# מבנה נתונים – פרויקט AVLTree – קובץ תיעוד ובדיקות

**מגישים:**

**Name: Idan Rahamim**

[redacted]

# תוכן עניינים

## Class Item

Stores (key-info). Will be used by AVLNode or by element in CircularList

- *Field Detail*

- *info*

```
private java.lang.String info
```

the info of the node

- *key*

```
private int key
```

the key of the node

**כלל הפעולות הבאות בסיבוכיות קבועה – השמה / החזרת שדה**

- *Constructor Detail*

- *Item*

```
public Item()
```

default constructor

- *Item*

- ```
  public Item(int key,
              java.lang.String info)
  ```

  constructor by key-info

  **Parameters:**

  `key - the key`

  `info - the info`

- *Method Detail*

- *getInfo*

```
public java.lang.String getInfo()
```

**Returns:**

`info value`

- *getKey*

```
public int getKey()
```

**Returns:** `key value`
```

## Class AVLTree.AVLNode

implements AVLTree.IAVLNode

class AVLNode implements IAVNode uses to creat nodes with fields: item-key+val,parent,left,right,height,size nodes uses by TreeList class to build a tree.

**כלל הפעולות הבאות הן ב(O(1 – השמה לשדה / החזרת שדה**

- *Constructor Detail*

- *AVLNode*

```
public AVLNode()
```

default constructor initialize a node, with default fields.

- *AVLNode*

- ```
  public AVLNode(int key,
                 java.lang.String info)
  ```

  constructor initialize a node to have Item with key and info as given

  **Parameters:**

  key- - nodes key

  info- - nodes info

- *Method Detail*

- *getBF*

```
private int getBF(AVLTree.AVLNode node)
```

get the balance factor of a node

**Parameters:**

node - to calculate its balance factor

**Returns:**

the Balance Factor of the node

**כיצד פועלת:** מחשבת את הגובה של הבן השמאלי פחות הגובה של הבן הימני
**סיבוכיות**: O(1) - נעשית גישה לבנים הישירים ע"מ להשיג את גובהם.

- *getHeight*

```
public int getHeight()
```

gets the height of the node

**Specified by:**

getHeight in interface AVLTree.IAVLNode

**Returns:**

height of the node
```

- *getItem*

```
public Item getItem()
```

**Returns:**

Item that contain key and val of the node.

- *getKey*

```
public int getKey()
```

**Specified by:**

getKey in interface AVLTree.IAVLNode

**Returns:**

key of the node.

- *getLeft*

```
public AVLTree.IAVLNode getLeft()
```

gets the left child of the node

**Specified by:**

getLeft in interface AVLTree.IAVLNode

**Returns:**

left child node

- *getParent*

```
public AVLTree.IAVLNode getParent()
```

gets the parent of the node

**Specified by:**

getParent in interface AVLTree.IAVLNode

**Returns:**

parent of the node

- *getRight*

```
public AVLTree.IAVLNode getRight()
```

gets the right child of the node

**Specified by:**

getRight in interface AVLTree.IAVLNode

**Returns:**

right child node

- *getSize*

```
public int getSize()
```

gets the size of the node

**Returns:**

size of the node

- *getValue*

```
public java.lang.String getValue()
```

**Specified by:**
getValue in interface AVLTree.IAVLNode

**Returns:**
val of the node.

- *setHeight*

```
public void setHeight(int height)
```

sets the height of the node

**Specified by:**
setHeight in interface AVLTree.IAVLNode

**Parameters:**
height - new height to be updated

- *setLeft*

```
public void setLeft(AVLTree.IAVLNode node)
```

sets the left child of the node

**Specified by:**
setLeft in interface AVLTree.IAVLNode

**Parameters:**
node - left node

- *setParent*

```
public void setParent(AVLTree.IAVLNode node)
```

sets the parent of the node

**Specified by:**
setParent in interface AVLTree.IAVLNode

**Parameters:**
node - parent node

- *setRight*

```
public void setRight(AVLTree.IAVLNode node)
```

sets the right child of the node

**Specified by:**
setRight in interface AVLTree.IAVLNode

**Parameters:**
node - right node

- *setSize*

```
public void setSize(int size)
```

sets the size of the node

**Parameters:**

```
size - new size to be updated
```

## Class AVLTree

An implementation of an AVL Tree
has a pointer to an AVLNode which is the root of the tree
implements search, insert, delete, in O(logn) by keeping the tree balanced (each AVLNode has Balance Factor smaller the 2 and greater the -2)

- *Field Detail*

- *maxNode*

```
private AVLTree.AVLNode maxNode
```

will point to the AVLNode with the maximum key in the tree

- *minNode*

```
private AVLTree.AVLNode minNode
```

will point to the AVLNode with the minimum key in the tree

- *root*

```
private AVLTree.AVLNode root
```

will contain a pointer to an AVLNode which is the root of the tree, enables access to other nodes by that

- *Constructor Detail*

- *AVLTree*

```
public AVLTree()
```

default constructor- initialize a tree with default fields (root = null)

**סיבוכיות:** O(1) - השמה

- *Method Detail*

- *checkAndCommitRotation*

```
public int checkAndCommitRotation(AVLTree.AVLNode node)
```

checks for a node if needs rotation.
if does - commit rotations, updating heights and sizes of the participant nodes and count number of rotations done.

**Parameters:**

```
node - to check if its avl criminal.
```

**Returns:**

```
returns how many rotations were made if needed.
```

**כיצד פועלת:** מקבלת מהפוקנציה AVLNode.getBF את ה balance factor של הצומת. אם הוא שונה מ2, 2- מחזירה 0 - לא נעשו גלגולים.

אחרת, אם BF = 2 אז בודקת את הBF של הבן השמאלי שלה בעזרת אותה פונ' עזר.

- אם הוא 1 או 0 מבצעת גלגול שמאלה בעזרת הפונקציה rotateLeft. (0 זה מקרה שמתאים למחיקה, אבל כאן לא נעשית הבחנה בין מחיקה לבין הכנסה שכן לא יתכן בהכנסה שBF של הצומת יהיה 2 (או 2-) ושל אחד הבנים בהתאמה יהיה 0). מחזירה 1 (נעשה גלגול יחיד)
- אחרת (שווה ל1-) מבצעת גלגול שמאלה ואז ימינה בעזרת הפונ' rotateLeftRight. מחזירה 2 (נעשו שני גלגולים - שמאלה וימינה)

אם BF = 2- אז בודקת את הBF של הבן הימני שלה בעזרת אותה פונ' עזר.

- אם הוא 1- או 0 מבצעת גלגול שמאלה בעזרת הפונקציה rotateRight. (0 זה מקרה שמתאים למחיקה, אבל כאמור הפונקציה לא מבחינה). מחזירה 1 - נעשה גלגול יחיד.
- אחרת (שווה ל1) מבצעת גלגול ימינה ואז שמאלה בעזרת הפונקציה rotateRightLeft

בכל אחד מהמקרים שומרת את מספר הגלגולים שנעשו (הוחזרו מפונ' העזר הרלוונטית). מחזירה 2 - נעשו 2 גלגולים ימינה ושמאלה.

**סיבוכיות**: O(1) . הפונקציה AVLNode.getBF עובדת בזמן קבוע שכן נעשית גישה ישירה לבנים. פונקציות הגלגול השונות גם עובדות ב O(1) שכן מתבצע שינוי פוינטרים בלבד ועדכון הגבהים והגדלים לפי הבנים הישירים.

- *delete*

```
public int delete(int k)
```

deletes an item with key k from the binary tree, if it is there

**Parameters:**

```
k - key of item requested to be deleted.
```

**Returns:**

```
the number of rebalancing operations, or 0 if no rebalancing
operations were needed, or -1 if an item with key k was not
found in the tree.
```

**כיצד פועלת:**

1. first finds the requested node by getNodeByKey(int key). If found then:

1.1. sets its children and parent and gets the node to start fix the height and sizes from - by the method setParentAndChildrenOnDelete_AndGetStartPoint(AVLNode node)

1.2 fixes the sizes and heights (and does rotations) from the node gotten on 1.1

by the method fixHeightAndSizeOnDelete_AndGetNumOfRotations(AVLNode startFixingFrom)

1.3 sets the fields minNode, maxNode by using getMinNode(root), getMaxNode(root)

**סיבוכיות**: O(logn). תחילה ב1 מתבצעת ירידה מהשורש למציאת הצומת המבוקש - במקרה הרע (הצומת לא נמצא / מדובר בעלה) ירידה של O(logn) רמות עד לעלים.
לאחר מכן ב1.1 מתבצע עדכון פוינטרים ב O(1). הפונ'
(setParentAndChildrenOnDelete_AndGetStartPoint(AVLNode node
רצה ב O(logn) במקרה הגרוע (יש לצומת שני בנים וצריך למצוא את העוקב שלו)
שלב 1.2 מתבצע בעליה במסלול המחיקה מהצומת הפיזי אל השורש- במקרה הרע (מחיקת עלה) כגובה העץ ולכן O(logn). ראה סיבוכיות הפונקציה המתאימה ב1.2.
שלב 1.3 מתבצע גם הוא ב O(logn) – ראה סיבוכיות הפונ' המתאימות (בקצרה: נעשית ירידה מהשורש כגובה העץ פעם אחת לקביעת המינימלי – האיבר השמאלי ביותר בעץ, ופעם אחת לקביעת המקס' – האיבר הימני ביותר בעץ)

- *empty*

```
public boolean empty()
```

returns true if and only if the tree is empty (down't contain any node)

**Returns:**

```
true- if the tree is empty, else- false
```

**סיבוכיות** - O(1): השוואה של root ל.null

- *exchangeOriginToSuccessor*

- `public void exchangedOriginToSuccessor(AVLTree.AVLNode origin, AVLTree.AVLNode successor)`

sets the successor node to exchange origin (in case node has to children)
means phisically deletion of successor and deletion of origin

**Parameters:**

```
origin - the node its place will be overidden

successor - the node which will be put instead of origin
```

**כיצד פועלת:** ההחלפה מתבצעת על ידי שינוי פוינטרים. נעשית אבחנה בין המקרים הבאים: origin הוא השורש, וכן אם successor הוא הבן הימני של origin.
**סיבוכיות**: O(1)- החלפת פוינטרים בלבד.

- *fixHeightAndSizeOnDelete_AndGetNumOfRotations*

```
public int fixHeightAndSizeOnDelete_AndGetNumOfRotations
(AVLTree.AVLNode startFixingFrom)
```

fixes heights, sizes, and does rotations from the given node and above in its path to the root

**Parameters:**

```
startFixingFrom - node to start fixing height, size and does
rotation from (and include)
```

**Returns:**

```
number of rotations done
```

**כיצד פועלת:**

for each node from the given in the path climbing up to the root do:
1. update height and size of the current node using updateSize(AVLNode), updateHeight(AVLNode);
2. commits rotations (checks before if needed) - both by checkAndCommitRotation(AVLNode), and sums them. Pay attention, this function fixes the heights and sizes too.

**סיבוכיות:** O(logn) או חסם יותר הדוק O(depth(given_node)): נעשית עליה מהצומת הנתון אל השורש במסלול העליה ובכל צומת נעשות פעולות קבועות של ביצוע רוטציה, עדכון הגובה והגודל.

- *getChildrenHeights*

```
private int[] getChildrenHeights(AVLTree.AVLNode node)
```

returns an array with the heights of the children of node.

**Parameters:**

```
node - the node its children heights needed.
```

**Returns:**

```
an array [leftHeight, rightHeight]
```

**סיבוכיות** -O(1): גישה לשדות ואתחול מערך באורך 2.

- *getChildrenSizes*

```
private int[] getChildrenSizes(AVLTree.AVLNode node)
```

returns an array with the sizes of the children of node.

**Parameters:**

```
node - the node its children sizes needed.
```

**Returns:**

```
an array [leftSize, rightSize]
```

**סיבוכיות** -O(1): גישה לשדות ואתחול מערך באורך 2.

- *getMaxInSubTree*

```
public AVLTree.AVLNode getMaxInSubTree(AVLTree.AVLNode origin)
```

returns the node with max key in the subtree which origin is the root.

**Parameters:**

```
origin - the root of sub-tree to find the node with the max
key in
```

**Returns:**

```
the node which has the highest key in the subtree
```

**כיצד פועלת:** מתחילה מהצומת שמתקבל ויורדת כל הזמן ימינה עד הצומת האחרון שאינו null
**סיבוכיות:** O(logn) או חסם יותר הדוק O(height(given_node)) - יש לרדת ימינה מהצומת במקרה הרע מס' רמות ששווה לגובהו של הצומת

- *getMinInSubTree*

```
private AVLTree.AVLNode getMinInSubTree(AVLTree.AVLNode origin)
```

return the min node in the subtree which origin is the root

**Parameters:**

```
origin - root of the tree
```

**Returns:**

```
the min node in the subtree which origin is the root. if
tree is empty, returns null
```

**כיצד פועלת:** מתחילה מהצומת שמתקבל ויורדת כל הזמן שמאלה עד הצומת האחרון שאינו null
**סיבוכיות**: O(logn) או חסם יותר הדוק O(height(given_node)).- יורדים שמאלה מהצומת במקרה הרע מס' רמות כגובהו.

- *getNodeByKey*

```
private AVLTree.AVLNode getNodeByKey(int key)
```

get a node by it key

**Parameters:**

```
key - key of the node being searched
```

**Returns:**

```
returns the node with key of the argument inserted if it
exists in the tree otherwise, returns null
```

**כיצד פועלת:** משתמשת בפונק עזר רקורסיבית getNodeByKeyRec.

**סיבוכיות**: O(logn). הסיבוכיות היא כאורך המסלול מהשורש עד לצומת. במקרה הגרוע הצומת לא נמצא בעץ ויש לרדת O(logn) רמות

- *getNodeByKeyRec*

- `private AVLTree.AVLNode getNodeByKeyRec(AVLTree.AVLNode node, int key)`

  recursive function as helper for getNodeByKey(int key)

  **Parameters:**

  ```
  node - root of a subtree

  key - key of the node being searched
  ```

  **Returns:**

  ```
  returns the node with key of the argument inserted if it
  exists in the subtree otherwise, returns null
  ```

**סיבוכיות**: O(logn) או חסם יותר הדוק O(height(given_node)): יורדת מהצומת הנתון במקרה הרע (הצומת המבוקש עלה / לא נמצא) מס' רמות ששווה לגובה הצומת.

- *getRoot*

  `public AVLTree.IAVLNode getRoot()`

  **Returns:**

  ```
  Returns the root AVL node, or null if the tree is empty
  precondition: none postcondition: none
  ```

**סיבוכיות** -O(1): גישה ישירה לשדה.

- *getSuccessorInCaseOfTwoChildren*

  `public AVLTree.AVLNode getSuccessorInCaseOfTwoChildren(AVLTree.AVLNode origin)`

  gets the successor of the node origin in case origin has 2 children

  **Parameters:**

  ```
  origin - node to find its successor. pre condition - the
  node has two children
  ```

  **Returns:**

  ```
  the successor of origin
  ```

**כיצד פועלת:** מחזיר את הצומת עם המפתח המינימלי בתת העץ הימני של הצומת node. משתמש בפונקציית העזר getMinInSubTree(AVLNode node)
**סיבוכיות**: O(logn) או חסם יותר הדוק O(height(given_node)). - יורדים במקרה הרע מס' רמות שהוא כגובה הצומת למציאת העוקב (במקרה הרע העוקב הוא עלה) . ראה סיבוכיות של getMinInSubTree (AVLNode)

- *infoToArray*

  `public java.lang.String[] infoToArray()`

  **Returns:**

  ```
  Returns an array which contains all info in the tree, sorted
  by their respective keys,
  or an empty array if the tree is empty.
  ```

**כיצד פועלת:** שולחת לפונקציה inorder מצביע לשורש העץ וכן רשימה ריקה ומקבלת בחזרה רשימה עם צמתי העץ מסודרים בסדר עולה על פי המפתח שלהם. נעשה מעבר על הרשימה כדי להכניס את את הinfo של צמתיה למערך.
**סיבוכיות**: O(n) - כשל inorder - נעשה מעבר על כל צמתי העץ - ראה סיבוכיות inorder

- *inOrder*

- `private void inOrder(AVLTree.AVLNode node, java.util.ArrayList<AVLTree.AVLNode> list)`

  fills the given list with the nodes sorted by their keys from small to higher.

  **Parameters:**
  ```
  node - the root of subtree

  list - contains the so-far sorted nodes by their keys
  ```

  **כיצד פועלת**: פועלת ריקורסיבית - תחילה פונה לתת העץ השמאלי של הצומת, אח"כ מוסיפה את הצומת הנוכחי לרשימה הנתונה, ואז פונה לתת העץ הימני
  **סיבוכיות**: O(n) - מבקרת בכל צומת 3 פעמים (הגעה אליו, ירידה ממנו שמאלה ועליה אליו בחזרה, ירידה ממנו ימינה ועלייה בחזרה אליו) ובכל צומת עבודה בזמן קבוע O(1) - הכנסה לרשימה.

- *insert*

- `public int insert(int k, java.lang.String i)`

  inserts an item with key k and info i to the AVL tree.

  **Parameters:**
  ```
  k - key of item inserted

  i - value of item inserted
  ```

  **Returns:**
  ```
  the number of rebalancing operations, or 0 if no rebalancing
  operations were necessary. -1 if an item with key k already
  exists in the tree.
  ```

  **כיצד פועלת:** מקבלת מפתח k ומחרוזת i, בודקת אם המפתח כבר קיים בעץ, אם לא יוצרת צומת חדש עם מפתח וערך זה ומכניס לעץ ומחזירה את מספר הגלגולים שהתבצעו, אחרת מחזירה -1. משתמשת בפונ העזר getKey כדי לקבל את המפתחות של הצמתים במסלול, ולאחר שמוצאת את מקום ההכנסה לפי חוקיות של עץ חיפוש בינארי יוצרת את הצומת החדש, מעדכנת גובה ומידה עם פונק העזר setHeight,setSize ומכניסה אותו לעץ. לבסוף חוזרת אחורה במסלול, מעדכנת סייז וגובה לצמתים, מבצעת גלגולים בעזרת פונ העזר checkAndComitRotations. לבסוף נעשית השמה לשדות minNode, maxNode ע"י הפונקציות getMinInSubTree, getMaxInSubTree שמופעלות על הארגומנט root.
  **סיבוכיות**: O(logn). גובה של מסלול מהשורש בעץ avl לעלה (הצומת נכנס כעלה) הוא O(logn), עוברים במסלול אחד כדי להכניס צומת, ואז חוזרים על אותו מסלול כדי לבצע גלגולים ולעדכן גובה ומידה, גלגולים ועדכונים עולים O(1). לבסוף נעשית ירידה כגובה העץ לקביעת הצומת המינימלי והמקסימלי. לכן סהכ O(logn).

- *keysToArray*

  `public int[] keysToArray()`

  **Returns:**
  ```
  Returns a sorted array which contains all keys in the tree,
  or an empty array if the tree is empty.
  ```

  **כיצד פועלת:** שולחת לפונקציה inorder מצביע לשורש העץ וכן רשימה ריקה ומקבלת בחזרה רשימה עם צמתי העץ מסודרים בסדר עולה על פי המפתח שלהם. נעשה מעבר על הרשימה כדי להכניס את את המפתחות של צמתיה למערך.
  **סיבוכיות**: O(n) - כשל inorder - נעשה מעבר על כל צמתי העץ - ראה סיבוכיות inorder

- *max*

```
public java.lang.String max()
```

**Returns:**

```
Returns the info of the item with the largest key in the
tree, or null if the tree is empty
```

**כיצד פועלת:** מחזירה את ערכו של השדה maxNode. אם השדה הוא null מחזירה null.
**סיבוכיות**: O(1) – גישה למצביע

- *min*

```
public java.lang.String min()
```

**Returns:**

```
Returns the info of the item with the smallest key in the
tree,
or null if the tree is empty
```

**כיצד פועלת:** מחזירה את המצביע ל minNode. אם השדה הוא null מחזירה null.
**סיבוכיות**: O(1) – גישה למצביע

- *rotateLeft*

```
private void rotateLeft(AVLTree.AVLNode x)
```

commits a rotation to the left

**Parameters:**

```
x - the node needs a rotation to the right.
```

**כיצד פועלת:** מבצעת גלגול שמאלה כשם שנלמד בכיתה.
**סיבוכיות**: O(1) - שינוי מצביעים בלבד - זמן קבוע. עדכון הגבהים והגדלים גם באופן קבוע - מסתמך על הבנים.

- *rotateLeftRight*

```
private void rotateLeftRight(AVLTree.AVLNode node)
```

commits two rotations, first to the left and then to the right.

**Parameters:**

```
node - the node needs rotations to the left and then to the
right.
```

**כיצד פועלת:** תחילה מבצעת גלגול שמאלה בעזרת הפונקציה rotateLeft עם הבן השמאלי של הצומת הנתון. לאחר מכן (מבנה העץ השתנה), מבצעת גלגול ימינה עם הצומת הנתון. נעשה עדכון heights, sizes של השותפים בגלגול.
**סיבוכיות**: O(1) - ראה סיבוכיות rotateRight, rotateLeft

- *rotateRight*

```
private void rotateRight(AVLTree.AVLNode x)
```

commits a rotation to the right

**Parameters:**

```
x - the node needs a rotation to the right.
```

**כיצד פועלת:** מבצעת גלגול ימינה כשם שנלמד בכיתה. עשה עדכון heights, sizes של השותפים בגלגול. בעזרת הפונקציות setHeightsOnRotation, setSizesOnRotation
**סיבוכיות**: O(1) - שינוי מצביעים בלבד - זמן קבוע. עדכון הגבהים והגדלים גם באופן קבוע - מסתמך על הבנים.

- *rotateRightLeft*

```
private void rotateRightLeft(AVLTree.AVLNode node)
```

commits two rotations, first to the right and then to the left.

**Parameters:**

```
node - the node needs rotations to the right and then to the
left.
```

**כיצד פועלת:** תחילה מבצעת גלגול ימינה בעזרת הפונקציה rotateRightעם הבן הימני של הצומת הנתון.
לאחר מכן (מבנה העץ השתנה), מבצעת גלגול שמאלהעם הצומת הנתון. נעשה עדכון heights, sizes של השותפים בגלגול.
**סיבוכיות**: O(1) - ראה סיבוכיות rotateRight, rotateLeft

- *search*

```
public java.lang.String search(int k)
```

returns the info of an item with key k if it exists in the tree

**Parameters:**

```
k - key of item being searched.
```

**Returns:**

```
node with key k otherwise, returns null
```

**כיצד פועלת:** קוראת לפונק העזר getNodeByKey כדי למצוא את הצומת וב get.Value כדי לקבל את הערך.
**סיבוכיות**: O(logn) כסיבוכיות getNodeByKey - ראה פירוט בתיעוד שלה.

- *setHeightsOnRotation*

- ```
  private void setHeightsOnRotation(AVLTree.AVLNode wasParent,
                                    AVLTree.AVLNode newParent)
  ```

sets the heights of the participants nodes in a rotation.

**Parameters:**

```
wasParent - node which used to be the parent and now will be
newParent's child

newParent - node which used to be wasParent's child and now
will become its parent
```

**כיצד פועלת:** קובעת את גובהו של newParent להיות כגובהו של wasParent.
ואת גובהו של wasParent להיות כמקסימום מבין גבהי הבנים שלה + 1 (עבור הצומת עצמו).
**סיבוכיות**: O(1) - גישה ישירה לבנים ופעולות אריתמטיות.

- *setParentAndChildrenOnDelete_AndGetStartPoint_InCaseOfTwoChildren*

```
private AVLTree.AVLNode setParentAndChildrenOnDelete_AndGetStartP
oint_InCaseOfTwoChildren(AVLTree.AVLNode node)
```

sets the parent and children of a node with two children on delete
and returns the node which is the first to start fixing size and height from

**Parameters:**

```
node - node which has to be deleted. pre condition - the
node has two children
```

**Returns:**

```
node which is the first to start fixing size and height from
```
```

**כיצד פועלת:**

first finds the successor by getSuccessorInCaseOfTwoChildren(AVLNode node).

If the given node is successor's right child then the returned node is the successor itself.

otherwise, returns successor's parent.

uses exchangedOriginToSuccessor(AVLNode node, AVLNode successor) to change the successor place to be instead of the given node (phisically deletion of successor)

**סיבוכיות** O(logn) או חסם יותר הדוק .O(height(given_node)): יורדים במקרה הרע מס' רמות שהוא כגובה הצומת למציאת העוקב - ראה סיבוכיות .getSuccessorInCaseOfTwoChildren שאר הפעולות (שינוי פוינטרים) נעשה ב O(1).

- *setParentAndChildrenOnDelete_AndGetStartPoint*

```
public AVLTree.AVLNode setParentAndChildrenOnDelete_AndGetStartPo
int(AVLTree.AVLNode node)
```

sets the parent and children of a node on delete
and returns the node which is the first to start fixing size and height from

**Parameters:**

`node - the node which has to be deleted`

**Returns:**

`node which is the first to start fixing size and height from`

**כיצד פועלת:**

seperates between cases:

1. has to children - uses setParentAndChildrenOnDelete_AndGetStartPoint_InCaseOfTwoChildren(AVLNode node)

2. has only right child

3. has only left child

4. doesn't have any children

**סיבוכיות**: O(logn ) או חסם יותר הדוק O(height(given_node)). במקרה הגרוע יש לצומת שני בנים וצריך למצוא את העוקב שלו (מקרה 1) - יש לרדת לכל היותר מס' רמות ששווה לגובהו - ראה סיבוכיות setParentAndChildrenOnDelete_AndGetStartPoint_InCaseOfTwoChildren(AVLNode node) שינויי הפוינטרים שנעשים בכל המקרים הם ב O(1).

- *setRoot*

```
public void setRoot(AVLTree.AVLNode newRoot)
```

sets a new root to the tree.

**Parameters:**

`newRoot - the new root`

**סיבוכיות** -O(1)השמה.

- *setSizesOnRotation*

- `private void setSizesOnRotation(AVLTree.AVLNode wasParent, AVLTree.AVLNode newParent)`

  sets the sizes of the participants nodes in a rotation.

  **Parameters:**

  `wasParent - node which used to be the parent and now will be newParent's child`

  `newParent - node which used to be wasParent's child and now will become its parent`

  **כיצד פועלת:** קובעת את גודלו של newParent להיות כגודלו של wasParent.
  ואת גודלו של wasParent להיות כסכום גדלי הבנים שלה + 1 (עבור הצומת עצמו).
  **סיבוכיות**: O(1) - גישה ישירה לבנים ופעולות אריתמטיות.

- *size*

  `public int size()`

  **Returns:**

  `Returns the number of nodes in the tree. precondition: none postcondition: none`

  **כיצד פועלת:** קוראת לפונק העזר getSize כדי לקבל את הסייז של השורש.
  **סיבוכיות** -O(1): גישה לשדה.

- *updateHeight*

  `public void updateHeight(AVLTree.AVLNode node)`

  updates the field height of node- node.

  **Parameters:**

  `node - the node its height need update`

  **כיצד פועלת:** מקבלת צומת node ומעדכנת את הגובה שלה לפי הגדרה - סכום הגובה של הבנים +1. קוראת לפונק העזר getChildrenheights
  **סיבוכיות** -O(1): פעולות אריתמטיות וגישה ישירה לבנים.

- *updateSize*

  `public void updateSize(AVLTree.AVLNode node)`

  updates the field size of node- node.

  **Parameters:**

  `node - the node its size need update`

  **כיצד פועלת:** מקבלת צומת node ומעדכנת את הסייז שלה לפי הגדרה - סכום הסייז של הבנים +1. קוראת לפונק העזר getChildrenSizes
  **סיבוכיות** -O(1): פעולות אריתמטיות וגישה ישירה לבנים.

## Class TreeList

An implementation of a ADT List by Ranked AVL tree (not necessarily BST) which holds Items: (key-info).
Implements methods: retrieve(index), insert(index, key, val), delete(index) in O(logn)

- *Field Detail*

- *rankedAvlTree*

```
private AVLTree rankedAvlTree
```

AVLTree (not necessarily BST). Stores nodes each consists of an Item- (key, info).
Used for searching\inserting\deleting elements by their index (equivalent to rank - 1) in O(logn).
Only the field root will be used, nor maxNode nor minNode because they not improve complexity, and requires maintaining

- *Constructor Detail*

- *TreeList*

```
public TreeList()
```

Constructor of TreeList class.
building an avl tree for every instance of the class.

**סיבוכיות** - O(1): יצירת אובייקט avltree חדש והשמה.

- *Method Detail*

- *delete*

```
public int delete(int i)
```

deletes an item in the ith posittion from the list.

**Parameters:**

```
i - index of the item requested to be deleted
```

**Returns:**

```
returns -1 if i smaller then 0 or i greater then n-1,
otherwise returns 0.
```

**כיצד פועלת:**

first finds the requested node by getNodeByIndex(int index). If not found returns -1.

If found then:

1.1. sets its children and parent and gets the node to start fix the height and sizes from - by the method AVLTree.setParentAndChildrenOnDelete_AndGetStartPoint(AVLNode node)

1.2 fixes the sizes and heights (and does rotations) from the node gotten on 1.1 -

by the method AVLTree.fixHeightAndSizeOnDelete_AndGetNumOfRotations (AVLNode)

**סיבוכיות:** O(logn) או חסם יותר הדוק: O(Max{depth(node), height(node)}): נעשה חיפוש אחר הצומת בסיבוכיות הנ"ל - ראה תיעוד .getNodeByIndex(int index).

לאחר מכן, מוצאים את הצומת שיש למחוק פיזית - O(height(given_node)) - ראה תיעוד AVLTree.setParentAndChildrenOnDelete_AndGetStartPoint(AVLNode node)

מתבצעת עליה במסלול מן הצומת שנמחק פיזית אל השורש. אורך מסלול זה הוא O(depth(phisically_deleted)), כעומק הצומת. בעליה מתבצעים גלגולים ותיקוני גובה וגודל בזמן קבוע. - ראה סיבוכיות AVLTree.fixHeightAndSizeOnDelete_AndGetNumOfRotations (AVLNode)

במקרה הרע הצומת שנמחק הוא עלה / יש לצומת שני ילדים והעוקב שלו הוא עלה ואז סיבוכיות הפעולה היא O(logn)

- *getNodeByIndex*

```
private AVLTree.AVLNode getNodeByIndex(int i)
```

Get node which is in place (starts from 0) of the given index in the tree.

**Parameters:**

```
i - index of requested item in the list
```

**Returns:**

```
Item which is in the ith position if it exists in the Tree.
Otherwise, returns null
```

**כיצד פועלת:** מקבלת אינדקס i ומחזירה את הצומת בעלת ה- rank i+1, בעזרת פונק העזר select.

**סיבוכיות** O(logn) :כסיבוכיות פונק העזר - select ראה פירוט בתיעוד שלה.

- *insert*

```
public int insert(int i,
                  int k,
                  java.lang.String s)
```

inserts an item to the ith position in list with key k and info s.

**Parameters:**

```
i - index to be filled with the item inserted

k - the key of the item inserted

s - the value of the item inserted
```

**Returns:**

```
-1 if i smaller then 0 or i greater then n, otherwise
returns 0
```

**כיצד פועלת:** מקבלת מפתח k, ערך s ואינדקס i בו היא נדרשת להכניס איבר חדש לרשימה. ראשית, בודקת אם הקלט חוקי, אם כן מחזירה בסוף הריצה 0, אחרת 1-. אם חוקי, יוצרת צומת חדש ובודקת לאן להכניס אותו.
אם האינדקס המבוקש i שווה למספר הצמתים בעץ, תכניס אותו בצומת ימנית ביותר בעזרת פונק' העזר insertLast.
אם i קטן ממספר הצמתים בעץ, הפונק' תחפש את הצומת g העכשווית באינדקס i עם פונק' העזר getNodeByIndex. לאחר מכן, נבדוק עם פונק' העזר getLeft אם אין לו בן שמאלי.
אם אין - נכניס את הצומת החדש כבנו השמאלי ונגדיר את אביו g עם פונק' העזר setParent.
אחרת - נמצא את הקודם של g עם פונק העזר AVLTree.getMaxInSubTree(AVLNode) שתופעל על תת העץ הימני של g, ונגדיר אותו כאבא של הצומת החדש.
לבסוף נבצע פעולות איזון עם פונק עזר updateSize, updateHeight שמעדכנות גובה וגודל, ו-

checkAndComitRotation שמבצעת גלגולים, אם יש צורך.

**סיבוכיות** O(logn): העץ הוא עץ avl, ולכן כדי לבצע הכנסה, אנו הולכים על מסלול אחד בעץ מהשורש לעלה (הצומת נכנס כעלה) שאורכו O(logn), מכניסים, ואז מבצעים פעולות איזון בזמן קבוע- שינוי מצביעים, על אותו מסלול.

- *insertLast*

```
private void insertLast(AVLTree.AVLNode newNode)
```

inserts a node to be with the hightest (most right) rank in the Tree.

**Parameters:**

```
newNode - node to be inserted most right of the tree.
```

**כיצד פועלת:** מקבלת צומת k ומכניסה אותה להיות הצומת הימנית ביותר בעץ (בעלת ה rank הגבוה ביותר). הפונק' קוראת למתודות העזר getRoot, getMaxInSubTree(AVLNode) על מנת למצוא את הצומת הימנית ביותר העכשווית, אם קיימת, נכניס את k להיות בנה הימני בעזרת מתודות העזר setRight,setParent, אחרת נכניס את k להיות השורש בעזרת מתודת העזר setRoot. (בחרנו לא להשתמש במצביע של rankedAvlTree לאיבר המינימלי ע"מ שלא נצטרך לתחזק אותו).
**סיבוכיות** O(logn): משום שהיא עוברת על מסלול אחד- הימני ביותר בעץ באורך עד O(logn), שאר הפעולות בזמן קבוע.

- *isIndexValidInsert*

```
private boolean isIndexValidInsert(int i)
```

checks if index i is valid for insert operation.

**Parameters:**

```
i - index to be checked if valid.
```

**Returns:**

```
true - if valid, else (smaller the 0 or greater then
rankedAvlTree size)- false
```

**סיבוכיות**: O(1).

- *isIndexValidRetrieveAndDelete*

```
private boolean isIndexValidRetrieveAndDelete(int i)
```

checks if index i is valid for retrieve and delete operations.

**Parameters:**

```
i - index to be checked if valid.
```

**Returns:**

```
true - if valid, else (smaller the 0 or greater or equals to
rankedAvlTree size)- false
```

**סיבוכיות**: O(1)

- *retrieve*

```
public Item retrieve(int i)
```

Get Item from the list which is in place (starts from 0) of the given index.

**Parameters:**

```
i - index of requested item in the list
```

**Returns:**

```
Item which is in the ith position if it exists in the list.
Otherwise, returns null
```

**כיצד פועלת:** ראשית, הפונק' בודקת שהקלט חוקי, אם לא מחזירה null. אם כן, מוצאת את הצומת באינקס זה בעץ בעזרת פונק' העזר getNodeByIndex ואז מחזירה את האיבר המבוקש בעזרת פונק העזר getItem.

**סיבוכיות** O(logn) :כסיבוכיות פונק העזר - getNodeByIndex ראה פירוט בתיעוד שלה.

- *select*

```
private AVLTree.AVLNode select(int i)
```

returns the node in position i (Rank) (starts from 1) in the Tree.
uses recursion by the inner method - selectRec().

**Parameters:**

```
i - index of requested node in the Tree
```

**Returns:**

```
node which is in the ith position if it exists in the Tree.
Otherwise, returns null
```

**כיצד פועלת:** הפונק' היא פונקציית מעטפת אשר קוראת לפונק' העזר הפנימית selectRec. כמו כן היא קוראת גם לפונק' העזר getRoot כדי לקבל את השורש של העץ ולשלחו לפונק' הפנימית.

**סיבוכיות** O(logn) :משום שהיא עוברת על מסלול אחד בעץ באורך O(logn)כפי שראינו בהרצאה , והשאר פעולות השוואה קבועות.

- *selectRec*

- ```
  private AVLTree.AVLNode selectRec(AVLTree.AVLNode node,
                                    int i)
  ```

returns the node in position i (Rank) (starts from 1) in the Tree that node its root.
uses recursion, inner method of select()

**Parameters:**

```
i - index of requested node in the Tree that node its root.

node - - the tree's root.
```

**Returns:**

```
node which is in the ith position if it exists in the Tree.
Otherwise, returns null
```

**כיצד פועלת:** פונקציה הפנימית של הפונק select, מקבלת צומת k ואינדקס i, ומחפשת את הצומת בעלת rank(i) בתת העץ ש- k הוא שורשו. לפירוט ראה- select .

**סיבוכיות**O(logn) :כאשר n הוא מספר הצמתים בתת העץ ששורשו k, משום שהיא עוברת על מסלול אחד בעץ באורך O(logn)כפי שראינו בהרצאה , והשאר פעולות השוואה קבועות.
```

## Class CircularList

An implementation of a ADT List by a Circular List which holds Items: (key-info) in a circular array (defined as an array in size maxLen, with a pointer to the start, and len which is the actual size of the array).
Implement methods: retrieve(index), insert(index, key, val), delete(index)

- *Field Detail*

- *arr*

```
private Item[] arr
```

will hold Items. The empty cells were point to null

- *len*

```
private int len
```

the actual size of the array - how many items are there. Will be given to us.

- *maxLen*

```
private int maxLen
```

the size of the array - updated in insert and delete

- *start*

```
private int start
```

points to the index that is the 'start point reading' from the array. can be changed in insert (in example if we insert an element as first the start will move back)

- *Constructor Detail*

- *CircularList*

```
public CircularList(int maxLen)
```

Constructor of CircularList class.
for every instance of the class building an array arr.
fields: maxLen- array's length, arr - java's array,
len - num of items in the list, start - index of the first item.

**Parameters:**

```
maxLen - the length of the array - max number of items can
be added to the list
```

**סיבוכיות: O(1) – השמה.**

- *Method Detail*

- *copyElementsInBackMoveDelete*

```
private void copyElementsInBackMoveDelete(int index)
```

help function for delete. moves all items after the index not include, one step back in the array arr.

**Parameters:**

```
index - index of the item that its all forward items needs
to move back one step.
```

**כיצד פועלת:** מקבלת אינדקס Index ומסיתה צעד אחד אחורה במערך arr את כל האיברים שאחריו לא כולל, על מנת לסגור מקום של האיבר הנמחק באינדקס זה.
הפונקצייה קוראת לפונק' העזר getCountAfterIndex, getPosition על מנת לבדוק כמה איברים יש לפני האינדקס לא כולל, ואז מבצעת את ההסתה אחורה.
**סיבוכיות**: O(n) במקרה הגרוע - כהסתה של חצי איברים של מערך (n מספר האיברים במערך).

- *copyElementsInBackMoveInsert*

```
private void copyElementsInBackMoveInsert(int index)
```

help function for insert. moves all items before the index and include one step back in the array arr.

**Parameters:**

```
index - index of the item that its and his all previous
items needs to move back one step.
```

**כיצד פועלת:** מקבלת אינדקס Index ומסיתה צעד אחד אחורה במערך arr את כל האיברים שלפניו כולל, על מנת לפנות מקום להכנסה של איבר באינדקס זה.
הפונקצייה קוראת לפונק' העזר getCountBeforeIndexAndInclude על מנת לבדוק כמה איברים יש לפני האינדקס כולל, ואז מבצעת את ההסתה אחורה, ועדכון ה start.
**סיבוכיות** (O(n :במקרה הגרוע - כהסתה של חצי איברים של מערך n) מספר האיברים במערך(

- *copyElementsInForwardMoveDelete*

```
private void copyElementsInForwardMoveDelete(int index)
```

help function for delete. moves all items before the index not include, one step forward in the array arr.

**Parameters:**

```
index - index of the item that its all previous items needs
to move forward one step.
```

**כיצד פועלת:** מקבלת אינדקס Index ומסיתה צעד אחד קדימה במערך arr את כל האיברים שלפניו לא כולל, על מנת לסגור מקום של האיבר הנמחק באינדקס זה.
הפונקצייה קוראת לפונק' העזר getCountBeforeIndex, getPosition על מנת לבדוק כמה איברים יש אחרי האינדקס לא כולל, ואז מבצעת את ההסתה קדימה.
**סיבוכיות** (O(n :במקרה הגרוע - כהסתה של חצי איברים של מערך n) מספר האיברים במערך.(

- *copyElementsInForwardMoveInsert*

```
private void copyElementsInForwardMoveInsert(int index)
```

help function for insert. moves all items after the index and include one step forward in the array arr.

**Parameters:**

```
index - index of the item that its and his all forward items
needs to move forward one step.
```

**כיצד פועלת:** מקבלת אינדקס Index ומסיתה צעד אחד קדימה במערך arr את כל האיברים שאחריו כולל, על מנת לפנות מקום להכנסה של איבר באינדקס זה.
הפונקצייה קוראת לפונק' העזר getCountAfterIndexAndInclude, getPosition על מנת לבדוק כמה איברים יש אחרי האינדקס כולל, ואז מבצעת את ההסתה קדימה.

**סיבוכיות**: O(n) במקרה הגרוע - כהסתה של חצי איברים של מערך (n מספר האיברים במערך).

- *delete*

```
public int delete(int i)
```

deletes an item in the ith posittion from the list.

**Parameters:**

```
i - index of the item requested to be deleted
```

**Returns:**

```
returns -1 if ismaller then 0 or i greater then n-1,
otherwise returns 0.
```

**כיצד פועלת:** ראשית קוראת לפונק' העזר validateRetrieveAndDelete על מנת לבדוק אם הקלט חוקי, אם כן בודקת אם באינדקס שהוכנס נמצא האיבר הראשון ברשימה, אם כן קוראת לפונק' העזר deleteFirst על מנת למחוק אותו, באותו אופן לגבי האיבר האחרון קוראת לפונק העזר deleteLast. אחרת, בודקת משיקולי סיבוכיות ע"י קריאה לפונק העזר deleteIsBetterToMoveBack אם כדאי לבצע הזזה אחורה של האיברים שאחרי האינדקס לא כולל ברשימה, אם כן מבצעת את ההזזה בעזרת פונק העזר copyElementsInBackMoveDelete ומעדכנת את האיבר האחרון שהיה ל null - מוצאת אותו בעזרת פונק' העזר getPositionOfLastItem. אם כדאי לבצע ההזה קדימה באופן דומה קוראת לפונק copyElementsInForwardMoveDelete, ומעדכנת שדה start.
לבסוף מעדכנת את השדה len, ובודקת אם הרשימה ריקה, אם כן מחזירה את אינדקס ההתחלה- start להיות 0.
**סיבוכיות**: O(min{i+1,n-i+1}) כאשר n=len, משום שבמקרה הגרוע נבצע הזזה של איברים במערך arr קדימה או אחורה, המינימלי מביניהם.

- *deleteFirst*

```
private void deleteFirst()
```

deletes the first item in the list

**כיצד פועלת:** מעדכנת שדות start ו len.
**סיבוכיות** - O(1)פעולות אריתמטיות והשמות.

- *deleteIsBetterToMoveBack*

```
private boolean deleteIsBetterToMoveBack(int index)
```

returns true if number of elements after the given index (i) is smaller than number of elements after (and include) i if they are equal return false (better to move forward)

**Parameters:**

```
index - index of the item requested to be deleted.
```

**Returns:**

```
true if its better to move the items after the index back
for deletion complexity. otherwise returns false
```

**כיצד פועלת:** בודקת משיקולי סיבוכיות האם כדאי להסית את האיברים שלפניו קדימה, או להסית את אלה שאחריו אחורה. הפונקצייה קוראת לפונקציית העזר getCountBeforeIndex כדי לדעת כמה איברים יש לפני האינדקס לא כולל, ולפונקציית העזר getCountAfterIndex כדי לדעת כמה איברים יש אחרי האינדקס לא כולל.

**סיבוכיות**: סיבוכיות - O(1)כסיבוכיות - getCountBeforeIndex, getCountAfterIndex ראה פירוט בתיעוד שלהן.

- *deleteLast*

```
private void deleteLast()
```

deletes the last item in the list

**כיצד פועלת:** קוראת לפונק' העזר getPosition על מנת למצוא איבר זה במערך arr.
**סיבוכיות** - O(1)פעולות אריתמטיות והשמות.

- *getCountAfterIndex*

```
private int getCountAfterIndex(int index)
```

returns the number of items after index-index in the list not include.

**Parameters:**

```
index - index of the requested number of items after him not
include.
```

**Returns:**

```
number of items after the index not include.
```

**כיצד פועלת:** קוראת לפונק העזר getCountBeforeIndex כדי לחשב את מספר האיברים שאחרי האינדקס כולל.
**סיבוכיות** O(1) - פעולות אריתמטיות + סיבוכיות פונק העזר ראה פירוט בתיעוד שלה.

- *getCountAfterIndexAndInclude*

```
private int getCountAfterIndexAndInclude(int index)
```

returns the number of items after index-index in the list and include.

**Parameters:**

```
index - index of the requested number of items after him and
include.
```

**Returns:**

```
number of items after the index and include.
```

**כיצד פועלת:** קוראת לפונק העזר getCountBeforeIndexAndInclude כדי לחשב את מספר האיברים שאחרי האינדקס כולל.

**סיבוכיות** - O(1)פעולות אריתמטיות + סיבוכיות פונק העזר ראה פירוט בתיעוד שלה.

- *getCountBeforeIndex*

```
private int getCountBeforeIndex(int index)
```

returns the number of items before index-index in the list not include.

**Parameters:**

```
index - index of the requested number of items before him
not include.
```

**Returns:**

```
number of items before the index not include.
```

**כיצד פועלת:** מחזירה את האינדקס אותו קיבלה (כי לפני האינדקס הi יש i איברים)

**סיבוכיות** - O(1)פעולת החזרה.

- *getCountBeforeIndexAndInclude*

```
private int getCountBeforeIndexAndInclude(int index)
```

returns the number of items before index-index in the list and include.

**Parameters:**

```
index - index of the requested number of items before him
and include.
```

**Returns:**

```
number of items before the index and include.
```

**כיצד פועלת:** מחזירה את האינדקס אותו קיבלה + 1

**סיבוכיות** - O(1)פעולת החזר + פעולות אריתמטיות.

- *getPosition*

```
private int getPosition(int index)
```

returns the position of the item in the array arr of the index- index of the list.

**Parameters:**

```
index - index of the list of the requested position in the
array arr.
```

**Returns:**

```
position of the item in the array arr of the index- index of
the list.
```

**סיבוכיות** - O(1)פעולות אריתמטיות (מודולו על maxLen)

- *getPositionOfLastItem*

```
private int getPositionOfLastItem()
```

returns the index of last item in the list

**Returns:**

```
index of last item in the list
```

**כיצד פועלת:** קוראת לפונק' העזר getPosition על מנת למצוא איבר זה במערך arr.
**סיבוכיות** O(1) - פעולות אריתמטיות + סיבוכיות פונק העזר ראה פירוט בתיעוד שלה.

- *insert*

- `public int insert(int i, int k, String s)`

inserts an item to the ith position in list with key k and info s.

**Parameters:**

```
i - index to be filled with the item inserted

k - the key of the item inserted

s - the value of the item inserted
```

**Returns:**

```
-1 if i smaller then 0 or greater the n or n=maxLen,
otherwise returns 0
```

**כיצד פועלת:** אחר בדיקת חוקיות הקלט הפונק' יוצרת איבר חדש של המחלקה Item עם מפתח k וערך s, ובודקת אם האינדקס i הוא 0, ואז נכניס את האיבר להיות האיבר הראשון ברשימה בעזרת פונק העזר insertAsFirst, לעומת זאת אם i שווה לאורך הרשימה נכניסו להיות האיבר האחרון בעזרת פונק העזר insertAsLast. אחרת, בודקת משיקולי סיבוכיות ע"י קריאה לפונק העזר IsBetterToMoveBack אם כדאי לבצע הזזה אחורה של האיברים שלפני האינדקס כולל ברשימה, אם כן מבצעת את ההזזה בעזרת פונק העזר copyElementsInBackMoveInsert ומעדכנת את התא באינדקס i ברשימה להכיל את ה items שהתבקש ע"י פונק העזר updateElementInIndex, לאחר מכן מעדכנת את השדה start עם פונק העזר updateStartInMovingBack. אחרת, אם כדאי לעשות הזזה קדימה, עושה זאת באופן סימטרי בעזרת פונק העזר copyElementsInForwardMoveInsert, updateElementInIndex. בסוף מעדכנת את len.
**סיבוכיות**: O(min{i+1,n-i+1}) כאשר n=len, משום שבמקרה הגרוע נבצע הזזה של איברים במערך arr קדימה או אחורה, המינימלי מביניהם.

- *insertAsFirst*

```
private void insertAsFirst(Item item)
```

inserts an item with key and info to be first item in the list.

**Parameters:**

```
item - item to be inserted as first in the list.
```

**כיצד פועלת:** משתמשת בפונק' העזר updateStartInMovingBack על מנת לעדכן את שדה ה start של המופע.
**סיבוכיות** - O(1)בדיקת השוואה, השמה ועדכון.start

- *insertAsLast*

```
private void insertAsLast(Item item)
```

inserts an item with key and val to be last item in the list.

**Parameters:**

```
item - item to be inserted as last in the list.
```

**כיצד פועלת:** משתמשת בפונק' העזר updateElementInIndex על מנת להכניסו לאינדקס האחרון על ידי חישובו.
**סיבוכיות** O(1) - כסיבוכיות updateElementInIndex - ראה פירוט בתיעוד שלה .

- *isBetterToMoveBackInsert*

```
private boolean isBetterToMoveBackInsert(int index)
```

returns true if number of elements until (and include) the given index (i) is smaller than number of elements after (and include) i if they are equal return false (better to move forward)

**Parameters:**

```
index - index of the item requested to be inserted.
```

**Returns:**

```
true if its better to move the items before the index back
for insertion complexity. otherwise returns false
```

**כיצד פועלת:** מקבלת אינדקס index בו התבקש להכניס איבר חדש, ובודקת משיקולי סיבוכיות האם כדאי להסית את האיברים שלפניו אחורה, או להסית את אלה שאחריו קדימה. הפונקצייה קוראת לפונקציית העזר getCountBeforeIndexAndInclude כדי לדעת כמה איברים יש לפני האינדקס כולל, ולפונקציית העזר getCountAfterIndexAndInclude כדי לדעת כמה איברים יש אחרי האינדקס כולל.

**סיבוכיות**: - O(1) כסיבוכיות - getCountBeforeIndexAndInclude, getCountAfterIndexAndInclude ראה פירוט בתיעוד שלהן.

- *retrieve*

```
public Item retrieve(int i)
```

Get Item from the list which is in place of the given index.

**Parameters:**

```
i - index of requested item in the list
```

**Returns:**

```
item in the ith position if it exists in the list.
otherwise, returns null
```

**כיצד פועלת:** ראשית, הפונק' בודקת שהקלט חוקי בעזרת פונק' העזר validateRetrieveAndDelete, אם לא מחזירה null. לאחר מכן מחשבת לאיזה אינדקס במערך תואם האינדקס i, ומחזירה את האיבר באינדקס זה.

**סיבוכיות** -O(1): גישה לשדות, פעולות אריתמטיות קבועות, וגישה לאינדקס במערך.

- *updateElementInIndex*

- ```
  private void updateElementInIndex(int index,
                                    Item item)
  ```

  updates the list of index-index to fill the Item-item.

  **Parameters:**

  ```
  index - index of the list to be updated
  ```

  ```
  item- - key and value to be filled in the list.
  ```

**כיצד פועלת:** מקבלת אינדקס index ואיבר item עם מפתח וערך, ומעדכנת את התא באינדקס זה ברשימה להכיל את האיבר הזה. קוראת לפונק' העזר getPosition על מנת למצוא איבר זה במערך arr.
**סיבוכיות** O(1) - פעולות אריתמטיות + סיבוכיות פונק העזר ראה פירוט בתיעוד שלה.

- *updateStartInMovingBack*

```
private void updateStartInMovingBack()
```

updates start field after moving items one step back in the array arr.

**סיבוכיות** - O(1)גישה לשדה ופעולות אריתמטיות.

- *validateInsert*

```
private boolean validateInsert(int index)
```

check if the index of the item requested to be inserted to the list is valid.

**Parameters:**

```
index - index of requested item to be inserted to the list
```

**Returns:**

```
true if index valid – not smaller then 0 and not greater
than len, else false
```

**סיבוכיות** - O(1)גישה לשדה ופעולות אריתמטיות.

- *validateRetrieveAndDelete*

```
private boolean validateRetrieveAndDelete(int index)
```

check if the index of the item requested to be retrieved or deleted is valid.

**Parameters:**

```
index - index of requested item to be retrieved or deleted.
```

**Returns:**

```
true if index valid - not smaller then 0 and not greater
than len - 1, else false
```

**סיבוכיות** - O(1)גישה לשדה ופעולות אריתמטיות.
```

# מדידות

הערות כלליות:
1. הזמנים הממוצעים הם ב nano sec.
2. n-מספר האיברים ברשימה ברגע נתון, אלא אם נאמר אחרת.
3. בוצע מיצוע על פני 1000 ריצות, מלבד בניסויים 2,3 בבדיקת ההכנסה לרשימה מעגלית בה בוצע מיצוע על פני 50 הכנסות בגלל הזמן הרב שנדרש לתוכנית (היא נתקעה לנו כאשר ניסינו להריץ עם 100 הרצות)

## ניסוי 1:

| מספר סידורי | מספר פעולות | זמן הכנסה ממוצע עבור רשימה מעגלית | זמן הכנסה ממוצע עבור רשימה עצית | כמות גלגולים ימינה ממוצעת עבור רשימה עצית | כמות גלגולים שמאלה ממוצעת עבור רשימה עצית |
|---|---|---|---|---|---|
| **1** | 10,000 | 38 | 128 | 0 | 0.9986 |
| **2** | 20,000 | 25 | 132 | 0 | 0.99925 |
| **3** | 30,000 | 22 | 136 | 0 | 0.9995 |
| **4** | 40,000 | 13 | 153 | 0 | 0.9996 |
| **5** | 50,000 | 10 | 162 | 0 | 0.99968 |
| **6** | 60,000 | 9 | 168 | 0 | 0.99973 |
| **7** | 70,000 | 12 | 168 | 0 | 0.999757 |
| **8** | 80,000 | 8 | 182 | 0 | 0.99978 |
| **9** | 90,000 | 10 | 181 | 0 | 0.99981 |
| **10** | 100,000 | 13 | 199 | 0 | 0.99983 |

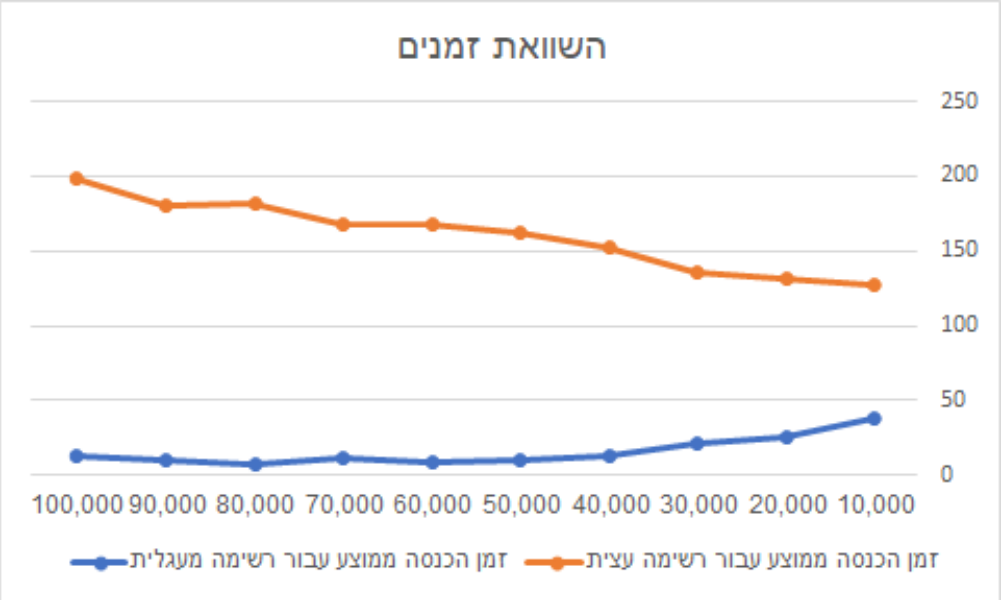
**מיקומי הכנסה:** על מנת להדגים את היתרון של רשימה מעגלית, בחרנו להכניס איברים רק לסוף הרשימה כל פעם.
זאת משום שסיבוכיות פעולת ההכנסה לסוף / התחלה של איבר בודד ברשימה מעגלית היא O(1), שכן היא כרוכה רק במספר קבוע של פעולות - הכנסה לשדה ועדכון אינדקס (לא מתבצעת הזזה של איברים ברשימה). כמו כן, הכנסה בסוף עדיפה במעט על הכנסה בהתחלה, מפאת מספר פעולות נמוך יותר בקבועים (שינוי המצביע start).
לעומת זאת, סיבוכיות הכנסה בודדת ברשימה עצית לכל מקום שהוא, ובפרט כאיבר האחרון, היא O(logn)(ראינו בתרגול שסיבוכיות כלל ההכנסות הוא O(nlogn) ולכן בממוצע להכנסה יחידה מדובר ב(O(logn).

**ציפיות**: ציפינו שזמן הריצה הממוצע של הכנסת איבר ברשימה מעגלית, על קלטים הולכים וגדלים לא יהיה תלוי בגודל הקלט, ושזמן הריצה הממוצע של הכנסת איבר ברשימה עצית, על קלטים הולכים וגדלים יגדל לוגריתמית. בנוסף, סברנו שיהיה פער הולך וגדל עם הכנסת קלטים גדולים יותר בין זמני הריצה, לטובת הרשימה המעגלית.
בנוסף, הכנסה לתחילת / סוף הרשימה תגרור מספר גלגולים מקסימלי- O(n)גלגולים סה"כ ((O(n גלגולים, משום שכמעט בכל הכנסה נוצר עבריין avl ודורש גלגול. זה שקול למספר הגלגולים הדרוש

על מנת להפוך עץ שרוך עם n צמתים לעץ מאוזן, כפי שהוכחנו באינדוקציה בתרגיל בית 2 שאלה 2.ג).
על כן, נצפה שכמות הגלגולים הממוצע שמאלה ברשימה העצית יהיה גבוה מאוד מהסיבה שתיארנו, ושלא יהיו כלל גלגולים ימינה (על פי סימולציות שעשינו).

**תוצאות ומסקנות:** ההשערה שלנו אוששה. ניתן לראות שהניתוח התיאורטי שלנו על הסיבוכיות האסימפטוטית, התממש בפועל במדידת זמני הריצה. זמן הריצה של הרשימה המעגלית עבור כל הקלטים נשאר די קבוע, כלומר לא תלוי בגודל הקלט, וכן ברשימה העצית ניתן לראות קו מגמה עולה באופן מתון, שכן פונקציה לוגריתמית גדלה לאט במספרים גדולים.
בנוסף, קיים פער גדול בין זמני הריצה של שתי הרשימות, לטובת הרשימה המעגלית, כאשר הפער מוסיף לגדול עם הגדלת הקלטים- פי 3.36 עבור 10,000 פעולות ופי 15.3 עבור 100,000 פעולות.
ומכאן המסקנה, שעבור דרישות הניסוי ובחירת מיקומי ההכנסה, יתרון לא מבוטל לרשימה מעגלית.
כמו כן, ההשערה לגבי כמות הגלגולים אוששה. נדרש בממוצע כמעט לכל איבר ברשימה העצית לעבור גלגול, מהסיבה שציינו.

**להלן תוצאות ניסוי 1 בגרפים:**



## ניסוי 2:

| מספר סידורי | מספר פעולות | זמן הכנסה ממוצע עבור רשימה מעגלית | זמן הכנסה ממוצע עבור רשימה עצית | כמות גלגולים ימינה ממוצעת עבור רשימה עצית | כמות גלגולים שמאלה ממוצעת עבור רשימה עצית |
|---|---|---|---|---|---|
| **1** | 10,000 | 14045 | 284 | 0.8103 | 0.8118 |
| **2** | 20,000 | 32593 | 265 | 0.8115 | 0.81195 |
| **3** | 30,000 | 42122 | 249 | 0.81163333 | 0.81226667 |
| **4** | 40,000 | 54362 | 231 | 0.8117 | 0.8125 |
| **5** | 50,000 | 69497 | 224 | 0.81202 | 0.81232 |
| **6** | 60,000 | 73583 | 246 | 0.8120833334 | 0.81233334 |
| **7** | 70,000 | 82556 | 246 | 0.8121285714285714 | 0.8123857142857143 |
| **8** | 80,000 | 92775 | 271 | 0.8121875 | 0.8123875 |
| **9** | 90,000 | 105681 | 238 | 0.8121777777777778 | 0.81242222222223 |
| **10** | 100,000 | 128359 | 233 | 0.81221 | 0.81241 |

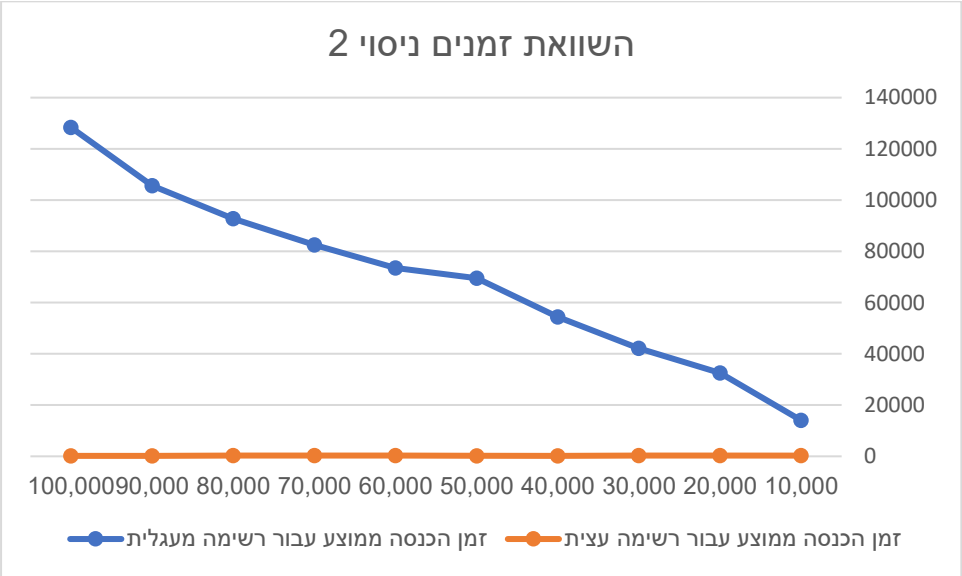
**מיקומי הכנסה:** הכנסנו איבר בכל פעם לאמצע הרשימה (מעוגל למטה), מכיוון שאז ברשימה המעגלית תתבצע הסתה של חצי מהאיברים שמאלה. במקרה כזה סיבוכיות ההכנסה הכוללת של כל n האיברים ברשימה המעגלית תהיה $O(n^2)$ כי לכל איבר נעשית הזזה של חצי מהאיברים (ערך תחתון של זה), כלומר סך ההזזות יהיה $\sum_{i=0}^{n}(\frac{i}{2})$ שזה לפי סכום סדרה חשבונית $O(n^2)$ ולכן בממוצע להכנסה בודדת של איבר יחיד O(n).
כמו כן, הכנסה לערך התחתון של הגודל תגרור פעולות בקבועים מקסימלי- עדכון start (לעומת עיגול מעלה, בה תתבצע הסתה של חצי מהאיברים ימינה ולא יעודכן שדה ה start).
זאת לעומת הציפיה לקבל סיבוכיות לוגריתמית במספר הצמתים ברשימה עצית, שכן כפי שהסברנו בניסוי הקודם הכנסה בודדת לכל מקום שהוא עולה O(logn).

**ציפיות**: ציפינו שזמן הריצה של רשימה עצית על קלטים הולכים וגדלים יגדל לוגריתמית, וזה של רשימה מעגלית יגדל באופן לינארי. בנוסף, סברנו שיהיה פער משמעותי הולך וגדל עם הכנסת קלטים גדולים יותר בין זמני הריצה, לטובת הרשימה העצית.
כמו כן, נצפה שמס' הגלגולים יהיה יחסית מאוזן בין ימין לשמאל וזאת בגלל שההכנסה באמצע גוררת לפעמים שצד ימין יהיה עמוס יותר ולפעמים שצד שמאל יהיה עמוס יותר. נצפה גם לקבל יחסית הרבה גלגולים (לאחר סימולציה שערכנו).

**תוצאות ומסקנות**: ההשערה שלנו אוששה ברובה. ניתן לראות שהניתוח התיאורטי שלנו על הסיבוכיות האסימפטוטית, התממש ברובו בפועל במדידת זמני הריצה.
ברשימה המעגלית ניתן לראות קו מגמה עולה לינארית באופן ברור.
ניתן לראות שקיים פער גדול בין זמני הריצה לטובת הרשימה העצית, כאשר הפער מוסיף לגדול עם הגדלת הקלטים- פי 85.1 עבור 10,000 פעולות ופי 550 עבור 100,000 פעולות. ומכאן המסקנה, שעבור דרישות הניסוי ובחירת מיקומי ההכנסה, אין ספק שיתרון משמעותי לרשימה עצית, בעקבות ההבדל בין פונ' לינארית ללוגריתמית.

לא ניתן לראות קו מגמה ברור בזמן הריצה של הרשימה העצית עבור הקלטים שבדקנו, אנו סבורים שהדבר נובע מהעובדה שעבור הקלטים הללו, זמן הריצה קטן מאוד, עד שקשה להבחין בקו מגמה, וזאת עקב שיקולים אחרים שמשפיעים על זמן הריצה, כמו חומרת המחשב ומערכת ההפעלה. לכן, נמליץ לבצע בדיקה זו על קלטים גדולים יותר. בנוסף, הגידול אמור להיות לוגריתמי, ובמס' גבוהים מאוד ההבדל אינו ניכר כי פונקציית הלוגריתם גדלה לאט. כמו כן, אין לנו הסבר מדוע החל מ90,000 חלה ירידה בזמן הריצה (בוצע מיצוע על פני 1000 ריצות!)

באשר למס' הגלגולים – ניתן לראות שהוא מאוזן בין ימין לשמאל, כנראה מהסיבה שציינו, ושאכן בוצעו יחסית הרבה גלגולים לכל צד. אין לנו הסבר מספר הגלגולים בכל ריצה הוא בערך 0.81 לכל צד ובסהכ 1.62.



## ניסוי 3:

| מספר סידורי | מספר פעולות | זמן הכנסה ממוצע עבור רשימה מעגלית | זמן הכנסה ממוצע עבור רשימה עצית | כמות גלגולים ימינה ממוצעת עבור רשימה עצית | כמות גלגולים שמאלה ממוצעת עבור רשימה עצית |
|---|---|---|---|---|---|
| **1** | 10,000 | 12000 | 450 | 0.3454 | 0.3441 |
| **2** | 20,000 | 19999 | 456 | 0.3453 | 0.3438 |
| **3** | 30,000 | 31024 | 452 | 0.35076666666666667 | 0.3526 |
| **4** | 40,000 | 40338 | 436 | 0.351975 | 0.35055 |
| **5** | 50,000 | 49785 | 443 | 0.34522 | 0.34332 |
| **6** | 60,000 | 59139 | 441 | 0.34815 | 0.348 |
| **7** | 70,000 | 69301 | 472 | 0.3479 | 0.3483 |
| **8** | 80,000 | 80029 | 539 | 0.3489375 | 0.34705 |
| **9** | 90,000 | 91222 | 502 | 0.34767777777777775 | 0.3486 |
| **10** | 100,000 | 97461 | 547 | 0.35032 | 0.35056 |

**ציפיות**: זמן ההכנסה הממוצע לכל איבר ברשימה עצית יגדל לוגריתמית כאשר גודל הקלט יגדל, שכן הכנסה בודדת לכל מיקום שהוא בעץ AVL (שמייצג את הרשימה העצית) הוא ב- O(logn), כאשר מיקומי הכנסה שונים משפיעים רק מבחינת הקבועים ולא מבחינה אסימפטוטית.
לעומת זאת, נצפה שזמן ההכנסה הממוצע לכל איבר ברשימה המעגלית יגדל לינארית כאשר גודל הקלט יגדל, שכן מירב הסיכויים שהרוב הכנסות לא יבוצעו לתחילת הרשימה או לסופה - ורק בהכנסות אלה ההכנסה היא בזמן שאינו תלוי בגודל הקלט, כלומר O(1). בהכנסות שאינן בהתחלה/לסוף יש תלות בגודל הקלט - המינימום בין i+1 ל n-i כאשר i הוא מקום ההכנסה, וחסם עליון על זה, הוא O(n) (ניתן גם להסביר לפי ההסבר שנתנו בניסוי מספר 2 לפי סיבוכיות ההכנסה הכוללת שהיא $O(n^2)$)
לכן נצפה שזמן הריצה הממוצע של רשימה מעגלית יהיה גדול בהרבה משל הרשימה העצית, שכן פונקציה לינארית גדלה הרבה יותר מהר מלוגריתמית.
כמו כן לגבי מס' הגלגולים הממוצע שמאלה וימינה - נצפה שיהיה פחות או יותר שווה בין גדלי הקלט השונים, ובתוך כל קלט. הסיבה: נצפה שגודל הקלט הגדול והאקראיות יגררו "התיישרות הסטטיסטיקה" כלומר למזעור הבדלים בין גלגולים שמאלה וימינה.

**תוצאות ומסקנות**: השערתנו אוששה ברובה. ניתן לראות גידול לינארי בזמן ההכנסה הממוצע ברשימה מעגלית. כלומר יש להניח ש O(n) הוא חסם יחסית הדוק עבור הכנסות שאינן בהתחלה/בסוף (שכאמור הכנסה בהתחלה/בסוף נדירות מאוד בהתפלגות אחידה).
ניתן לראות הבדלים עצומים בין זמן הריצה ברשימה מעגלית לרשימה העצית לטובת הרשימה העצית (למשל עבור n=100000 פי 180), וזה קשור לגידול לוגריתמי מול לינארי.
באשר למגמת זמן הריצה ברשימה העצית כתלות בגודל הקלט לא ניתן לראות מגמה ברורה כנראה בגלל הזמן הקצר שנדרש לפעולה, וכן בעקבות הגידול האיטי של פונ' לוגריתם במס' גדולים.
השערתנו לגבי מס' הגלגולים אוששה - מס' הגלגולים ימינה/ שמאלה עבור כל n פחות או יותר זהה, וגם בין הח-ים השונים. הסיבה היא כנראה האקראיות וגודל הקלט - ככל שהמס' גדולים וישנה אקראיות אין הבדלים גדולים בין הכנסה שמאלה וימינה.

מעניין לשים לב כי סכום כמות הגלגולים הממוצעת לכל n שואפת ל0.7 - תוצאה שניתן לראות הסבר לגביה במאמר הבא בעמוד 30 https://people.mpi-inf.mpg.de/~mehlhorn/ftp/AmortizedAnalysisAVL.pdf

לאחר התייעצות פרטנית עם אמיר רובינשטיין על המאמר ותוצאותיו שתואמות את שלנו - הוא כולל פרטים רבים שלא יילמדו בקורס זה ולכן אין צורך לפרט בעניין.