מבני נתונים – תרגיל בית מעשי מספר 1

פרטי המגישות:

מגישה 1:

שם משתמש: koslowsky

(בעברית - איילה קוסלובסקי)

שם המגישה: איילה קוסלובסקי

תעודת זהות: 207618323

מגישה 2:

שם משתמש: ronishamai

(בעברית – רוני שמאי)

שם המגישה: רוני שמאי

תעודת זהות: 3190936909

תיעוד חיצוני ופירוט סיבוכיות זמן-ריצה של הקוד:

המנשק IAVLNode:

המתודה	הסבר
int getKey()	מחזירה את השדה key, את המפתח, של הצומת (-1 עבור צומת ווירטואלי)
String getValue()	מחזירה את השדה info של הצומת (null עבור צומת ווירטואלי)
void setLeft(IAVLNode node)	מעדכנת את המצביע לילד השמאלי של הצומת, ל IAVLNode שמקבלת
IAVLNode getLeft()	מחזירה את השדה left של הצומת, כלומר מחזירה את המצביע ל IAVLNode שהינו הבן השמאלי של הצומת; אם לא קיים, מחזירה null
void setRight(IAVLNode node)	מעדכנת את המצביע לילד הימני של הצומת, ל IAVLNode שמקבלת
IAVLNode getRight()	מחזירה את השדה right של הצומת, כלומר מחזירה את המצביע ל IAVLNode שהינו הבן הימני של הצומת; אם לא קיים, מחזירה null
void setParent(IAVLNode node)	מעדכנת את המצביע להורה של הצומת, ל IAVLNode שמקבלת
IAVLNode getParent()	מחזירה את השדה parent של הצומת, כלומר מחזירה את המצביע ל IAVLNode שהינו ההורה של הצומת; אם לא קיים, מחזירה null
boolean isRealNode()	מחזירה true אם הצומת הוא לא צומת ווירטואלי, ו false אחרת
void setHeight(int height)	מעדכנת את הגובה של הצומת
int getHeight()	מחזירה את הגובה של הצומת (-1 עבור צומת ווירטואלי)
int getSize()	מחזירה את ה"גודל" של הצומת – סכום הצמתים בתתי העצים שלו +1 עבור עצמו
void setSize(int newSize)	מקבלת מספר, ומעדכנת אותו להיות ה"גודל" החדש של הצומת (שאמור להיות סכום הצמתים בתתי העצים שלו +1 עבור עצמו)

בקוד שלנו, ישנן שתי מחלקות אשר מממשות את המנשק **IAVLNode**:

1) **AVLNode** המייצגת צמתים שאינם ווירטואליים
2) **VirtualNode** המייצגת צמתים ווירטואליים בלבד (מחלקה סטטית, final)

זאת כאשר המחלקה **AVLTree** משתמשת באובייקטים מהמחלקות לעיל.

המחלקה AVLNode:

- השדות במחלקה AVLNode:

String info
int key
IAVLNode left
IAVLNode right
IAVLNode parent
int rank
int size
boolean realNode

- בנאים של המחלקה:
  - בנאי המקבל שני ערכים (int key, String info) – מעדכן את המפתח של הצומת ואת שדה ה info שלו להיות הערכים המתקבלים בבנאי. את שאר השדות נעדכן באופן הבא: הבן השמאלי וכן הבן הימני של הצומת יהיו צומת ווירטואלי, ההורה יהיה null, הדרגה 0, הגודל 1 ו realNode יעודכן להיות True.
  - בנאי המקבל ערך אחד בלבד (int key) – וקורא לבנאי מעלה, עם ערך info שהינו null.

- פירוט המתודות במחלקה AVLNode (<u>מימוש מתודות המנשק IAVLNode</u>):

המתודה	הסבר	סיבוכיות זמן ריצה
int getKey()	מחזירה את השדה key, את המפתח, של הצומת	סיבוכיות זמן קבועה – O(1), מחזירים מצביע
String getValue()	מחזירה את השדה info של הצומת	
void setLeft(IAVLNode node)	מעדכנת את המצביע לילד השמאלי של הצומת, ל IAVLNode שמקבלת	
IAVLNode getLeft()	מחזירה את השדה left של הצומת, כלומר מחזירה את המצביע ל IAVLNode שהינו הבן השמאלי של הצומת; אם לא קיים, מחזירה null	
void setRight(IAVLNode node)	מעדכנת את המצביע לילד הימני של הצומת, ל IAVLNode שמקבלת	
IAVLNode getRight()	מחזירה את השדה right של הצומת, כלומר מחזירה את המצביע ל IAVLNode שהינו הבן הימני של הצומת; אם לא קיים, מחזירה null	
void setParent(IAVLNode node)	מעדכנת את המצביע להורה של הצומת, ל IAVLNode שמקבלת	
IAVLNode getParent()	מחזירה את השדה parent של הצומת, כלומר מחזירה את המצביע ל IAVLNode שהינו ההורה של הצומת; אם לא קיים, מחזירה null	
boolean isRealNode()	מחזירה true	
void setHeight(int height)	מעדכנת את הגובה של הצומת	
int getHeight()	מחזירה את הגובה של הצומת	
int getSize()	מחזירה את ה"גודל" של הצומת – סכום הצמתים בתתי העצים שלו +1 עבור עצמו	
void setSize(int newSize)	מקבלת מספר, ומעדכנת אותו להיות ה"גודל" החדש של הצומת (שאמור להיות סכום הצמתים בתתי העצים שלו +1 עבור עצמו)	

<u>המחלקה VirtualNode:</u>

- פירוט המתודות במחלקה VirtualNode (<u>מימוש מתודות המנשק IAVLNode</u>):

המתודה	הסבר	סיבוכיות זמן ריצה
int getKey()	מחזירה -1	סיבוכיות זמן קבועה – O(1), מחזירים ערך
String getValue()	מחזירה null	
void setLeft(IAVLNode node)	מימוש ריק (לא מבצעת כלום) – על מנת שלא יהיה ניתן לעדכן בן שמאלי עבור צומת ווירטואלי	
IAVLNode getLeft()	מחזירה null	
void setRight(IAVLNode node)	מימוש ריק (לא מבצעת כלום) – על מנת שלא יהיה ניתן לעדכן בן ימני עבור צומת ווירטואלי	

IAVLNode getRight()	מחזירה null	
void setParent(IAVLNode node)	מימוש ריק (לא מבצעת כלום) – על מנת שלא יהיה ניתן לעדכן הורה עבור צומת ווירטואלי	
IAVLNode getParent()	מחזירה null	
boolean isRealNode()	מחזירה False	
void setHeight(int height)	מימוש ריק (לא מבצעת כלום) – על מנת שלא יהיה ניתן לעדכן גובה עבור צומת ווירטואלי	
int getHeight()	מחזירה 1-	
int getSize()	מחזירה 0	
void setSize(int newSize)	מימוש ריק (לא מבצעת כלום) – על מנת שלא יהיה ניתן לעדכן גודל עבור צומת ווירטואלי	

במחלקה virtualNode מימשנו את הפונקציות באופן הבא על מנת שלא ניתן יהיה ליצור שינויים לא רצויים בצומת הוירטואלי, שהינו בעל "מופע יחיד". השמנו בו את השדות הרצויים, ומעבר לכך – לא נרצה שיבצעו שינויים נוספים.

<u>המחלקה AVLTree</u>:

- השדות במחלקה AVLTree:

static IAVLNode virtualNode
private IAVLNode root
private int size

(השדה virtualNode הוא שדה אחיד לכלל העצים, המייצג צומת ווירטואלי; קראנו לבנאי שלו ואתחלנו אותו עם הגדרת השדה).

- בנאים במחלקה AVLTree:
  - בנאי ריק, המאתחל את שדות העץ באופן הבא: השורש יצביע ל null, וגודל העץ יהיה אפס. אלו הם המאפיינים שהגדרנו עבור עץ ריק.
  - בנאי המקבל IAVLNode, אותו נאתחל להיות שורש העץ. את גודל העץ נעדכן בהתאם לשדה size של השורש. כמו כן – נעדכן את אב השורש להצביע ל null. כך למעשה אנו מאתחלים עץ על ידי קבלת צומת לשורש, שיגדיר את העץ הנוכחי.

- מתודות המחלקה <u>AVLTree</u>:

*נציין כי כלל פונק' העזר ופירוטן מופיעות תחת ההסבר על הפונק' שקוראת להן.

המתודה	**הסבר + מתודות העזר בהן השתמשנו, ופירוט אודותן**	**סיבוכיות זמן הריצה**
boolean empty()	מחזירה True אם העץ ריק (אם השורש מאותחל להיות null); אחרת, מחזירה False	סיבוכיות זמן קבועה – **O(1)**, ניגשים לשדה root של העץ ומחזירים ערך בוליאני
String search(int k)	מחזירים את השדה info של הצומת אם האובייקט עם המפתח k נמצא בעץ; אחרת, מחזירים null: - אם העץ ריק: נחזיר null, אין אובייקטים בעץ ובפרט אין צומת עם מפתח k בעץ. - אחרת: נקרא לפונקציית עזר רקורסיבית - **search**(IAVLNode node, int k) עם השורש של העץ ועם k, שמחזירה את הצומת בעל המפתח k בעץ אם קיים כזה, אחרת null.	כסיבוכיות זמן הריצה של פונקציית העזר הרקורסיבית search (מפורט מטה - O(log(n))), שכן שאר הפעולות הינן בסיבוכיות זמן ריצה קבוע. לכן: **O(log(n))**
	IAVLNode search(IAVLNode node, int k) פונקציית עזר רקורסיבית, אשר מקבלת צומת בעץ ואת המפתח אותו אנו מחפשים בעץ. אם המפתח שאנו מחפשים שווה למפתח של הצומת, נחזיר את הצומת. אם המפתח קטן ממנו, נקרא לפונקציה הרקורסיבית עם הבן השמאלי של הצומת ואם גדול ממנו, נקרא לה עם הבן הימני של הצומת; כלומר,	סיבוכיות זמן הריצה הינה כאורך המסלול שנבצע – לכל היותר כגובה העץ, מכיוון שבכל פעם נפעיל את המתודה על אחד הבנים, עד שנמצא את הצומת שאנו מחפשים, או עד שנגיע לעלה ווירטואלי. במקרה הגרוע, נתחיל את

	נבצע את החיפוש בתת העץ הימני או השמאלי של הצומת, בהתאם לחיפוש בינארי. אם הגענו לצומת ווירטואלי – נחזיר null, לא מצאנו את המפתח בעץ (סיימנו "לסרוק" את העץ בחיפוש בינארי).	החיפוש מהשורש, ונגיע לצומת ווירטואלי. נזכיר גם כי גובה עץ AVL הינו לוגריתמי במספר הצמתים (מאוזן). לכן: **O(log(n))**
int insert(int k, String i)	נוסיף את הערך k לעץ, במידה והוא לא קיים בו; העץ לאחר ההכנסה יהיה מאוזן, ונחזיר את מספר פעולות האיזון שנדרשו. תחילה, ניצור צומת חדש עם הkey וה-value שקיבלנו. נחפש את הצומת בעץ בעזרת הפונ' **search**: אם הוא קיים בעץ, לא נוסיף אותו; נסיים, ונחזיר 1-. אחרת, הוא לא קיים בעץ. במקרה זה נמשיך לסדרת הפעולות הבאות במטרה להוסיף אותו לעץ באופן תקין: ראשית, נמצא את הצומת אליו נרצה להוסיף את הצומת שלנו בעזרת הפונ' **treePosition**: אם הפונק' החזירה null, העץ ריק ולכן נעדכן את שורש העץ להיות הצומת שנרצה להוסיף לו, ונעדכן את השדות הרלוונטיים בהתאם. במקרה זה סיימנו את ההכנסה ללא פעולות איזון, ולכן נסיים, ונחזיר 0. אחרת, הפונקציה החזירה צומת אליו נרצה להוסיף את הצומת שלנו כבן (שמאלי או ימני). נבדוק האם הצומת אליו נרצה להכניס את הצומת שלנו הוא עלה או לא (בעזרת פונק' עזר **isLeaf**): בכל מקרה, נוסיף את הצומת אליו בעזרת פונק' העזר **insertChild**, ונעדכן את גודל העץ. אם הצומת אליו הכנסנו את הצומת שלנו לא היה עלה, נחזיר 0 – מפני שקיבלנו עץ תקין ללא פעולות איזון. אחרת, הוא לא עלה, ולכן נבצע פעולות איזון לעץ על ידי קריאה לפונק' העזר **insertRebalance** (המבצעת זאת, ומחזירה את מספר פעולות האיזון הנדרשות – הערך אותו נחזיר).	ראשית אנו מבצעים פעולות שלוקחות זמן קבוע O(1). לאחר מכן search לוקחת O(log(n)), וגם treePosition. כעת קוראים ל- insertChild שלוקחת גם O(log(n)) ואז insertBalancing שגם היא בסיבוכיות O(log(n)). אנו מבצעים אותן אחת אחרי השנייה ולכן סה"כ הסיבוכיות היא O(log(n)).
	IAVLNode search(IAVLNode node, int k) תיארנו למעלה.	**O(log(n))**
	IAVLNode treePosition(IAVLNode node) אם העץ ריק, מחזירים null. אחרת, ישנו מצביע x ומצביע y (מסוג IAVLNode). x מתקדם כל פעם לעבר המקום אליו צריך להכניס את הצומת ו-y "עוקב" אחרי x עד שמגיעים (על ידי x) לצומת וירטואלי. x יעודכן להתקדם לתת העץ הימני או השמאלי שלו בהתאם לאלגוריתם חיפוש בינארי. לבסוף, נחזיר את y – שיהיה המקום המתאים להכניס את node.	מתחילים מהשורש ובכל פעם פונים בעץ או שמאלה או ימינה עד שמגיעים לצומת וירטואלי, לכן הסיבוכיות היא כגובה העץ ומכיוון שזהו עץ מאוזן הסיבוכיות היא – **O(log(n))**

	boolean isLeaf(IAVLNode node) מחזירים true אם הצומת הוא עלה, כלומר אם שני הבנים של node צמתים ווירטואליים. אחרת, מחזירה false.	ניגשים לשדות: בן ימני ושמאלי, מפעילים עליהם את המתודה: isRealNode שהינה בסיבוכיות זמן קבועה (מחזירה את השדה realNode של הצומת). בסה"כ – **O(1)**
	void insertChild(IAVLNode parent, IAVLNode child) מכניסים את child כבן ימני או שמאלי של parent, בהתאם לערכי המפתחות שלהם; ההכנסה – כוללת עדכון אחד השדות של parent של ההורה ל child, ואת עדכון השדה parent של child ל parent. לאחר מכן קוראים ל**increaseSize** – על מנת לעדכן את גדלי הצמתים שהכנסנו את הצומת כצאצא שלהם, בהתאם.	ניגשים לשדות הרלוונטיים, ומבצעים להם עדכון – סיבוכיות זמן קבועה. קוראים לפונק' העזר increaseSize – שנפרט בהמשך, כי הינה בסיבוכיות זמן ריצה O(log(n)). לכן: **O(log(n))**
	void increaseSize(IAVLNode node) מגדילים את השדה size של כל הצמתים החל מאב הצומת שהכנסנו, עד לשורש, ב - 1.	במקרה הגרוע, המסלול שנעשה כלפי מעלה יהיה כגובה העץ, ולכן: **O(log(n))**
	int insertRebalance(IAVLNode x) אם x הוא השורש – לא נדרשים איזונים, ולכן נסיים ונחזיר 0. אחרת, נקרא לפונק' איזון מתאימות (שמאזנות את העץ לאחר ההכנסה, ומחזירות את מספר האיזונים שנדרשו), לפי מקרים: אם x בן שמאלי, קוראת ל **insertRebalanceLeft** עם x, אחרת x בן ימני, וקוראת ל **insertRebalanceRight** עם x. את הבדיקה האם הוא בן ימני או שמאלי, היא מבצעת על ידי קריאה לפונק' העזר **isLeftChild** עם x.	isLeftChild היא בסיבוכיות זמן קבועה (נראה למטה). סיבוכיות זמן הריצה של insertRebalanceLeft ו insertRebalanceRight היא O(log(n)) (נראה למטה). לכן: **O(log(n))**
	boolean isLeftChild(IAVLNode node) מחזירה true אם x הוא בן שמאלי, אחרת מחזירה false.	פעולות בסיבוכיות זמן קבועה בלבד (תנאים בוליאניים וגישה ולשדות), ולכן: **O(1)**
	int insertRebalanceLeft(IAVLNode x) נבצע איזון לעץ לאחר הכנסה, עבור בן שמאלי, תחת הפרדה למקרים לפי הפרשי הדרגות בעזרת הפונ' **rankDiff**. נבצע גלגולים ימינה, שמאלה, העלאה והורדה בדרגה, בהתאם למקרה הספציפי, בעזרת פונק' העזר: **rightRotate**, **leftRotate**, **demote** ו **promote**. אם הבעיה נפתרה והעץ מאוזן לאחר	נראה מטה כי כל אחת מהפונק' הנקראות במסגרת ריצת המתודה, הינן בסיבוכיות זמן קבועה. מדובר בפונקציה רקורסיבית, אשר במקרה הגרוע נקראת מס' פעמים כגובה העץ (במסגרת גלגול הבעיה כלפי מעלה), לכן:

	הפעולה שבצענו: נחזיר את מספר הפעולות שנדרשו. אחרת: נחזיר את מספר הפעולות שבצענו + נקרא לפונקציה **insertRebalance** שוב, כאשר גלגלנו את הבעיה במעלה העץ.	**O(log(n))**
	int insertRebalanceRight(IAVLNode x) פועלת באופן סימטרי לפונק' **insertRebalanceLeft** (איזון עבור בן ימני).	בדומה לפונק' הסימטרית לה: **O(1)**
	int rankDiff(IAVLNode parent, IAVLNode child) מחזירה את הפרש הדרגות של שני הצמתים.	מחסירים בין הערכים המתקבלים מ getHeight על כל אחד מהצמתים (ניגשת למצביע לשדה בלבד), לכן: **O(1)**
	void rightRotate(IAVLNode y, IAVLNode x) מבצעים גלגול ימינה בין הצמתים y ל x (y הוא ההורה של x), בהתאם לפעולות הנדרשות לגלגול ימינה שראינו במסגרת ההרצאה – עדכון השדות הרלוונטיים להחלפה, באופן שלמדנו.	משנים מספר קבוע של מצביעים לכן: **O(1)**
	void leftRotate(IAVLNode y, IAVLNode x) מבצעים גלגול שמאלה בין הצמתים y ל x (y הוא הילד של x), בהתאם לפעולות הנדרשות לגלגול שמאלה שראינו במסגרת ההרצאה – עדכון השדות הרלוונטיים להחלפה, באופן שלמדנו.	משנים מספר קבוע של מצביעים לכן: **O(1)**
	void promote(IAVLNode node) מעדכנת את הגובה של הצומת להיות גדול ב- 1 על ידי קריאה ל getHeight ו setHeight על node.	getHeight ו setHeight בסיבוכיות זמן קבועה, לכן: **O(1)**
	void demote(IAVLNode node) מעדכנת את הגובה של הצומת להיות קטן ב- 1 על ידי קריאה ל getHeight ו setHeight על node.	getHeight ו setHeight בסיבוכיות זמן קבועה, לכן: **O(1)**
int delete(int k)	נמחק את הצומת בעל המפתח k מהעץ, אם הוא קיים בו. העץ לאחר המחיקה יהיה מאוזן. נחזיר את מספר פעולות האיזון שנדרשו. תחילה, נבדוק אם העץ ריק באמצעות הפונק' **empty** – אם כן, נחזיר 1- כנדרש. נחפש את הצומת בעל המפתח k בעץ על ידי הפונק' **search** – אם לא קיים בעץ, נחזיר 1- כנדרש. אחרת: אם הצומת היא צומת פנימית (עם שני ילדים) - אנו קוראים לפונק' **replaceWithSuccessor** שמחליפה את הצומת עם היורש שלה. כעת, הצומת שנאלץ למחוק יהיה עלה או אונארי.	Search, replaceWithSuccessor בסיבוכיות - O(log(n)). replaceWithSuccessor, deleteLeaf, ו-deleteUnary, בסיבוכיות זמן ריצה O(log(n)) כל אחת (נראה בהמשך). כל שאר הפעולות בסיבוכיות זמן קבועה. לכן: **O(log(n))**

	נבדוק אם הצומת הוא עלה על ידי הפונק' **isLeaf**; אם כן נבצע את המחיקה באמצעות הפונק' **deleteLeaf**, אחרת, נבצע את המחיקה באמצעות הפונק' **deleteUnary**. המתודות הנ"ל מבצעות הן מחיקה, והן מחזירות את מס' פעולות האיזון שנדרשו לשם כך. לאחר שמחקנו את הצומת באמצעות אחת מהמתודות לעיל, נוריד את גודל העץ ב1 ונחזיר את כמות פעולות האיזון.	
	void replaceWithSuccessor(IAVLNode node) נמצא את היורש של הצומת – צעד אחד ימינה ואז שמאלה עד שמגיעים לעלה. נפריד למקרים: אם היורש הוא בן ישיר של הצומת או צאצא שלו, ונחליף את המצביעים בהתאם כך שנחליף את המקום של שניהם בעץ.	במקרה הגרוע הצומת שנרצה למחוק היא השורש והיורש שלה הוא עלה, ואז מציאת היורש תעלה O(log(n)), שאר הפעולות בעלות O(1) (החלפה של מצביעים), ולכן: **O(log(n))**
	int deleteLeaf(IAVLNode y) מחיקת עלה, נסמנו y: אם y הוא השורש, נהפוך את העץ לריק. אחרת, נבדוק בעזרת **isLeftChild** אם y הוא בן שמאלי או ימני של אבא שלו, ובהתאם נעדכן את המצביע של ההורה – במקום להציע ל y, להצביע לצומת וירטואלי. כעת נקרא ל-**decreaseSize** על מנת לעדכן את גדלי כל הצמתים ש y היה צאצא שלהם (נפחית מהם 1), ואז נפעיל את **deleteRebalane** לצורך איזון העץ, פונק' אשר גם מחזירה את מספר פעולות האיזון שנדרשו לשם כך.	כלל הפעולות בסיבוכיות זמן קבועה, למעט **decreaseSize** ו **deleteRebalane** שנראה בהמשך שהן בסיבוכיות זמן O(log(n)). בסה"כ: **O(log(n))**
	int deleteUnary(IAVLNode y) מחיקת צומת אונארי בעץ, y: נבדוק אם ל-y, יש ילד שמאלי או ימני ומסמנים את x להיות הילד השמאלי או הימני בהתאם. אם y הוא השורש, נגדיר את השורש להיות הבן של y. אחרת, נקרא ל **replaceChild** – כך שאבא של y יצביע לבן של y. כעת נקרא ל-**decreaseSize** על מנת לעדכן את גדלי כל הצמתים ש y היה צאצא שלהם (נפחית מהם 1), ואז נפעיל את **deleteRebalane** לצורך איזון העץ, פונק' אשר גם מחזירה את מספר פעולות האיזון שנדרשו לשם כך.	כלל הפעולות בסיבוכיות זמן קבועה, למעט **decreaseSize** ו **deleteRebalane** שנראה בהמשך שהן בסיבוכיות זמן O(log(n)). בסה"כ: **O(log(n))**
	void replaceChild(IAVLNode parent, IAVLNode child) הפונקציה מקבלת שני צמתים – אבא ובן, ומגדירה את child להיות הבן של parent. אם העץ הוא ריק אז נגדיר את ההורה של child להיות null וסיימנו. אחרת נבדוק אם המפתח של child גדול או קטן מהמפתח של ההורה ונגדיר אותו כבן ימני או שמאלי שלו בהתאם. לבסוף נגדיר את האבא של הילד להיות הצומת parent.	אנו משנים מספר קבוע של מצביעים ומשתנים ולכן: **O(1)**

	void decreaseSize(IAVLNode node) מקטינים את השדה size של כל הצמתים החל מאב הצומת שהכנסנו, עד לשורש, ב - 1.	במקרה הגרוע, המסלול שנעשה כלפי מעלה יהיה כגובה העץ, ולכן: **O(log(n))**
	int deleteRebalance(IAVLNode z) מקבלת צומת וקוראת לאחת מהפונק': **deleteRebalance**, **deleteRebalanceLeft** או **deleteRebalanceRight** בהתאם להפרשי הדרגות של הצומת עם הבנים שלו (חישבנו בעזרת הפונק' **rankDiff**). אם הצומת הוא null (הגענו לשורש), אנו מחזירים 0. אם ההפרש בין האבא לילדים הוא 2,2 – אנו מורידים את הדרגה של האבא, ומגלגלים את הבעיה למעלה (קוראים שוב לפונ' עם האבא) ומחזירים את התוצאה המתקבלת (מס' פעולות האיזון) + 1 (על הורדת הדרגה). אם ההפרש הוא 3,1 אנו קוראים ל-deleteRebalanceLeft, ומחזירים את הערך שלו. אם ההפרש הוא 1,3 אנו קוראים ל-deleteRebalanceRight ומחזירים את הערך שלו.	בכל פעם אנו קוראים או לאיזון השמאלי או לאיזון הימני וכל אחד מהם או פותר את הבעיה או קורא שוב לפונק' ומגלגל את הבעיה למעלה. כל אחת מפונק' האיזון שקראנו לה – בסיבוכיות זמן ריצה O(log(n)). לכן: **O(log(n))**
	int deleteRebalanceLeft(IAVLNode z) נבצע איזון לעץ לאחר מחיקה "למקרה השמאלי" (שראינו בהרצאה), תחת הפרדה למקרים לפי הפרשי הדרגות בעזרת הפונ' **rankDiff**. נבצע גלגולים ימינה, שמאלה, העלאה והורדה בדרגה, בהתאם למקרה הספציפי, בעזרת פונק' העזר: **rightRotate**, **leftRotate**, **demote** ו **promote**. אם הבעיה נפתרה והעץ מאוזן לאחר הפעולה שבצענו: נחזיר את מספר הפעולות שנדרשו. אחרת: נחזיר את מספר הפעולות שבצענו + נקרא לפונקציה **deleteRebalance** שוב, כאשר גלגלנו את הבעיה במעלה העץ.	כל אחת מהפונק' הנקראות במסגרת ריצת המתודה, הינן בסיבוכיות זמן קבועה. מדובר בפונקציה רקורסיבית, אשר במקרה הגרוע קוראת ל deleteRebalance מס' פעמים כגובה העץ (במסגרת גלגול הבעיה כלפי מעלה), לכן: **O(log(n))**
	int deleteRebalanceRight(IAVLNode z) פועלת באופן סימטרי לפונק' **deleteRebalanceLeft.**	באופן דומה ל deleteRebalanceLeft: **O(log(n))**
string Min()	נחזיר את ה info של האיבר המינימלי בעץ; אם העץ ריק, נחזיר null (נבדוק על ידי הפונק' **empty** בהתחלה). כיצד נמצא את האיבר המינימלי בעץ אם הוא לא ריק? כל עוד לא הגענו לצומת שהבן השמאלי שלו הוא ווירטואלי, נעדכן את הצומת להיות הבן השמאלי שלו, בלולאה. הצומת שנקבל הוא הצומת השמאלי ביותר בעץ, והוא האיבר המינימלי.	כאורך המסלול מהשורש עד לעלה השמאלי ביותר – בעץ מאוזן יהיה: **O(log(n))**

string Max()	נחזיר את ה info של האיבר המקסימלי בעץ; אם העץ ריק, נחזיר null (נבדוק על ידי הפונק' **empty** בהתחלה). כיצד נמצא את האיבר המקסימלי בעץ אם הוא לא ריק? כל עוד לא הגענו לצומת שהבן הימני שלו הוא ווירטואלי, נעדכן את הצומת להיות הבן הימני שלו, בלולאה. הצומת שנקבל הוא הצומת הימני ביותר בעץ, והוא האיבר המקסימלי.	כאורך המסלול מהשורש עד לעלה הימני ביותר – בעץ מאוזן יהיה: **O(log(n))**
Int[] keysToArray()	אם העץ ריק, מחזירים מערך ריק. אם לא, נקרא ל **nodeToArray**, שמעדכנת את המערך וממלאת אותו בצמתים של העץ, לפי סדר מהקטן לגדול. לאחר מכן הפונ' עוברת עם לולאה על המערך ומוסיפה למערך חדש בגודל העץ את המפתח של כל צומת, לפי הסדר.	קוראים לפונק' **nodeToArray** שבסיבוכיות זמן ריצה O(n) (נראה למטה). לאחר מכן, עוברים על מערך בגודל n. שאר הפעולות בסיבוכיות זמן קבועה, לכן: **O(n)**
	int nodeToArray(IAVLNode node, IAVLNode[] nodes, int index) עוברת על העץ in-order ומכניסה את כל הצמתים מסודרים, מהקטן לגדול, למערך.	עוברת על כל הצמתים בעץ ומוסיפה אותם לרשימה, לכן הסיבוכיות כמס' הצמתים בעץ: **O(n)**
String[] infoToArray()	אם העץ ריק, מחזירים מערך ריק. אם לא, נקרא ל **nodeToArray**, שמעדכנת את המערך וממלאת אותו בצמתים של העץ, לפי סדר מהקטן לגדול. לאחר מכן הפונ' עוברת עם לולאה על המערך ומוסיפה למערך חדש בגודל העץ את ה info של כל צומת, לפי הסדר.	קוראים לפונק' **nodeToArray** שבסיבוכיות זמן ריצה O(n). לאחר מכן, עוברים על מערך בגודל n. שאר הפעולות בסיבוכיות זמן קבועה, לכן: **O(n)**
int size()	נחזיר את השדה size של העץ – מספק הצמתים בעץ.	גישה למצביע והחזרתו: **O(1)**
AVLTree[] split(int x)	פונק' שמפצלת את העץ לשני עצים לפי המפתח k ומחזירה מערך עם שני עצים – האחד עם מפתחות שקטנים מ x ועץ עם מפתחות שגדולים מ x. ראשית נמצא את הצומת עם המפתח k בעזרת הפונ' **search**. כעת ניצור שני עצים – smaller שהוא תת העץ השמאלי של הצומת x ו-bigger שהוא תת העץ הימני שלו. נקרא לפונק' העזר **splitRec** עם הצומת שלנו ושני העצים ולבסוף נחזיר כמערך את שני העצים שקיבלנו – שיהיו התוצאה הרצויה.	כלל הפעולות בסיבוכיות זמן קבועה, למעט search בסיבוכיות O(log(n)) ו splitRec שנראה בהמשך שבסיבוכיות O(log(n)) (ראו למטה), לכן: **O(log(n))**
	void splitRec(IAVLNode x, AVLTree smaller, AVLTree bigger) פונק' עזר רקורסיבית שמקבלת צומת בעץ - x, ושני עצים – smaller, bigger. אם x בן שמאלי, אז המפתח של אבא שלו וכלל המפתחות בתת העץ הימני של האבא גדולים ממנו, ולכן אז נחבר אותם לעץ bigger בעזרת הפונ' **join** (כאשר האב יהווה את מפתח החיבור). אם x בן ימני, אז המפתח של אבא שלו וכלל המפתחות בתת העץ השמאלי של האבא קטנים ממנו, ולכן אז נחבר אותם לעץ smaller בעזרת הפונ' **join** (כאשר האב יהווה את מפתח החיבור). לאחר מכן נעבור לאבא ונבצע עליו את הפונק' ברקורסיה עד שנגיע לשורש של העץ. לבסוף נקבל שני עצים שמהווים את התוצאה הרצויה.	בצענו k פעולות join (k הוא לכל היותר log(n)), וכל פעם אנחנו מחברים שני עצים ומקבלים עץ שדרגתו שווה לדרגת העץ הגבוה יותר בחיבור. לכן אם נסכום את עלות כלל פעולות ה join במהלך ה split, ומפני שעלות join היא הפרשי הגבהים ועוד 1 (כפי שנסביר בתיעוד של join) – העלות תהיה הדרגה של העץ הגבוה ביותר עליו ביצענו join פחות הדרגה של העץ הנמוך ביותר עליו ביצענו join + k (בצענו k פעולות join), לכן בסה"כ: O(rank(Tk)-rank(T1)+k) =

		O(log(n)) ראינו את חישוב זה בהרצאה – שקופית 72
Int join(AVLNode x, AVLTree t)	מאחדת בין העץ לבין העץ t והצומת x – ומחזירה את סיבוכיות זמן הריצה. אם אחד העצים ריק, נוסיף לעץ שאינו ריק את x בעזרת הפונקציה **insert;** אם העץ שלנו היה ריק, אז גם נעדכן את השורש שלנו להיות השורש של t לאחר ההוספה. אם שניהם ריקים נגדיר את השורש להיות x. אחרת, נבדוק באיזה עץ המפתחות קטנים יותר (נתון שבעץ אחד כל המפתחות קטנים מ-x ובשני גדולים מ-x) ונקרא לפונק' **joinByOrder** כך שהעץ עם המפתחות הקטנים יותר מועבר ראשון, לאחר מכן x ואז העץ עם המפתחות הגדולים יותר. ונחזיר את הפרש הדרגות ועוד 1 כנדרש (joinByOrder מחזירה לנו את ערך זה).	בצענו insert רק במקרה שבו אחד מהעצים הוא ריק – במקרה זה, הפרשי גבהי העצים יהיה כגובה העץ שאינו ריק: **O(log(n))** (n = מס' הצמתים בעץ שאינו ריק). בכל מקרה אחר: המסלול שנבצע למציאת צומת החיבור (posToJoin לה נקרא בתוך הקריאות בפונק') יהיה באורך הפרשי הגבהים של העצים, וכך גם מסלול איזון העץ כלפי מעלה לאחר החיבור בין העצים. לכן: **O(|rank(Tree1)–rank(Tree2)| +1)**
	int joinByOrder(AVLTree T1, IAVLNode x, AVLTree T2) אם הפרשי הדרגות של השורש של שני העצים קטן או שווה ל1, נקרא ל- **joinSameSizeTrees** עם העצים והצומת. אחרת, אם הדרגה של העץ הימני יותר גדולה נקרא ל-**joinRightTreeIsBigger** עם העצים והצומת, ואם הדרגה של השמאלי יותר גדולה, נפעיל את **joinRightTreeIsSmaller** על העצים והצומת.	קוראים לאחת מהפונקציות joinSameSizeTrees (O(1)), joinRightTreeIsBigger או joinRightTreeIsSmaller (שתיהן O(|rank(Tree1)–rank(Tree2)| +1)). לכן: **O(|rank(Tree1)–rank(Tree2)| +1)** (פירוט סיבוכיות הפונק' לעיל מפורטות למטה)
	int joinSameSizeTrees(AVLTree T1, IAVLNode x, AVLTree T2) איחוד העצים והצומת יתבצע כך: - נגדיר את השורשים של שני העצים להיות ילדים של x בעזרת **insertChild**. - נעדכן את השדה של הגודל והדרגה של x בהתאם לעדכון שדות הבנים שלו (גובה: הגובה המקסימלי מבין שניהם +1, גודל: גודל תת העץ הימני + גודל תת העץ השמאלי + 1). - נעדכן את שדות העץ: את הגודל של העץ כולו להיות הגודל של העץ הראשון ועוד הגודל של העץ השני ועוד 1, ומעדכנים את השורש להיות x. לבסוף מחזירים את הפרשי הדרגות של העצים המקוריים,ועוד 1.	אנו מבצעים רק פעולות בסיבוכיות זמן קבועה, כלומר O(1). גם קריאה לinsertChild בסיבוכיות O(1). לכן: **O(1)**

	int joinRightTreeIsBigger(AVLTree T1, IAVLNode x, AVLTree T2) נסמן ב-k את הדרגה של השורש של העץ הנמוך יותר, במקרה הזה T1. כעת נקרא **לposToJoin** עם העץ הגבוה יותר, במקרה הזה T2, ועם k, ו-true (ערך המציין כי נלך על הצלע השמאלית של העץ), ונמצא את c – הצומת שבה אנו רוצים לעשות את החיבור עם x. נגדיר את b להיות הילד של c וכעת בעזרת **insertChild** נגדיר את b ואת השורש של T1 להיות ילדים של x ואת x להיות ילד של c. נעדכן את השורש ואת הגודל של העץ לערכים הנכונים בהתאם. לאחר מכן נבצע איזון לעץ בעזרת **insertRebalance** החל מ x. נעדכן את גודל תת העץ של x ואז נקרא **לfixSize** כדי לעדכן את השדה של הגודל בכל הצמתים החל מ x עד השורש בהתאם. לאחר מכן נחזיר את הפרשי הגבהים של העצים המקוריים + 1 כנדרש.	posToJoin, fixSize ו insertRebalance מבצעות לכל היותר מסלול שאורכו הפרשי הגבהים בין העצים. כעת פעולות החיבור של הצמתים והקריאה ל- insertChild בסיבוכיות O(1) (שינוי של מצביעים). בנוסף עדכון השורש והגודל של העץ בסיבוכיות O(1) גם כן. לכן: **O(|rank(Tree1)–rank(Tree2)| +1)**
	int joinRightTreeIsSmaller(AVLTree T1, IAVLNode x, AVLTree T2) הפונ' סימטרית לחלוטין לפונקציה הקודמת אך מתמודדת עם המקרה שבו השורש של העץ הימני הוא מדרגה קטנה יותר. לכן נסמן ב k את הדרגה של T2, ונפעיל את **posToJoin** על T2, עם k ו-false (ערך המציין כי לא נלך על הצלע השמאלית של העץ, אלא על הימנית) ונמצא את c – הצומת שבה אנו רוצים לעשות את החיבור עם x. נגדיר את b להיות הילד של c וכעת בעזרת **insertChild** נגדיר את b ואת השורש של T2 להיות ילדים של x ואת x להיות ילד של c. נעדכן את השורש ואת הגודל של העץ לערכים הנכונים בהתאם. לאחר מכן נבצע איזון לעץ בעזרת **insertRebalance** החל מ x. נעדכן את גודל תת העץ של x ואז נקרא **לfixSize** כדי לעדכן את השדה של הגודל בכל הצמתים החל מ x עד השורש בהתאם. לאחר מכן נחזיר את הפרשי הגבהים של העצים המקוריים + 1 כנדרש.	באופן דומה לפונק' הקודמת (סימטריות): **O(|rank(Tree1)–rank(Tree2)| +1)**
	IAVLNode posToJoin(IAVLNode node, int k, Boolean isLeft) הפונ' מקבלת שורש של העץ הגבוה יותר, דרגה k וערך בוליאני – isLeft. כל עוד הדרגה של הצומת אותה קיבלנו גדולה מ k, אנו יורדים שמאלה בעץ או ימינה, בהתאם לערך הבוליאני שקיבלנו. אם הערך הוא true אנו יורדים שמאלה ואם false אז יורדים ימינה, עד שאנו מגיעים לצומת בעל דרגה k או קטנה מ k ונחזיר אותו.	כל פעם אנו יורדים למטה בעץ, עד שנגיע לצומת שגובהה כגובה העץ הנמוך לכן: **O(|rank(Tree1)–rank(Tree2)| +1)**
	void fixSize(IAVLNode node) הפונ' מקבלת צומת בעץ (השתמשנו בה רק לצורך join, ולכן הצומת הינה "צומת החיבור" של join בין שני העצים), בכל פעם עולה לאבא עד לשורש ומתקנת בכל צומת את הגודל להיות גודל תת העץ השמאלי ועוד תת העץ הימני ועוד 1.	אורך המסלול מצומת החיבור עד לשורש = הפרשי הגבהים בין העצים שנבצע ביניהם join, ולכן: **O(|rank(Tree1)–rank(Tree2)| +1)**
IAVLNode getRoot()	מחזירה את השדה root של המחלקה.	**O(1)**

<u>חלק ניסויי (תיאורטי)</u>

<u>שאלה 1:</u>

א.

מספר סידורי	מספר חילופים במערך ממוין -הפוך	עלות החיפושים במיון *AVL* עבור מערך ממוין-הפוך	מספר חילופים במערך מסודר אקראית	עלות החיפושים במיון *AVL* עבור מערך מסודר אקראי
1	1,999,000	38884	*992,147*	*36380*
2	7,998,000	85764	*3,989,351*	*81810*
3	31,996,000	187524	*16,020,644*	*174979*
4	127,992,000	407044	*63,692,207*	*383037*
5	511,984,000	878084	*255,549,263*	*821566*

ב. **מספר החילופים במערך ממוין** הפוך יהיה $\binom{n}{2}$, מכיוון שכל זוג איברים במערך הוא היפוך: לכל שני אינדקסים j,i במערך, יתקיים ש- i<j וגם A[i]>A[j] (כאשר A הוא מערך ממוין-הפוך), ולכן כל זוג איברים ייספר כחילוף. **זה אכן תואם למספר החילופים שציינו בטבלה.**

עלות החיפושים של ה AVL במקרה של מערך ממוין הפוך:
בכל הכנסה נבצע פעולת חיפוש בעץ כדי למצוא את המיקום בו נרצה להכניס את האיבר.
נבצע את החיפוש בצורת finger-tree, כלומר באופן הבא –
נתחיל מהאיבר המקסימלי ונעלה למעלה בעץ, עד שנגיע לאיבר שהמפתח של ההורה שלו קטן מהערך שאנו רוצים להכניס לעץ, או עד שנגיע לשורש, ואז נבצע search (כפי שלמדנו בהרצאה - חיפוש בעץ בינארי) על הצומת אליו הגענו.
המערך ממוין הפוך, לכן כל איבר שנכניס יהיה האיבר הכי קטן בעץ בעת הכנסתו. מכאן, שבהתאם למה שהסברנו למעלה, בכל הכנסה נצטרך לעלות עד לשורש, ומשם נבצע חיפוש רגיל בעץ בינארי כמו שראינו בהרצאה.
עבור האיבר ה- i: כשנכניס אותו יהיו i איברים בעץ ולכן החיפוש יעלה O(log(i)). ככה עבור כל איבר נוסף שנכניס, ולכן הסיבוכיות הכוללת היא:

$$\sum_{i=1}^{n} log(i) = log(1 \cdot 2 \cdot \ldots \cdot n) = log(n!) = \mathbf{\Theta(n\, log(n))}.$$

(את המעבר האחרון בשוויון הוכחנו בתרגול מס' 1).

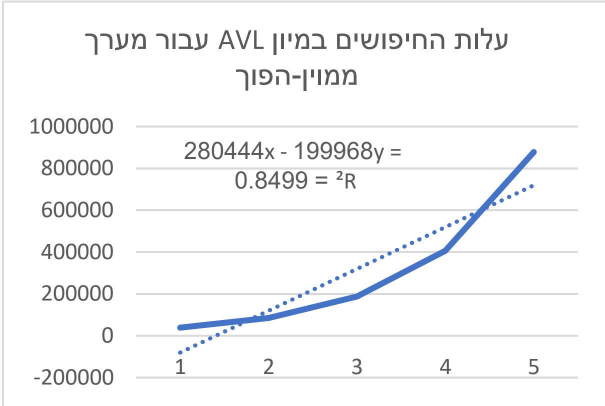
ג. כן, **הערכים בטבלה מסעיף א' והניתוחים מסעיף ב' תואמים.**

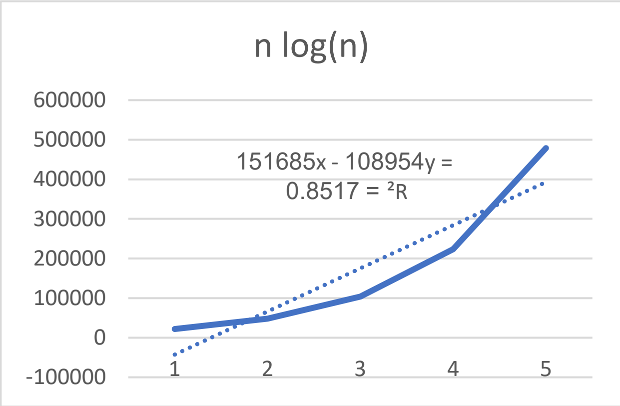
- מספר החילופים במערך ממוין הפוך:

	n = 2000	n = 4000	n = 8000	n = 16000	n = 32000
התוצאות שיצאו בטבלה	1,999,000	7,998,000	31,996,000	127,992,000	511,984,000
התוצאות בניתוח התאורטי (הצבה ב $\binom{n}{2}$)	$\binom{2000}{2} = 1{,}999{,}000$	$\binom{4000}{2} = 7{,}998{,}000$	$\binom{8000}{2} = 31{,}996{,}000$	$\binom{16000}{2} = 127{,}992{,}000$	$\binom{32000}{2} = 511{,}984{,}000$

ניתן לראות כי **הערכים תואמים.**

- עלות החיפושים במערך ממוין הפוך:
מימין, הגרף וקו המגמה של התוצאות שקיבלנו בטבלה, ומשמאל, גרף הפונק' nlog(n) (הסיבוכיות מסעיף ב):





ניתן לראות כי **הערכים תואמים**.

ד. נתון מערך שמספר החילופים בו הוא h. נהדק את החסם-העליון על עלות המיון באמצעות עץ ה AVL- במונחי n, h.
נסמן עבור איבר כללי i במערך את כמות האיברים שגדולים ממנו ונמצאים לפניו במערך ב- h_i. נשים לב ש- $h_1 + ... + h_n = h$ כאשר h שווה לסך כמות החילופים במערך.
לאיבר ה - i ישנם h_i חילופים. כלומר, כאשר אנו מכניסים אותו לעץ יש h_i איברים שגדולים ממנו ונמצאים כבר בעץ (במילים אחרות, האיבר שאנחנו מכניסים הוא האיבר ה $h_i + 1$ בגודלו, מהסוף להתחלה). בהתאם לאופן חיפוש finger-tree שמתחיל מהאיבר המקסימלי (העלה הימני), נעלה בעץ לכל היותר $h_i + 1$ צמתים (עד שנגיע לצומת שערך המפתח של ההורה שלו קטן מערך ההכנסה). משם, נמשיך לחיפוש בינארי רגיל – על עץ בגובה $h_i + 1$ לכל היותר ← חסם עליון לעלות החיפוש: $O(log(h_i + 1))$.
אנו מבצעים כך עבור כל איבר במערך. לכן סה"כ עבור n איברים הסיבוכיות הכוללת היא:

$$\sum_{i=1}^{n} log(h_i + 1) = log(h_1 + 1) + ... + log(h_n + 1) = log(\prod_{i=1}^{n}(h_i + 1))$$

כעת נבצע על הנוסחה שקיבלנו שורש n-י ונעלה בחזקת n ואז נוכל לקבל חסם עליון מאי-שוויון הממוצעים באופן הבא:

$$\sqrt[n]{\prod_{i=1}^{n} h_i + 1}^{\,n} \le \left(\frac{\sum_{i=1}^{n} h_i + 1}{n}\right)^n = \left(\frac{h+n}{n}\right)^n$$

לכן, חסם עליון לסיבוכיות הוא: $log\left(\left(\frac{h+n}{n}\right)^n\right) = n\, log\left(\frac{h+n}{n}\right) = \boldsymbol{O}\left(\boldsymbol{n\, log}\left(\frac{\boldsymbol{h+n}}{\boldsymbol{n}}\right)\right)$

ה. בסעיף ד' ביקשנו חסם עליון בלבד מפני שהחסם התחתון לאו דווקא תלוי בh. זאת מפני שאם בכל פעם נוסיף איבר שגדול מכל האיברים שקיימים עד כה במערך, נבצע בסך הכל שתי פעולות (השוואה לאיבר המקסימלי והכנסת האיבר החדש תחתיו), זאת על אף שמספר החילופים במקרה זה הוא 0 (מערך ממוין הפוך). לכן חסם תחתון על עלות חיפוש בודד הוא 2 ואינו תלוי ב h ולכן חסם תחתון לעלות החיפושים הכוללת גם הוא לא יהיה תלוי ב h.

<u>שאלה 2</u>

<u>סעיף א'</u>

מספר סידורי	**עלות join ממוצע עבור split אקראי**	**עלות join מקסימלי עבור split אקראי**	**עלות join ממוצע עבור split של האיבר מקסימלי בתת העץ השמאלי**	**עלות join מקסימלי עבור split של איבר מקסימלי בתת העץ השמאלי**
1	2.66	7	3.1	12
2	2.25	5	3.0	13
3	2.83	7	3.45	14
4	2.9	4	2.92	15
5	2.92	5	3.25	17
6	2.85	8	3.142857	17
7	2.57	5	3.125	18
8	2.66	6	2.94	19
9	3.0	6	3.2666667	20
10	2.66	6	3.2	20

<u>סעיף ב'</u>

ראשית, נתייחס ל**עלות join ממוצע עבור split של האיבר מקסימלי בתת העץ השמאלי**:

מפני שהצומת עליו אנחנו מבצעים join הוא המקסימלי בתת העץ השמאלי: או שאין לו בנים כלל (עלה), או שיש לו בן אחד – שמאלי.

ה split עליו מתבצע באופן הבא:

1. נייצא לשני עצים smaller ו-bigger, את תת העץ הימני והשמאלי של הצומת בהתאמה: שניהם ריקים או ש-bigger ריק וsmaller הוא מגודל 1.
2. לאחר מכן, כל עוד לא הגענו לשורש העץ:
   - נבצע join בין smaller לבין תת העץ השמאלי של האבא של הצומת הנוכחי; נשים לב שבכל פעם הפרש הגבהים של שני העצים שנחבר יהיה לכל היותר 2 ולכל הפחות 1 (לפי המבנה של עץ AVL תקין).
   - נעדכן את הצומת להיות האבא שלו.
3. הגענו לשורש העץ: כעת, נבצע join בין תת העץ הימני של השורש (גובה (log(n)) לבין bigger (גובה 0 מפני שלא הוספנו לו איברים עד כה).

כפי שראינו בהרצאה, סיבוכיות זמן הריצה הכוללת של split כנ"ל תהיה:

$$O\left(rank(T_k) - rank(T_1) + k\right) \;=\; O(\log n)$$

כאשר Tk הינו העץ בעל הדרגה הגבוהה ביותר (=העץ הגבוה ביותר) שחיברנו במהלך ה split, ו T1 הינו העץ בעל הדרגה הנמוכה ביותר (=העץ הנמוך ביותר) שחיברנו במהלך ה split, וכפי שהסברנו מעלה:

Rank(Tk) = log(n), Rank(T1) = 0

ובצענו בין1 + log(n)\2 לבין log(n) + 1 פעולות join בסה"כ ולכן נקבל שסיבוכיות זמן הריצה בהתאם לנוסחה לעיל הינה –

O(log(n))

לכן, סה"כ – העלות הכוללת היא O(log(n)), ובצענו O(log(n)) פעולות ← העלות הממוצעת לפעולת join במקרה זה היא **O(1) =** O(log(n))/ O(log(n)).

ניתן לראות שזה אכן מתיישב עם התוצאות בטבלה, שכן מדובר בקבוע - ללא תלות בגודל העץ.

כעת, נתייחס לעלות join ממוצע עבור split אקראי:

יהי צומת אקראי בעץ, x, שדרגתו הינה k. נניח כי גובה העץ הינו h.

- כמה פעולות join נבצע במהלך ה split? הפרש הגבהים בין הצומת לבין השורש הינו h-k, והפרש ה rankים האפשרי בין כל צומת לבין ההורה שלו הוא 1 או 2. לכן נבצע בסה"כ בין (h-k)/2 לבין (h-k) פעולות join, שכן מבצעים פעולה אחת כזו בכל "עלייה" למעלה בעץ (מהבן להורה), עד שמגיעים לשורש העץ.
- מה עלות פעולות ה join הכוללת במהלך ה split? נחלק ל 2 סכומים, שיחדיו מהווים את הסכום הנ"ל:
נציין קודם לכן, כי כפי שראינו בהרצאה, העלות הכוללת של פעולות join על m עצים היא הפרש הגבהים בין העץ הגבוה ביותר לבין גובה העץ הנמוך ביותר + מספר פעולות ה join שבצענו בחלק זה.
  - סכום פעולות ה join ל smaller (מערך המפתחות הקטנים מערכו של x):
גובה הצומת x הוא k. נניח בלי הגבלת הכלליות כי גובה תת העץ השמאלי שלו הוא k-1 וגובה תת העץ הימני שלו הוא k-2. לכן, העץ הנמוך ביותר עליו נבצע join כנ"ל יהיה בגובה k-1, והעץ הגבוה ביותר יהיה תת העץ השמאלי של השורש, בגובה, בה"כ h-2.
  - סכום פעולות ה join ל bigger (מערך המפתחות הגדולים מערכו של x):
גובה הצומת x הוא k. הנחנו בלי הגבלת הכלליות כי גובה תת העץ השמאלי שלו הוא k-1 וגובה תת העץ הימני שלו הוא k-2. לכן, העץ הנמוך ביותר עליו נבצע join כנ"ל יהיה בגובה k-2, והעץ הגבוה ביותר יהיה תת העץ הימני של השורש, בגובה (בה"כ ובהתאם לאמור לעיל) h-1.

לכן בסה"כ –

- עבור smaller, ההפרש בין גבהי העצים (הגבוה ביותר והנמוך ביותר) הוא: h-k-1 = (h-2)-(k-1)
- עבור bigger, ההפרש בין גבהי העצים (הגבוה ביותר והנמוך ביותר) הוא: h-k+1 = (h-1)-(k-2)
- כמות פעולות ה join הכוללת היא כ בין (h-k)/2 לבין (h-k)

ומכיוון שהעלות הכוללת של פעולות join על m עצים היא הפרש הגבהים בין העץ הגבוה ביותר לבין גובה העץ הנמוך ביותר + מספר פעולות ה join שבצענו, נקבל: O(h-k).

מכיוון שגם בצענו O(h-k) פעולות join בסה"כ, הסיבוכיות הממוצעת היא: **O(1)** = O(h-k)\ O(h-k).

ניתן לראות שזה אכן מתיישב עם התוצאות בטבלה שכן מדובר בקבוע - ללא תלות בגודל העץ.

<u>סעיף ג'</u>

נתייחס לעלות של join מקסימלי עבור split של האיבר המקסימלי בתת העץ השמאלי:

במהלך הsplit, אנו מסמנים את תת העץ השמאלי של הצומת להיות העץ smaller ואת העץ הימני של הצומת להיות העץ bigger. הצומת ממנה התחלנו היא או עלה או צומת אונרי. לכן, העץ bigger הוא ריק בהתחלה והעץ smaller הוא או ריק או עם צומת אחד. הצומת שלנו הוא המקסימלי בתת העץ השמאלי של העץ המקורי, ולכן כל הצמתים עד השורש קטנים ממנו, ומכאן שאת כל תתי העצים עד השורש נחבר לעץ smaller ונקבל בכל פעם הפרש גבהים נמוך מאוד בין שני העצים ובהתאם גם סיבוכיות נמוכה (מפאת הפרשי ה rankים המותרים בעץ AVL). כאשר מגיעים לשורש מחברים את תת העץ הימני של השורש עם העץ bigger שהוא כרגע עץ ריק ולכן מדרגה 0. תת העץ הימני של השורש הוא מדרגה שקטנה ב 1 או ב 2 מהשורש שהוא מדרגה log(n) כאשר n כמות הצמתים בעץ. לכן סיבוכיות join זה כפי שלמדנו הוא – O((log(n)-1)-0+1) או O((log(n)-2)-0+1) כלומר O(log(n)).

זהו הjoin עם הסיבוכיות המקסימלית כי אנו מחברים עצים שהפרשי הדרגות שלהם כגובה העץ, ולא ייתכנו שני עצים עם הפרש גדול מזה בתוך העץ.

נשים לב **שאכן זה מתיישב עם התוצאות שקיבלנו בטבלה** – למשל עבור n=2000: $\log_2 2000 + 1 = 10.96 + 1 \approx$ 12, בדיוק כפי שקיבלנו בטבלה.