מימוש עץ AVL

<u>תיעוד המתודות במחלקות:</u>

<u>המחלקה AVLNode:</u>

המחלקה מממשת את המנשק AVLNode ומכילה את השדות הבאים:

value – הערך בתוך הצומת.

height – גובה הצומת בתוך העץ. באתחול הצומת הגובה הוא 1- לצומת וירטואלי ו-0 לצומת לא וירטואלי.

Size – מספר הצמתים בתת העץ שבו הצומת הנוכחי משמש כשורש, כולל הצומת עצמו.

בנוסף, מכיל 3 מצביעים לAVLNode עבור הבן הימני של הצומת, הבן השמאלי ועבור ההורה.

המתודות במחלקה:

1. get וset עבור ההורה, הבן הימני והבן השמאלי, כולן בסיבוכיות O(1). בset של הבן הימני והשמאלי אנחנו מעדכנים גם את ההורה להיות הצומת עליו הפעלנו את המתודה.
2. Get וset עבור השדות של הצומת, value, height, size וכולן בסיבוכיות O(1).
3. הוספת ילדים וירטואלים לצומת בסיבוכיות O(1).
4. isRealNode – מחזירה האם הצומת הוא אמיתי או וירטואלי, בסיבוכיות O(1).
5. newHeight – מעדכנת את הגובה של הצומת הנוכחי לפי המקסימום של גבהי הבנים שלו, פלוס 1, בסיבוכיות O(1).
6. newSize – מעדכנת את הגודל של הצומת הנוכחי, לפי סכום הגדלים של הבנים שלו, פלוס 1, בסיבוכיות O(1).
7. getBalanceFactor – קבלת ההפרש בין גבהי הבנים של הצומת הנוכחי, בסיבוכיות O(1).
8. leftRotate וrightRotate – מבצעות רוטציות כמו שנלמד בכתה, בסיבוכיות O(1). המתודה מעדכנת את השדות של הsize, הheight והמצביעים לבנים ולהורה של כל הצמתים בהתאם. המתודה מחזירה 1 כחלק מספירה של כמות הסיבובים שנדרשו לתיקון העץ (בהמשך).
9. FindRoot - מתודה שמוצאת את השורש של העץ. עולה מצומת אחת דרך הוריו עד שמוצאת צומת ללא הורה (קיים יחיד כזה- השורש). תעבור לכל היותר בlogn צמתים ולכן הסיבוכיות שלה היא O(logn).
10. leftThenRightRotation - מתודה אשר מגיעה לבן השמאלי של הצומת, מסובבת אותו לשמאל ולאחר מכן מסובבת את הצומת עצמה לימין, נעזרת במתודות של סיבוב לימין וסיבוב לשמאל. מחזירה 2 כמספר הסיבובים שמבצעת. מבצעת מספר סופי של פעמים פעולות בסיבוכיות של O(1) ולכן הסיבוכיות שלה היא O(1).

מימוש עץ AVL

11. rightThenLeftRotation - מתודה אשר מגיעה לבן הימני של הצומת, מסובבת אותו לימין ולאחר מכן מסובבת את הצומת עצמה לשמאל, נעזרת במתודות של סיבוב לימין וסיבוב לשמאל. מחזירה 2 כמספר הסיבובים שמבצעת. מבצעת מספר סופי של פעמים פעולות בסיבוכיות של o(1) ולכן הסיבוכיות שלה היא o(1).
12. Predecessor - מתודה שמחזירה את האיבר הקודם לצומת כלשהו.

- נלך לבן השמאלי ולאחר מכן לבן הכי ימני שלו ונחזיר אותו. אם לא קיים בן שמאלי, כל עוד הצומת הוא בן שמאלי של אבא אחר נמשיך לעלות לאבא שלו עד שנעלה לאבא שהצומת מהווה בן ימני שלו, איבר זה יהיה הקודם.
- הסיבוכיות תלויה בגובה האיבר בעץ, שבהכרח קטן שווה לגובה העץ. נניח כי הצומת נמצא בגובה h אז הסיבוכיות הינה o(h) כי אנחנו מתקדמים למעלה בעץ דרך האבות, בהכרח קטן יותר מO(logn).

13. Successor - מתודה שמחזירה את האיבר הבא מצומת כלשהו. אופן הפעולה סימטרי לזו של predecessor, הסיבוכיות זהה.
14. fixTree - מתודת מעטפת למתודה רקורסיבית אשר מאזנת את העץ כדי שלכל צומת יהיה ערך balance factor בטווח הנדרש כדי להיות עץ AVL, מחזירה את כמות הסיבובים שנדרשו לשם כך.

- המתודה עוברת מהצומת דרך הוריו עד לשורש בעזרת רקורסיה.
- המתודה מחשבת את הbalance factor (BF) בעזרת מתודת עזר ומסובבת את העץ בצורה כזו שאם יש הפרה באיזון ההפרה תתוקן.
- אם הגענו לצומת לא אמיתית נחזיר 0.
- נעדכן את הגובה והגודל של הצומת בעזרת פעולות עזר newHeight, newSize בסיבוכיות של o(1).
- לתיקון צד ימין גדול משמאל נשתמש בסיבוב לשמאל או בסיבוב לימין ואז לשמאל, לתיקון צד שמאל גדול מימין נשתמש בסיבוב לימין או בסיבוב לשמאל ואז לימין כנלמד בכיתה, כולן פעולות עזר בעלות של o(1).
- לאורך הדרך נסכום את כמות הסיבובים שנדרשו לביצוע הפעולה, כמות של כל סיבובית הנדרשו לכל פעולת עזר מוחזרות לאחר שימוש בפועלת העזר.
- כל הפעולות שאנו משתמשים בהם הן של חישוב והשוואה, וכך גם פעולות העזר שנעזרים בהם במתודה הזו. בסה"כ אנו מריצים את הרקורסיה לכל היותר o(logn) פעמיים כי אנחנו מתקדמים מצומת אחת דרך ההורה עד לשורש כך שהסיבוכיות לפעולה אחת של הרקורסיה היא o(1) ולכן בסה"כ הסיבוכיות היא o(logn).

Eden Pollinger 318688991 עבודה מעשית 1 במבני נתונים 318895877 Ron Mondshein

מימוש עץ AVL

המחלקה AVLTree:

המחלקה מכילה את השדות הבאים:

3 מצביעים – אחד לשורש העץ, אחד למינימום של הרשימה ואחד למקסימום של הרשימה (במינימום ובמקסימום מדובר בצומת הראשון והאחרון בעץ בסריקה תוכית, אין קשר לvalue של הצומת.

בנוסף, size של העץ – מספר הצמתים בעץ, וגובה העץ.

המתודות במחלקה:

1. בנאי – המתודה מאתחלת עץ חדש, המצביעים למינימום, המקסימום ושורש העץ מאותחלים לNone, גובה העץ הוא 1- והגודל שלו הוא 0.
2. empty(self) – לא מקבלת ערך ומחזירה ערך בוליאני. אם השורש ריק או צומת וירטואלי, יוחזר true, אחרת, יוחזר false. הסיבוכיות היא O(1).
3. TreeSelectRec(self, AVLNode, int) – מתודת עזר למתודת retrieve. מתודת select כמו שנלמד בכתה. המתודה עוברת על הצמתים בעץ, בדומה לחיפוש בינארי אך במקום להסתכל על ערכי הצמתים, היא מסתכלת על הsize שלהם. אם גודל תת העץ השמאלי קטן מהמספר שאנחנו מחפשים, נמשיך לתת העץ הימני, אחרת, נמשיך לשמאלי.
   - אם סך הכל מספר הצמתים שעברנו עליהם הוא המספר שקיבלנו, אפשר לעצור.
   - סיכויות זמן הריצה היא o(logn) כגובה העץ.
4. Retrieve_node(int) – המתודה מקבלת אינדקס ומחזירה את הצומת במקום i ברשימה, על ידי קריאה למתודת העזר treeSelectRec במקום i, נשלח את i+1.
5. Retrive_node(int) – המתודה עובדת באופן זהה למתודה בסעיף 4, אבל מחזירה את ערך הצומת במקום את הצומת עצמו.
6. Insert(I, s) - המתודה מכניסה את הערך s לאינדקס I ברשימה, ומחזירה את מספר פעולות האיזון שנדרשו לשם תיקון העץ להיות AVLTree תקין לאחר הפעולה.
   - ראשית, ניצור צומת חדשה בעלת בנים וירטואליים בסיבוכיות של o(1).
   - במקרה והמיקום הנדרש הוא הראשון או האחרון נכניס את האיבר הנ"ל למקום המתאים ונעדכן את המינימום או המקסימום בהתאם.

מימוש עץ AVL

- אחרת נמצא את האיבר שנמצא במקום הנדרש לפני ההכנסה בעזרת פעולת עזר (retrieve_node בעלת סיבוכיות o(logn) נוסיף את האיבר החדש להיות בנו השמאלי (ובעצם לרשת את המקום שלו כאיבר הi). במידה ואי אפשר להכניס מיידית את הערך הנדרש כבן שמאלי, נחפש את האיבר הקודם לאיבר במקום הi ונכניס את הערך להיות בנו הימני ובעצם לקבל את המיקום הi כנדרש. חיפוש האיבר הקודם לאיבר הרצוי (presecessor) נעשה בסיבוכיות של o(logk) כך ש k הוא גובה הצומת בעץ ומתקיים k<=n.
- לאחר ההוספה במקום הרצוי מופעלות 2 פעולות עזר: תיקון העץ (fixTree) ומציאת השורש החדש של העץ במידה והשתנה (findRoot). תיקון העץ נעשה כדי לאזן את העץ כעץ AVL לאחר ההכנסה (במידת הצורך). 2 הפעולות הן לכל היותר בסיבוכיות של o(logn).
- fixTree מחזירה את מספר פעולות האיזון, את ערך זה אנחנו מחזירים כדי לדעת כמה פעולות איזון נדרשו לאחר ההכנסה.
- סה"כ נעשות כמות סופית של פעולות מסיבוכיות של לכל היותר o(logn) ולכן הסיבוכית היא o(logn).

7. Delete(i) - המתודה מוחקת את האיבר באינדקס i ברשימה, ומחזירה את מספר פעולות האיזון שנדרשו לשם כך, במידה והמיקום לא קיים המתודה מחזירה 1-.

- תחילה, אם הצומת הנדרש לא קיים מחזירים 1- כנדרש, אם יש צומת 1 ואינדקס לצומת זו מוחקים אותה ומחזירים 0 פעולות איזון.
- נחלץ את הצומת הנדרשת בעזרת פעולת עזר (retrieve_node) בעלות של o(logn). נמצא את הצומת שצריכה להחליף אותה.
- ראשית, במידה ויש לצומת בן ימני וגם בן שמאלי נמצא את העוקב לצומת, (פעולת successor שנעשית בעלות של o(logn), נשים את ערך העוקב בערך הצומת שאנחנו רוצים למחוק ונצביע על הצומת העוקבת כצומת אותה אנחנו רוצים למחוק. כך נוכל למחוק את העוקב במקום את הצומת שאנחנו רוצים למחוק תוך כדי שמירה על כל הערכים והמיקומים שלהם.
- נמצא את האבא של הצומת אותה רוצים למחוק ואת הבן הימני, נשמור אותו כצומת אותה אנחנו רוצים להחליף.
- אם לא קיים כזה בן ימני, נראה אם יש לצומת בן שמאלי ונשמור אותה בצומת אותה אנחנו רוצים להחליף. אחרת, נחליף את הצומת ב None.
- את ההמשך נחלק למקרים:
- במידה ומחקנו את השורש (צומת שאין לו אבא) נמצא את הקודם (פעולת עזר predecessor שפועלת בסיבוכיות של o(logn). אם הצומת אותה אנחנו רוצים להחליף היא בן ימני והצומת הקודמת קיימת נגדיר את הצומת שאותה אנחנו רוצים להחליף כצומת הקודמת, נגדיר את בנה הימני להיות הבן הימני של השורש אותו רוצים למחוק ונגדיר את המיקום הנוכחי שלנו כהורה של הקודם. נעדכן את הsize של כל הצמתים עד לשורש

(עד o(logn) מעברים בסיבוכיות של o(1)). נגדיר את הבן השמאלי של הקודם להיות הבן השמאלי של השורש אותו אנחנו רוצים למחוק. נגדיר את השורש להיות הצומת שאנחנו רוצים להחליף, ונבצע תיקון לעץ תוך כדי שמירה על מספר הסיבובים שעשינו כדי לחזור למצב של AVLTree. נשתמש בפעולת עזר fixTree שפועלת בסיבוכיות של o(logn). נחשב את הגודל החדש של השורש בפעולה בסיבוכיות של o(1).

- במקרה השני, בו אנחנו מוחקים צומת שהיא איננה שורש. נראה אם אנחנו מוחקים בן ימני או בן שמאלי ונעדכן את ההורה בצומת המוחלפת בהתאם. נבצע תיקון לעץ כמו בחלק הראשון, נמצא את השורש בפועלת עזר findroot בסיבוכיות של o(logn).
- נעדכן לאחר כל השינויים את הצומת במקום הראשון והאחרון (המינימום והמקסימום) בעזרת פעולת עזר (retrieve_node) בעלות של o(logn).
- נחזיר את מספר הסיבובים שסובבנו כדי להחזיר את העץ להיות עץ AVL תקין.
- הסיבוכיות של כל אחת מפעולות העזר היא לכל היותר o(logn) והן נעשות בכמות סופית ולכן סה"כ סיבוכיות . היא o(logn).

8. First() – המתודה מחזירה את האיבר הראשון ברשימה, אצלנו זה ערך המינימום. כיוון שיש אליו מצביע המתודה מתבצעת ב o(1).
9. last() – המתודה מחזירה את האיבר האחרון ברשימה, אצלנו זה ערך המקסימום. כיוון שיש אליו מצביע המתודה מתבצעת ב o(1).
10. listToArray() - מתודת מעטפת למתודה רקורסיבית המחזירה מערך המכיל את איברי הרשימה לפי סדר האינדקסים, או מערך ריק אם הרשימה ריקה.
    - מהלך המתודה הינו ברקורסיה.
    - מקרה עצירה- אם הצומת ריקה או לא אמיתית.
    - מהלך הרקורסיה- קריאה רקורסיבית לאיברים משמאל לאיבר הנוכחי (הקטנים ממנו), הוספה של האיבר הנוכחי, קריאה רקורסיבית לאיברים מימין לאיבר הנוכחי (הגדולים ממנו). בצורה כזו האיברים יסודרו מהקטן ביותר לגדול ביותר לפי הסדר. הערך הראשון שהרקורסיה מקבלת הוא את השורש.
    - הסיבוכיות הינה o(n) היות ואנחנו עוברים ברקורסיה פעם אחת על כל איבר ושמים אותו במערך, סה"כ n איברים, n קריאות רקורסיביות בסיבוכיות של o(1) לכל אחת.
11. Length() – המתודה מחזירה את אורך הרשימה. ערך השדה size של השורש מכיל את מספר הצמתים בעץ, כיוון שיש מצביע לשורש, הסיבוכיות של המתודה היא O(1).
12. listToAVL(lst) – מתודת עזר למתודות הinsert והpermustation שמכניסה ערכי רשימה לתוך עץ AVL בסיבוכיות של O(n) לפי האלגוריתם שנלמד בכיתה.

- ניקח את אמצע הרשימה להיות השורש של העץ.
- נקרא למתודה ברקורסיה עבור החצי השמאלי והחצי הימני של הרשימה.

13. Sort() – המתודה ממיינת את הרשימה המייצגת את העץ.

- ניצור רשימה מהעץ הנתון שלנו על ידי שימוש במתודה listToArray().
- נמיין את הרשימה על ידי מיון mergeSort, ידוע ממבוא למדמ"ח שהמיון הוא בסיבוכיות של O(nlogn).
- ניצור עץ חדש בסיבוכיות של O(1) ונכניס אליו את הערכים הממוינים עם המתודה של listToAVL בסיבוכיות של O(n).
- סך הכל, הסיבוכיות של המתודה היא O(nlogn).

14. Permutation() – המתודה מחזירה את אותם האיברים ברשימה בסדר אקראי.

- ניצור רשימה מהעץ הנתון שלנו על ידי שימוש במתודה listToArray().
- בלולאה באורך מספר האיברים ברשימה, נגריל מספר רנדומלי בין אינדקס האינטרציה שאנחנו נמצאים בה לבין אורך הרשימה.
- נחליף את האיבר במקום אינדקס האינטרציה עם האיבר במספר שהוגרל. כך נוודא שאנחנו לא מגרילים את אותו צומת פעמיים.
- ניצור עץ חדש בסיבוכיות של O(1) ונכניס אליו את הערכים החדשים עם המתודה של listToAVL בסיבוכיות של O(n).
- סך הכל, הסיבוכיות של המתודה היא O(n).

15. join(lst) - מתודת עזר לconcat. מתודה אשר מצרפת שני עצים בעזרת האיבר האחרון בעץ אליו רוצים לצרף.

- אם העץ שרוצים לצרף ריק, מחזירים את האיבר האחרון למקום שלו וסיימנו, אם העץ אליו רוצים לצרף ריק נעדכן את האיבר המינימלי שלו באיבר שהוצאנו (לשימוש עתידי) ונמשיך כרגיל.
- נטפל ב2 מקרים סימטריים באותה הדרך: במקרה בו גובה העץ הקיים קטן מגובה העץ שרוצים לצרף או ההפך. נסביר את המקרה הראשון, השני סימטרי אליו.
- במקרה זה, ראשית, ניקח את הצומת המקשרת להיות השורש של העץ הגדול. נקדם אותו בלולאה להיות הבן השמאלי של עצמו עד אשר הוא יהיה שורש לתת עץ השווה בגובהו לגובה העץ הקיים. ניקח את הצומת שהוצאנו מהעץ המקורי ונחבר לה משמאל את העץ המקורי ומימין את התת עץ הזה. נגדיר את הצומת המקשרת כשורש ונעדכן את הגודל, גובה ואת האיבר המקסימלי. (פעולות בסדר גודל של o(1)).
- המתודה פועלת בסיבוכיות של o(height_difference) ותלויה בהפרש הגבהים בין העצים, כפי שנלמד בכיתה.

מימוש עץ AVL

16. concat(lst) - מתודה אשר מקבלת רשימה ומשרשרת אותה אל סוף הרשימה המקורית. המתודה מחזירה את הפרש הגבהים בין עצי הAVL שמוזגו.

- אם העצים ריקים לא משנה כלום ומחזירה אפס. אם אחד מהם ריק מעדכנת את העץ להיות העץ הלא ריק ומחזירה את הגובה של העץ הלא ריק.
- אחרת, מחשבת את הפרש הגבהים, מוחקת את הצומת הגדולה ביותר בעץ המקורי ומשתמשת בו ובפעולת העזר join על מנת לחבר ביניהם לעץ המקורי וליצירת עץ AVL משותף.
- המתודות פועלות לפי סיבוכיות של o(1) למעט delete וjoin שפועלות פעם אחת כל אחת. Delete בסיבוכיות של o(logn) וjoin בסיבוכיות של o(height_difference). נניח כי lst בסדר גודל של n ולכן הסיבוכיות הכלליות של המתודה הינה o(logn) .

17. Search(val) – המתודה מקבלת ערך ומחזירה את האינדקס הראשון ברשימה עם אותו הערך.

- ניצור רשימה מהעץ הנתון שלנו על ידי שימוש במתודה listToArray().
- נעבור על האיברים ברשימה בלולאת for עד שנמצא את האינדקס הראשון בו הערך הוא הערך שקיבלנו.
- הסיבוכיות היא O(n).

18. getRoot() – המתודה מחזירה את השורש של העץ. כיוון שיש לנו מצביע לשורש, נעשה בסיבוכיות של O(1).

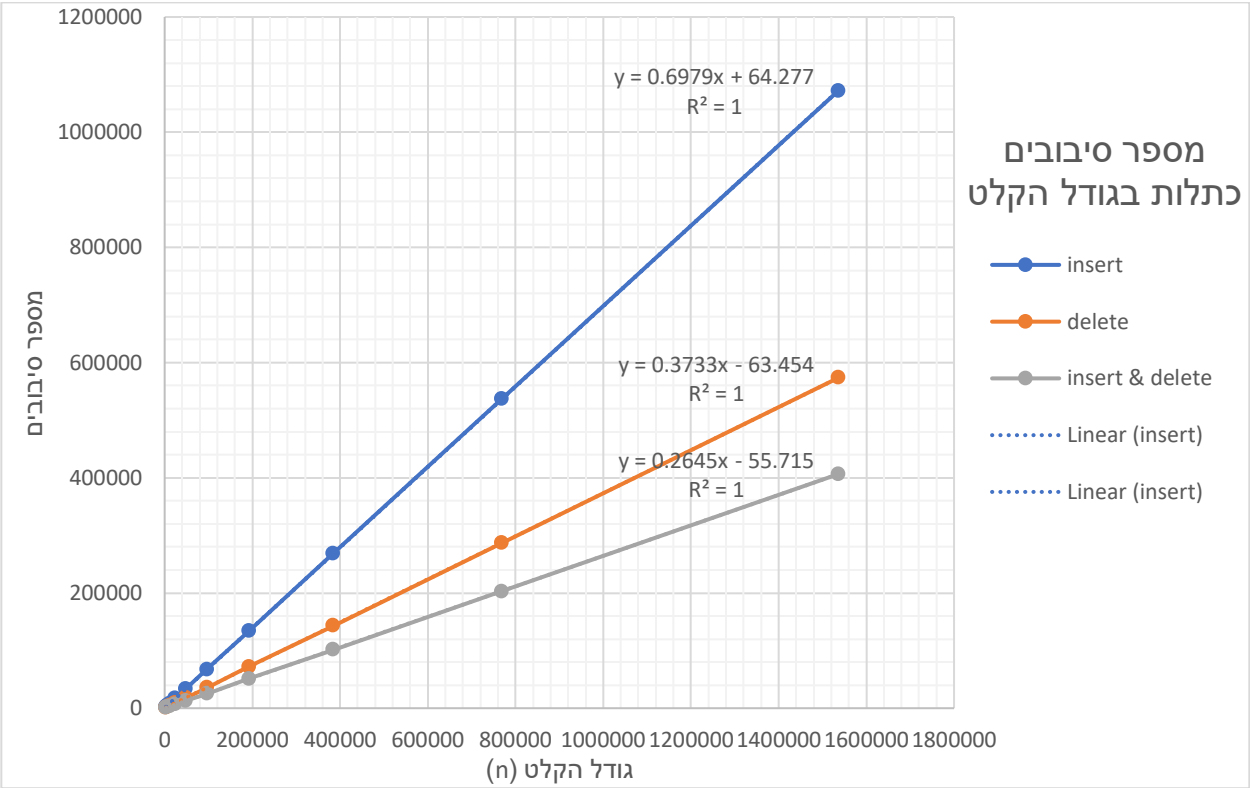
Eden Pollinger 318688991 עבודה מעשית 1 במבני נתונים 318895877 Ron Mondshein

מימוש עץ AVL

החלק התאורטי:

שאלה 1

insert & delete	delete	insert	n	index
759	1116	2178	3000	1
1645	2321	4174	6000	2
3180	4473	8321	12000	3
6309	8864	16673	24000	4
12535	17967	33384	48000	5
25043	35558	67201	96000	6
51162	71695	134050	192000	7
101380	143109	268323	384000	8
202909	286523	536432	768000	9
406361	573471	1071828	1536000	10



אנחנו מצפים להתאמה מהסוג הבא-

אם נסמן את i כאינדקס של מספר ההכנסה וn כגודל הקלט כך ש $n = 1500 * 2^i$ והפעולות insert וdelete נכתבו בסיבוכיות של $o(logn)$ כך שn מייצג את גודל הקלט נצפה לתלות לינארית מהצורה הבאה-

$$\log(1500 * 2^{i}) = \log(1500) + \log(2^{i}) = 10.55 + i$$

אכן קיבלנו תלות לינארית. ניתן לראות כי ההתאמה שלנו טובה לפי מדד הR^2 ששווה בדיוק ל1.

מימוש עץ AVL

שאלה 2

הכנסות לתחילת מבני הנתונים-

Array	Linked List	AVLTree	Average time/index
6.15E-07	8.27E-07	7.02E-05	1
1.11E-06	1.09E-06	7.36E-05	2
1.57E-06	9.53E-07	7.65E-05	3
1.70E-06	8.65E-07	7.90E-05	4
2.47E-06	9.33E-07	8.18E-05	5
2.86E-06	9.94E-07	7.89E-05	6
3.30E-06	9.40E-07	8.30E-05	7
4.21E-06	1.00E-06	8.45E-05	8
3.97E-06	9.02E-07	8.36E-05	9
4.41E-06	8.72E-07	8.26E-05	10

מבחינת סיבוכיות זמן ריצה לפעולה זו אנחנו יודעים ש linked list פועלת בסיבוכיות של o(1), array ב o(n) וAVLTree בסיבוכיות של o(logn) ביחס למקרה הממוצע של הכנסה לתחילת המבנה. לכן נצפה לזמן ריצה מאוד טוב לlinked list וגרוע לarray . ניתן לראות ש linked list אכן עומדת בציפיות וזמן הריצה הממוצע שלה הוא הקצר ביותר. מה שלא ציפינו לראות הוא שזמן ההרצה הממוצע של AVLTree ארוך יחסית לArray . ניתן ליחס זאת למימוש מאוד יעיל שפייתון כנראה מממש את array.

הכנסות אקראיות למבני הנתונים-

Array	Linked List	AVLTree	Average time/index
1.30685E-06	0.00012065	7.49987E-05	1
1.66003E-06	0.00026295	7.91596E-05	2
1.76907E-06	0.00041754	8.73947E-05	3
1.92018E-06	0.00058847	8.4864E-05	4
2.13677E-06	0.00073537	8.6564E-05	5
2.35144E-06	0.00090203	8.86711E-05	6
2.83255E-06	0.00104137	9.45277E-05	7
2.74082E-06	0.00122166	9.33106E-05	8
2.96527E-06	0.00137335	9.55164E-05	9
3.1268E-06	0.00158996	9.86893E-05	10

מבחינת סיבוכיות זמן ריצה לפעולה זו linked list וarray פועלות בסיבוכיות של o(n-i+1) כך ש i הוא המיקום האקראי להכנסת האיבר, AVLTree רצה בסיבוכיות של o(logn). נצפה לזמן ריצה ממוצע זהה לlinked list ול array וזמן ריצה טוב יותר לAVLTree . אם מסתכלים על linked list ועל AVLTree קל לראות שAVLTree עובדת

Eden Pollinger 318688991 עבודה מעשית 1 במבני נתונים 318895877 Ron Mondshein

מימוש עץ AVL

בזמן ריצה ממוצע מהיר הרבה יותר מlinked list . אם זאת ובניגוד למה שציפינו Array עובדת בצורה יותר מהירה מAVLTree . ניתן להסביר זאת על ידי כך שפייתון מממשת בצורה טובה מאוד את array בידיעה שזה כלי שמשתמשים בו בצורה תדירה.

<u>הכנסות לסוף מבני הנתונים-</u>

Array	Linked List	AVLTree	Average time/index
8.32876E-08	6.0194E-05	6.73207E-05	1
8.59896E-08	7.80767E-05	7.15057E-05	2
7.68767E-08	0.000114807	7.41413E-05	3
8.40028E-08	0.000166337	7.3369E-05	4
1.04523E-07	0.000190867	8.21589E-05	5
8.28902E-08	0.000226983	7.86279E-05	6
8.21749E-08	0.000265396	7.98638E-05	7
7.83404E-08	0.000305929	7.98476E-05	8
8.06738E-08	0.000469216	8.21253E-05	9
1.00581E-07	0.000374062	8.11597E-05	10

מבחינת סיבוכיות זמן הריצה linked list פועלת ב$o(1)$, array ב$o(n+1)$ וAVLTree ב $o(logn)$. לכן נצפה ש linked list תהיה המהירה ביותר לאחר מכן AVLTree ולבסוף array . AVLTree אכן נמצאת מבחינת זמן ריצה בין linked list לבין array אבל array מהירה הרבה יותר מlinked list בניגוד מוחלט לציפיות. ההסבר היחיד הוא ההסבר שהסברו בסעיפה הקודמים של השאלה.