<u>מטלה מעשית 1 מבנה נתונים</u>

<u>יעקב גולדשמידט [329575427] והילה לוין [206823163]</u>

מחלקת AVLTree

AVLTree

- סיבוכיות של פונ' זו היא $O(1)$, כי אנחנו מאתחלים אובייקט מסוג AVLNode ומשתנה של גודל להיות 0.

Empty

- סיבוכיות של פונ' זו היא $O(1)$, כי אנחנו קוראים לפונ' אחרת שהינה מסיבוכיות $O(1)$ גם.
- הפונ' בודקת האם השורש של העץ הוא צומת אמיתי, אם כן, היא תחזיר שהעץ לא ריק. אחרת תחזיר שהוא ריק.

search

- סיבוכיות של פונ' זו היא $O(\log n)$ משום שאנו קוראים לפונ' עזר מסיבוכיות זו (המופיעה למטה).
- אנו מבצעים את פעולת החיפוש ע"י קריאה לפונ' עזר, שמאתרת את הnode שאנו מחפשים, כשלבסוף נחזיר בפונ' המקורית את הערך של אותו הnode. כפי שציינו, בפונ' עזר נבצע חיפוש בינארי בעץ על מנת למצוא את הnode עם המפתח המתאים, כשכמובן שאם לא מצאנו נחזיר node ריק.

search_node

- סיבוכיות של פונ' זו היא $O(\log n)$. בפונ' זו אנו מבצעים חיפוש הזהה לחיפוש בינארי, עד למציאת/אי מציאת האיבר עם הערך המתאים. לכן, כפי שלמדנו הסיבוכיות היא O(logn).
- אנו מבצעים חיפוש בינארי על העץ במטרה לאתר את הnode עם הערך שאנו מחפשים. אם מצאנו, נחזיר את הnode. אחרת, נחזיר node ריק.

Insert(int k, Boolean s): הפונקציה מכניסה איבר חדש למבנה בעל מפתח k וערך s. סיבוכיות זמן פונקציית הכנסה היא Olog n . הפונקציה פועלת בארבעה שלבים וננתח את הסיבוכיות של כל שלב בנפרד:

1. הפונקציה מחפשת את המקום הנכון להכניס את האיבר החדש. החיפוש מתחיל מהשורש וכל עוד שלא הגענו לצומת וירטואלי (כלומר כל עוד אנחנו בצמתים, כלומר עד שנגיע לעלה) נשווה בין המפתח של הצומת הנוכחי לבין k. אם k גדול נרד לבן הימני של אותו צומת ונמשיך. אם k קטן נרד לבן השמאלי. אם יש שוויון, אנו נדע שיש כבר צומת קיים במבנה בעל מפתח k ולכן נסיים את הפונקציה ונחזיר 1-. אם הגענו לעלה והמפתח של העלה שונה מ-k אנו נדע שמצאנו את מקומו של הצומת החדש. סך הכל ביצענו O(logn) פעולות, כמו חיפוש בינארי רגיל בגובה העץ, וגובה העץ של עץ AVL הוא מובטח להיות O(logn) .
2. נאתחל צומת חדש ונכניסו בתור הבן הימני או שמאלי של אותו עלה (שמאלי אם k קטן ממפתח העלה, ימני אם גדול). סיבוכיות זמן O(1).
3. עתה הפונקציה מתחילה מהצומת החדש שהוכנס ובודקת מה הBalanceFactor של הצומת. אם הוא לא ±2 אז הצומת לא עבריין AVL. כעת נבדוק אם היה שינוי בגובה של אותו צומת. (השדה "גובה" עבור כל צומת מתוחזק על ידי בדיקת השדה "גובה" עבור שני בנים של אותו צומת, לוקחת מקסימום מבניהם ומוסיפה 1. ככה כפי

שלמדנו תחזוק השדה ובדיקת שינוי הגובה מתבצעות ב(O(1.) מבצעים את החישוב של הגובה החדש ובודקים אם הוא שווה לגובה הקיים. אם אין שינוי בגובה, הוכחנו בכיתה כי לא נדרש לבצע רוטציה כי הBalanceFactor של כל צומת בעץ נשאר תקין לכן נסיים את הפונקציה. אחרת, כלומר אם היה שינוי בגובה של אותו צומת אנו נעדכן את הגובה ונמשיך עם האיטרציה הבאה עם ההורה של אותו צומת. אם נמצא שהBalanceFactor שווה ל 2 או 2- גילינו עבריין AVL

4. ונצטרך לבצע רוטציה ימנית או שמאלית או רוטציה כפולה כתלות באותו בן בצד של העבריינות כפי שלמדנו את ארבעת הרוטציות בשיעור. הרוטציות מתבצעות דרך פונקציית העזר leftRotation או rightRotation שהן בסיבוכיות זמן (O(1 ונסביר עליהן בנפרד.
5. אכן אחרי ביצוע רוטציה(או רוטציה כפולה) הוכחנו בהרצאה כי העץ תקין כעת ומעל לאותם צמתים אין שינויי גובה. לכן ניתן להפסיק את הפונקציה. במקרה הגרוע נעלה עד לשורש ואז בשורש: או שנבצע רוטציה או רטציה כפולה. בכל אופן בשני המקרים הנ"ל, בכל מקרה גרוע קיבלנו סיבוכיות של (O(logn. סך הכל חיפוש + תיקונים + רוטציות $= O(\text{logn}) + O(\text{logn}) + O(1) = O(\text{logn})$

leftRotation(AVLNode pivot, AVLNode right_son, AVLNode parent): הפונקציה מבצעת רוטציה שמאלית סביב צומת הציר בשם pivot כאשר right_son הוא הבן שלו וparent הוא האב של הציר. אנחנו קובעים שהאב של right_son עכשיו יהיה parent והאב של הציר יהיה right_son. בנוסף, התת עץ השמאלי של right_son אם קיים, הוא אכן גדול מ right_son כי right_son הוא הבן הימני של pivot. סך הכל שינינו מספר קבוע של פויינטרים ולכן סיבוכיות הזמן הינה (O(1.

rightRotation(AVLNode pivot, AVLNode left_son, AVLNode parent): הפונקציה מבצעת רוטציה ימנית סביב צומת הציר בשם pivot כאשר left _son הוא הבן שלו וparent הוא האב של הציר. אנחנו קובעים שהאב של left_son עכשיו יהיה parent והאב של הציר יהיה left _son. בנוסף, התת עץ השמאלי של left _son אם קיים, הוא אכן קטן מ left _son כי left _son הוא הבן השמאלי של pivot. סך הכל שינינו מספר קבוע של פויינטרים ולכן סיבוכיות הזמן הינה (O(1.

Delete(int k): אנו נבצע מחיקה של צומת בעץ בעלת מפתח k אם קיים. הפונקציה תתבצע בארבעה שלבים וסך הכל סיבוכיות זמן של (O(logn .

1. נחפש את הצומת עלינו למחוק תוך שימוש בפונקציה (search_node(int k. הוכחנו שהסיבוכיות היא (O(logn. נסמנו ב-to_del. אם הצומת לא קיים במבנה נסיים את הפונקציה.
2. נמחק את to_del. אם יש לו בן יחיד נקשר אותו לאבא של to_del תוך שימוש בפונקציה change_kid שפועל ב(O(1 המתוארת למטה. אם יש לו שני בנים, אנו נחליף אותו בעוקב שלו תוך שימוש בפונקציית successor מסיבוכיות (O(logn שיתואר בהמשך. מכך שיש לto_del שני בנים מובטח שיש לו עוקב. מהגדרת העוקב, העוקב ימצא בתת עץ של to_del (משום שיש לו שני בנים). לפי מה שראינו בהרצאה, לעוקב לא יכול להיות בן שמאלי ולכן בהכרח יש לו במקסימום בן ימני בלבד.

3. עתה נעלה מהצומת הנמחק (או אם החלפנו אותו בעוקב שלו אנו נעלה מהעוקב) ובכל איטרציה נבדוק את הBalanceFactor והאם השתנה הגובה (תוך כדי הבדיקה של הגובה אנו נעדכן את הגובה של הצומת הנוכחית כתלות בגובה של הבנים שלו לכן תחזוק השדה גובה ובדיקת שינוי גובה מתבצעת בO(1)).
4. אם נמצא עבריין אנו נבצע רוטציה מתאימה או רוטציה כפולה מתאימה תוך הפונקציות עזר leftRotation ו-rightRotation שהסברנו למעלה O(1). הפעם לא נעצור אחרי רוטציה אחת אלא נעצור רק אם נמצא צומת שלא השתנה הגובה שלו או שנגיע עד לשורש (כולל).

לכן סך הכל עלינו במקרה הגרוע את גובה העץ ובמקרה הגרוע ביצענו רוטציה בכל צומת במסלול לכן סך הכל O(logn) כולל החיפוש.

changeKid

- פונקציה זו מקבלת שתי צמתים, Node ו-New_kid ומשנה את המצביעים כך שהמצביע Left או right של ההורה של Node מצביע עתה לNew_kid כתלות באם Node היה הבן הימני או הבן השמאלי של ההורה שלו. אם Node הייתה השורש של העץ, כלומר אם לNode אין הורה (שזה מובטח לקרות רק כאשר Node הוא שורש של עץ) הפונקציה תשנה את העץ כך שהשדה root יצביע לNew_kid. הפונקציה גם קובעת שהשדה Parent של New_kid עתה מצביע להורה של Node. פונקציה זו תעזור לנו בפעולת המחיקה. סיבוכיות זמן : O(1).

Min

- סיבוכיות של פונ' זו היא O(1), משום שיש לנו שדה המחזיק מצביע של המינימום של העץ, לכן נוכל להחזיר את הערך שלו. אם העץ ריק, נחזיר

Max

- סיבוכיות של פונ' זו היא O(1), משום שיש לנו שדה המחזיק מצביע של המקסימום של העץ, לכן נוכל להחזיר את הערך שלו. אם העץ ריק, נחזיר null.

HasRightSon

- הפונ' בודקת האם הבן הימיני של צומת נתון הוא node אמיתי. כלומר, האם לצומת נתון קיים בן ימיני. פעולה זו היא מסיבוכיות של O(1).

HasLeftSon

- הפונ' בודקת האם הבן השמאלי של הצומת הנתון הוא node אמיתי. כלומר, האם לצומת נתון קיים בן שמאלי. פעולה זו היא מסיבוכיות של O(1).

keysToArray

- סיבוכיות של פונ' זו היא O(n). אנו נבצע מספר פעולות מסיבוכיות זניחה, עד שנקרא לפונ' העזר שלנו keysToArrayRec, שהינה מסיבוכיות O(n). פעולה זו תבנה לנו את הרשימה הדרושה. לבסוף, נעבור על איברי הרשימה ונעביר אותם בסדר זה למערך, בהתאם לחתימת הפונ'. פעולה זו גם מסיבוכיות של O(n). לכן זוהי הסיבוכיות של הפונ' בסה"כ.

- במידה והעץ הוא עץ ריק, נסיים אוטומטית ונחזיר מערך ריק. אם העץ אינו ריק, נקרא כאמור לפונ' עזר שתכניס לפי הסדר הרצוי את המפתחות לרשימה. הפונ' עזר הרקורסיבית מבצעת סיור inorder באיברי הרשימה, ומכניסה את מפתחותיהם לרשימה לפי הסדר הרצוי. כשנסיים את הסיור, נעבור איבר איבר ברשימה ונכניס כל איבר למערך חדש לאותו המיקום שבו היה ברשימה. לבסוף, נחזיר את המערך הנדרש.

keysToArrayRec

- סיבוכיות של פונ' זו היא O(n). פונ' זו, עוברת inorder על האיברים בעץ ומכניסה את מפתחותיהם לרשימה בהתאם לכך (add בList היא מסיבוכיות O(1)). למדנו במבוא מורחב כי פעולה זו מסיבוכיות O(n).
- אם העץ אינו ריק, הפונ' תכניס לפי הסדר הרצוי את המפתחות לרשימה. הפונ' הרקורסיבית מבצעת סיור inorder באיברי הרשימה, ומכניסה את מפתחותיהם לרשימה לפי הסדר הרצוי.

infoToArray

- סיבוכיות של פונ' זו היא O(n). אנו נבצע מספר פעולות מסיבוכיות זניחה, עד שנקרא לפונ' העזר שלנו infoToArrayRec שהינה מסיבוכיות O(n). פעולה זו תבנה לנו את הרשימה הדרושה. בדומה לחישוב סיבוכיות של keysToArray, נקבל שסה"כ הסיבוכיות היא O(n).
- במידה והעץ הוא עץ ריק, נסיים אוטומטית ונחזיר מערך ריק. אם העץ אינו ריק, נקרא כאמור לפונ' עזר שתכניס לפי הסדר הרצוי את הערכים לרשימה. הפונ' עזר הרקורסיבית מבצעת סיור inorder באיברי הרשימה, ומכניסה את ערכיהם לרשימה לפי הסדר הרצוי. כשנסיים את הסיור, נעבור איבר איבר ברשימה ונכניס כל איבר למערך חדש לאותו המיקום שבו היה ברשימה. לבסוף, נחזיר את המערך הנדרש.

infoToArrayRec

- סיבוכיות של פונ' זו היא O(n). פונ' זו, עוברת inorder על האיברים בעץ ומכניסה את ערכיהם לרשימה בהתאם לכך (add בList היא מסיבוכיות O(1)). למדנו במבוא מורחב כי פעולה זו מסיבוכיות O(n).
- גם פונ' זו, עוברת inorder על האיברים בעץ ומכניסה את הערכים של כל איבר לרשימה בהתאם לכך. בדומה לחישוב של keysToArrayRec.

size

- סיבוכיות של פונ' זו היא O(1). זאת משום שאנו מתחזקים שדה המייצג את גודל העץ.
- לכל עץ, המיוצג ע"י צומת של שורש, קיים משתנה size . לכן, נוכל להחזיר בקלות את משתנה זה. עוד נציין כי אנו מתחזקים את שדה זה בפעולות כמו insert, delete ואתחול עץ חדש.

getRoot

- סיבוכיות של פונ' זו היא O(1), משום שיש לנו שדה המייצג את השורש.
- כל עץ מיוצג ע"י צומת של שורש. לכן ישנו משתנה המייצג את השורש, ונוכל להחזירו. כמובן ששדה זה משתנה בהתאם לפעולות המתבצעות.

prefixXor

- פונקציה הזו מחזירה את הXOR של מספר הצמתים עם הערך true שהמפתחות שלהם קטנים מצומת הקלט. סיבוכיות זמן: O(log(n)). הפונקציה משתמשת בשדה בתוך AVLNode בשם trues_in_sub_tree שאנחנו מתחזקים ונסביר כיצד בהמשך. השדה שומר את מספר הצמתים בעלי הערך true בתת שדה של אותו צומת, כולל הצומת עצמו. אנו מתחילים בשורש באיטרציות. אם k קטן מהמפתח של הצומת הנוכחית, נרד שמאלה ונתחיל שוב את האיטרציה. אם k גדול מהמפתח של אותו הצומת, נוסיף לספירה שלנו את הערך trues_in_sub_tree של הבן השמאלי של הצומת הנוכחי, וכמו כן נוסיף 1 בנוסף אם הערל של הצומת הנוכחית הוא true. נרד ימינה ונמשיך. אם k שווה למפתח של הצומת הנוכחי, נוסיף לספירה שלנו את הערך trues_in_sub_tree של הבן השמאלי של הצומת הנוכחי, וכמו כן נוסיף 1 אם הערך של אותו הצומת הוא true. לאחר שלב זה, נסיים.

  ככה הגענו לספירה של מספר הצמתים בעל ערך true שמפתחותיהם קטנים או שווים לk. השדה trues_in_sub_tree של צומת AVLNode מתוחזק כתלות אך ורק בבנים הישירים של אותו צומת. לכן אנו מתחזקים אותו בהכנסה כאשר אנו יורדים עד לעלה ומחפשים את המקום להכניס את הצומת ומעדכנים את השדה של כל צומת במסלול כתלות באם הערך של הצומת החדש אותו אנו רוצים להכניס לתת עץ של הצומת במסלול הוא true או false. ואנחנו מתחזקים את השדה במחיקה כאשר אנו עולים למעלה מהצומת הנמחק כדי לבצע גלגולים. לכן השדה מוחזק בסיבוכיות $O(1)$ ולא משפיע על הסיבוכיות של הפונקציות האחרות.

ascendUpdateTrues

- הפונקציה הזאת היא בשימוש כאשר אנו בפונקציית מחיקה, עולים כדי לבצע גלגולים ומצאנו תנאים מספיקים שלא צריכים עוד לעלות ולתקן כך שניתן לסיים את פונקציית delete. במקרה הזה אנחנו רוצים להמשיך ולעלות עד לשורש ולעדכן את השדות של הצמתים האלו trues_in_sub_tree. זאת לא משפיעה על הסיבוכיות של delete מכך שבמקרה הגרוע delete רץ לאורך כל הגובה של העץ עד השורש.
- שימוש נוסף של הפונקציה הזאת הוא כאשר בפונקציית הכנסה ירדנו בחיפוש בינארי ובירידה עדכנו את השדות trues_in_sub_tree של כל צומת שעברנו בו כתלות באם הערך שאנו רוצים להכניס הוא true או false. אבל אם במהלך החיפוש מצאנו צומת בעל מפתח k, כלומר אנו מנסים להכניס צומת בעל מפתח שכבר קיים במבנה, נעצור את החיפוש ונקרא לפונקצייה הנ"ל כדי לעלות בחזרה אל השורש ולעדכן את השדות של כל הצמתים שעברנו בהם. סיבוכיות זמן של הפונקציה היא O(log(n)) במקרה הגרוע ולכן לא משפיע על סיבוכיות המחיקה או הכנסה.

Predecessor

- סיבוכיות זמן: O(log(n)). הפונקציה הזו מחזירה את המקדם של הצומת Node בסדר הממוין של הצמתים. אם המקדם שלו לא קיים היא מחזיר Null. הפונקציה הולכת למקסימום של התת עץ השמאלי של הצומת אם קיים בו תת עץ, אחרת היא עולה למעלה בעץ עד שהיא עושה פנייה שמאלה כלומר עד שהיא מוצאת את הצומת הראשון כך שהצומת המקורי הוא בתת עץ הימני שלו. וככה אותו צומת הוא העוקב. זו לפי השיטה שנלמדת בכיתה. ישנם שני מקרים גרועים: מקרה ראשון: אם בתת עץ השמאלי של הצומת אנו יורדים ימינה את כל גובה

העץ להגיע למקסימום. מקרה שני: אם לצומת אין בן שמאלי ובעליה למעלה אנו לעולם לא פונים שמאלה עד שנגיע לשורש. במקרה הזה גילינו שאין לצומת עוקב. בשני המקרים עברנו בגובה העץ ולכן לפי החוקיות של עץ AVL קיבלנו סיבוכיות O(log(n)).

Successor

- פונקציה זאת מוצאת את העוקב של הצומת המבוקש. סיבוכיות של פונ' זו היא O(logn) במקרה הגרוע. אנו נתחיל בצומת המבוקש ויש שני מקרים: אם לצומת יש בן ימני אנו נרד אליו ואז שמאלה עד הסוף. כך נמצא את הצומת בעל המפתח המינימלי בתת עץ הימני של הצומת המבוקש וכך מצאנו את האיבר בעל המפתח הכי קטן שהוא גדול מהמפתח של הצומת המקורי ומצאנו את העוקב. אם אין לצומת המבוקש בן ימני אנו נתחיל לעלות מעלה. אם במהלך עלייתנו מעלה נפנה שמאלה, סימן שהצומת המקורי שלנו נמצא בתת עץ הימני של הצומת הנוכחי ועוד לא מצאנו צומת גדול מהמבוקש. אם נעלה בפניה ימינה, סימן שהצומת המקורי נמצא בתת עץ השמאלי של הצומת המקורי ומצאנו את הצומם בעל המפתח הכי קטן שהוא גדול מהמפתח של הצומת המקורי וזה העוקב. כל זה מבוסס על מה שנלמד בהרצאה. במקרה הגרוע, אנו עולים מעלה למעלה אל השורש או לחילופין מהשורש מטה אל עלה וכך קיבלנו סיבוכיות לוגריטמי. במקרה שלא מצאנו עוקב סימן שהצומת המבוקש הוא הצומת המקסימלי בעץ ונחזיר null.

succPrefixXor

- זאת פונקציה שמקבלת מפתח k ומחזירה את הxor של מספר הצמתים בעל מפתח קטן או שווה לk ובעל ערך true. הפונקציה פועלת בצורה נאיבית כך שהיא מתחילה בצומת בעל המפתח המינימלי $O(1)$ לפי הפונקציה min_node) ואז קוראת לפונקציה successor ($O(\text{logn})$) עד שאנחנו מגיעים לצומת בעל המפתח k. ובדרך אנחנו סופרים את הצמתים בעל הערך true. במקרה הגרוע נקרא n פעמים לפונקציה successor לכן סיבוכיות הזמן הינה $O(\text{nlogn})$

מחלקת AVLNode

getBalanceFactor

- סיבוכיות של פונ' זו היא $O(1)$. מטרת הפונ' היא להחזיר את הbalance factor. אם העץ ריק, נחזיר 0. אחרת, נחזיר את ההפרשי גבהים בין הבן השמאלי לבן הימיני של השורש. חישוב זה מסיבוכיות $O(1)$, כמו הבדיקה האם העץ ריק. לכן $O(1)$ היא הסיבוכיות הכללית.

(int k, Boolean val)AVLNode

- הבנאי של AVLNode מקבל מספר k וערך בוליאני val. הבנאי יוצר אוביקט חדש מסוג AVLNode ומעדכן את הערך שלו להיות val ואת המפתח שלו להיות k. בנוסף אנו מאתחלים את השדה height של הצומת להיות 0 ואת השדה trues_in_sub_tree להיות 1 או 0 כתלות בvala. בנוסף במקרה אחד אנו רוצים ליצור צומת וירטואלי (שלא חשוף למשתמש). המימוש שלנו היא כך שהערך של הצומת הוירטואלי הוא null והמפתח שלו הוא 1-. ככה אנחנו נזהה

בהמשך שאנחנו בצומת וירטואלי ויש רק אחד כזה. לכן במקרה שיוצרים את הצומת הוירטואלי (פעם אחת בלבד) אנו נקבע שהגובה שלו הוא 1- וה trues_in_sub_tree הוא 0.

getTrues_in_sub_tree

- הפונקציה הזאת מחזירה את השדה trues_in_sub_tree כלומר את מספר הצמתים בעל ערך true בתת עץ של אותו צומת כולל הצומת עצמו. סיבוכיות זמן: $O(1)$ משום שיש לנו שדה המתחזק ערך זה.

updateTrues_in_sub_tree

- הפונקציה הזאת מעדכנת את השדה trues_in_sub_tree על ידי כך שהיא סוכמת את השדה trues_in_sub_tree של שתי הבנים שלו (אם קיימים) ומוסיפה 1 אם לצומת עצמו יש ערך true. ככה אנו מתחזקים את השדה כתלות אך ורק בבנים הישירים. סיבוכיות זמן: $O(1)$ משום שיש לנו שדה המתחזק ערך זה.

getKey

- החזרת המפתח של node. סיבוכיות של פונ' זו היא $O(1)$, משום שיש לכל node ישנו שדה המייצג את המפתח שלו.

getValue

- החזרת הערך של node. סיבוכיות של פונ' זו היא $O(1)$, משום שיש לכל node ישנו שדה המייצג את הערך שלו.

setLeft/getLeft

- החזרת/עדכון הבן השמאלי של node. סיבוכיות של פונ' זו היא $O(1)$ משום שלכל node ישנו שדה המייצג את הבן השמאלי. לכן, ניתן לעדכנו או להחזירו.

setRight/getRight

- החזרת/עדכון הבן הימיני של node. סיבוכיות של פונ' זו היא $O(1)$, משום שלכל node ישנו שדה המייצג את הבן הימיני. לכן, ניתן לעדכנו או להחזירו.

setParent/getParent

- החזרת/עדכון ההורה של node. סיבוכיות של פונ' זו היא $O(1)$, משום שלכל node ישנו שדה המייצג את ההורה שלו. לכן, ניתן לעדכנו או להחזירו.

isRealNode

- סיבוכיות של פונ' זו היא $O(1)$ כפי שכבר ציינו, מפני שפעולת ההשוואה היחידה שאנחנו מבצעים היא מסיבוכיות זו.

getHeight/setHeight

- החזרת/עדכון הגובה של node. סיבוכיות של פונ' זו היא $O(1)$, משום שלכל node ישנו שדה המייצג את הגובה שלו. לכן, ניתן לעדכנו או להחזירו.

updateHeight

- אם הnode הוא Node אמיתי, נעדכן את הגובה שלו להיות המקסימום בין הגובה של הבן הימיני שלו, לבין השמאלי שלו + 1. השוואת הגבהים הינה מסיבוכיות $O(1)$ וכך גם פעולת החיבור. לכן, הסיבוכיות הינה $O(1)$

מדידות

מדידה ראשונה

מספר סידורי	עלות prefixXor ממוצעת כל הקריאות	עלות succPrefixXor ממוצעת כל הקריאות	עלות prefixXor ממוצעת 100 קריאות ראשונות	עלות succPrefixXor ממוצעת 100 קריאות ראשונות
1	39	378	39	374
2	49	793	44	450
3	51	1210	50	509
4	44	1545	33	650
5	46	2018	46	547

**התוצאות בננו-שניות. המדידות התבצעו בו זמנית וכמו כן, התוצאות נמדדו אחרי כ100 הרצות של המדידות.

הסבר לתוצאות ומסקנות: אנו הסברנו למעלה שסיבוכיות הזמן של prefixXor היא לוגריתמי במספר האיברים ואנחנו רואים את זה בתוצאות (הן מקבילות גם לתוצאות ממוצע זמן פעולת ההכנסה למטה שהיא גם לוגריתמי). הסברנו גם למעלה שSuccPrefixXor פועל בסיבוכיות זמן של O(nlogn). ולכן הזמנים גדולים יותר וגם עולים לפי מספר האיברים. (בprefixXor אנו טוענים כי לפי גודל מספר האיברים בעץ שזמן הפעולה גדלה אבל בהפרשים שקטנים וקשים לזהות את העלייה בזמנים אבל היא שם. ב3 כאשר הכנסנו 300 איברים אקראיים יש תוצאה חריגה העלולה לקרות כאשר מדובר באיברים אקראיים. דוגמא אחת לכך היא כאשר הרבה מהאיברים שהוכנסו הם בעלי ערך true ולכן חיבור הספירה שלהם היא במספרים יותר גדולים ולכן לוקחת יותר זמן.) במקרה שאנו בודקים את ממוצע עלות הפעולות עבור 100 קריאות הראשונות אנו מבינים שבממוצע יש פחות איברים הקטנים מהאיבר עליו מריצים את הפונקציה ולכן התוצאות של prefixXor במקרה הזה יותר קטנים וגם של succPrefixXor.

מדידה שנייה

מספר סידורי	עץ AVL סדרה חשבונית	עץ ללא מנגנון איזון סדרה חשבונית	עץ AVL סדרה מאוזנת	עץ ללא מנגנון איזון סדרה מאוזנת	עץ AVL סדרה אקראית	עץ ללא מנגנון איזון סדרה אקראית

1	40	816	49	42	91	69
2	58	1892	46	42	97	73
3	38	2871	50	42	104	91
4	38	3925	56	44	115	80
5	38	4883	49	42	113	83

**התוצאות בננו-שניות. המדידות התבצעו בו זמנית וכמו כן, התוצאות נמדדו אחרי כ100 הרצות של המדידות.

הסבר לתוצאות ומסקנות: אנו רואים כי בסדרה חשבונית העץ ללא מנגנון איזון יצור עץ לא מאוזן כך שהגובה שלו יהיה n. אבל בעץ AVL הגובה תמיד נשאר log(n) לכן פעולות הכנסה תמיד יהיו בסיבוכיות זמן של O(log(n)). כמובן שיש בעץ AVL פעולות נוספות כמו תיקונים וגלגולים אבל הוכחנו שהן עולות גם סיבוכיות לוגריתמי במקרה הגרוע לכן במקרה הגרוע אנו מבצעים בערך 2log(n)) פעולות בממוצע בפעולת הכנסה. בנוסף אנחנו רואים שההכנסה בעץ ללא מנגנון איזון מתבצע בזמן שהוא מקביל למספר האיברים (1000, 2000... , 5000). בהכנסה של סדרה מאוזנת שתי העצים יישארו מאוזנים וכך הזמן הממוצע של הפעולות של שני העצים היא בערך לוגריתמי ביחס למספר האיברים. בסדרה אקראית למדנו בהרצאה שעץ ללא מנגנון איזון שומר על איזון רלטיבי ולכן הפעולות מתבצעות בזמן לוגריתמי בתוחלת ולכן בממוצע הן בפועל בערך פי שנים מהזמנים בהכנסת סדרה מאוזנת אבל עדיין לוגריתמי במונחי O. העץ AVL גם שומר על ממוצע זמן לוגריתמי במונחי O אבל בפועל יש לה יותר פעולות כמו פעולות איזון ותחזוק שדות כמו גובה ושדות אחרות לצורך סעיפים אחרים במטלה. לכן בפועל הוא טיפה יותר איטי מהעץ ללא מנגנון איזון.