Data Structures – Project #1

Oded Neeman – odedneeman – 207110479
Roy Mayan – roymayan – 206483554

חלק מעשי

המחלקה AVLNode

תיאור: AVLNode מייצגת צומת בעץ AVLTreeList.

שדות:

שדה	תפקיד
value	הערך (info) של הצומת.
left	הבן השמאלי של הצומת.
right	הבן הימני של הצומת.
parent	ההורה של הצומת.
height	הגובה של הצומת.
size	הדרגה של הצומת (כמות האיברים בתת-העץ שהצומת הוא שורשו).
is_real	האם הצומת מייצג צומת אמיתי בעץ (צומת שאינו וירטואלי).

מתודות:

מתודה	תיאור	סיבוכיות
__init__(value,is_real)	בנאי המחלקה. יוצרת צומת חדש בעל הערך value. אם is_real הוא True, הצומת שייווצר הוא צומת אמיתי (ולכן ייווצרו לו שני בנים וירטואליים). אחרת, הצומת שייווצר הוא צומת וירטואלי.	$O(1)$ מתבצעות בדיקות והשמות בזמן קבוע.
getLeft()	מחזירה את הבן השמאלי של הצומת.	$O(1)$ גישה לשדה בזמן קבוע.
getRight()	מחזירה את הבן הימני של הצומת.	$O(1)$ גישה לשדה בזמן קבוע.
getParent()	מחזירה את ההורה של הצומת.	$O(1)$ גישה לשדה בזמן קבוע.
getValue()	מחזירה את הערך של הצומת.	$O(1)$ גישה לשדה בזמן קבוע.
getHeight()	מחזירה את הגובה של הצומת.	$O(1)$ גישה לשדה בזמן קבוע.
getSize()	מחזירה את הדרגה של הצומת.	$O(1)$ גישה לשדה בזמן קבוע.
setLeft(node)	מעדכנת את הבן השמאלי של הצומת להיות node.	$O(1)$ השמה לשדה בזמן קבוע.
setRight(node)	מעדכנת את הבן הימני של הצומת להיות node.	$O(1)$ השמה לשדה בזמן קבוע.
setParent(node)	מעדכנת את ההורה של הצומת להיות node.	$O(1)$ השמה לשדה בזמן קבוע.
setValue(value)	מעדכנת את הערך של הצומת להיות value.	$O(1)$ השמה לשדה בזמן קבוע.
setHeight(h)	מעדכנת את הגובה של הצומת להיות h.	$O(1)$ השמה לשדה בזמן קבוע.
setSize(s)	מעדכנת את הדרגה של הצומת להיות s.	$O(1)$ השמה לשדה בזמן קבוע.

isRealNode()	מחזירה True אם הצומת אמיתי, False אחרת.	$O(1)$ גישה לשדה בזמן קבוע.
getBF()	מחזירה את ה-Balance Factor של הצומת (הפרש הגבהים בין הבן השמאלי של הצומת לבין הבן הימני שלו).	$O(1)$ גישה לגבהים של הבנים בזמן קבוע.
updateSize()	מעדכנת את הדרגה של הצומת לפי הדרגות של הבנים שלו. חישוב זה מבוסס על התכונה הלוקאלית: דרגה של צומת היא סכום הדרגות של בניו, בתוספת 1.	$O(1)$ גישה לדרגות של הבנים בזמן קבוע.
updateHeight()	מעדכנת את הגובה של הצומת לפי הגבהים של הבנים שלו. חישוב זה מבוסס על התכונה הלוקאלית: גובה של צומת הוא המקסימלי מבין הגבהים של בניו, בתוספת 1.	$O(1)$ גישה לגבהים של הבנים בזמן קבוע.

המחלקה AVLTreeList

תיאור: AVLTreeList מייצגת מימוש של ADT רשימה באמצעות עץ AVL. כל איבר ברשימה הזו מיוצג על ידי AVLNode.

שדות:

שדה	תפקיד
root	מצביע לשורש העץ (AVLNode).
first_node	מצביע לאיבר המינימלי בעץ (AVLNode).
last_node	מצביע לאיבר המקסימלי בעץ (AVLNode).

מתודות:

מתודה	תיאור	סיבוכיות
selectRec(x, k)	מחזירה את הצומת עם rank של k בתת העץ שהשורש שלו הוא x. מימוש זהה ל--Tree Select שראינו בהרצאה. מחזירה איבר מסוג AVLNode. תנאי קדם: העץ ששורשו x מכיל לפחות k איברים.	$O(logn)$ כמו שראינו עבור Tree-Select, לינארי בגובה העץ שבמקרה שלנו תמיד $O(logn)$.
getSuccessor(x)	מחזירה את האיבר העוקב של x ברשימה (מחזירה אותו כטיפוס AVLNode). אם ל-x יש ילד ימני, אז הולכים אליו ואז שמאלה עד שתת העץ נגמר. אם אין לו ילד ימני, אז עולים במעלה העץ עד הפעם הראשונה שהפנייה היא ימינה (נבדוק זאת כל פעם שנעלה לצומת ונבדוק אם הילד שעלינו ממנו הוא הבן הימני או השמאלי. אם הוא הבן השמאלי- אז פנינו ימינה).	$O(logn)$ לכל היותר נטפס את מלוא גובה העץ עד השורש. בכל צומת העבודה היא קבועה.
getPredecessor(x)	מוצאת את האיבר הקודם ל-x ברשימה (מחזירה אותו כטיפוס AVLNode). מימוש דומה לפונקציה הקודמת: אם יש ילד שמאלי, אז ללכת אליו ואז ימינה עד הסוף. אם אין- לעלות עד שפונים שמאלה.	$O(logn)$ לכל היותר נטפס את מלוא גובה העץ עד השורש. בכל צומת העבודה היא קבועה.
inOrderRec(node, lst)	פונקציה רקורסיבית שמוסיפה לרשימה lst את האיברים שבתת העץ של הצומת node, לפי סדר האינדקסים שלהם. אנו מוסיפים את ה-value של האיברים ולא את הצמתים עצמם כמובן. אם הצומת הוא לא אמיתי, זה סימן שהגענו לקצה של תת העץ, ולכן אין מה להוסיף בקריאה זו, ונסיים אותה. אחרת, נבצע קריאה רקורסיבית של הפונקציה על תת העץ השמאלי של node (וכך יתווספו לרשימה האיברים שלפני node), נוסיף את ה-value של node עצמו, ואז נקרא לפונקציה על תת העץ הימני של node.	$O(n)$ עוברים על כל איבר בתת העץ בדיוק פעם אחת. לכן לינארי בכמות האיברים בתת העץ.
sortedArrayToAVL(arr, l, r)	פונקציה רקורסיבית המקבלת מערך ממוין, ויוצרת עץ AVL חדש שמכיל את איברי המערך. הפונקציה מבצעת זאת בהתאם למימוש שראינו בתרגול 5, בעזרת לקיחת החציון, ובניית העץ באופן רקורסיבי על ידי "פיצול" לשתי קריאות על חלקי המערך עם האיברים הקטנים מהחציון והגדולים ממנו (בעזרת האינדקסים l, r).	$O(n)$ כפי שניתחנו בתרגול.
merge(lst1, lst2)	מיזוג שתי רשימות ממוינות לרשימה אחת (ממוינת), תוך מעבר על שתי הרשימות במקביל והכנסת האיבר הקטן יותר מבין שתי הרשימות.	$O(n+m)$ הלולאה רצה כסכום אורכי הרשימות.

mergeSort(lst)	פונקציה רקורסיבית הממיינת רשימה באמצעות חלוקה הרשימה לשתיים (בחציון), מיון רקורסיבי של כל תת-רשימה ומיזוגן.	$O(nlogn)$ בהתאם לניתוח שראינו במבוא מורחב.
__init__()	בנאי המחלקה. יוצרת עץ AVL ריק. מתבצעת השמה של ערכי ברירת-מחדל (None) לשדות המחלקה.	$O(1)$ השמות בזמן קבוע.
empty()	מחזירה True אם העץ ריק ו-False אם העץ לא ריק. עץ רק אמ"מ השורש שלו הוא None ולכן זו הבדיקה שנעשית.	$O(1)$ יש מצביע לשורש לכן הבדיקה נעשית בזמן קבוע.
retrieve(i)	מחזירה את ערך האיבר במקום ה-i ברשימה. הפונקציה קוראת ל-selectRec על שורש העץ, ודרגה מבוקשת $i+1$. בעץ, אנו "סופרים" את האיברים החל מ-1 (כך הוגדר rank של צומת) וברשימה החל מ-0. לכן אינדקס של איבר ברשימה יהיה 1 פחות מה-rank שלו בעץ ולכן קוראים ל-selectRec עם $i+1$. בסוף אנו מחזירים את ה-value של הצומת שקיבלנו מ-selectRec.	$O(logn)$ קריאה בודדת ל-selectRec, לכן אותה סיבוכיות שלה.
rotate(A, B)	הפונקציה כוללת את פעולות החלפת המצביעים המשותפות לסיבוב ימינה ולסיבוב שמאלה על מנת למנוע שכפול קוד. B הוא הצומת הפושע שגילינו לו פקטור איזון של 2, A הוא הבן הימני או השמאלי, בהתאם לתרחיש האיזון הדרוש, שיהיה מעורב בסיבוב כך ש-A יחליף את B בסופו. הפעולות נעשות בסוף הסיבוב: אנו מעדכנים את ההורה של A להיות ההורה של B, בודקים אם ההורה הוא שורש ומעדכנים שדות בהתאם, וכן מעדכנים את ההורה לשעבר של A כך שיהיה עכשיו ההורה של B (בצד המתאים), כמו כן מעדכנים את שדות הגובה וה-size של A ושל B. הם היחידים שמשתנים לאחר שמבצעים סיבוב.	$O(1)$ בודקים ומשנים שדות ומצביעים. כמות הבדיקות והשינויים חסומה.
rightRotation(criminal)	מבצעת סיבוב ימינה. B מסומן כצומת הפושע ו-A כבן השמאלי שלו (שבסוף הסיבוב יהיה במקום של B), AR הוא הבן הימני של A שמועבר להיות הבן השמאלי של B (יחד עם כל תת העץ שלו כמובן). מצביעי הילדים וההורים הדרושים מתעדכנים, חלקם במתודה עצמה וחלקם עם קריאה לפונקציית העזר rotate.	$O(1)$ בודקים ומשנים שדות ומצביעים. כמות הבדיקות והשינויים חסומה.
leftRotation(criminal)	מצבעת סיבוב שמאלה. באותו אופן כמו סיבוב ימינה רק שכעת A הוא הבן הימני של B וניגשים לבן השמאלי של A, AL, במקום לבן הימני.	$O(1)$ בודקים ומשנים שדות ומצביעים. כמות הבדיקות והשינויים חסומה.
performRotations(criminal)	פונקציה זו מקבלת צומת שידוע שהוא פושע (התגלה ב-fixTree שה-bf שלו הוא $\|2\|$). היא בודקת איזה סיבוב/ים נדרש לבצע ומבצעת אותו/אותם ע"י קריאה לפונקציות עזר. הבדיקה נעשית כמו שלמדנו: בודקים שני מקרים ($bf=2$ או $bf=-2$), ובכל מקרה כזה בדוקים את ה-bf של הבנים. המימוש תואם את המקרים השונים בדיוק כמו שראינו, על ידי קריאה לפונקציות הסיבוב בהתאם. נשים לב כי הפונקציה מטפלת בכל מקרים, גם בכנסה וגם במחיקה. כמו כן הפונקציה מחזירה את מספר הסיבובים שבוצעו, שהוא 1 או 2.	$O(1)$ כל שנעשה הוא בדיקת bf של הפושע ושל אחד הבנים, ואז פעולת סיבוב אחת או שתיים.
fixTree(start_node, is_delete=False)	הפונקציה מבצעת את כל התיקונים הדרושים על העץ כך שלאחר הפעולה שנעשתה העץ יהיה עץ AVL תקין וכל השדות יהיו מעודכנים. אנחנו מתחילים מההורה של start_node שמיושם ל-y, ומשם מטפסים כל פעם להורה של y עד שמגיעים לשורש. בכל צומת y, מעדכנים את הגובה ואת ה-size שלו. אם ה-balance factor לא חוקי (הפך ל-2 בערך מוחלט- הוא לא יכול להיות יותר מ-2), אז מבצעים את הסיבובים הדורשים ומוסיפים אותם לספירת הסיבובים שנעשו. לפונקציה ארגומנט אופציונלי של is_delete: כאשר הוא מקבל False: אזי התיקונים מתאימים לפעולת insert - אנו יודעים שנצטרך לעשות לכל היותר פעולת תיקון אחת, וכאשר מגיעים לצומת y שגובהה לא השתנה, אנו יודעים שכל גבהי הצמתים מעליה גם לא השתנו ולכן העץ חוקי. עם זאת, אנו עדיין צריכים לתקן את הגובה וה- size של הצמתים במסלול עד השורש, ולכן נמשיך להתקדם עליו. נדע שאנו לא צריכים לבדוק אם צריך לבצע תיקונים אחרים בעזרת המשתנה fixed_criminals שמתחיל כ-False ומשתנה ל- True אם מגיעים לצומת שגובהו לא השתנה או שמבצעים פעולת תיקון.	$O(logn)$ עוברים על צמתים כגובה העץ לכל היותר, כי מתחילים מצומת כלשהו בעץ ומטפסים עד השורש. בכל צומת שעוברים עליו נעשית עבודה קבועה- בדיקת שדות ו-bf, עדכון שדות, ולפעמים פונקציית performRotations שמתבצעת בזמן קבוע.

	אם is_delete=True: אזי התיקונים מתאימים לפעולת מחיקה או concat. ההבדל היחיד מתיקונים של פעולת הכנסה, הוא שייתכן שנתקן צומת או שנגיע לצומת שגובהו לא השתנה, אולם יהיו דרושים עוד תיקונים אחר כך, ולכן fixed_criminals יהיה תמיד False.	
insert(i, val)	מכניסה איבר עם הערך val למקום ה-i ברשימה. ראשית הופכים את הערך מהקלט לטיפוס של AVLNode שמיושם למשתנה z. נבדקים תחילת מקרי קצה: אם העץ ריק אזי מעדכנים את השדות של השורש, האיבר הראשון והאיבר האחרון להיות z. אם i=0, כלומר z מוכנס לתחילת הרשימה, אזי ניגשים לאיבר הראשון ומעדכנים את z להיות הבן שלו ואת הראשון להיות אבא שלו וכן מעדכנים את שדה האיבר הראשון להיות z. אם i=self.length(), כלומר z מוכנס לסוף הרשימה, אזי מבצעים פעולות דומות, רק הפעם z יהיה בן ימני של האיבר האחרון. שאר המקרים: אנו מגיעים לאיבר במקום ה-i ברשימה (בעץ הוא ב-rank ה-i+1), בעזרת הפונקציה selectRec. Z שלנו צריך להיות מוכנס בדיוק לפני האיבר הזה, לכן אם אין לו בן שמאלי אזי מגדירים את z להיות בנו השמאלי, ואם יש לו בן שמאלי אזי מגיעים לאיבר הקודם שלו ע"י פונקציית העזר getPredecessor, ואז מהגדרתו אין לו איבר ימני, וזה יהיה בדיוק המקום המתאים להכנסת z. בסוף פעולת ההכנסה, לא משנה באיזה תרחיש, נקראת הפונקציה fixTree עליה הרחבנו, ומוחזר ערך ההחזרה שלה שהוא מספר פעולות התיקון שנעשו.	$O(logn)$ במקרה הגרוע, ישנה קריאה אחת ל-selectRec, קריאה אחת ל-getPredecessor, וקריאה אחת ל-fixTree. כל הפונקציות האלה רצות בזמן לוגריתמי. חוץ מפונקציות אלה, ישנן פעולות של שינוי מצביעים בכמות חסומה ולכן ללא השפעה אסימפטוטית.
deleteLeafOrSingle(node, child)	הפונקציה מקבלת צומת שיש לו לכל היותר ילד אחד ומוחקת אותו ע"י bypass כפי שראינו בהרצאה. אם לפונקציה יש ילד אחד, אז child יהיה הילד הזה (באופן טכני - האמיתי מבין השניים). בודקים אם node היה הבן הימני או השמאלי של ההורה שלו ואז לפי זה מתקנים שכעת הילד בצד הזה יהיה child (אם node הוא עלה אז child הוא צומת ווירטואלי, לכן הפונקציה מתאימה לשני המקרים), ושההורה של child יהיה ההורה של node. ***הפונקציה לא תעבוד אם node הוא שורש העץ, אך בקוד שלנו לא תקרה סיטואציה כזו.**	$O(1)$ אנו מקבלים מצביע על הצומת, לכן כל הפעולות הן בדיקות של שדות ושינוי מצביעים בכמות חסומה.
delete(i)	הפונקציה מוחקת מהרשימה את האיבר במקום ה-i. אם i גדול/שווה מאורך הרשימה, אזי אין איבר באינדקס הזה ונחזיר 1- (תקף גם לכל קלט כאשר הרשימה ריקה). נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס $i+1$ (כאמור זה נובע ממהבדל בין מקום ברשימה ל-rank בעץ) ונסמן את הצומת ב-z. בודקים אם אחד הילדים של העץ לא אמיתי. אם זה המקרה: בודקים מי האמיתי ואז מפעילים את deleteLeafOrSingle עם קלט של הצומת והילד האמיתי (ואם אף אחד מהילדים לא אמיתי, אז לא משנה איזה מהילדים ניתן כקלט). כמו כן טיפלנו לפני כן במקרי קצה: אם z הוא שורש העץ, אזי נמחק אותו על ידי שינוי של מעט המצביעים הדרושים ונתחזק בהתאם את המצביע על השורש. אם z הוא האיבר הראשון או האחרון, גם נתחזק את המצביע המתאים בעזרת getSuccessor או getPredecessor בהתאמה, ואז נבצע מחיקה כמו קודם. אם ל-z יש שני ילדים (אמיתיים - לא ווירטואליים): נמצא את העוקב של z (בעזרת getSuccessor), לו בהגדרה יהיה בן אחד. נמחק אותו ונשים אותו במקום של z. נתקן את כל המצביעים והשדות כך שהעוקב יהיה במקום של z ושנמחק את הצומת מהמקום הקודם שלו, ובסוף גם נעדכן את שדות הגובה וה-size של העוקב. כמו כן נדאג לבדוק אם צריך לתחזק את שדה השורש של העץ למקרה שמחקנו אותו ונעשה זאת במקרה הצורך. לאחר פעולת המחיקה נקרא לfixTree כדי לתקן את העץ. נתחיל את התיקונים מ-z אם לא היו לו שני ילדים (אנחנו רוצים להתחיל מההורה שלו, אליו עדיין יש לנו מצביע מ-z למרות שהוא לא בעץ), ומהבן של ה-successor במקרה של-z היו שני ילדים (מצביע שדאגנו לשמור מבעוד מועד).	$O(logn)$ במקרה הגרוע אנחנו מבצעים קריאות ל-selectRec, getSuccessor (או getPredecessor אם z היה האיבר האחרון), ולבסוף קריאה ל-fixTree, פונקציות שכולן רצות בזמן לוגריתמי. לכל פונקציה מביניהן נקרא לכל היותר פעם אחת, ושאר הפעולות הן פונקציות עזר שעובדות בזמן קבוע (ונקרא לכמות חסומה שלהן), או פעולות שכוללות בדיקה ושינויים של מצביעים ושדות, גם כן בכמות קבועה.
first()	אנחנו מתחזקים מצביע לצומת שמייצג את האיבר הראשון ברשימה תמיד, ולכן קריאה לפונקציה תחזיר את ה-value של הצומת שהמצביע מצביע אליו (או None אם העץ ריק).	$O(1)$ שימוש בשדה קיים.
last()	כמו first, אנו מתחזקים מצביע גם לאיבר האחרון בעץ.	$O(1)$ שימוש בשדה קיים.

listToArray()	מחזירה מערך שבו איברי הרשימה לפי סדר האינדקסים שלהם. המערך ריק אם הרשימה ריקה. הפונקציה יוצרת מערך ריק. אם הרשימה ריקה היא מחזירה אותו ואם לא, היא קוראת לפונקציה inOrderRec על שורש הרשימה והמערך הריק שיצרנו.	$O(n)$ מה שיוצר את הסיבוכיות היא הקריאה לפונקציה inOrderRec אותה ניתחנו.
length()	מחזירה את מספר האיברים ברשימה. מספר האיברים ברשימה הוא מספר הצמתים (הלא ווירטואליים) בעץ שמייצג אותה. גודל זה הוא שקול ל-size של שורש העץ הזה. יש לנו גישה מיידית לשורש ולכן גם ל-size שלו, וכך נדע את מספר האיברים (במידה ואין שורש, כלומר העץ ריק, נחזיר 0).	$O(1)$ גישה לשדה בזמן קבוע.
permutation()	הפונקציה מחזירה עץ חדש עם אותם הערכים בפרמוטציה אקראית. תחילה, אנו ממירים את העץ לרשימה ב-$O(n)$, מערבבים את הרשימה באמצעות בחירת אינדקס רנדומלי והחלפת הערכים (Fisher–Yates). נתייחס לרשימה כממוינת כשלעצמה, כדי שנוכל ליצור עץ בזמן $O(n)$ באמצעות קריאה לפונקציה sortedArrayToAVL. נעדכן את המצביעים first, last ונחזיר את העץ החדש.	$O(n)$ המרת העץ לרשימה ויצירת עץ מרשימה ממוינת בזמן לינארי (באמור).
sort()	הפונקציה מחזירה עץ חדש ממוין לפי ערכי העץ הנוכחי. תחילה, אנו ממירים את העץ לרשימה ב-$O(n)$, ממיינים את הרשימה באמצעות mergeSort ב-$O(nlogn)$ ומוסיפים את הערכים לרשימה בסוף – n הכנסות שכל אחת ב-$O(logn)$.	$O(nlogn)$ בשל סיבוכיות mergeSort והכנסות הערכים לרשימה.
concat(lst)	משרשרת את הרשימה שמתקבלת כקלט לסוף הרשימה הנוכחית. אופן פעולת הפונקציה דומה מאוד לפונקציה Join שראינו בביתה. אנחנו משתמשים בצומת x שאנחנו יוצרים, כ-dummy שנועד לעזור לחבר בין שני הצמתים, ובסוף מוחקים אותו. נסמן ב-h1 את גובה העץ שמחזיק את self וב-h2 את גובה העץ שמחזיק את lst. אם h2=>h1: נעשה בדיוק מה שראינו בביתה: מהשורש של lst, נרד על הדופן השמאלי עד שנגיע לצומת הראשונה שגובהה הוא h1 או פחות (יכול להיות h1 או h1-1) נסמנה b. נגדיר את b להיות הילד השמאלי של x, ואת השורש של self להיות הילד הימני של x. כמו כן נגדיר את ההורה הקודם של b להיות כעת ההורה של x (אם b היה השורש אז אין הורה, נדאג לבדוק זאת). נעדכן את השדות של x משום שנצטרך אותם לתיקון העץ, וגם את שורש העץ אם השתנה (במקרה הזה, רק אם הם היו באותו גובה). אם h1>h2: נפעל באופן דומה- הפעם נתחיל מהשורש של self, ונרד על הדופן הימני עד שנגיע לצומת הראשון שגובהו h2 או פחות, נסמנו b. נגדיר את b להיות הבן השמאלי של x ואת שורש lst להיות בנו הימני של x וכן הלאה באותו אופן. לא משנה איזה מבין שני המקרים התקיים, לאחר מכן נבצע תיקון של העץ, ואז נמחק את x ממנו, וגם נעדכן את האיבר האחרון להיות האיבר האחרון של lst.	$O(logn)$ נטען אפילו ליותר טוב מכך: $O(\|h1-h2\|+1)$ ה"טיול" שנעשה על הדופן הימני או השמאלי של אחד העצים נעשה לאורך (כמעט) בדיוק הפרש הגבהים. כל חיבורי ותחזוקי השדות כמובן נעשים בזמן קבוע. קריאת ה-fixTree היא על x שנמצא בגובה שהוא גם כן בערך הפרש הגבהים, ולכן הטיפוס לשורש העץ הנוכחי יעבור על מספר צמתים כהפרש הגבהים. המחיקה של x: מחיקה לא תמיד חייבת לקרות בזמן לינארי לגובה הצומת הנמחק, בעיקר בגלל שעלולים ללכת לעוקב שלו. אך כעת אנו יודעים מי העוקב שלו. זהו צומת b (בשני המקרים, כל פעם זו b אחרת אך מיקומה דומה), שמחוברת ישירות אליו, והמחיקה תעשה משם ומעלה. כלומר לעולם לא נתרחק מהשורש למרחק שהוא יותר מהפרש הגבהים + 2, ואז גם התיקונים שיעשו שוב יעשו על מסלול שלא ארוך מכך.
search(val)	הפונקציה מחזירה את האינדקס הראשון ברשימה בו מופיע val, ואם הוא לא נמצא מחזירה -1. תחילה, אנו ממירים את העץ לרשימה ב-$O(n)$, ולאחר מכן מבצעים בה חיפוש על ידי קריאה ל-index.	$O(n)$ חיפוש ברשימה.
getRoot()	מחזירה את שורש העץ המייצג את הרשימה.	$O(1)$ שימוש בשדה קיים.

חלק תיאורטי

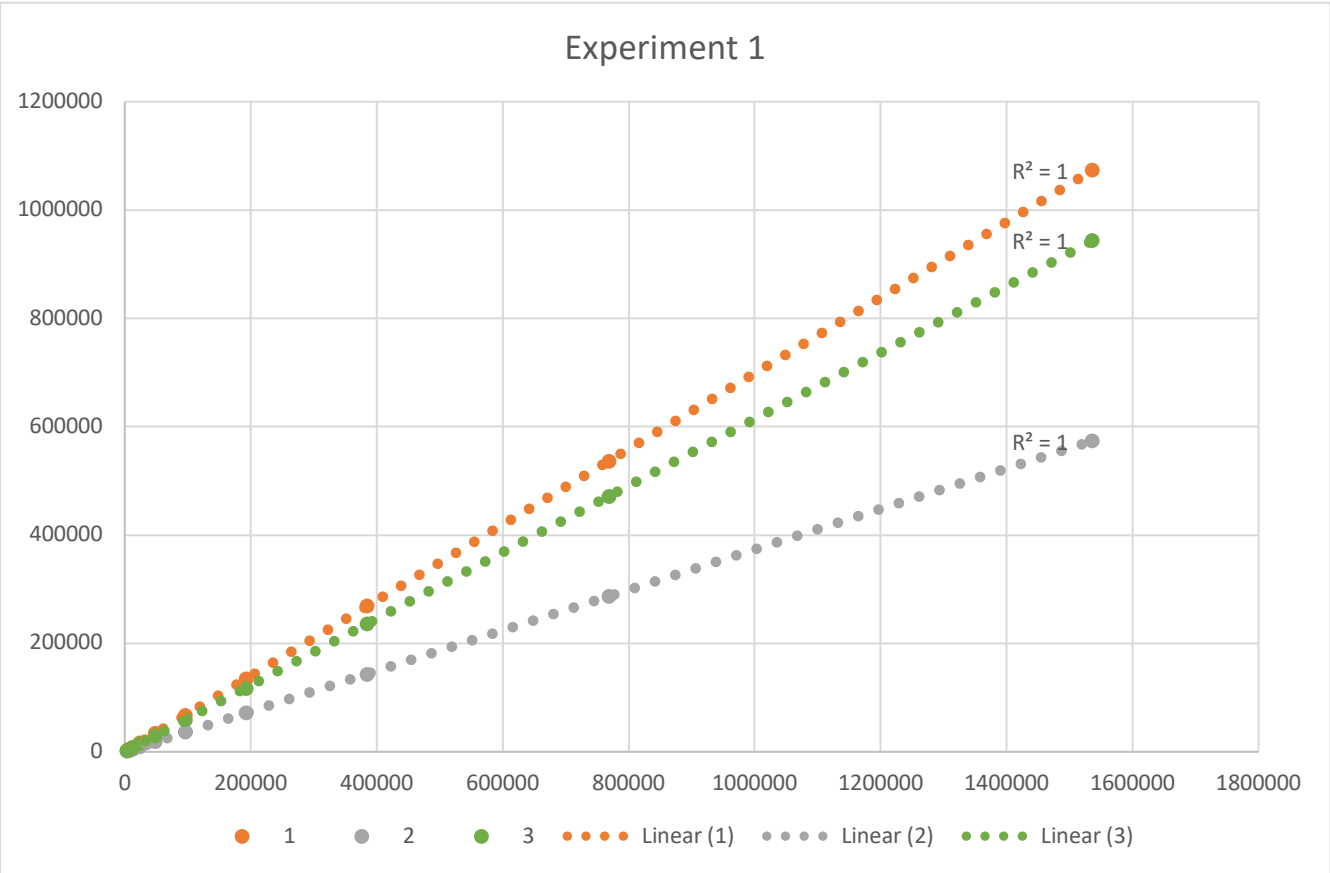
<u>שאלה 1:</u>

מספר פעולות האיזון שנדרשו כדי לתקן את העץ:

מספר סידורי i	ניסוי 1 – הכנסות	ניסוי 2 – מחיקות	ניסוי 3 – הכנסות ומחיקות לסירוגין	
			הכנסות:	הכנסות ומחיקות:
1	2071	1130	1097	785
2	4177	2262	2068	1553
3	8322	4531	4232	3198
4	16823	8814	8297	6307
5	33516	18077	16661	12472
6	67121	35885	33391	25412
7	134263	71628	66582	50651
8	268671	143074	134462	101442
9	535635	286395	268371	202704
10	1073270	574009	537124	405982

הביטויים האסימפטוטיים המתאימים הם (ניתן לראות גם בתרשים המצורף):

ניסוי 1 – הכנסות	ניסוי 2 – מחיקות	ניסוי 3 – הכנסות ומחיקות לסירוגין
$O(n)$	$O(n)$	$O(n)$



שאלה 2:

נציג את תוצאות ההכנסות במקרים השונים ואת מסקנותינו.

באופן כללי בעץ AVL, נצפה לקצב גדילה שתואם ל-$O(logn)$ בכל תרחיש של הכנסה, כיוון שבכל מקרה נקרא בסוף ל-fixTree שעוברת על כל צומת במסלול החל ממקום ההכנסה עד השורש.

הכנסה בהתחלה:

זמן ריצה בממוצע / מספר סידורי i	עץ AVL – הכנסות להתחלה e-06 sec	רשימה מקושרת – הכנסות להתחלה e-07 sec	מערך – הכנסות להתחלה e-06 sec
1	5.04	3.22	0.45
2	5.41	2.65	0.80
3	5.18	3.39	1.14
4	5.27	3.14	1.49
5	5.39	2.93	1.84
6	5.44	2.78	2.37
7	6.64	2.91	2.55
8	5.73	3.28	2.90
9	5.71	2.72	3.18
10	5.74	2.87	3.57

- עץ AVL – ישנה מגמת עליה קלה משציפינו.
- רשימה מקושרת – במימוש שבדקנו ישנו מצביע לאיבר הראשון ברשימה, ולכן ההכנסה מתבצעת ב-$O(1)$. התוצאות תואמות ציפייה זו.
- מערך – מדובר במקרה הגרוע, כי יש לבצע הזזה של האיברים ברשימה כדי להכניס איבר במקום הראשון. ציפינו כאן ל-$O(n)$ ואכן נראה שכך המצב.

הכנסה באקראי:

זמן ריצה בממוצע / מספר סידורי i	עץ AVL – הכנסות באקראי e-06 sec	רשימה מקושרת – הכנסות באקראי e-05 sec	מערך – הכנסות באקראי e-06 sec
1	8.30	0.62	0.55
2	6.69	1.10	0.63
3	6.31	1.69	0.81
4	6.38	1.39	0.98
5	6.74	2.97	1.16
6	7.72	3.57	1.34
7	7.47	4.41	1.57
8	6.90	4.94	1.67
9	8.07	5.49	1.86
10	7.28	6.19	2.01

- עץ AVL – הנתונים בהכנסה אקראית לא תואמים את הציפייה שלנו. לא נראית מגמת עליה. אולי עבור עצים גדולים משמעותית, נוכל לראות עליה.
- רשימה מקושרת – בתוחלת אנו מכניסים כל פעם באמצע הרשימה, ונצפה לקצב שתואם $O\left(\frac{n}{2}\right) = O(n)$. אכן הנתונים מראים קצב גדילה שכזה (פחות מהיר מאשר במקרה של הכנסה בסוף שהיא המקרה הגרוע ביותר).
- מערך – בתוחלת אנו מכניסים כל פעם באמצע הרשימה, ובאופן דומה לרשימה מקושרת נצפה לגדילה נמוכה יותר מהמקרה הגרוע ביותר שהוא הכנסה בהתחלה.

<u>הכנסה בסוף:</u>

זמן ריצה בממוצע / מספר סידורי i	עץ AVL – הכנסות לסוף e-06 sec	רשימה מקושרת – הכנסות לסוף e-05 sec	מערך – הכנסות לסוף e-08 sec
1	4.84	0.98	5.77
2	5.31	1.93	5.03
3	5.34	2.95	4.76
4	5.40	3.99	4.79
5	5.53	4.96	4.71
6	6.35	5.98	4.71
7	5.61	7.02	4.43
8	5.69	8.01	4.47
9	6.81	8.97	4.59
10	6.34	10.03	4.46

- עץ AVL – ישנה מגמת עליה התואמת את הציפיה שלנו.
- רשימה מקושרת – במימוש שבדקנו אין מצביע לאיבר האחרון ברשימה, ולכן הכנסה זו דורשת מעבר על כל הרשימה ב-$O(n)$. אכן ניתן לראות מגמת עליה לינארית התואמת את הציפייה הזו.
- מערך – הכנסה לסוף המערך מתבצעת ב-$O(1)$. נראה שיש מגמת ירידה קלה. השערה: תדירות הגדלת המערך (שדורשת העתקה של כולו למערך חדש) יורדת כאשר גודל המערך גדל.