Data Structures – Project #1 Oded Neeman – odedneeman – 207110479 Roy Mayan – roymayan – 206483554

חלק מעשי

המחלקה AVLNode

.AVLTreeList מייצגת צומת בעץ AVLNode <u>תיאור:</u>

<u>שדות:</u>

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

<u>מתודות:</u>

סיבוכיות	תיאור	מתודה
0(1)	בנאי המחלקה. יוצרת צומת חדש בעל הערך value. אם is_real הוא True,	init(value,is_real)
מתבצעות בדיקות והשמות בזמן קבוע.	הצומת שייווצר הוא צומת אמיתי (ולכן ייווצרו לו שני בנים וירטואליים).	
	אחרת, הצומת שייווצר הוא צומת וירטואלי.	
0(1)	מחזירה את הבן השמאלי של הצומת.	getLeft()
גישה לשדה בזמן קבוע.		
0(1)	מחזירה את הבן הימני של הצומת.	getRight()
גישה לשדה בזמן קבוע.		
0(1)	מחזירה את ההורה של הצומת.	getParent()
גישה לשדה בזמן קבוע.		
0(1)	מחזירה את הערך של הצומת.	getValue()
גישה לשדה בזמן קבוע.		
0(1)	מחזירה את הגובה של הצומת.	getHeight()
גישה לשדה בזמן קבוע.		
0(1)	מחזירה את הדרגה של הצומת.	getSize()
גישה לשדה בזמן קבוע.		
0(1)	מעדכנת את הבן השמאלי של הצומת להיות node.	setLeft(node)
השמה לשדה בזמן קבוע.		
0(1)	מעדכנת את הבן הימני של הצומת להיות node.	setRight(node)
השמה לשדה בזמן קבוע.		
0(1)	מעדכנת את ההורה של הצומת להיות node.	setParent(node)
השמה לשדה בזמן קבוע.		
0(1)	מעדכנת את הערך של הצומת להיות value.	setValue(value)
השמה לשדה בזמן קבוע.		
0(1)	מעדכנת את הגובה של הצומת להיות h.	setHeight(h)
השמה לשדה בזמן קבוע.		
0(1)	מעדכנת את הדרגה של הצומת להיות s.	setSize(s)
השמה לשדה בזמן קבוע.		

0(1)	מחזירה True אם הצומת אמיתי, False אחרת.	isRealNode()
גישה לשדה בזמן קבוע.		
0(1)	מחזירה את ה-Balance Factor של הצומת (הפרש הגבהים בין הבן	getBF()
גישה לגבהים של הבנים בזמן קבוע.	השמאלי של הצומת לבין הבן הימני שלו).	
0(1)	מעדכנת את הדרגה של הצומת לפי הדרגות של הבנים שלו. חישוב זה	updateSize()
גישה לדרגות של הבנים בזמן קבוע.	מבוסס על התכונה הלוקאלית: דרגה של צומת היא סכום הדרגות של בניו,	
	בתוספת 1.	
0(1)	מעדכנת את הגובה של הצומת לפי הגבהים של הבנים שלו. חישוב זה	updateHeight()
גישה לגבהים של הבנים בזמן קבוע.	מבוסס על התכונה הלוקאלית: גובה של צומת הוא המקסימלי מבין הגבהים	
	של בניו, בתוספת 1.	

<u>המחלקה AVLTreeList</u>

.AVLNode מייצגת מימוש של ADT רשימה באמצעות עץ AVL. כל איבר ברשימה הזו מיוצג על ידי AVLNode <u>תיאור:</u> AVLTreeList מייצגת מימוש של ADT ש<u>דות:</u>

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

<u>מתודות:</u>

סיבוכיות	תיאור		
O(logn)	מחזירה את הצומת עם rank של k בתת העץ שהשורש שלו הוא x. מימוש זהה לTree	selectRec(x, k)	
כמו שראינו עבור Tree-Select, לינארי	Select שראינו בהרצאה. מחזירה איבר מסוג AVLNode.		
בגובה העץ שבמקרה שלנו תמיד	תנאי קדם: העץ ששורשו x מכיל לפחות k איברים.		
.O(logn)			
O(logn)	מחזירה את האיבר העוקב של x ברשימה (מחזירה אותו כטיפוס AVLNode).	getSuccessor(x)	
לכל היותר נטפס את מלוא גובה העץ עד	אם ל-x יש ילד ימני, אז הולבים אליו ואז שמאלה עד שתת העץ נגמר. אם אין לו ילד ימני,		
השורש. בכל צומת העבודה היא קבועה.	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)	מוצאת את האיבר הקודם ל-x ברשימה (מחזירה אותו כטיפוס AVLNode).	getPredecessor(x)	
לכל היותר נטפס את מלוא גובה העץ עד	מימוש דומה לפונקציה הקודמת: אם יש ילד שמאלי, אז ללכת אליו ואז ימינה עד הסוף. אם		
השורש. בכל צומת העבודה היא קבועה.	.אין- לעלות עד שפונים שמאלה		
O(n)	פונקציה רקורסיבית שמוסיפה לרשימה lst את האיברים שבתת העץ של הצומת node,	inOrderRec(
עוברים על כל איבר בתת העץ בדיוק	לפי סדר האינדקסים שלהם. אנו מוסיפים את ה-value של האיברים ולא את הצמתים	node, lst)	
פעם אחת. לכן לינארי בכמות האיברים	עצמם כמובן.		
בתת העץ.	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)	חדש שמכיל את איברי AVL פונקציה רקורסיבית המקבלת מערך ממוין, ויוצרת עץ	sortedArrayToAVL(
כפי שניתחנו בתרגול.	מרך. הפונקציה מבצעת זאת בהתאם למימוש שראינו בתרגול 5, בעזרת לקיחת arı		
	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+m)	מיזוג שתי רשימות ממוינות לרשימה אחת (ממוינת), תוך מעבר על שתי הרשימות	merge(lst1, lst2)	
הלולאה רצה כסכום אורכי הרשימות.	במקביל והכנסת האיבר הקטן יותר מבין שתי הרשימות.		

O(nlogn)	פונקציה רקורסיבית הממיינת רשימה באמצעות חלוקה הרשימה לשתיים (בחציון), מיון	mergeSort(Ist)
פרתאם לניתוח שראינו במבוא מורחב.	פונקבידו קוו טיבית דוממיינות שימוד באמצעות דולקודדוו שימוד לשוניים (בדוביון), מיון רקורסיבי של כל תת-רשימה ומיזוגן.	mergesort(ist)
0(1)	דקוו סיבי <i>סל בל ומכרו סיבור ובריו</i> גן. בנאי המחלקה. יוצרת עץ AVL ריק.	init()
רד) ט השמות בזמן קבוע.	בנאי וומוזקוו. יוצדת עץ TAVL יק. מתבצעת השמה של ערכי ברירת-מחדל (None) לשדות המחלקה.	()
0(1)	מועבצטור ושכנוו של ערבי ברידו ל (מאוטאו) לשרוות ותמוזלקות. מחזירה True אם העץ ריק ו-False אם העץ לא ריק.	empty()
יש מצביע לשורש לכן הבדיקה נעשית	נוחוידה דום אם העץ דיק ה-raise אם העץ לא דיק. עץ רק אמ"מ השורש שלו הוא None ולכן זו הבדיקה שנעשית.	empty()
	עץ דק אנד מדושוו ש שלדוווא שווטאו ולבן ודרובו יקור שנעשיונ.	
בזמן קבוע. $O(logn)$	מחזירה את ערך האיבר במקום ה-i ברשימה.	retrieve(i)
קריאה בודדת ל-selectRec, לכן אותה	מוחידו האת עוך האיבו במקום היד בו שימות. מבוקשת $i+1$. בעץ, אנו "סופרים" selectRec- הפונקציה קוראת ל	retrieve(i)
קו יאוז בורדונ ל-selectivec, לבן אוונוד סיבוביות שלה.	וופונקציה קוו את selectikec-7 על שוו ש העץ, ודרגה מבתקשת ד + 1. בעץ, אנד סופרים את האיברים החל מ-1 (בך הוגדר rank של צומת) וברשימה החל מ-0. לבן אינדקס של	
טיבוניווו פיווו.	אונדהאיבו ים הודל נו-1 (בן הוגדו rank של צומונ) ובו שימה הודל נו-0. לבן אינדקט של אונדהאיבו $i+1$ עם $i+1$.	
	איבו בו שינוד ידידו ביוות מודיאווגד שלדבעץ ולכן קוו אים ל-selectRec עם ד + ז. בסוף אנו מחזירים את ה-value של הצומת שקיבלנו מ-selectRec.	
0(1)	בסוף אנו נוזוזיו ים אול זו-value של הצומות שקיבלנו מ-selectivec. הפונקציה כוללת את פעולות החלפת המצביעים המשותפות לסיבוב ימינה ולסיבוב	rotato/A D)
(ב) ט בודקים ומשנים שדות ומצביעים. כמות	הפונקציה בוללוג את פעולות החלפת המצביעים המשותפות לסיבוב ימינה ולסיבוב שמאלה על מנת למנוע שכפול קוד.	rotate(A, B)
	·	
הבדיקות והשינויים חסומה.	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	
0(1)	A ושל B. הם היחידים שמשתנים לאחר שמבצעים סיבוב.	
0(1)	מבצעת סיבוב ימינה.	rightRotation(
בודקים ומשנים שדות ומצביעים. כמות	B מסומן כצומת הפושע ו-A כבן השמאלי שלו (שבסוף הסיבוב יהיה במקום של B), AR	criminal)
הבדיקות והשינויים חסומה.	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	
0(1)	לפונקציית העזר rotate.	I-faD-t-ti/
0(1)	מצבעת סיבוב שמאלה.	leftRotation(
בודקים ומשנים שדות ומצביעים. כמות	באותו אופן כמו סיבוב ימינה רק שכעת A הוא הבן הימני של B וניגשים לבן השמאלי של	criminal)
הבדיקות והשינויים חסומה.	A, A, במקום לבן הימני.	f
0(1)	פונקציה זו מקבלת צומת שידוע שהוא פושע (התגלה ב-fixTree שה 2).	performRotations(
כל שנעשה הוא בדיקת bf של הפושע	היא בודקת איזה סיבוב/ים נדרש לבצע ומבצעת אותו/אותם ע"י קריאה לפונקציות עזר.	criminal)
ושל אחד הבנים, ואז פעולת סיבוב אחת	הבדיקה נעשית כמו שלמדנו: בודקים שני מקרים ($f=2$ או $f=-2$), ובכל מקרה	
או שתיים.	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.	<u> </u>
O(logn)	הפונקציה מבצעת את כל התיקונים הדרושים על העץ כך שלאחר הפעולה שנעשתה העץ	fixTree(start_node,
עוברים על צמתים כגובה העץ לכל	יהיה עץ AVL תקין וכל השדות יהיו מעודכנים. 	is_delete=False)
היותר, כי מתחילים מצומת כלשהו בעץ	אנחנו מתחילים מההורה של start_node שמיושם ל-y, ומשם מטפסים כל פעם להורה 	
ומטפסים עד השורש. בכל צומת	של y עד שמגיעים לשורש. בכל צומת y, מעדכנים את הגובה ואת ה-size שלו. אם ה-	
שעוברים עליו נעשית עבודה קבועה-	balance factor לא חוקי (הפך ל-2 בערך מוחלט- הוא לא יכול להיות יותר מ-2), אז	
בדיקת שדות ו-bf, עדכון שדות, ולפעמים	מבצעים את הסיבובים הדורשים ומוסיפים אותם לספירת הסיבובים שנעשו. 	
פונקציית performRotations	לפונקציה ארגומנט אופציונלי של is_delete:	
שמתבצעת בזמן קבוע.	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.	

	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		
O(logn)	מכניסה איבר עם הערך val למקום ה-i ברשימה.	insert(i, val)	
במקרה הגרוע, ישנה קריאה אחת ל- 			
selectRec, קריאה אחת ל-			
-, וקריאה אחת ל getPredecessor	·		
fixTree. כל הפונקציות האלה רצות בזמן			
לוגריתמי. חוץ מפונקציות אלה, ישנן	שדה האיבר הראשון להיות z. אם ()i=self.length, כלומר z מוכנס לסוף הרשימה, אזי		
פעולות של שינוי מצביעים בכמות	מבצעים פעולות דומות, רק הפעם z יהיה בן ימני של האיבר האחרון.		
חסומה ולכן ללא השפעה אסימפטוטית.	שאר המקרים: אנו מגיעים לאיבר במקום ה-i ברשימה (בעץ הוא ב-rank ה-1+1), בעזרת		
	הפונקציה Z .selectRec 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.		
	עליה fixTree בסוף פעולת ההכנסה, לא משנה באיזה תרחיש, נקראת הפונקציה		
	הרחבנו, ומוחזר ערך ההחזרה שלה שהוא מספר פעולות התיקון שנעשו.		
0(1)	 הפונקציה מקבלת צומת שיש לו לכל היותר ילד אחד ומוחקת אותו ע"י bypass כפי	deleteLeafOrSingle(
אנו מקבלים מצביע על הצומת, לכן כל	י - י - י י י י י י י י י י י י י י י י	node, child)	
י הפעולות הן בדיקות של שדות ושינוי	י חיבים איניים). בודקים אם node היה הבן הימני או השמאלי של ההורה שלו ואז	,	
י י מצביעים בכמות חסומה.	לפי זה מתקנים שכעת הילד בצד הזה יהיה child (אם node הוא עלה אז child הוא צומת		
	יהיה ההורה של child ווירטואלי, לכן הפונקציה מתאימה לשני המקרים), ושההורה של		
	node. *הפונקציה לא תעבוד אם node הוא שורש העץ, אך בקוד שלנו לא תערה		
	סיטואציה כזו. סיטואציה כזו.		
O(logn)	ס שואב זו בוו: הפונקציה מוחקת מהרשימה את האיבר במקום ה-i.	delete(i)	
כונטקור) במקרה הגרוע אנחנו מבצעים קריאות ל-	וופונקבידו מווזקות מווו שימוד את דואיבו במקום דויו. אם i גדול/שווה מאורך הרשימה, אזי אין איבר באינדקס הזה ונחזיר 1- (תקף גם לכל קלט	delete(i)	
	אם דגוול לשווד מאודן דוו שימוז, אוי אין איבר באינדולוס ווודדונדוויד ב- (ומוף גם לכל ולכ		
IN) gotSuccessor selectRec	בעוער ברועומב בובב/		
או) getSuccessor ,selectRec	באשר הרשימה ריקה). משני מעבוע על בעומת במבות במתועם בעזבת בבועת coloctRos לעונדבת 1 . i		
אם z היה האיבר getPredecessor	i+1 לאינדקס selectRec נשיג מצביע על הצומת במקום המתאים בעזרת קריאת		
אם z היה האיבר getPredecessor האחרון), ולבסוף קריאה ל-fixTree,	i+1 לאינדקס selectRec נשיג מצביע על הצומת במקום המתאים בעזרת קריאת z . (כאמור זה נובע ממהבדל בין מקום ברשימה ל-rank בעץ)		
getPredecessor אם z היה האיבר האחרון), ולבסוף קריאה ל-fixTree, פונקציות שכולן רצות בזמן לוגריתמי.	i+1 לאינדקס selectRec נשיג מצביע על הצומת במקום המתאים בעזרת קריאת z - נשיג מצביע על הצומת ב-z. (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		
getPredecessor אם z היה האיבר האחרון), ולבסוף קריאה ל-fixTree, פונקציות שכולן רצות בזמן לוגריתמי. לכל פונקציה מביניהן נקרא לכל היותר	i+1 לאינדקס selectRec נשיג מצביע על הצומת במקום המתאים בעזרת קריאת אינדקס selectRec לאינדקס z -2. (כאמור זה נובע ממהבדל בין מקום ברשימה ל-rank בעץ) ונסמן את הצומת ב- z -2. 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		
אם z היה האיבר 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	 i + 1 לאינדקס selectRec לאינדקס 1 לאינדקס selectRec לאינדקס 1 בעזרת קריאת אינדקס 1 בעץ 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). 		
אם z היה האיבר מgetPredecessor 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	 i + 1 לאינדקס selectRec לאינדקס בעזרת קריאת אינדקס 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 		
אם z היה האיבר האחרון), ולבסוף קריאה ל-fixTree, 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	 i + 1 לאינדקס selectRec לאינדקס בעזרת קריאת selectRec לאינדקס i + 1 (באמור 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). במו 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 		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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		
אם z היה האיבר האחרון), ולבסוף קריאה ל-fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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 getSuccessor		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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): (בעזרת sanget (בעזרת sanget) צו במצא את העוקב של c (בעזרת sanget) (פנמצא את העוקב של c (בעזרת sanget) (בעזרת sanget).		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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 z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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): (בעזרת sanget (בעזרת sanget) צו במצא את העוקב של c (בעזרת sanget) (פנמצא את העוקב של c (בעזרת sanget) (בעזרת sanget).		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	 נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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). במו 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 		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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 size size.		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 - 2. (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 במו קודם. אם ל-z יש שני ילדים (אמיתיים - לא ווירטואליים): נמצא את העוקב של z (בעזרת getSuccessor), לו בהגדרה יהיה בן אחד. נמחק אותו ושנמחק את הצומת מהמקום הקודם שלו, ובסוף גם נעדכן את שדות הגובה וה-size של size 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		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1. ב-2. באמור זה נובע ממהבדל בין מקום ברשימה ל-rank בעץ) ונסמן את הצומת ב-2. 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 2 הוא שורש העץ, אזי נמחק אותו על ידי שינוי של מעט המצביעים הדרושים ונתחזק בהתאם את המצביע על השורש. אם 2 הוא האיבר הראשון או האחרון, גם נתחזק את המצביע המתאים בעזרת getSuccessor או getPredecessor בהתאמה, ואז נבצע מחיקה כמו קודם. אם ל-2 יש שני ילדים (אמיתיים - לא ווירטואליים): (פעזרת Ryana) (לו בהגדרה יהיה בן אחד. נמחק אותו נמצא את העוקב של 2 (בעזרת getSuccessor), לו בהגדרה יהיה בן אחד. נמחק אותו ונשים אותו במקום של 2 נתקן את כל המצביעים והשדות כך שהעוקב יהיה במקום של צ size 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		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec (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 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 (בעזרת לתקן 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.		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec לאינדקס 1 + 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). 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 getSuccessor 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 (בעזרת דפל 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. hand מההורה שלו, עלו עדיין יש לנו מצביע מ-z אם לאחר פעולת המחיקה נקרא להתחיל מההורה שלו, אליו עדיין יש לנו מצביע מ-z אם לא היו לו שני ילדים (אנחנו רוצים להתחיל מההורה שלו, אליו עדיין יש לנו מצביע מ-z		
אם z היה האיבר האחרון), ולבסוף קריאה ל-rixTree, fixTree, 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	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec (כאמור זה נובע ממהבדל בין מקום ברשימה ל-rank) 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 מפעילים את אמיתי, אז לא משנה איזה מהילדים ניתן בקלט). במו 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 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 (בעזרת המקום שלו, ובסוף גם נעדכן את שדות הגובה וה-size של ושנמחק את הצומת מהמקום הקודם שלו, ובסוף גם נעדכן את שדות הגובה וה-size של אותון ונעשה זאת במקרה הצורך. אותו ונעשה זאת במקרה הצורך. לאחר פעולת המחיקה נקרא לfixTree בדי לתקן את העץ. נתחיל את התיקונים מ-z אם לאחר פעולת המחיקה נקרא להתחיל מההורה שלו, אליו עדיין יש לנו מצביע מ-z למרות שהוא לא בעץ), ומהבן של ה-successor במקרה של-z היו שני ילדים (אנחנו רוצים להתחיל מההורה שלו, אליו עדיין יש לנו מצביע מ-z שדאגנו לשמור מבעוד מועד).	first()	
אם z היה האיבר האחרון), ולבסוף קריאה ל-fixTree, 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.	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec (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 הילדים של העץ לא אמיתי. אם זה המקרה: בודקים מי האמיתי (ואם אף אחד מפעילים את deleteLeafOrSingle עם קלט של הצומת והילד האמיתי (ואם אף אחד מהילדים לא אמיתי, אז לא משנה איזה מהילדים ניתן כקלט). במו 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 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 (בעזרת 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. אותו ונעשה זאת במקרה הצורך. לא היו לו שני ילדים (אנחנו רוצים להתחיל מההורה שלו, אליו עדיין יש לנו מצביע מ-z 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).	first()	
מיה האיבר האחרון), ולבסוף קריאה ל-fixTree, 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.	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת נסמן את הצומת ב-z. (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 מפעילים את מאיתי, 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 petPredecessor 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 (בעזרת להמצביעים והשדות כך שהעוקב יהיה במקום של z ונשנמחק את הצומת מהמקום הקודם שלו, ובסוף גם נעדכן את שדות הגובה וה-size של ושנמחק את הצומת מהמקום הקודם שלו, ובסוף גם נעדכן את שדות הגובה וה-size של אותו ונעשה זאת במקרה הצורך. אותו ונעשה זאת במקרה הצורך. לאחר פעולת המחיקה נקרא fixTree להתחיל מההורה שלו, אליו עדיין יש לנו מצביע מ-z 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). אנחנו מתחזקים מצביע לצומת שמייצג את האיבר הראשון ברשימה תמיד, ולכן קריאה אנחנו מתחזיר את ה-valu של הצומת שהמצביע מצביע אליו (או None).		
אם z היה האיבר האחרון), ולבסוף קריאה ל-fixTree, 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.	נשיג מצביע על הצומת במקום המתאים בעזרת קריאת selectRec (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 הילדים של העץ לא אמיתי. אם זה המקרה: בודקים מי האמיתי (ואם אף אחד מפעילים את deleteLeafOrSingle עם קלט של הצומת והילד האמיתי (ואם אף אחד מהילדים לא אמיתי, אז לא משנה איזה מהילדים ניתן כקלט). במו 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 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 (בעזרת 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. אותו ונעשה זאת במקרה הצורך. לא היו לו שני ילדים (אנחנו רוצים להתחיל מההורה שלו, אליו עדיין יש לנו מצביע מ-z 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).	first()	

O(n)	מחזירה מערך שבו איברי הרשימה לפי סדר האינדקסים שלהם. המערך ריק אם הרשימה	listToArray()
O(n)מה שיוצר את הסיבוכיות היא הקריאה		list to Array()
מוד שיוצד אול ווסיבוניוול וויא דוקו יאוד לפונקציה inOrderRec אותה ניתחנו.	ריקה. הפונקציה יוצרת מערך ריק. אם הרשימה ריקה היא מחזירה אותו ואם לא, היא קוראת	
לפונקביוז שאוישטוסווו אווטו ניוטונו.	דופונקצידו יוצדות נעודן דיק. אם דוו שימדדו יקודדויא מחזידדו אותרואם לא, דויא קוו את לפונקציה inOrderRec על שורש הרשימה והמערך הריק שיצרנו.	
0(1)	לפונקציה der kec לפונקציה לפונקציה מספר האיברים ברשימה הוא מספר הצמתים (הלא מחזירה את מספר האיברים ברשימה הוא מספר הצמתים (הלא	length()
(ב) ט גישה לשדה בזמן קבוע.	נווזיו זו אול מספר דואיבו ים בו שינווז. מספר דואיבו ים בו שינווד ווא מספר דוצמונים (וולא ווירטואליים) בעץ שמייצג אותה. גודל זה הוא שקול ל-size של שורש העץ הזה. יש לנו	iengui()
גישוו לשרוז בונון לןבוע.	וויו טואליים) בעץ שנוייצג אוונוז. גוול זוד וווא שקול ל-size של שוו ש דועץ דוווז. יש לנו גישה מיידית לשורש ולכן גם ל-size שלו, וכך נדע את מספר האיברים (במידה ואין שורש,	
	גישוז מייז זו לשוו ש זלבן גם ל-size שלו, ובן נדע אול מטפר דואיבו ים (במידוד ואין שוו ש, בלומר העץ ריק, נחזיר 0).	
0(n)	ביזנור דועד דיקן, נדודיד טן. הפונקציה מחזירה עץ חדש עם אותם הערכים בפרמוטציה אקראית.	permutation()
המרת העץ לרשימה ויצירת עץ מרשימה	יופונקביזי מוואידי על יוו ט עם אוונם וזעו כים בפו מוסביזי און איני. $O(n)$, מערבבים את הרשימה באמצעות בחירת	permutation()
ממוינת בזמן לינארי (כאמור).	ינורילוז, אנו ממידים אולדועץ לו סיפור ביילה) ס, מעו בבים אולדוו סיפור באמצעות בודידת אינדקס רנדומלי והחלפת הערכים (Fisher–Yates). נתייחס לרשימה כממוינת	
נונוו נולבונון ל באו "לבאנוו").	אינוקט דנו נמילי ווווילפול דועו בים (מילונים לינות היידים לינות במבווינות בשלעצמה, בדי שנוכל ליצור עץ בזמן $O(n)$ באמצעות קריאה לפונקציה	
	ביסלעבנווז, בוי סנוכל קיבור עץ בזמן (זו) ס באמצפות קויאחר לבומקבירו sortedArrayToAVL. נעדכן את המצביעים first, last ונחזיר את העץ החדש.	
O(nlogn)	יסט זיסט זיסט נפו בן אול דונצב ע בי זיטו , זיטו דיטו זיטו ויסט זיטו זיטו פונקציה מחזירה עץ חדש ממוין לפי ערכי העץ הנוכחי. תחילה, אנו ממירים את העץ	sort()
והכנסות mergeSort בשל סיבוביות	הבומן ביז מווא דו על דוו ט ממוין לפי עו כי זועל דומיבוזי דווא דו על דוו מארים אול דועל $O(nlogn)$, ממיינים את הרשימה באמצעות $O(nlogn)$ ומוסיפים	3011()
בשל 6 בוב ות mergeson ווובנסות הערכים לרשימה.	יו שינויו או אוויים או אוויים או אוויים או הינים או אוויים אווי	
O(logn)	אונדושו בים לו <i>סימור</i> בסוף יי דרובנסוול <i>סבל אוורב (toght) ס.</i> משרשרת את הרשימה שמתקבלת בקלט לסוף הרשימה הנובחית.	concat(lst)
נטען אפילו ליותר טוב מכך:	נוסו סדול אול דוו סינודו סנוולןבלול בקלט לטוף דוו סינודו דונובוזיית. אופן פעולת הפונקציה דומה מאוד לפונקציה Join שראינו בכיתה. אנחנו משתמשים	concactisty
O(h1 - h2 + 1)	אופן פעולת חפונקבית דומות מאוד לפונקבית חוסנ סו אינו בכיזמה. אנחנו מסותמסים בצומת x שאנחנו יוצרים, כ-dummy שנועד לעזור לחבר בין שני הצמתים, ובסוף מוחקים	
ה"טיול" שנעשה על הדופן הימני או	בצונות ג שאנדונו יוצו ינו, ב-dummy שנועד לעווד לחבר בין שני דוצנותים, ובסוף נוודוקים אותו.	
השמאלי של אחד העצים נעשה לאורך	אוויני. נסמן ב-h1 את גובה העץ שמחזיק את self וב-h2 את גובה את העץ שמחזיק את lst.	
(כמעט) בדיוק הפרש הגבהים. כל חיבורי	נטמן ביבוז אול גוברו זועץ שמו זויקן אול וופצ וביבוז אול גוברו אול זועץ שמו זויקן אול זכו. אם h2=>h1:	
ותחזוקי השדות כמובן נעשים בזמן קבוע.	אם דורכ–זוו. נעשה בדיוק מה שראינו בכיתה: מהשורש של lst, נרד על הדופן השמאלי עד שנגיע	
היא על x אנמצא fixTree היא על	נעשוד בריוק מודשו אינו בכיומה. מודשורש של זמ, נדד על דודופן וושמאלי עד שנגיע לצומת הראשונה שגובהה הוא h1 או פחות (יכול להיות h1 או h1-1) נסמנה b. נגדיר את	
לן אולדו פורארודוא <i>פרא סבמב</i> א בגובה שהוא גם כן בערך הפרש הגבהים,	לצומול דוו אשונדו שגוברוד דווא בדוו ליבול לדויות בדו או ביבדון נטמנדו ט. נגריד אות b להיות הילד השמאלי של x, ואת השורש של self להיות הילד הימני של x. כמו כן נגדיר	
ולבן הטיפוס לשורש העץ הנוכחי יעבור	ט לוויות וויכוני של 2, אות דושוו ש של seii עם בנויים של b בנו בן בגויי b את ההורה הקודם של b להיות בעת ההורה של x (אם b היה השורש אז אין הורה, נדאג	
על מספר צמתים כהפרש הגבהים.	אול דודוו דד דוקודם של טלדדיות בעול דודוו של א לאם טדידי דוטוו ש או אין דוו דון, נו אג לבדוק זאת). נעדכן את השדות של x משום שנצטרך אותם לתיקון העץ, וגם את שורש	
לי מסבו במולם בוזבו סיווגבוזים: המחיקה של x: מחיקה לא תמיד חייבת	לבו זק זאות). נערכן אות דופרות של א נופום שנצטרן אותם לתיקון וזעץ, וגם את שוו ש העץ אם השתנה (במקרה הזה, רק אם הם היו באותו גובה).	
לקרות בזמן לינארי לגובה הצומת	וועץ אם ווסוגנוו (בניקורו ווווי, דיק אם ווס וויו באווט גובוו). אם h1>h2:	
הנמחק, בעיקר בגלל שעלולים ללכת	אם בחיבות. נפעל באופן דומה- הפעם נתחיל מהשורש של self, ונרד על הדופן הימני עד שנגיע	
לעוקב שלו. אך כעת אנו יודעים מי	נבשל באובן רונוור רובעם נולדר מחופה של האל, ונדיר את b להיות הבן השמאלי של x ואת לצומת הראשון שגובהו h2 או פחות, נסמנו b. נגדיר את	
ישוקב שלו. זהו צומת b (בשני המקרים,	לבונות או אסון שגובווו בוז או פרוות נסמנו טי בגויד אורט לדאות דובן דושמאלי של א אות שורש lst להיות בנו הימני של x וכן הלאה באותו אופן.	
כל פעם זו b אחרת אך מיקומה דומה),	פוו פוזנו ליו וולבנו זו נוני פל אובן ולאוור באוונן אובן. לא משנה איזה מבין שני המקרים התקיים, לאחר מכן נבצע תיקון של העץ, ואז נמחק את	
שמחוברת ישירות אליו, והמחיקה תעשה	א מסנו אוו נעדכן את האיבר האחרון להיות האיבר האחרון של lst. x	
משם ומעלה. כלומר לעולם לא נתרחק	א ממבות ואם בעו בן אורוא בו וואווו ון לו ווירוא בו יואווו ון לו אורוא בו	
מהשורש למרחק שהוא יותר מהפרש		
הגבהים + 2, ואז גם התיקונים שיעשו		
שוב יעשו על מסלול שלא ארוך מכך.		
, ,		
O(n)	ואם הוא לא נמצא val הפונקציה מחזירה את האינדקס הראשון ברשימה בו מופיע	search(val)
חיפוש ברשימה.	י מחזירה -1 . תחילה, אנו ממירים את העץ לרשימה ב- $O(n)$, ולאחר מכן מבצעים בה	. ,
	חיפוש על ידי קריאה ל-index.	
0(1)	מחזירה את שורש העץ המייצג את הרשימה.	getRoot()
שימוש בשדה קיים.	·	

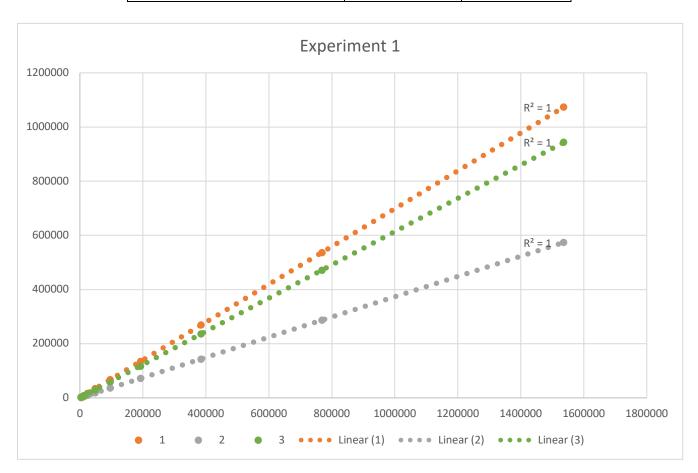
חלק תיאורטי

שאלה 1: מספר פעולות האיזון שנדרשו כדי לתקן את העץ:

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

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

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



:2 שאלה

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:

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

- עץ AVL − ישנה מגמת עליה קלה משציפינו. •
- רשימה מקושרת במימוש שבדקנו ישנו מצביע לאיבר הראשון ברשימה, ולכן ההכנסה מתבצעת ב-0(1). התוצאות תואמות ציפייה זו.
- מערך מדובר במקרה הגרוע, כי יש לבצע הזזה של האיברים ברשימה כדי להכניס איבר במקום הראשון. ציפינו כאן ל- $\mathcal{O}(n)$ ואכן נראה שכך המצב.

הכנסה באקראי:

רשימה מקושרת –	– AVL עץ	זמן ריצה בממוצע
הכנסות באקראי	הכנסות באקראי	
e-05 sec	e-06 sec	מספר סידורי i
0.62	8.30	1
1.10	6.69	2
1.69	6.31	3
1.39	6.38	4
2.97	6.74	5
3.57	7.72	6
4.41	7.47	7
4.94	6.90	8
5.49	8.07	9
6.19	7.28	10
	е-05 sec 0.62 1.10 1.69 1.39 2.97 3.57 4.41 4.94 5.49	е-05 secе-06 sec0.628.301.106.691.696.311.396.382.976.743.577.724.417.474.946.905.498.07

- 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>

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

- 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-0(1). נראה שיש מגמת ירידה קלה. השערה: תדירות הגדלת המערך (שדורשת העתקה של בולו למערך חדש) יורדת באשר גודל המערך גדל.