

מגישה 2: סתיו פרידמן

תעודת זהות: TODO

**תיעוד**

**AVLNode**

* מתודה: \_\_init\_\_  
  קלט:
  1. key
     + טיפוס: int או None.
     + תיאור: המפתח של הצומת שתיווצר.
  2. value
     + טיפוס: any.
     + תיאור: המידע שתכיל הצומת שתיווצר.

פלט: אין.  
תיאור: בנאי של המחלקה AVLNode  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות.

* מתודה: create\_leaf\_with\_virtual\_nodes   
  קלט:
  1. key
     + טיפוס: int או None
     + תיאור: המפתח של הצומת שתיווצר
  2. value
     + טיפוס: any
     + תיאור: המידע שתכיל הצומת שתיווצר

פלט:

* + טיפוס: AVLNode
  + תיאור: הצומת שנוצרה עם המפתח key והערך value, עם שני בנים וירטואליים.

תיאור: מייצרת צומת עם המפתח key והערך value ועם שני בנים וירטואליים.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות וקוראת לבנאי המחלקה מספר קבוע של פעמים שסיבוכיות זמן הריצה שלו היא O(1).

* מתודה: compute\_size\_and\_update\_property  
  קלט: אין.

פלט: אין.

תיאור: מחשבת את הsize של self על בסיס הsize של הבן ימני שלו והsize של הבן השמאלי שלו.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות וסכומים.

* מתודה: compute\_balance\_factor  
  קלט: אין.

פלט:

* + טיפוס: int
  + תיאור: הbalance\_factor של self.

תיאור: מחשבת את הbalance\_factor של self, ומוודאת שהheight והsize שלו מותאמים עם של בניו על ידי עדכון השדות.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות וחישובים. בנוסף, המתודה קוראת למתודות compute\_size\_and\_update\_property, is\_real\_node פעם אחת וסיבוכיות זמן הריצה שלהן היא O(1).

* מתודה: is\_empty\_node  
  קלט:
  1. node
     + טיפוס: AVLNode או None
     + תיאור: צומת לבדיקה

פלט:

* + טיפוס: bool
  + תיאור: מחזירה True עם הצומת היא None או צומת וירטואלית וFalse אחרת.

תיאור: בודקת אם node הוא None או צומת וירטואלית.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השוואות. המתודה קוראת גם מתודה is\_real\_node פעם אחת שסיבוכיות זמן הריצה שלה היא O(1).

* מתודה: is\_leaf  
  קלט: אין.

פלט:

* + טיפוס: bool
  + תיאור: מחזירה True אם שני הבנים של self הם None או צמתים וירטואליים (שילוב של None וצומת וירטואלי גם יחזיר True) וFalse אחרת.

תיאור: בודקת אם שני הבנים של self הם None או צמתים וירטואליים, כאשר שילוב של None וצומת וירטואלי גם תופס.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות וקוראת פעמיים למתודה is\_empty\_node שסיבוכיות זמן הריצה שלה היא O(1).

* מתודה: has\_one\_child  
  קלט: אין.

פלט:

* + טיפוס: bool
  + תיאור: מחזירה True אם לself יש בן אדם אחד בלבד שאינו None או צומת וירטואלית ומחזירה False אחרת.

תיאור: בודקת אם בן אחד בלבד של self אינו None או צומת וירטואלית.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות וקוראת פעמיים למתודה is\_empty\_node שסיבוכיות זמן הריצה שלה היא O(1).

* מתודה: is\_left\_child\_of\_parent  
  קלט: אין.

פלט:

* + טיפוס: bool
  + תיאור: מחזירה True אם self הוא הבן השמאלי של האב שלו וFalse אחרת.

תיאור: בודקת אם self הוא הבן השמאלי של האב שלו.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות והשוואות.

* מתודה: update\_parents\_child  
  קלט:
  1. old\_child
     + טיפוס: AVLNode
     + תיאור: הצומת שצריך להחליף.
  2. new\_child
     + טיפוס: AVLNode
     + תיאור: הצומת שתחליף את old\_child.

פלט: אין.  
תיאור: בודקת אם old\_child הבן השמאלי של האב שלו ומעדכנת את new\_child להיות במקום old\_child.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השוואות. בנוסף, המתודה קוראת למתודות is\_left\_child\_of\_parent, set\_left/set\_right פעם אחת בלבד וסיבוכיות זמן הריצה שלהן היא O(1).

* מתודה: get\_key  
  קלט: אין.

פלט:

* + טיפוס: int או None
  + תיאור: השדה key של self.

תיאור: מחזירה את השדה key של self.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מחזירה שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: get\_value  
  קלט: אין.

פלט:

* + טיפוס: any
  + תיאור: השדה value של self.

תיאור: מחזירה את השדה value של self.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מחזירה שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: get\_left  
  קלט: אין.

פלט:

* + טיפוס: AVLNode או None
  + תיאור: השדה left של self.

תיאור: מחזירה את השדה left של self.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מחזירה שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: get\_right  
  קלט: אין.

פלט:

* + טיפוס: AVLNode או None
  + תיאור: השדה right של self.

תיאור: מחזירה את השדה right של self.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מחזירה שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: get\_parent  
  קלט: אין.

פלט:

* + טיפוס: AVLNode או None
  + תיאור: השדה parent של self.

תיאור: מחזירה את השדה parent של self.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מחזירה שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: get\_height  
  קלט: אין.

פלט:

* + טיפוס: int
  + תיאור: השדה height של self.

תיאור: מחזירה את השדה height של self (אם self הוא צומת וירטואלי יוחזר -1).  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מחזירה שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: get\_size  
  קלט: אין.

פלט:

* + טיפוס: int
  + תיאור: השדה size של self.

תיאור: מחזירה את השדה size של self (אם self הוא צומת וירטואלית יוחזר 0).  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מחזירה שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: set\_key  
  קלט:
  1. key
     + טיפוס: int או None.
     + תיאור: מפתח חדש לself.

פלט: אין.  
תיאור: מעדכנת את השדה key של self להיות key שניתן כקלט.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מעדכנת שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: set\_value  
  קלט:
  1. value
     + טיפוס: any.
     + תיאור: מידע חדש לself.

פלט: אין.  
תיאור: מעדכנת את השדה value של self להיות value שניתן כקלט.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מעדכנת שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: set\_left  
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: בן שמאלי חדש לself.

פלט: אין.  
תיאור: מעדכנת את השדה left של self להיות node שניתן כקלט ומעדכנת את node כך שself יהיה האב שלו בעזרת המתודה set\_parent.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מעדכנת שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1) וקוראת למתודה set\_parent שסיבוכיות זמן הריצה שלה היא O(1).

* מתודה: set\_right  
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: בן ימני חדש לself.

פלט: אין.  
תיאור: מעדכנת את השדה right של self להיות node שניתן כקלט ומעדכנת את node כך שself יהיה האב שלו בעזרת המתודה set\_parent.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מעדכנת שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1) וקוראת למתודה set\_parent שסיבוכיות זמן הריצה שלה היא O(1).

* מתודה: set\_parent  
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: הצומת שתהיה האב של self בסיום ריצת המתודה.

פלט: אין.  
תיאור: מעדכנת את השדה parent של self להיות node שניתן כקלט, או None אם node הוא None או צומת וירטואלית. בהנחה שnode אינו None או צומת וירטואלית: מעדכנת גם את השדות size וheight של node בהתאם למיקומו הנוכחי בעץ.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות, סכומים ומעדכנת שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1). בנוסף, המתודה קוראת למתודות is\_empty\_node, set\_height\_set\_size שסיבוכיות זמן הריצה שלהן היא O(1).

* מתודה: set\_height  
  קלט:
  1. h
     + טיפוס: int.
     + תיאור: גובה חדש לself.

פלט: אין.  
תיאור: מעדכנת את השדה height של self להיות h שניתן כקלט.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מעדכנת שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: set\_size  
  קלט:
  1. s
     + טיפוס: int.
     + תיאור: גודל חדש לself.

פלט: אין.  
תיאור: מעדכנת את השדה size של self להיות s שניתן כקלט.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מעדכנת שדה של self שהגישה אליו היא בסיבוכיות זמן ריצה של O(1).

* מתודה: is\_real\_node  
  קלט: אין.

פלט:

* + טיפוס: bool.
  + תיאור: מחזירה False אם self הוא node וירטואלי, אחרת מחזירה True

תיאור: בודקת אם המפתח של self שונה מNone, כלומר הוא לא צומת וירטואלי.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השוואות ופונה לשדה של self שסיבוכיות זמן הריצה בגישה אליו היא O(1).

**AVLTree**

* מתודה: \_\_init\_\_  
  קלט: אין.

פלט: אין.  
תיאור: בנאי של המחלקה AVLTree.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות.

* מתודה: set\_root

קלט:

* 1. node
     + טיפוס: AVLNode או None
     + תיאור: הצומת שתשמש כהשורש החדש של self בסוף ריצת המתודה.

פלט: אין.  
תיאור: מעדכנת את node להיות השורש של self.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מעדכנת את שדה root של self שסיבוכיות זמן הריצה בגישה אליו היא O(1) וקוראת למתודה set\_parent שסיבוכיות זמן הריצה שלה היא O(1).

* מתודה: rotate\_right

קלט:

* 1. old\_root
     + טיפוס: AVLNode
     + תיאור: הצומת שעל תת העץ שלה נעשה רוטציה ימינה.

פלט: אין.  
תיאור: מבצעת רוטציה ימינה של תת העץ של old\_root.  
הרוטציה ימינה מקדמת הבן השמאלי (נקרא לו new\_root) של old\_root, להיות במקומו, כך שכעת old\_root הוא הבן הימני של new\_root והאב של new\_root הוא האב שהיה לold\_root לפני השינוי. בנוסף, תת העץ הימני של new\_root לפני השינוי, הופך להיות תת העץ השמאלי של old\_root. במקרה בו old\_root היה השורש שלself, אנחנו מעדכנים את new\_root להיות השורש של self.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות והשוואות, ובנוסף, קוראת למתודות get\_left, get\_right, get\_parent. update\_parents\_child, set\_root, set\_left, set\_right שסיבוכיות זמן הריצה שלהן היא O(1), מספר קבוע של פעמים, ולכן, סיבוכיות זמן הריצה הכוללת היא O(1).

* מתודה: rotate\_left

קלט:

* 1. old\_root
     + טיפוס: AVLNode
     + תיאור: הצומת שעל תת העץ שלה נעשה רוטציה שמאלה.

פלט: אין.  
תיאור: מבצעת רוטציה שמאלה של תת העץ של old\_root.  
הרוטציה שמאלה מקדמת הבן הימני (נקרא לו new\_root) של old\_root, להיות במקומו, כך שכעת old\_root הוא הבן השמאלי של new\_root והאב של new\_root הוא האב שהיה לold\_root לפני השינוי. בנוסף, תת העץ השמאלי של new\_root לפני השינוי, הופך להיות תת העץ הימני של old\_root. במקרה בו old\_root היה השורש שלself, אנחנו מעדכנים את new\_root להיות השורש של self.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות והשוואות, ובנוסף, קוראת למתודות get\_left, get\_right, get\_parent. update\_parents\_child, set\_root, set\_left, set\_right שסיבוכיות זמן הריצה שלהן היא O(1), מספר קבוע של פעמים, ולכן, סיבוכיות זמן הריצה הכוללת היא O(1).

* מתודה: rotate\_left\_then\_right

קלט:

* 1. old\_root
     + טיפוס: AVLNode
     + תיאור: הצומת שעל תת העץ שלה נעשה את הרוטציות.

פלט: אין.  
תיאור: מבצעת left\_rotation על הבן השמאלי של old\_root ולאחר מכן right\_rotation על old\_root.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות. בנוסף, קוראת למתודות get\_left, rotate\_left, rotate\_right שסיבוכיות זמן הריצה שלהן היא O(1), מספר קבוע של פעמים, ולכן, סיבוכיות זמן הריצה הכוללת היא O(1).

* מתודה: rotate\_right\_then\_left

קלט:

* 1. old\_root
     + טיפוס: AVLNode
     + תיאור: הצומת שעל תת העץ שלה נעשה את הרוטציות.

פלט: אין.  
תיאור: מבצעת right\_rotation על הבן הימני של old\_root ולאחר מכן left\_rotation על old\_root.  
סיבוכיות זמן ריצה: O(1).  
הסבר לסיבוכיות: המתודה מבצעת מספר קבוע של השמות. בנוסף, קוראת למתודות get\_right, rotate\_left, rotate\_right שסיבוכיות זמן הריצה שלהן היא O(1), מספר קבוע של פעמים, ולכן, סיבוכיות זמן הריצה הכוללת היא O(1).

* מתודה: find\_successor\_for\_node\_with\_two\_childs   
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: צומת עם שני בנים (לא וירטואליים) שנחפש את הsuccessor שלה.

פלט:

* + טיפוס: AVLNode
  + תיאור: מחזירה את האב של הsuccessor.

תיאור: מוצאת את הsuccessor של node.  
המתודה הולכת לבן הימני של node (הוא קיים כי יש לnode 2 בנים לא וירטואליים) ומשם ממשיכה שמאלה, בתת העץ של הבן הימני, עד שהיא מגיע לצומת וירטואלית ואז מחזירה את האב שלה.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במקרה הגרוע, node יהיה השורש של העץ והsuccessor שלו יהיה בתחתית העץ, במקרה זה אנו רצים בלולאת while ומבצעים get\_left כגובה העץ - פעמים. בנוסף, אנו קוראים למתודות get\_right, get\_parent שסיבוכיות זמן הריצה של כל אחת מהן היא O(1), מספר קבוע של פעמים. לכן, סיבוכיות זמן הריצה הכוללת של המתודה היא .

* מתודה: search  
  קלט:
  1. key
     + טיפוס: int
     + תיאור: מפתח לחפש בעץ.

פלט:

* + טיפוס: AVLNode או None (בחתימה שניתנה לנו רשום שזה any, אבל, בקונטקסט של המטלה זה יהיה מה שרשמנו פה).
  + תיאור: אם קיים בעץ צומת שהמפתח שלה הוא key, צומת זו תוחזר, אחרת יוחזר None.

תיאור: המתודה יורדת בעץ החל מהשורש ומחפשת צומת עם המפתח key, באופן הבא:

* אם הגענו לצומת עם המפתח key, נחזיר אותה.
* אם המפתח של הצומת הנוכחית קטן מkey נרד ימינה.
* אם המפתח של הצומת הנוכחית גדול מkey נרד שמאלה.

כך, עד שמגיע לNone או לצומת וירטואלית. במקרה בו הגיע לNone או לצומת וירטואלית, יוחזר None.

סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במקרה הגרוע, key לא קיים בעץ ונרד החל מהשורש, עד לתחתית העץ בלולאת while בעזרת קריאה למתודות get\_right, get\_key, is\_real\_node, get\_left (בהתאם למפתח של הצומת עליה אנו נמצאים באותו הרגע) ומספר קבוע של השמות והשוואות. גובה עץ AVL הוא O(logn). סיבוכיות זמן הריצה של המתודות שמשתתפות בלולאת הwhile ושל מספר קבוע של השמות והשוואות היא O(1), ולכן, סיבוכיות זמן הריצה הכוללת של לולאת הwhile היא O(logn). חוץ מלולאת הwhile, אנחנו מבצעים מספר קבוע של השמות ומחזירים None – סיבוכיות זמן הריצה של O(1). לכן, סיבוכיות זמן הריצה הכוללת של המתודה היא O(logn).

* מתודה: insert  
  קלט:
  1. key
     + טיפוס: int
     + תיאור: מפתח שלא נמצא בעץ.
  2. val
     + טיפוס: any
     + תיאור: הערך שיאוחסן עם המפתח.

פלט:

* + טיפוס: int.
  + תיאור: מספר התיקונים (רוטציות) שבוצעו בעץ כדי לשמור על תכונת עצי AVL.

תיאור: המתודה מכניסה לעץ צומת עם המפתח key וערך val.  
המתודה מתחילה ביצירת עלה עם בנים וירטואליים שהמפתח שלו הוא key והערך שלו הוא val. לאחר מכן, אם הroot של self הוא None או צומת וירטואלי, אז העלה שיצרנו יהיה שורש העץ ויוחזר 0. אחרת, אנחנו מכניסים את הצומת לעץ בעזרת המתודה physical\_insert, ולאחר מכן, קוראים למתודה fix\_tree שמתקנת את העץ, החל מהאב של הצומת שהכנסנו. המתודה fix\_tree מחזירה לנו את כמות הרוטציות שביצענו במסגרת התיקון של העץ כך שישאר עץ AVL גם לאחר ההכנסה, וזה המספר גם שהמתודה מחזירה.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במקרה הגרוע, המתודה קוראת למתודות create\_leaf\_with\_virtual\_nodes, is\_empty\_node שסיבוכיות זמן הריצה שלהן היא O(1), ובנוסף, קוראת למתודות physical\_insert וfix\_tree שסיבוכיות זמן הריצה של כל אחת מהן היא במקרה הגרוע O(logn), ולכן, סיבוכיות זמן הריצה הכוללת היא O(logn).

* מתודה: find\_parent\_for\_insert  
  קלט:
  1. key
     + טיפוס: int
     + תיאור: מפתח לחפש בעץ.

פלט:

* + טיפוס: AVLNode
  + תיאור: האב של העלה הוירטואלי שבמקומו ניתן להכניס צומת עם המפתח key.

תיאור: המתודה יורדת בעץ החל מהשורש ומחפשת את העלה הוירטואלי שבמקומו ניתן להכניס צומת עם המפתח key, באופן הבא:

* אם המפתח של הצומת הנוכחית קטן מkey נרד ימינה.
* אם המפתח של הצומת הנוכחית גדול מkey נרד שמאלה.

כך, עד שנגיע לצומת וירטואלית, ולבסוף, נחזיר את האב שלה.

סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במקרה הגרוע, אנו יורדים לצומת וירטואלית מספר רמות כגובה העץ. במקרה זה, נרד החל מהשורש, עד לתחתית העץ בלולאת while בעזרת קריאה למתודות get\_right, get\_key, is\_real\_node, get\_left (בהתאם למפתח של הצומת עליה אנו נמצאים באותו הרגע) ומספר קבוע של השמות והשוואות. גובה עץ AVL הוא O(logn). סיבוכיות זמן הריצה של המתודות שמשתתפות בלולאת הwhile ושל מספר קבוע של השמות והשוואות היא O(1), ולכן, סיבוכיות זמן הריצה הכוללת של לולאת הwhile היא O(logn). חוץ מלולאת הwhile, אנחנו מבצעים מספר קבוע של השמות וקוראים למתודה get\_parent – סיבוכיות זמן הריצה של O(1). לכן, סיבוכיות זמן הריצה הכוללת של המתודה היא O(logn).

* מתודה: find\_parent\_with\_illegal\_balance\_factor  
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: צומת שהחל ממנה נעלה ונתחיל לחפש צומת עם Balance Factor "לא חוקי".

פלט:

* + טיפוס: AVLNode או None
  + תיאור: צומת עם Balance Factor "לא חוקי" במסלול שמהשורש לnode או None במקרה שאין כזו.

תיאור: המתודה עולה מnode עד לאב של שורש העץ (None) ובודקת אם קיימת צומת שהBalance Factor שלה "לא חוקי" קרי, גדול מ1. כשהמתודה מוצאת צומת מהסוג האחרון, היא מחזירה אותו. אחרת, המתודה תחזיר None.

סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במקרה הגרוע, node בתחתית העץ, ואין צומת במסלול מהשורש לnode שהBalance Factor גדול מ1. במקרה זה, לולאת הwhile תרוץ כגובה העץ פעמים - O(logn) פעמים – ובכל איטרציה תיקרא למתודות get\_parent, compute\_balance\_factor שסיבוכיות זמן הריצה שלהן היא O(1). בנוסף, בכל איטרציה של הלולאה אנו מבצעים מספר קבוע של השמות והשוואות – סיבוכיות זמן ריצה של O(1). לכן, סיבוכיות זמן הריצה הכוללת של לולאת הwhile היא O(logn), וסיבוכיות זמן הריצה הכוללת של המתודה גם היא O(logn), כי מעבר ללולאה אנחנו רק מחזירים ערך וסיבוכיות זמן הריצה של פעולה זו היא O(1).

* מתודה: fix\_tree\_of\_illegal\_root  
  קלט:
  1. illegal\_root
     + טיפוס: AVLNode או None
     + תיאור: צומת "לא חוקית" בעץ, שהBalance Factor שלה גדול מ1, או None.

פלט:

* + טיפוס: int
  + תיאור: מספר הרוטציות שביצענו בתת עץ של illegal\_root כדי לתקנו להיות עץ AVL חוקי.

תיאור: אם illegal\_root הוא None המתודה מחזירה 0 כי לא ביצענו רוטציות כלל. אחרת, הפונקציה קובעת את הרוטציות הנדרשות לאיזון תת העץ של illegal\_root על בסיס הBalance Factor שלו ושל בניו.

סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במקרה הגרוע, illegal\_root שונה מNone ונדרש לעשות רוטציה ימינה ואז שמאלה או לחילופין רוטציה שמאלה ואז ימינה. נניח בה"כ שאנו מבצעים רוטציה ימינה ואז שמאלה. במקרה זה, אנו קוראים למתודות compute\_balance\_factor, get\_right מספר קבוע של פעמים וגם פעם אחת למתודה rotate\_right\_then\_left, סיבוכיות זמן הריצה של מתודות אלו הוא O(1). בנוסף, אנו מבצעים מספר קבוע של השמות, השוואות ומחזירים ערך בסוף – סיבוכיות זמן ריצה של O(1). סך הכל, אנו מבצעים מספר קבוע של פעמים פעולות בסיבוכיות זמן ריצה של O(1), ולכן, סיבוכיות זמן הריצה הכוללת היא O(1).

* מתודה: fix\_tree\_of\_illegal\_root  
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: צומת שהחל ממנה נעלה למעלה ונתקן תתי עצים של צמתים שהBalance Factor שלהם לא חוקי קרי, גדול מ1.

פלט:

* + טיפוס: int
  + תיאור: מספר הרוטציות שביצענו סך הכל כדי לתקן את העץ החל מnode.

תיאור: המתודה עולה מעלה בעץ מnode, עד לאב של השורש (None) ובצורה איטרטיבית מתקנת את העץ בכל פעם שמוצאת צומת שהBalance Factor שלא לא חוקי.

סיבוכיות זמן ריצה: O(logn).  
הסבר לסיבוכיות: במקרה הגרוע, node ברמה הנמוכה ביותר של העץ, וכמעט בכל פעם שנעלה רמה בעץ, נמצא צומת עם Balance Factor לא חוקי (נסביר את השימוש במילה "כמעט" בהמשך ההסבר). אנו רצים בלולאת while מnode ועד לאב של השורש. כדי לעלות ולמצוא צומת שהBalance Factor שלה לא חוקי, אנחנו משתמשים במתודה find\_parent\_with\_illegal\_balance\_factor שסך הכל סיבוכיות זמן הריצה שלה בכל איטרציות לולאת הwhile יהיה O(logn) כי היא בסך הכל מעלה אותנו מעלה בעץ על ידי שימוש בget\_parent ומעבר לכך מבצעת השוואות שסיבוכיות זמן הריצה שלהן היא O(1) בכל פעם שמעלה אותנו רמה בעץ. בנוסף, בכל פעם שfind\_parent\_with\_illegal\_balance\_factor אכן מוצא צומת עם Balance Factor לא חוקי, תרוץ fix\_tree\_of\_illegal\_root שסיבוכיות זמן הריצה שלה היא O(logn). השימוש במילה "כמעט" קודם לכן, נובע מכך שראינו בשיעור שכשאנו מתקנים עץ באופן המתואר, מספר התיקונים בסופו של דבר יהיה קבוע כלומר, לא תלוי בn או בגובה העץ. לכן, fix\_tree\_of\_illegal\_root תרוץ מספר קבוע של פעמים. כלומר, קיבלנו במקרה הגרוע שfind\_parent\_with\_illegal\_balance\_factor תבצע סך הכל O(logn) "עבודה" וfix\_tree\_of\_illegal\_root תבצע סך הכל גם כן O(logn) "עבודה" מספר קבוע של פעמים. לכן, סיבוכיות זמן הריצה הכוללת של המתודה היא O(logn), כי מעבר לפעולות שציינו אנו מבצעים השמות שלא משפיעים על סיבוכיות זמן הריצה של לולאת הwhile ומחזירים ערך שסיבוכיות זמן הריצה של פעולה זו היא O(1).

* מתודה: physical\_insert  
  קלט:
  1. leaf\_for\_insert
     + טיפוס: AVLNode
     + תיאור: צומת להכניס לעץ

פלט:

* + טיפוס: AVLNode.
  + תיאור: האב של leaf\_for\_insert לאחר ההכנסה לעץ.

תיאור: המתודה מכניסה את leaf\_for\_insert לעץ.  
המתודה יורדת למטה בעץ ועל בסיס המפתח של leaf\_for\_insert, מכריעה איפה הוא צריך להיות.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במקרה הגרוע, leaf\_for\_insert צריך להיות בתחתית העץ. במקרה זה, אנו מבצעים קריאה למתודות get\_key, set\_left/set\_right שסיבוכיות זמן הריצה שלהם היא O(1), מספר קבוע של פעמים, ובנוסף, מבצעים מספר קבוע של השוואות והשמות שסיבוכיות זמן הריצה שלהן היא O(1). מעבר לכך, אנו קוראים למתודה find\_parent\_for\_insert שבמקרה הגרוע המתואר, תרוץ בסיבוכיות זמן ריצה של O(logn), פעם אחת. לכן, סיבוכיות זמן הריצה הכוללת היא O(logn).

* מתודה: delete  
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: צומת בעץ.

פלט:

* + טיפוס: int.
  + תיאור: מספר התיקונים (רוטציות) שבוצעו בעץ כדי לשמור על תכונת עצי AVL.

תיאור: המתודה מוחקת את node מהעץ.  
אם לnode יש בן יחיד או שהוא עלה, המתודה פשוט מוחקת אותו מהעץ ובסוף מתקנת את העץ החל מהאב שהיה לו לפני המחיקה. אחרת, המתודה מוצאת את הsuccessor של node, מוחקת את הsuccessor מהעץ, שמה אותו במקום node, ולבסוף, מתקנת את העץ החל מהאב שהיה לsuccessor לפני ההחלפה.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במקרה הגרוע, node הוא השורש, יש לו 2 בנים והsuccessor שלו נמצא בתחתית העץ. במקרה זה, ירוצו המתודות:

1. find\_successor\_for\_node\_with\_two\_childs – סיבוכיות זמן ריצה של O(logn).
2. physical\_delete – סיבוכיות זמן ריצה של O(1).
3. replace\_node\_in\_tree – סיבוכיות זמן ריצה O(1).
4. fix\_tree – סיבוכיות זמן ריצה O(logn).

כל אחת מהן, פעם אחת. מעבר לכך, אנו מבצעים מספר קבוע של השמות והשוואות שסיבוכיות זמן הריצה שלהן היא O(1). לכן, סיבוכיות זמן הריצה הכוללת היא O(logn).

* מתודה: physical\_delete  
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: צומת בעץ שבסוף ריצת המתודה כבר לא תהיה בעץ.

פלט:

* + טיפוס: AVLNode או None.
  + תיאור: האב של הצומת שנמחקה מהעץ.

תיאור: המתודה מוחקת node שיש לו לכל היותר בן אחד (צומת וירטואלית לא נחשבת בן של הצומת). המתודה מוחקת את node לפי המקרים הבאים:

* אם node הוא לא השורש של self – במקרה זה ננתק את הצומת מהעץ ואת הבן שלה נחבר לאב שהיה לnode לפני המחיקה, במקום בו node היה.
* אם node הוא השורש של self – אם לnode יש בן, הוא יהפוך להיות השורש, אחרת, נכווין השורש להצביע לNone.

סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במהלך ריצת המתודה אנחנו עושים מספר קבוע של השמות, השוואות וקוראים למתודות: get\_left, get\_right, get\_parent, is\_real\_node, set\_root, set\_left, set\_right מספר קבוע של פעמים. סיבוכיות זמן הריצה של המתודות והפעולות האחרות שתיארנו היא O(1). לכן, סיבוכיות זמן הריצה הכוללת של המתודה היא O(1).

* מתודה: replace\_node\_in\_tree  
  קלט:
  1. old\_node
     + טיפוס: AVLNode
     + תיאור: הצומת בעץ שנחליף.
  2. new\_node
     + טיפוס: AVLNode
     + תיאור: הצומת שתחליף את old\_node.

פלט: אין.

תיאור: המתודה מחליפה את old\_node עם new\_node תוך שמירה על הבנים והאב של old\_node כלומר, כעת, הבנים של old\_node יהיו הבנים של new\_node והאב של old\_node יהיה האב של new\_node. בנוסף, אם old\_node לא היה השורש, new\_node יהיה הבן הימני/השמאלי (תלוי מה old\_node היה) של האב במקום old\_node. אם old\_node היה השורש אנו מעדכנים את השורש להיות new\_node.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: במהלך ריצת המתודה אנחנו עושים מספר קבוע של השמות, השוואות וקוראים למתודות: set\_root, get\_parent, set\_left, set\_right, get\_left, get\_right ,update\_parents\_child מספר קבוע של פעמים. סיבוכיות זמן הריצה של הפעולות שציינו היא O(1), ולכן, סיבוכיות זמן הריצה הכוללת היא O(1).

* מתודה: avl\_to\_array  
  קלט: אין.

פלט:

* + טיפוס: list[Tuple (int, any)]
  + תיאור: רשימה ממוינת, לפי האיבר הראשון בtuple (key), של tuples מהצורה (key, value) המתארת את המפתחות והערכים בעץ.

תיאור: עוברת על העץ בעזרת פונקציית עזר רקורסיבית avl\_to\_array\_rec ויוצרת רשימה ממוינת, לפי האיבר הראשון בtuple (key), של tuples מהצורה (key, value) המתארת את המפתחות והערכים בעץ.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: המתודה יוצרת רשימה ריקה – סיבוכיות זמן ריצה של O(1) - קוראת למתודה avl\_to\_array\_rec עם הרשימה שיצרה - סיבוכיות זמן הריצה של avl\_to\_array\_rec היא O(n) כאשר n זה מספר האיברים self- ומחזירה את הרשימה – סיבוכיות זמן ריצה של O(1). לכן, סיבוכיות זמן הריצה הכוללת של המתודה היא O().

* מתודה: avl\_to\_array\_rec  
  קלט:
  1. node
     + טיפוס: AVLNode או None
     + תיאור: הצומת שמבצעים עליו את הפעולה בקריאה הנוכחית.
  2. array
     + טיפוס: list[Tuple(int, any)]
     + תיאור: רשימה אליה נדחוף tuples בקריאות הרקורסיביות.

פלט:

* + טיפוס: list[Tuple(int, any)]
  + תיאור: רשימה ממוינת, לפי האיבר הראשון בtuple (key), של tuples מהצורה (key, value) המתארת את המפתחות והערכים בעץ.

תיאור: עוברת על העץ בצורת in-order ובסוף מחזירה רשימה ממוינת ,לפי האיבר הראשון בtuple (key), של tuples מהצורה (key, value) שמתארת את המפתחות והערכים בעץ.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: המתודה עוברת באופן רקורסיבי על כל הצמתים בעץ. בכל, קריאה של המתודה, היא קוראת למתודות is\_empty\_node, get\_left, get\_right, get\_key, get\_value ודוחפת איבר לסוף הרשימה array. סיבוכיות זמן הריצה של כל הפעולות שהמתודה מבצעת בקריאה אחת היא O(1), ולכן, סיבוכיות זמן הריצה של קריאה אחת למתודה היא O(1). בעץ self יש n צמתים, ולכן, יבוצעו סך הכל n קריאות רקורסיביות למתודה כלומר, נבצע n פעמים פעולות בסיבוכיות זמן ריצה של O(1). מכאן, סיבוכיות זמן הריצה הכוללת היא O().

* מתודה: size  
  קלט: אין.

פלט:

* + טיפוס: int
  + תיאור: מספר האיברים בעץ.

תיאור: המתודה מחזירה את מספר האיברים בעזרת הsize של השורש של self.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: המתודה קוראת למתודות get\_root, get\_size שסיבוכיות זמן הריצה שלהם היא O(1), מספר קבוע של פעמים, ולכן, סיבוכיות זמן הריצה הכוללת היא O(1).

* מתודה: split  
  קלט:
  1. node
     + טיפוס: AVLNode
     + תיאור: צומת בעץ, שעל בסיסה נפריד לשני עצים.

פלט:

* + טיפוס: list[AVLTree]
  + תיאור: רשימה באורך 2 שהאיבר הראשון שלה הוא העץ של כל הצמתים שהיו בself שהמפתחות שלהם קטנים מהמפתח של node והאיבר השני שלה הוא העץ של כל הצמתים שהיו בself שהמפתחות שלהם גדולים מהמפתח של node.

תיאור: מפצלת את self לשני עצים לפי node שניתן כקלט. לאחר הפיצול תתקבל רשימה באורך 2 שהאיבר הראשון שלה הוא העץ של כל הצמתים שהיו בself שהמפתחות שלהם קטנים מהמפתח של node והאיבר השני שלה הוא העץ של כל הצמתים שהיו בself שהמפתחות שלהם גדולים מהמפתח של node.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: המתודה מבצעת קריאה למתודות get\_right, get\_left, create\_tree, get\_parent, שסיבוכיות זמן הריצה שלהן היא O(1), מספר קבוע של פעמים. בנוסף, המתודה "עולה למעלה" בעץ מהצומת node שניתנה כקלט, עד לשורש בלולאת while ובכל איטרציה של לולאת הwhile מבצעת פעולות בסיבוכיות זמן ריצה של O(1) מספר קבוע של פעמים. לכן, סיבוכיות זמן הריצה של לולאת הwhile היא O(logn). יתר על כן, אנו מבצעים פעמיים קריאה למתודה join\_tree\_with\_array\_of\_node\_tuples שסיבוכיות זמן הריצה שלה היא O(logn). לבסוף, אנו יוצרים ומחזירים מערך עם 2 איברים – סיבוכיות זמן ריצה של O(1). לכן, סיבוכיות זמן הריצה הכוללת של המתודה היא O(logn).

* מתודה: join\_tree\_with\_array\_of\_node\_tuples  
  קלט:
  1. array\_of\_node\_tuples
     + טיפוס: List[Tuple(AVLNode, AVLNode)]
     + תיאור: רשימה של Tuples כך שהאיבר הראשון בTuple מתאר צומת מקשר בין העצים והאיבר השני בTuple מתאר את השורש של העץ שנעשה איתו join לself.

פלט: אין.

תיאור: פונקצית עזר של split. המתודה מאחדת את self עם העצים ברשימה כשהצמתים המקשרים מקשרים ביניהם, באמצעות המתודה join.  
סיבוכיות זמן ריצה: O().  
הסבר לסיבוכיות: זו פונקציית עזר של split, ולכן, ננתח את סיבוכיות זמן הריצה בהינתן שsplit קראה למתודה זו. המתודה רצה בלולאת for לכל היותר O(logn) פעמים – כגובה העץ – ובכל איטרציה של הלולאה קוראת למתודה create\_tree שסיבוכיות זמן הריצה שלה היא O(1) ולjoin שסיבוכיות זמן הריצה שלה היא O(logn), אך, ראינו בשיעור שכשאנו קוראים לפונקציה זו מsplit במקרה הגרוע פעולות הjoin יסכמו להיות בסיבוכיות זמן ריצה של O(logn), ולכן, על פני כל הלולאה אנו מבצעים O(logn) "עבודה" רק עם פעולות הjoin שאנו עושים, ובנוסף, אנו עושים O(logn) עבודה עם בO(logn) קריאות שאנו מבצעים לcreate\_tree. לכן, סיבוכיות זמן הריצה הכוללת היא O(logn).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 בנוסחה הבאה:

חישבנו את הדרגה של האיבר במסגרת החיפוש של המיקום בעץ אליו אנחנו רוצים להכניס אותו. לכן התחלנו מהאיבר המקסימלי, ועלינו למעלה לכיוון השורש (לאורך הענף הימיני בעץ), עד שמצאנו את המפתח הראשון בעץ שקטן מהמפתח של האיבר. אז חיסרנו 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.

ניתוח תיאורטי של מספר החילופים במקרה של מערך ממוין-הפוך:

נראה שמספר החילופים במקרה של מערך ממוין הפוך הוא .

כפי שהסברנו קודם, כדי לחשב את מספר החילופים, אפשר לסכום את מספר האיברים שכל איבר חדש שאנחנו מכניסים לעץ מתחלף איתם. מכיוון שהמערך ממוין הפוך, בכל הכנסה לעץ האיבר שאנחנו מכניסים יהיה האיבר הקטן ביותר בעץ, ולכן הוא יתחלף עם כל האיברים שכבר בעץ.

כלומר, כל איבר i יתחלף עם i איברים.

לכן נקבל שמספר החילופים הוא:

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 מהנוסחה:

*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.*

כפי שניתן לראות בגרפים הבאים, הערכים בטבלה מסעיף א׳ והניתוח מסעיף ב׳ מתאימים.

**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.