מסמך תיעודים

סטודנט 1: עלי שוואהנה 326683885

סטודנט 2: מוחמד מנסור 324293331

<u>מחלקה ראשונה: AVLNode</u>

תיאור: מחלקה שמייצגת צמתים בעץ AVL, כלומר כל אובייקט מטיפוס AVLNode הינו צומת בעץ AVL.

שדות:

Key – המפתח של הצומת.

value – הערך של הצומת.

left – מצביע לבן השמאלי של הצומת.

right – מצביע לבן הימני של הצומת.

parent – מצביע להורה של הצומת.

height – גובה הצומת בעץ.

<u>מתודות:</u>

__init__(self, key, value)

מתודת בנאי, שמקבלת מפתח key וערך value ומייצרת צומת חדש ומחזירה אותו. המפתח של הצומת המיוצר הוא key והערך value, וגובהו 1-, וללא בנים (בן שמאלי ובן ימני שמצביעים ל- null), וללא הורה (ההורה מצביע ל- null).

סיבוכיות: ההשמות מתבצעות בזמן קבוע ולכן סיבוכיות הפונקציה הינה $O(1)$.

is_real_node(self)

פונקציה שמחזירה שקר אם הצומת הוא virtual node ואחרת, מחזירה אמת.

סיבוכיות: הפונקציה מבצעת בדיקה אחת בזמן קבוע ולכן הפונקציה מסיבוכיות $O(1)$.

מחלקה שנייה: AVLTree

root – מצביע לשורש העץ.

tree_size– שדה שמחזיק את גודל העץ, כלומר מספר הצמתים שנמצאים בעץ.

max – מצביע לצומת המקסימלי בעץ.

מתודות:

__init__(self)

מתודת בנאי, שמייצרת עץ חדש ומחזירה אותו. השורש של העץ המיוצר הוא None וה-size שווה 0.

סיבוכיות: המטודה מסיבוכיות $O(1)$.

get_root(self)

מתודת getter, שמחזירה את שורש העץ.

סיבוכיות: המתודה רק מחזירה מצביע לשורש, ולכן מסיבוכיות $O(1)$.

tree_size (self)

פונקציה שמחזירה את הערך השמור בשדה tree_size של העץ.

סיבוכיות: המתודה רק מחזירה את הערך השמור בשדה tree_size, ולכן מסיבוכיות $O(1)$.

max_node(self)

פונקציה שמחזירה מצביע לצומת בעל המפתח המקסימלי בעץ.

סיבוכיות: בשורה הראשונה מתבצעת השמה בזמן קבוע. לולאת ה- while מבצעת $O(height) = O(\log n)$ איטרציות שבכל אחת מהן מתבצעות שתי בדיקות בזמן קבוע,

השמה אחת בזמן קבוע, וקריאה ל- *is_real_node* שגם היא מתבצעת בזמן קבוע. ולכן הלולאה רצה ב- $O(\log n)$ זמן, ולכן הפונקציה מסיבוכיות $O(\log n)$.

get_height(self, node)

פונקציה שמקבלת מצביע לצומת *node* שנמצא בעץ, ומחזירה את הגובה שלו בעץ.

סיבוכיות: במקרה הגרוע ביותר, כאשר המצביע אינו מצביע ל- *None* ואינו מצביע לצומת וירטואלי, הפונקציה תצטרך לבצע גם בדיקה וגם קריאה ל- *is_real_node* ולאחר מכן לגשת לשדה הגובה של הצומת. שתי פעולות הראשונות מתבצעות בזמן קבוע, וכך גם הגישה לשדה הגובה. ולכן בסה״כ הפונקציה מסיבוכיות $O(1)$.

get_balance_factor(self, node)

פונקציה שמקבלת מצביע לצומת בעץ ומחזירה את ההפרש בין הגבהים של הבן השמאלי של הצומת והבן הימני שלו.

סיבוכיות: הפונקציה בודקת אם המצביע מצביע על *None*, ובמקרה הגרוע ביוצר (כאשר אינו מצביע על *None*), הפונקציה תבצע שתי קריאות ל- *get_height* ותבצע גם פעולה אריתמטית אחת. הבדיקה הראשונה מתבצעת בזמן קבוע, כך גם שתי הקריאות (ניתחנו את סיבוכיות הפונקציה), וגם הפעולה האריתמטית. ולכן הפונקציה מסיבוכיות $O(1)$.

update_height(self, node)

פונקציה שמקבלת מצביע לצומת בעץ. אם המצביע מצביע ל- *None* או לצומת וירטואלי.
אחרת, הפונקציה תעדכן את הגובה של הצומת, בעזרת הגבהים של שני הבנים שלו.
הפונקציה תחזיר *None*. נשים לב שגובהו של הצומת הוא כגובה הבן הגבוה ביותר פלוס 1.

סיבוכיות: הפונקציה בודקת אם המצביע מצביע ל- *None*, ובודקת אם הוא צומת וירטואלי. במקרה הגרוע ביותר (כאשר הצומת אמיתי), מתבצעות שתי קריאות ל- *get_height* ושתי הגשות לשדות של הצומת. שתי הקריאות מתבצעת בזמן קבוע כל אחת, וכך גם ההגשות לשדות. בנוסף, מתבצעת קריאה ל- *max* בזמן קבוע, ועוד פעולה אריתמטית. ולבסוף מתבצעת השמה לערך המעודכן לתוך שדה הגובה, שגם זה מתבצע ב- $O(1)$. בסה״כ, הפונקציה מסיבוכיות $O(1)$.

rotate_left(self, node)

פונקציה שמקבלת מצביע לצומת *node* בעץ, ומבצעת *left rotation* ביחס לצומת זה. לפי האלגוריתם שראינו בשיעור.

סיבוכיות: הפונקציה מבצעת מספר סופי של עדכוני מצביעים ומספר סופי של גישות לשדות של צמתים (ועדכונם, בחלק מהמקרים). פעולות אלו מתבצעות בזמן קבוע. בנוסף, מתבצעות לכל היותר שתי בדיקות למצביעים (בודקות אם מצביעים ל- *None*) ולכל היותר בדיקה אחת שמשווה בין שני צמתים. כל הבדיקות מתבצעות בזמן קבוע. מתבצעות גם שתי קריאות ל- *update_height* בזמן קבוע (כפי שהראינו קודם). ולכן הפונקציה מסיבוכיות $O(1)$.

rotate_right(self, node)

פונקציה שמקבלת מצביע לצומת *node* בעץ, ומבצעת *left rotation* ביחס לצומת זה. לפי האלגוריתם שראינו בשיעור.

סיבוכיות: מאותם השיקולים שבניתוח הסיבוכיות של *rotate_left*, נקבל כי הפונקציה מסיבוכיות $O(1)$.

create_virtual_node(self)

פונקציה שמייצרת צומת וירטואלי (*virtual node*) ומחזירה אותו

סיבוכיות: ייצור צומת מתבצע בזמן קבוע. ולכן הפונקציה מסיבוכיות $O(1)$.

rebalance(self, node)

הפונקציה מקבלת מצביע לצומת מסוים בעץ, ומחזירה את מספר ה- *promotions* שנעשו במהלך ההרצה. הפונקציה מאזנת את העץ החל מהצומת שקיבלה, לפי האלגוריתם שראינו בשיעור, כלומר, מבצעת סיבוב אחד ו/או שני סיבובים ו/או סדרה של *promotions*.

סיבוכיות: שתי השורות הראשונות רצות בזמן קבוע. הלולאה מבצעת לכל היותר $O(\log n)$ איטרציות אשר בכל אחת מהן מתבצע מספר סופי של פעולות ובדיקות מסיבוכיות $O(1)$ כל אחת ומתבצע מספר סופי של קריאות ל- *get_balance_factor* בזמן קבוע גם הן. ולכן הפונקציה מסיבוכיות $O(\log n)$.

insert(self, key, value)

פונקציה שמקבלת מפתח *key* וערך *value* ומכניסה לעץ צומת חדש שהמפתח שלו הוא *key* ושדה הערך שלו (*value*) מכיל את *value*. הפונקציה תחזיר מצביע לצומת החדש ומספר הקשתות בהן עברנו לאורך כל ההרצה וגם מספר ה- *promotions* שעשינו לאורך כל ההרצה (נעשו בתוך ה- *rebalance*). הצומת החדש יוסף כעלה. קודם כל מחפשים את המקום המתאים בעץ להכנסת העלה החדש, חיפוש זה יתבצע באמצעות חיפוש בינארי שהרי עץ *AVL* הוא בפרט גם עץ חיפוש בינארי. ולאחר מכן, נכניס את הצומת למקום הנכון.

סיבוכיות: ראשית ניצר צומת חדש בזמן קבוע, ונבדוק אם העץ הוא עץ ריק. אם כן, נבצע מספר סופי של פעולות מסיבוכיות $O(1)$ ונסיים. אחרת, נבצע לולאת *while* שתבצע $O(height) = O(\log n)$ איטרציות, ובכל איטרציה יתבצעו פעולות מסיבוכיות $O(1)$ (שהרי גם *is_real_node* רצה בזמן קבוע). לאחר מכן יתבצעו מספר סופי של בדיקות ופעולות אשר עולות זמן קבוע כל אחת. ולבסוף נבצע *rebalance* בסיבוכיות $O(\log n)$, ולכן בסה״כ הפונקציה מסיבוכיות $O(\log n)$.

get_min(self, node)

פונקציה שמקבלת מצביע node לשורש של תת-עץ, ומחזירה מצביע לצומת בעל המפתח המינימלי בתת העץ.

סיבוכיות: בשורה הראשונה מתבצעת השמה בזמן קבוע. לולאת ה- while מבצעת $O(height) = O(\log n)$ איטרציות (שהרי המקרה הגרוע ביותר הוא המקרה שבו node הוא השורש של העץ הראשי) שבכל אחת מהן מתבצעת בדיקה אחת בזמן קבוע, השמה אחת בזמן קבוע, ושתי קריאות ל- *is_real_node* שגם הן מתבצעות בזמן קבוע. ולכן הלולאה רצה ב- $O(\log n)$ זמן, ולכן הפונקציה מסיבוכיות $O(\log n)$.

delete(self,node)

פונקציה שמקבלת מצביע לצומת בעץ, ומוחקת אותו מהעץ.

מקרה 1: אם הצומת הוא עלה, מוחקים אותו מהעץ ומבצעים איזון על ההורה.

מקרה 2:אם לצומת בן יחיד, נעדכן את המצביע של ההורה שמצביע לצומת (המצביע לבן השמאלי או הימני) להצביע לבן היחיד שלו (של הצומת) ומבצעים איזון על ההורה.

מקרה 3: (לצומת שני בנים), נחליף את הצומת ב- *successor* שלו, שהרי ל- *successor* יש לכל היותר בן יחיד ולכן מחיקתו תתבצע בהתאם לאחד משני המקרים הקודמים. ומבצעים איזון להורה המקורי של ה- *successor* (לפני ההחלפה).

סיבוכיות: ראשית, הפונקציה מבצעת פעולות שלוקחות זמן קבוע. (כולל קריאה ל- *is_real_node*). לאחר מכן, בכל אחד משלושת המקרים, מתבצעות בדיקות וקריאות שלוקחות זמן קבוע (השוואות צמתים, קריאות ל- *is_real_node*, וגם הגשות לשדות של צמתים ועדכונם) בנוסף לקריאות ל- *create_virtual_node* שגם הן מסיבוכיות $O(1)$. בנוסף, בכל אחד משלושת המקרים מתבצעת קריאה ל- *rebalance* שמתבצעות ב- $O(\log n)$. בנוסף, במקרה השלישי נבצע קריאה ל- *get_min* אשר גם היא מתבצעת ב- $O(\log n)$. ובסה״כ הפוהקציה מסיבוכיות $O(\log n)$.

search(self, key)

פונקציה שמקבלת מפתח של צומת שנמצא בעץ, מחפשת את הצומת בעל מפתח זה ומחזירה אותו, ומחזירה גם את האורך של המסלול מהשורש עד לצומת (בקשתות). החיפוש מתבצע לפי אלגוריתם חיפוש בינארי שהרי עץ *AVL* הוא בפרט גם עץ חיפוש בינארי.

סיבוכיות: שתי הפעולות הראשונות מתבצעות בזמן קבוע. לולאת ה- *while* מבצעת לכל היותר $O(height) = O(\log n)$ איטרציות, שהרי בכל איטרציה יורדים רמה אחת בעץ. בכל איטרציה מתבצעות פעולות שלוקחות זמן קבוע (פעולות אריתמיטיות וקריאה ל- *is_real_node*), ולכן הפונקציה מסיבוכיות $O(\log n)$.

avl_to_array(self)

הפונקציה מחזירה מערך ממוין (in-order) ע״פ המפתחות (של האיברים במילון) כאשר כל איבר מיוצג ע״י זוג סדור של (key, value). הפונקציה הינה פונקציית מעטפת, אשר קוראת ל- *in_order_traversal* .

סיבוכיות: הפונקציה מייצרת מערך ריק בזמן קבוע, ולאחר מכן קוראת ל- *get_root* אשר לוקחת זמן קבוע. בנוסף, מתבצעת קריאה לפונקציה *in_order_traversal* שלוקחת זמן $O(n)$ ולכן הפונקציה מסיבוכיות $O(n)$.

in_order_traversal(node)

פונקציית עזר רקורסיבית. הפונקציה מקבלת שורש של תת-עץ ומחזירה מערך ממוין כמתואר בהסבר של הפונקציה העוטפת *avl_to_array*. בכל צומת, באופן רקורסיבי, הפונקציה מוסיפה למערך את האיברים שמייצגים את הצמתים שנמצאים בתת העץ השמאלי של הצומת הנוכחי, לאחר מכן מוסיפה איבר שמייצג את הצומת הנוכחי ואחר כך מוסיפה למערך איברים שמייצגים את הצמתים שבתת העץ הימני של הצומת הנוכחי.

סיבוכיות: הפונקציה תעבור על כל צומת בעץ פעם אחת בדיוק, ותעבור גם על הבנים הווירטואליים (מספר העלים הווירטואליים $<$ (מספר הצמתים בעץ) $* 2$). ובכל צומת כזה, מתבצעת עבודה שלוקחת זמן קבוע (בדיקת מקרה העצירה לוקחת זמן קבוע וכך גם הקריאות הרקורסיביות וההכנסה למערך). ולכן הפונקציה מסיבוכיות $O(2n) = O(n)$.

split(self, x)

הפונקציה מקבלת מצביע לצומת x בעץ. ומפצלת את העץ לשניים $t_1,\ t_2$ כך ש- t_1 יכיל את המפתחות הקטנים מ- x , ו- t_2 יכיל את המפתחות הגדולים מ- x. הפונקציה מחזירה את $(t_1,\ t_2)$. ראשית נסמן $curr = x$ ונסמן את תת העץ השמאלי של $curr$ ב- t_1 ואת תת העץ הימני שלו ב- t_2, ונתחיל לעלות למעלה. בכל פעם שעלינו למעלה (כלומר $curr =$ $curr.parent$) אם $curr$ הוא בן ימני של ההורה, אזי t_1 יהפוך להיות עץ ששורשו הוא

$curr.parent$, בנו השמאלי הוא הבן השמאלי שלו בעץ הראשי, ובנו הימני הוא השורש של t_1 לפני שעלינו למעלה. אם $curr$ הוא בן שמאלי של ההורה, אזי t_2 יהפוך להיות עץ ששורשו הוא $curr.parent$, בנו הימני הוא הבן הימני שלו בעץ הראשי, ובנו השמאלי הוא השורש של t_2 לפני שעלינו למעלה. נמשיך כך עד שנגיע לשורש.

סיבוכיות: שתי הבדיקות הראשונות מתבצעות בזמן קבוע, כנ״ל גם יצירת העצים החדשים. הבדיקה השלישית והרביעית מתבצעת בזמן קבוע אף הן, ובמקרה הגרוע ביותר (כאשר התנאים באחד הבדיקות מתקיימים) תתבצע קריאה ל- *get_tree_size_subtree* בסיבוכיות $O(n)$, ויתבצעו פעולות נוספות שרצות בזמן קבוע. בכל איטרציה של לולאת ה-*while* נעלה רמה אחת בעץ, ולכן היא מבצעת לכל היותר $O(height) = O(\log n)$ איטרציות. מלבד הקריאות ל- *rebalance* ול- *get_tree_size_subtree*, כל הפעולות שמתבצעות בלולאה הן פעולות מסיבוכיות $O(1)$. נשים לב שבכל פעם שמבצעים *rebalance* אנחנו מבצעים אותה על שורש של עץ (t_1 או t_2) ולכן הפעולה מתבצעת בזמן קבוע (בפרט ב- $O(\log n)$) שזה גם לא ישנה את הזמן הכולל) לפי הדרך בה מימשנו את הפעולה (כמפורט למעלה). נשים לב גם, שבכל קריאה לפונקציה *get_tree_size_subtree*, אנחנו קוראים לה על שורשו של תת עץ מסוים, כאשר תתי העצים האלה הם זרים (לפי האלגוריתם המפורט למעלה), ומכאן נקבל כי בסה״כ ולאורך כל הלולאה, הפונקציה *get_tree_size_subtree* תרוץ בזמן $O(n) = O\big(\text{סכום הצמתים בכל תתי העצים}\big)$. ולכן בסה״כ הפונקציה מסיבוכיות $O(\log n + n + n) = O(n)$.

get_tree_size_subtree(self, node)

פונקציה שמקבלת מצביע לצומת בעץ $node$, ומחזירה את מספר הצמתים בתת העץ אשר שורשו הוא $node$. בנוסף, הפונקציה מעדכנת את הגבהים של כל הצמתים שנמצאים בתת עץ זה. בכל צומת, הפונקציה מחשבת באופן רקורסיבי את מספר הצמתים בתת העץ השמאלי של הצומת הנוכחי, ואת מספר הצמתים בתת העץ הימני של הצומת הנוכחי, סוכמת אותם ומוסיפה 1 (השורש), בנוסף לכך, בכל צומת הפונקציה מעדכנת את הגובה של הצומת הנוכחי, בעזרת הגבהים של שני הבנים שלו. נשים לב שגובהו של צומת הוא כגובה הבן הגבוה ביותר פלוס 1.

סיבוכיות: הפונקציה תעבור בכל צומת פעם אחת בדיוק, ותעבור גם בכל הצמתים הווירטואליים, כל אחד פעם אחת בדיוק. מספר העלים הווירטואליים $<$ (מספר הצמתים בעץ) $*$ 2 . בכל צומת הפונקציה מבצעת עבודה שלוקחת זמן קבוע - בדיקה אם המצביע

מצביע על *None*, קריאות ל- *is_real_node*, פעולה אריתמטית ושתי קריאות רקורסיביות, וקריאה אחת לפונקציית *max*. ולכן הפונקציה מסיבוכיות $O(3n) = O(n)$.

join(self, tree2, key, val)

הפונקציה מקבלת עץ *tree2*, מפתח *key* וערך *value*. הפונקציה מייצרת צומת חדש עם המפתח והערך שקיבלה. נתון כי המפתחות באחד העצים קטנים ממש מכל המפתחות בעץ השני ונתון כי *key* נמצא ביניהם ממש.

מקרה 1: לכל *key1* מפתח ב- *tree1* (העץ הראשי) ולכל *key2* מפתח ב- *tree2* מתקיים $key1 < key < key2$. **1.1)** גובהו של *tree1* גדול או שווה לגובהו של *tree2*. נתחיל משורשו של *tree1* ונתחיל לרדת ממנו ימינה (לבן הימני בכל פעם) עד שנגיע לצומת שגובהו שווה לגובהו של העץ השני או קטן ממנו באחד. **1.2)** נתחיל משורשו של *tree2* ונתחיל לרדת ממנו שמאלה (לבן השמאלי בכל פעם) עד שנגיע לצומת ($curr$) שגובהו שווה לגובהו של העץ השני או קטן ממנו באחד.

מקרה 2: לכל *key1* מפתח ב- *tree1* (העץ הראשי) ולכל *key2* מפתח ב- *tree2* מתקיים $key1 > key > key2$. **2.1)** גובהו של *tree1* גדול או שווה לגובהו של *tree2*. נתחיל משורשו של *tree1* ונתחיל לרדת ממנו שמאלה (לבן השמאלי בכל פעם) עד שנגיע לצומת שגובהו שווה לגובהו של העץ השני או קטן ממנו באחד. **2.2)** נתחיל משורשו של *tree2* ונתחיל לרדת ממנו ימינה (לבן הימני בכל פעם) עד שנגיע לצומת ($curr$) שגובהו שווה לגובהו של העץ השני או קטן ממנו באחד.

כאשר הגענו לצומת המתאים בעץ המתאים: במקרה 1.1 וגם במקרה 2.2 הבן הימני של *curr.parent* יהיה הצומת החדש, הבן השמאלי של הצומת החדש יהיה *curr* והימני יהיה שורשו של *tree2* ב- 1.1 או שורשו של *tree2* ב- 2.2 . במקרה 1.2 וגם במקרה 2.1 הבן השמאלי של *curr.parent* יהיה הצומת החדש, הבן הימני של הצומת החדש יהיה *curr* והשמאלי יהיה שורשו של *tree1* ב- 1.2 או שורשו של *tree2* ב- 2.1 . ונחזיר את העץ הגבוה ביותר (העץ אליו חיברנו את העץ השני).

סיבוכיות:

שלושת הבדיקות הראשונות מתבצעות בזמן קבוע. במקרה הגרוע ביותר (בו התנאי של אחת הבדיקות מתקיים) נבצע פעולת הוספה בזמן קבוע ופעולות נוספות, אשר כולן

מתבצעות בזמן קבוע. לאחר מכן מתבצע מספר סופי של פעולות אשר כל אחת מהן מתבצעת בזמן קבוע. בדיקת התנאי האחרון מתבצעת בזמן קבוע אף היא, בין אם התנאי מתקיים ובין אם לא (*first case or second case* לפי הסימונים בקוד) תתבצע לולאת *while* מאותה הסיבוכיות (בשני המקרים). הלולאה תבצע לכל היותר $O(\log n)$ איטרציות ובכל אחת מהן תבצע עבודה בזמן קבוע. בנוסף לכך, בשני המקרים אחרי הלולאה יתבצעו פעולות בזמן קבוע, וגם פעולת *rebalance*, וגם פעולת *max_node* שמתבצעות בזמן $O(\log n)$ כל אחת ולכן הסיבוכיות היא $O(\log n)$.

finger_insert(self, k, v)

הפונקציה מקבלת מפתח וערך. הפונקציה מחזירה מצביע לצומת החדש וגם מספר הקשתות שנמצאות על מסלול החיפוש, וגם מספר ה-*promotions* שעשינו לאורך ההוספה (שנעשו בפעולת האיזון). הפונקציה מבצעת חיפוש על העץ לפי אותו האלגוריתם של *finger_search* בדיוק. הפונקציה מייצרת צומת חדש עם המפתח והערך שקיבלנו ותולה אותו כעלה במקום המתאים, כלומר המקום שמצאנו כי הוא המקום המתאים להכנסת האיבר כאשר ביצענו את אלגוריתם *finger_search*.

סיבוכיות: ראשית, מתבצעות שתי פעולות בזמן קבוע, ובדיקת תנאי בזמן קבוע, ובמקרה שהתנאי מתקיים מתבצע גם מספר סופי של פעולות שלוקחות זמן קבוע בנוסף לפעולת *max_node* בזמן $O(\log n)$. לאחר מכן, מתבצעת עוד בדיקה בזמן קבוע, ובמקרה הגרוע ביותר (בו תנאי הבדיקה אינו מתקיים), יתבצעו שתי לולאות *while*, הראשונה תבצע $O(\log n)$ איטרציות שבכל אחת מהן תתבצע עבודה בזמן קבוע, כנ״ל גם הלולאה השנייה. אחרי ה- *else* מתבצעות פעולות בזמן קבוע בנוסף לקריאה ל- *max_node* וקריאה ל- *rebalance* אשר כל אחת מהן מתבצעת ב- $O(\log n)$. ולכן הפונקציה מסיבוכיות $O(\log n)$.

finger_search(self, k)

הפונקציה מקבלת מפתח k ומחפשת אותו בעץ. הפונקציה מחזירה מצביע לצומת בעל המפתח k אם ישנו צומת כזה בעץ, ומחזירה את אורך המסלול מהשורש ועד אליו 1+ (האורך בקשתות). אחרת, הפונקציה מחזירה *None* ומספר הקשתות שעברנו עליהם לאורך כל מסלול החיפוש 1+. הפונקציה מתחילה מהאיבר המקסימלי בעץ, מחפשת בתת העץ שהאיבר הזה הוא שורשו (שבשלב הראשון הוא רק האיבר הנוכחי $current$), בכל שלב אנחנו עולים למעלה בעץ ובודקים אם $k \geq current.key$. אם כן, נחפש את המפתח בתת-

העץ ששורשו $current$, חיפוש כזה יעלה לנו $O(curremt.height)$ אחרת, נמשיך לעלות למעלה. אם הגענו לשורש, נחפש אותו בעץ הראשי.

סיבוכיות: בדיקת התנאי בשורה הראשונה לוקח זמן קבוע, לולאת ה- *while* מבצעת לכל היותר $O(\log n)$ איטרציות (במקרה שבו המשכנו לעלות עד שהגענו לשורש), ובכל איטרציה נבצע פעולות שלוקחות זמן קבוע. גם בדיקת התנאי שאחרי הלולאה מתבצע בזמן קבוע. לולאת ה- *while* השנייה תבצע גם היא לכל היותר $O(\log n)$ איטרציות (במקרה שבו המשכנו לעלות עד שהגענו לשורש וירדנו עד לאחד העלים) ובכל איטרציה תבצע פעולות שלוקחות זמן קבוע. ולכן, במקרה הגרוע, הפונקציה מסיבוכיות $O(\log n)$.

חלק ניסויי

שאלה 1

מספר סידורי i	עלות איזון במערך ממוין	עלות איזון במערך ממוין-הפוך	עלות איזון במערך מסודר אקראית	עלות איזון במערך עם היפוכים סמוכים אקראיים
1	430	430	385.9	426.6
2	873	873	776.65	862.2
3	1760	1760	1569.9	1734.95
4	3535	3535	3149.25	3495.5
5	7086	7086	6330.2	7002.9
6	14189	14189	12660.4	14019.8
7	28396	28396	25353.36	28070.65
8	56811	56811	50709.5	56164.3
9	113642	113642	101374.8	112314.55
10	227305	227305	202974.15	224678.6

החסם העליון על סך עלויות האיזון כולל גלגולים הוא O(מספר ההכנסות) שהרי ראינו בהרצאה שעלות אמורטייזיד של סדרת התיקונים שמתבצעים אחרי הכנסת איבר חדש היא $O(1)$. ולכן O(מספר ההכנסות) הוא חסם אמורטייזיד לעלות של סך כל סדרות התיקונים שיתבצעו לאורך כל סדרת ההכנסות. החסם אכן תואם לערכים שבטבלה, שהרי נוכל לראות שכאשר אורך המערך גדל לינארית (מוכפל ב-2), גם מספר ה- *promotions* גדל לינארית (מוכפל כמעט ב-2).

מספר הגלגולים לא משנה אסימפטוטית כיוון שלכל הכנסה $O(1)$ הינו חסם אמורטייזיד לעלות סדרת התיקונים הנדרשים שכוללת גם את הגלגולים וגם את ה-*promotions*. וזה הוא החסם העליון הקטן ביותר, מכאן שהוא גם חסם עליון למספר ה- *promotions*. ולכן הוספת עלות הגלגולים לא תשנה את העלות אסימפטוטית.

שאלה 2

מספר סידורי i	מספר היפוכים במערך ממוין	מספר היפוכים במערך ממוין-הפוך	מספר היפוכים במערך מסודר אקראית	מספר היפוכים במערך עם היפוכים סמוכים אקראיים
1	0	24531	12100.55	111.4
2	0	98346	49046.94	221.3
3	0	393828	197082.1	439.65
4	0	1576200	793090.25	888.85
5	0	6306576	3142891.85	1771.96

שאלה 3

מספר סידורי i	עלות חיפוש במערך ממוין	עלות חיפוש במערך ממוין-הפוך	עלות חיפוש במערך מסודר אקראית	עלות חיפוש במערך עם היפוכים סמוכים אקראיים
1	221	2694	2423	400.1
2	443	6272	5704.6	797.3
3	887	14316	13088.75	1600.35
4	1775	32180	30106.7	3214.65
5	3551	71460	67007.8	6426.9

שאלה 4-א

לכל $1 \le i \le n$, עבור איבר במקום j (כאשר $i > j$) שקטן מהאיבר במקום i (כלומר $A[j]$ נספר ב- d_i) אז הוא והאיבר באינדקס i מהווים היפוך שנספר ב- I.

בנוסף, לכל שני אינדקסים $i > j$, אם $A[i] < A[j]$, כלומר האיבר $A[j]$ מופיע לפני האיבר $A[i]$ והוא גדול ממנו, אז $A[j]$ נספר ב- d_i.

מכאן נובע ששני הצדדים סופרים בדיוק את אותם ההיפוכים.

שאלה 4-ב

כאשר אנחנו מחפשים את $A[i]$ בעץ, מכיוון שאנחנו מבצעים *finger-search*, נתחיל את החיפוש מהאיבר המקסימלי, ונעלה בעץ עד שנגיע לשורשו של תת העץ שצריך להכיל את $A[i]$ במידה והוא נמצא בעץ.

מקרה 1: אם $d_i = 0$ או $d_i = 1$ אזי החיפוש יעלה $O(1)$.

מקרה 2: כאשר אנחנו יודעים שישנם בעץ $d_i > 0$ איברים שגדולים מ- $A[i]$, נצטרך לעלות בעץ לכל היותר $\lceil \log_2 d_i \rceil$ איברים, לפי הגדרת עץ *AVL*. בנוסף אחרי שעלינו $\lceil \log_2 d_i \rceil$ והגענו לצומת x כלשהו, נצטרך לחפש את $A[i]$ בתת העץ ששורש x. תת עץ זה הוא בגובה $\lceil \log_2 d_i \rceil$ ולכן חיפוש בינארי בעץ יעלה לכל היותר $O(\lceil \log_2 d \rceil)$. מכאן נקבל כי סך העלות היא $2\lceil \log_2 d_i \rceil < 2(1+\log_2 d_i) = O(2+2\log_2 d_i) = O(\log d_i)$.

ולכן סך העלות הוא $O(\max(1, \log d_i))$.

ולכן סך עלויות החיפוש לאורך כל סדרת ההכנסות חסום ע״י:

$$\sum_{i=1}^{n} O(\max(1, \log d_i)) = \sum_{i=1}^{n} O(\log(d_i+2)) = O\left(\sum_{i=1}^{n}(\log(d_i+2))\right)$$
$$= O\left(\log\left(\prod_{i=1}^{n}(d_i+2)\right)\right)$$

כאשר המעברים נובעים מההבהרה, וחוקי לוגריתמים.

שאלה 4-ג

מאי-שוויון הממוצעים מתקיים:

$$\sqrt[n]{\prod_{i=1}^{n}(d_i+2)} \le \frac{\sum_{i=1}^{n}(d_i+2)}{n} \;\rightarrow\; \prod_{i=1}^{n}(d_i+2) \le \left(\frac{\sum_{i=1}^{n}(d_i+2)}{n}\right)^n$$

והמונוטוניות של פונקציית log ומחוקי לוגריתמים:

$$\log\left(\prod_{i=1}^{n}(d_i+2)\right) \le \log\left(\left(\frac{\sum_{i=1}^{n}(d_i+2)}{n}\right)^n\right) = n\log\left(\frac{\sum_{i=1}^{n}(d_i+2)}{n}\right)$$
$$= n\log\left(\frac{\sum_{i=1}^{n}(d_i) + \sum_{i=1}^{n}(2)}{n}\right) = n\log\left(\frac{I+2n}{n}\right)$$

שאלה 4-ד

נסמן ב- f_1 את הפונקציה שחוסמת מלמעלה את עלות האיזונים של סדרת n ההכנסות.

ונסמן ב- f_2 את הפונקציה שחוסמת מלמעלה את עלות החיפושים לאורך סדרת n ההכנסות.

$$f_1(n) = n\ , \quad f_2(n) = n\log\left(\frac{I + 2n}{n}\right)$$

מתקיים $f_1 = O(f_2)$. ומכיוון ש- $f_2 = \theta(f_2)$ אז $f_1 + f_2 = O(2f_2) = O(f_2)$ וגם $f_2 = O(f_1 + f_2)$. ולכן $f_1 + f_2 = \theta(f_2)$ ואכן עלות האיזונים זניחה ביחס לעלות החיפוש ואינה משנה אסימפטוטית.