מבנה נתונים- תיעוד תרגיל מעשי 1

מגישים:

בר פקולה (ID : 316294297, user name : barpakula)

בר שגב (ID : 312456262, user name : barsegev)

# חלק א- תיעוד פונקציות

## מחלקת AVLTree

### שדות AVLTree

אתחול בהתאם לערכי ברירת המחדל הבאים:

* virtualLeaf – צומת ווירטואלית אליה יצביעו עלים וצמתים אונריים.
* root- מצביע לשורש העץ, מאותחל מצביע ל- virtualLeaf.
* min – מצביע לצומת שבה הערך המינימלי, אם העץ ריק אז השדה מצביע ל- virtualLeaf.
* max – מצביע לצומת שבה הערך המקסימלי, אם העץ ריק אז השדה מצביע ל- virtualLeaf.

### empty()

**תיאור כללי**- הפונקציה מחזירה ערך TRUE אם ורק אם העץ ריק.

**עקרון מימוש**- הפונקציה:

* קוראת ל-isRealNode() (פועלת ב-) על הצומת ששמורה בשורש העץ.
* אם היא virtualLeaf מחזירה true, אחרת מחזירה false.

**סיבוכיות**-

### treePosition(int k, IAVLNode root)

**תיאור כללי**- הפונקציה מחפשת איבר בעל המפתח k. אם קיים איבר כזה, היא מחזירה את v הצומת, אחרת היא מחזירה את ההורה של הצומת.

**עקרון מימוש**- הפונקציה מבצעת חיפוש בינארי בעץ:

* יוצרת מצביע p שתחילה מצביע לשורש העץ.
* בודקת אם המצביע לא בעלה ווירטואלי באמצעות הפונקציה isRealNode() ()
* בודקת אם k קטן, גדול או שווה למפתח של המצביע.
  + אם המפתח שווה מחזירה את הצומת.
  + אם קטן, מעדכנת את המצביע לבן השמאלי שלו.
  + אם גדול מעדכנת לבן הימני שלו.

האיטרציה ממשיכה כל עוד לא הגענו לצומת שהמפתח שווה ל-k או עד שהגענו לצומת שלא יכולנו לעשות צעד ימינה או שמאלה בהתאם לאם k גדול או קטן (הבן המתאים הוא virtualLeaf, בדקנו עם isRealNode() ב-).

אם נמצאה צומת ש-k שווה לערך בה מחזירים את הצומת. אחרת מחזירים את הצומת שהגענו אליה (ההורה של המקום המתאים ל-k).

**סיבוכיות**- (חיפוש בינארי)

### search(int k)

**תיאור כללי**- הפונקציה מחפשת איבר בעל המפתח k. אם קיים איבר כזה, היא מחזירה את הערך השמור עבורו, אחרת היא מחזירה null.

**עקרון מימוש**- הפונקציה:

* קוראת ל-treePosition(k) (( למציאת המקום של k בחיפוש בינארי.
* משווה את המפתח של הצומת שהוחזרה ל-k:
  + אם הם זהים, מחזירה את שדה ה-info של הצומת.
  + אחרת, מחזירה null.

**סיבוכיות**-

### insert(int k, String s)

**תיאור כללי**- הכנסת איבר בעל ערך s ומפתח k לעץ, אם המפתח לא קיים. הפונקציה מחזירה את מספר פעולות האיזון שנדרשו בסה"כ בשלב תיקון העץ על מנת להשלים את הפעולה (גלגולי LR ו- RL נחשבים ל- 2 פעולות איזון). אם קיים איבר בעל מפתח k בעץ הפונקציה מחזירה -1 ולא מתבצעת הכנסה.

**עקרון מימוש**- הפונקציה:

* יוצרת משתנה בשם rebalanceActions מסוג int עם הערך 0.

אם העץ ריק (הפונקציה empty() מחזירה true ב-):

* יוצרת אובייקט AVLNode חדש עם המפתח k והערך info.
* מעדכנת את שדה ה-root של העץ להצביע לצומת זו.
* מעדכנת את שדות ה-min וה-max של העץ להצביע לצומת החדשה.
* מעדכנת את שדות הצומת החדשה (כל פונקציה פועלת ב-):
  + setRight – שיצביע לעלה הווירטואלי.
  + setLeft – שיצביע לעלה הווירטואלי.
  + setParent – שיצביע לעלה הווירטואלי.
  + setHight – שיהיה שווה ל-0.
  + setSize – שיהיה שווה ל-1.

אחרת:

* קוראת ל-treePosition(k) (שפועלת ב-) ושומרת במצביע בשם parent.
  + בודקת אם המפתח של הצומת שהוחזרה שווה ל-k (getKey פועלת ב-). אם כן אז הצומת כבר בעץ ולכן מחזירה (-1).
  + אחרת-
    - יוצרת אובייקט AVLNode חדש עם המפתח k והערך info ושומרת במצביע בשם child.
    - מעדכנת את שדות הצומת החדשה (כל פונקציה פועלת ב-):
      * getLeft – שיצביע לעלה הווירטואלי.
      * setRight – שיצביע לעלה הווירטואלי.
      * setParent – שיצביע לצומת ההורה.
      * setHight – להיות שווה ל-0.
      * setSize – שיהיה שווה ל-1.
    - משווה בין המפתח של ה-parent ל-k (getKey פועלת ב-). אם k גדול מהמפתח מגדירה את הבן הימני של parent להצביע ל-child. אם k קטן מהמפתח הזה מגדירה שהבן השמאלי של parent יצביע ל-child (עם setRight או setLeft שפועלות ב-).
    - **עדכון המינימום-** משווה בין k למפתח ששמור בשדה min (getKey פועלת ב-). אם הוא קטן יותר- מעדכנת את min להצביע לצומת החדשה.
    - **עדכון המקסימום-** משווה בין המפתח של הצומת החדשה למפתח ששמור בשדה max (getKey פועלת ב-). אם הוא גדול יותר- מעדכנת את max להצביע לצומת החדשה.
    - **מאזנת את העץ-** קוראת ל- rebalance(parent)שמבצעת את פעולות האיזון של העץ וסופרת אותן, ומעדכנת את שדה ה-size בכל צומת רלוונטית (פועלת ב-). שומרת את הערך שהיא מחזירה ב- rebalanceActions.
* מחזירה את rebalanceActions

**סיבוכיות**- שנובע מפעולת ה-rebalance.

### delete(int k)

**תיאור כללי**- מחיקת איבר בעל המפתח k בעץ, אם הוא קיים. הפונקציה מחזירה את מספר פעולות האיזון שנדרשו בסך-הכל בשלב תיקון העץ על מנת להשלים את הפעולה.אם לא קיים איבר בעל המפתח k בעץ הפונקציה מחזירה .

**עקרון מימוש**- הפונקציה מבצעת את הפעולות הבאות:

* יוצרת משתנה בשם rebalanceActions מסוג int עם הערך 0.
* קוראת ל-treePosition(k) (שפועלת ב-) ושומרת במצביע בשם toDelete.
* בודקת אם המפתח של הצומת שהוחזרה שונה מ-k (getKey פועלת ב-). אם כן אז אין בעץ צומת עם המפתח k ולכן מחזירה (-1).
* בודקת אם בשדות min ו-max של העץ שמורה אותה הצומת. אם כן אז בעץ יש רק צומת בודדת ולכן נעדכן את שדות min ו-max להצביע ל-virtualLeaf.
* **עדכון המינימום-** קוראת ל- findSuccessor(min) שמחפשת את ה-successor ב-. מעדכנת את min להצביע לצומת המוחזרת.
* **עדכון המקסימום-** קוראת ל- findPredecessor(max) שמחפשת את ה-predecessor ב- . מעדכנת את max להצביע לצומת המוחזרת.
* יוצרת מצביע לצומת בשם startRebalance.
* בודקת אם toDelete עלה, צומת אונרית או צומת עם שני בנים. היא מפעילה את פונקציות getRight.isRealNode ו-getLeft.isRealNode שרצות ב- כל אחת.
  + אם toDelete זה עלה (שני הילדים הם עלים ווירטואליים) – קוראת ל- deleteLeaf(toDelete) שרצה ב- ושומרת את הצומת המוחזרת במשתנה startRebalance.
  + אם toDelete זו צומת אונרית – קוראת ל- deleteUnar(toDelete) שרצה ב- ושומרת את הצומת המוחזרת במשתנה startRebalance.
  + אחרת – קוראת ל- deleteDouble(toDelete) שרצה ב- ושומרת את הצומת המוחזרת במשתנה startRebalance.
* **מאזנת את העץ-** קוראת ל- rebalance(startRebalance) שמבצעת את פעולות האיזון של העץ וסופרת אותן, ומעדכנת את שדה ה-size בכל צומת רלוונטית (פועלת ב-) ושומרת את הערך שהיא מחזירה ב- rebalanceActions.
* מחזירה את rebalanceActions.

**סיבוכיות**- שנובע מפעולת ה-rebalance.

### deleteLeaf(IAVLNode child)

**תיאור כללי**- הפונקציה מקבלת עלה ומוחקת אותו מהעץ.

**עקרון מימוש**- הפונקציה:

* שומרת את ההורה של child במשתנה parent עם קריאה ל- getParent(child) שרצה ב-.
* בודקת אם child מצביע לשורש העץ. אם כן אז משנה את המצביע של root להיות virtualLeaf (אם הצומת היא גם עלה וגם השורש- במחיקתה העץ הופך לריק).
* אחרת- בודקת אם child הוא בן ימני או בן שמאלי של parent (באמצעות getRight ו-getLeft שרצות ב-) ובהתאם מעדכנת את הבן הימני או השמאלי להיות צומת וירטואלית (באמצעות setRight או setLeft שרצות ב-).
* מעדכנת את ההורה והילדים של child להצביע ל-null על מנת לנתק אותו לחלוטין מהעץ עם setParent, setRight, setLeft ().
* מחזירה את parent.

**סיבוכיות**-

### deleteUnar(IAVLNode child)

**תיאור כללי**- הפונקציה מקבלת צומת אונרית ומוחקת אותה מהעץ.

**עקרון מימוש**- הפונקציה:

* שומרת את ההורה של child במשתנה parent עם קריאה ל- getParent(child) שרצה ב-.
* בודקת אם child מצביע לשורש העץ:
  + אם כן אז משנה את המצביע של root להיות הבן היחיד של child
  + מעדכנת את ההורה של הבן הזה להיות עלה וירטואלי

(באמצעות getRight, getLeft, isRealNode, setParent, setRight, setLeft שרצות כל אחת ב-).

* אחרת
  + בודקת אם child הוא בן ימני או בן שמאלי של parent ובהתאם מעדכנת את הבן הימני או השמאלי של parent להיות הבן היחיד של child (באמצעות getRight, getLeft, isRealNode, setRight, setLeft, setParent שרצות כל אחת ב-).
  + מעדכנת את הבן המתאים של child להצביע ל-null על מנת לנתק אותו לחלוטין מהעץ עם setLeft או setRight ().
* מעדכנת את ההורה והילדים של child להצביע ל-null על מנת לנתק אותו לחלוטין מהעץ עם setParent, setRight, setLeft ().
* מחזירה את parent.

**סיבוכיות**-

### deleteDouble(IAVLNode child)

**תיאור כללי**- הפונקציה מקבלת צומת עם שני בנים ומוחקת אותה מהעץ. היא מבצעת זאת באמצעות מציאת ה-successor ומחיקתו והחלפת המקומות ביניהם כפי שראינו בהרצאה.

**עקרון מימוש**- הפונקציה:

* קוראת לפונקציית findSuccessor(child) ושומרת את הערך המוחזר ב-successor (נשים לב ש- successor לא יכול להיות null כי זה אומר שזו צומת ה-max והיא בהכרח עלה או אונרית וטיפלנו במקרה הזה. מצד שני ראינו כי successor זו צומת אונרית או עלה בעצמה).
* שומרת את ההורה של successor במשתנה successorParent.
* בודקת:
  + אם successor עלה- קוראת ל- deleteLeaf(successor) שפועלת ב-.
  + אם successor אונרית- קוראת ל- deleteUnar(successor) שפועלת ב-.
* מעדכנת את השדות של successor להיות אלו של child באמצעות קריאה לפונקציות getRight, getLeft, getParent, setRight, setLeft, setParent, setHeight, getHeight שרצות כל אחת ב-.
* אם child הוא שורש העץ:
  + מעדכנת את שדה root להצביע ל- successor.
  + אחרת- בהתאם לאם child הוא בן ימני או בן שמאלי מעדכנת את המצביע המתאים של ההורה של child להצביע ל- successor, באמצעות קריאה לפונקציות getRight, getLeft, setRight, setLeft שרצות כל אחת ב-.
* מעדכנת את ההורה והילדים של child להצביע ל-null על מנת לנתק אותו לחלוטין מהעץ עם setParent, setRight, setLeft ().
* מחזירה את successorParent.

**סיבוכיות**-

### rebalance(IAVLNode node, int countActions)

**תיאור כללי**- מבצעת את פעולת האיזון של העץ. נקראת על ידי Insert ו-delete. סופרת את כמות פעולות האיזון שהיא עושה.

**עקרון מימוש**- פונקציה רקורסיבית שמבצעת פעולת איזון לוקלית בהתאם לאלגוריתם שנלמד בכיתה. שומרת ספירה של כמות פעולות האיזון שביצענו במשתנה countActions. מעדכנת תוך כדי את שדה size של הצמתים מהצומת שהוכנסה ועד השורש.

* תנאי עצירה:
  + אם ההורה עלה ווירטואלי זה אומר שהגענו לשורש העץ ותפסיק הקריאה הרקורסיבית. במקרה הזה סיימנו לאזן את העץ ולעדכן את שדות size של הצמתים הרלוונטיים ונחזיר 0.
* מחשבת את הפרש ה-rank בין הצומת לכל אחד מהבנים שלה באמצעות קריאה לפונקציות getHeight, getLeft, getRight שרצות כל אחת ב-.
* אם הפרש ה-rank בין הצומת לבנים שלה הוא מהצורה (1,1), (1,2) או (2,1):
  + מעדכנת את שדה ה-size של node באמצעות setSize שרצה ב-.
  + קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
* אם הפרש ה-rank בין הצומת לבנים שלה הוא מהצורה (0,1) או (1,0):
  + קוראת ל-promote על node שרצה ב- ומוסיפה את הערך המוחזר ל- countActions.
  + מעדכנת את שדה ה-size של node באמצעות setSize שרצה ב-.
  + קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
* אם הפרש ה-rank בין הצומת לבנים שלה הוא מהצורה (2,2):
  + קוראת ל- demoteעל node שרצה ב- ומוסיפה את הערך המוחזר ל- countActions.
  + מעדכנת את שדה ה-size של node באמצעות setSize שרצה ב-.
  + קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
* אם הפרש ה-rank בין הצומת לבנים שלה הוא מהצורה (0,2):
  + אם הפרש ה-rank בין הבן השמאלי לבנים שלו הוא מהצורה (1,1) (רלוונטי ל-join).
    - מבצעת סיבוב ימינה ואז promote לבן השמאלי שרצות כל אחת ב-, ומוסיפה את הערך שמחזירות ל- countActions.
    - מעדכנת את שדה ה-size של node ושל הבן השמאלי באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
  + אם הפרש ה-rank בין הבן השמאלי לבנים שלו הוא מהצורה (1,2):
    - מבצעת סיבוב ימינה ואז demote ל-node שרצות כל אחת ב-, ומוסיפה את הערך שמחזירות ל- countActions.
    - מעדכנת את שדה ה-size של node ושל הבן השמאלי באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
  + אם הפרש ה-rank בין הבן השמאלי לבנים שלו הוא מהצורה (2,1):
    - מבצעת סיבוב כפול ימינה, demote ל-node ולבן השמאלי, ו-promote להורה החדש של הבן השמאלי (במקור היה הבן הימני של הבן השמאלי). כל פעולה רצה ב-. מוסיפה את הערך שמחזירות ל- countActions.
    - מעדכנת את שדה ה-size של node, של הבן השמאלי ושל ההורה החדש של הבן השמאלי (במקור היה הבן הימני של הבן השמאלי) באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
* אם הפרש ה-rank בין הצומת לבנים שלה הוא מהצורה (2,0):
  + אם הפרש ה-rank בין הבן הימני לבנים שלו הוא מהצורה (1,1) (רלוונטי ל-join).
    - מבצעת סיבוב שמאלה ואז promote לבן הימני שרצות כל אחת ב-, ומוסיפה את הערך שמחזירות ל-countActions.
    - מעדכנת את שדה ה-size של node ושל הבן הימני באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
  + אם הפרש ה-rank בין הבן הימני לבנים שלו הוא מהצורה (2,1):
    - מבצעת סיבוב שמאלה ואז demote ל-node שרצות כל אחת ב-, ומוסיפה את הערך שמחזירות ל-countActions.
    - מעדכנת את שדה ה-size של node ושל הבן הימני באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
  + אם הפרש ה-rank בין הבן הימני לבנים שלו הוא מהצורה (1,2):
    - מבצעת סיבוב כפול שמאלה, demote ל-node ולבן הימני, ו-promote להורה החדש של הבן הימני (במקור היה הבן השמאלי של הבן הימני). כל פעולה רצה ב-. מוסיפה את הערך שמחזירות ל- countActions.
    - מעדכנת את שדה ה-size של node, של הבן הימני ושל ההורה החדש של הבן הימני (במקור היה הבן השמאלי של הבן הימני) באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
* אם הפרש ה-rank בין הצומת לבנים שלה הוא מהצורה (3,1):
  + אם הפרש ה-rank בין הבן הימני לבנים שלו הוא מהצורה (1,1):
    - מבצעת סיבוב שמאלה, demote ל-node ו-promote לבן הימני. כל פעולה רצה ב-. מוסיפה את הערך שמחזירות ל-countActions.
    - מעדכנת את שדה ה-size של node ושל הבן הימני באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
  + אם הפרש ה-rank בין הבן הימני לבנים שלו הוא מהצורה (2,1):
    - מבצעת סיבוב שמאלה, ו-demote ל-node פעמיים. כל פעולה רצה ב-. מוסיפה את הערך שמחזירות ל-countActions.
    - מעדכנת את שדה ה-size של node ושל הבן הימני באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
  + אם הפרש ה-rank בין הבן הימני לבנים שלו הוא מהצורה (1,2):
    - מבצעת סיבוב כפול שמאלה, demote ל-node פעמיים, demote לבן הימני, ו-promote להורה החדש של הבן הימני (במקור היה הבן השמאלי של הבן הימני). כל פעולה רצה ב-. מוסיפה את הערך שמחזירות ל- countActions.
    - מעדכנת את שדה ה-size של node, של הבן הימני ושל ההורה החדש של הבן הימני (במקור היה הבן השמאלי של הבן הימני) באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
* אם הפרש ה-rank בין הצומת לבנים שלה הוא מהצורה (1,3):
  + אם הפרש ה-rank בין הבן השמאלי לבנים שלו הוא מהצורה (1,1):
    - מבצעת סיבוב ימינה, demote ל-node ו-promote לבן השמאלי. כל פעולה רצה ב-. מוסיפה את הערך שמחזירות ל-countActions.
    - מעדכנת את שדה ה-size של node ושל הבן השמאלי באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
  + אם הפרש ה-rank בין הבן השמאלי לבנים שלו הוא מהצורה (1,2):
    - מבצעת סיבוב ימינה, ו-demote ל-node פעמיים. כל פעולה רצה ב-. מוסיפה את הערך שמחזירות ל-countActions.
    - מעדכנת את שדה ה-size של node ושל הבן השמאלי באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
  + אם הפרש ה-rank בין הבן השמאלי לבנים שלו הוא מהצורה (2,1):
    - מבצעת סיבוב כפול ימינה, demote ל-node פעמיים, demote לבן השמאלי, ו-promote להורה החדש של הבן השמאלי (במקור היה הבן הימני של הבן השמאלי). כל פעולה רצה ב-. מוסיפה את הערך שמחזירות ל- countActions.
    - מעדכנת את שדה ה-size של node, של הבן השמאלי ושל ההורה החדש של הבן השמאלי (במקור היה הבן הימני של הבן השמאלי) באמצעות setSize שרצה ב-.
    - קוראת רקורסיבית ל-rebalance על ההורה של node (getParent רצה ב-( ומוסיפה את הערך המוחזר ל- countActions.
* מאפסת את המצביעים לבנים של virtualLeaf שיצביעו ל-null (באמצעות setRight, setLeft שרצות ב-(.
* מחזירה את countActions.

**סיבוכיות**-

### promote(IAVLNode node)

**תיאור כללי**- הפונקציה מבצעת את פעולת ה- promote ומעלה את גובה הצומת שקיבלה ב-1.

**עקרון מימוש**- מקבלת צומת ומגדילה את ה-height שלה ב-1 באמצעות הפונקציות getHeight ו-setHeight שרצות כל אחת ב-. מחזירה 1 שזה כמות פעולות האיזון שביצעה.

**סיבוכיות**-

### demote(IAVLNode node)

**תיאור כללי**- הפונקציה מבצעת את פעולת ה- demote ומורידה את ה של הצומת שקיבלה ב-.1

**עקרון מימוש**- מקבלת צומת ומקטינה את ה-height שלה ב-1 באמצעות הפונקציות getHeight ו-setHeight שרצות כל אחת ב-. מחזירה 1 שזה כמות פעולות האיזון שביצעה.

**סיבוכיות**-

### rotateRight(IAVLNode node, IAVLNode left)

**תיאור כללי**- ביצוע גלגול יחיד ימינה לוקלית.

**עקרון מימוש**- node מצביע לשורש, left מצביע לבן השמאלי של node. הפונקציה:

* אם node מצביע לאותה צומת כמו ה-root (שורש העץ)- מעדכנת את השורש של העץ להצביע לבן השמאלי.
* אחרת-
  + יוצרת מצביע parent שמצביע להורה של node.
  + בהתאם לאם node הוא בן שמאלי או ימני מעדכנת את המצביע המתאים של parent שיצביע לבן השמאלי באמצעות פונקציות getRight, setRight, getLeft, setLeft שרצות כל אחת ב-.
* מבצעת את הגלגול על ידי עדכון המצביעים:
  + מעדכנת את ההורה של הבן השמאלי להיות ההורה של node.
  + מעדכנת את הבן השמאלי של node להיות הבן הימני של הבן השמאלי.
  + מעדכנת את ההורה של הבן השמאלי החדש של node להצביע ל-node.
  + מעדכנת את הבן ימני של הבן השמאלי להצביע ל-node.
  + מעדכנת את ההורה של node להיות הבן השמאלי.

עדכון המצביעים מתבצע באמצעות הפונקציות setParent, getParent, setLeft, getLeft, setRight, getRight שרצות כל אחת ב-.

* מחזירה 1 שזה כמות פעולות האיזון שביצעה.

**סיבוכיות**-

### rotateLeft(IAVLNode node, IAVLNode right)

**תיאור כללי**- ביצוע גלגול יחיד שמאלה לוקלית.

**עקרון מימוש**- x node מצביע לשורש, right מצביע לבן הימני של node. הפונקציה:

* אם node מצביע לאותה צומת כמו ה-root (שורש העץ)- מעדכנת את השורש של העץ להצביע לבן הימני.
* אחרת-
  + יוצרת מצביע parent שמצביע להורה של node.
  + בהתאם לאם node הוא בן שמאלי או ימני מעדכנת את המצביע המתאים של parent שיצביע לבן הימני באמצעות פונקציות getRight, setRight, getLeft, setLeft שרצות כל אחת ב-.
* מבצעת את הגלגול על ידי עדכון המצביעים:
  + מעדכנת את ההורה של הבן הימני להיות ההורה של node .
  + מעדכנת את הבן הימני של node להיות הבן השמאלי של הבן הימני.
  + מעדכנת את ההורה של הבן הימני החדש של node להצביע ל-node.
  + מעדכנת את הבן השמאלי של הבן הימני להצביע ל-node.
  + מעדכנת את ההורה של node להיות הבן הימני.

עדכון המצביעים מתבצע באמצעות הפונקציות setParent, getParent, setLeft, getLeft, setRight, getRight שרצות כל אחת ב-.

* מחזירה 1 שזה כמות פעולות האיזון שביצעה.

**סיבוכיות**-

### doubleRotateRight(IAVLNode node, IAVLNode left)

**תיאור כללי**- ביצוע גלגול כפול ימינה.

**עקרון מימוש**- node מצביע לשורש, left מצביע לבן השמאלי של node. הפונקציה:

* יוצרת מצביע לבן הימני של הבן השמאלי.
* אם node מצביע לאותה צומת כמו ה-root (שורש העץ)- מעדכנת את השורש של העץ להצביע לבן הימני של הבן השמאלי.
* מבצעת סיבוב שמאלה על הבן השמאלי והבן הימני שלו.
* מבצעת סיבוב ימינה על node והבן הימני של הבן השמאלי שלו.

כל סיבוב מתבצע ב- פעולות.

* מחזירה 2 שזה כמות פעולות האיזון שביצעה.

**סיבוכיות**-

### doubleRotateLeft(IAVLNode node, IAVLNode right)

**תיאור כללי**- ביצוע גלגול כפול שמאלה.

**עקרון מימוש**- node מצביע לשורש, right מצביע לבן השמאלי של node. הפונקציה:

* יוצרת מצביע לבן השמאלי של הבן הימני.
* אם node מצביע לאותה צומת כמו ה-root (שורש העץ)- מעדכנת את השורש של העץ להצביע לבן השמאלי של הבן הימני.
* מבצעת סיבוב ימינה על הבן הימני והבן השמאלי שלו.
* מבצעת סיבוב שמאלה על node והבן השמאלי של הבן הימני שלו.

כל סיבוב מתבצע ב- פעולות.

* מחזירה 2 שזה כמות פעולות האיזון שביצעה.

**סיבוכיות**-

### findSuccessor(IAVLNode node)

**תיאור כללי**- הפונקציה מחזירה את הצומת עם המפתח העוקב שאחרי node.

**עקרון מימוש**- הפונקציה מקבלת node.

* אם node היא צומת ה-max – מחזירה virtualLeaf (אין לה עוקב אחרי).
* יוצרת מצביע curr שמצביע תחילה על node.
* אם הבן הימני קיים אז (getRight ו-isRealNode רצות ב-):
  + עושה צעד אחד ימינה בעץ באמצעות getRight ().
  + עושה לולאה של צעדים שמאלה בעץ כל עוד לא הגענו לעלה באמצעות getLeft (). לכל היותר תבצע צעדים שמאלה.
  + כשהגיעה לעלה השמאלי ביותר- מחזירה אותו.
* אחרת:
  + יוצרת מצביע נוסף parent שמצביע על ההורה של הצומת (getParent רצה ב-).
  + מתחילה לולאה-
    - כל עוד הצומת היא בן ימני של ההורה, ולא הגענו לשורש, מקדמים את שני המצביעים צעד למעלה.
    - כאשר הצומת היא בן שמאלי- מחזירים את ההורה.

מבצעים את פעולת עדכון המצביעים באמצעות isRealNode, getParent, getRight, getLeft שרצות ב-.

כמות האיטרציות היא לכל היותר .

**סיבוכיות**-

### finsPredecessor(IAVLNode node)

**תיאור כללי**- הפונקציה מחזירה את הצומת עם המפתח העוקב שלפני node.

**עקרון מימוש**- הפונקציה מקבלת את node.

* אם node היא צומת ה-min – מחזירה virtualLeaf (אין לה עוקב לפני).
* יוצרת מצביע curr שמצביע תחילה על node.
* אם הבן השמאלי קיים אז (getLeft ו-isRealNode רצות ב-):
  + עושה צעד אחד שמאלה בעץ באמצעות getLeft ().
  + עושה לולאה של צעדים ימינה בעץ כל עוד לא הגענו לעלה באמצעות getRight (). לכל היותר תבצע צעדים ימינה.
  + כשהגיעה לעלה הימני ביותר- מחזירה אותו.
* אחרת:
  + יוצרת מצביע נוסף parent שמצביע על ההורה של הצומת (getParent רצה ב-).
  + מתחילה לולאה-
    - כל עוד הצומת היא בן שמאלי של ההורה, ולא הגענו לשורש, מקדמים את שני המצביעים צעד למעלה.
    - כאשר הצומת היא בן ימני- מחזירים את ההורה.

מבצעים את פעולת עדכון המצביעים באמצעות isRealNode, getParent, getRight, getLeft שרצות ב-.

כמות האיטרציות היא לכל היותר .

**סיבוכיות**-

### min()

**תיאור כללי**- מחזירה את ערך (info) האיבר בעץ בעל המפתח המינימלי, או null בעץ ריק.

**עקרון מימוש**- הפונקציה מחזירה את הערך השמור בשדה min. (עדכון השדה מתבצע במידת הצורך בהכנסה ומחיקה של צומת מהעץ).

**סיבוכיות**-

### max()

**תיאור כללי**- מחזירה את ערך (info) האיבר בעץ בעל המפתח המקסימלי, או null בעץ ריק.

**עקרון מימוש**- הפונקציה מחזירה את הערך השמור בשדה max. (עדכון השדה מתבצע במידת הצורך בהכנסה ומחיקה של צומת מהעץ).

**סיבוכיות**-

### getMin()

***תיאור כללי-*** *מחזירה את הצומת שבה המפתח המינימלי של העץ.*

***עקרון מימוש-*** *מחזירה את שדה min שמצביע לצומת שבה המפתח המינימלי של העץ.*

***סיבוכיות-***

### getMax()

***תיאור כללי-*** *מחזירה את הצומת שבה המפתח המקסימלי של העץ.*

***עקרון מימוש-*** *מחזירה את שדה max שמצביע לצומת שבה המפתח המקסימלי של העץ.*

***סיבוכיות-***

### keysToArray()

**תיאור כללי**- הפונקציה מחזירה מערך ממוין המכיל את כל המפתחות בעץ, או מערך ריק אם העץ ריק.

**עקרון מימוש**- הפונקציה:

* יוצרת מערך ריק למפתחות בגודל n.
* יוצרת מערך ריק לצמתים בגודל n.
* קוראת לפונקציה nodeToArray שממלאת את המערך הריק של הצמתים בצמתי העץ ממוינים לפי המפתחות שלהם. היא רצה ב-.
* עוברת באיטרציה על מערך הצמתים ומוסיפה את המפתח שלהם למערך המפתחות בהתאם. סך הכל מתבצעות פעולות getKey שלוקחות כל אחת.
* מחזירה את מערך המפתחות.

**סיבוכיות**-

### infoToArray()

**תיאור כללי**- הפונקציה מחזירה מערך מחרוזות המכיל את כל המחרוזות בעץ, ממוינות על פי סדר המפתחות. כלומר הערך ה j במערך הוא המחרוזת המתאימה למפתח שיופיע במיקום ה j במערך הפלט של הפונקציה keysToArray() . גם הפונקציה הזאת מחזירה מערך ריק אם העץ ריק.

**עקרון מימוש**- הפונקציה:

* יוצרת מערך ריק לערכים בגודל n.
* יוצרת מערך ריק לצמתים בגודל n.
* קוראת לפונקציה nodeToArray שממלאת את המערך הריק של הצמתים בצמתי העץ ממוינים לפי המפתחות שלהם. היא רצה ב-.
* עוברת באיטרציה על מערך הצמתים ומוסיפה את הערך שלהם למערך הערכים בהתאם. סך הכל מתבצעות פעולות getValue שלוקחות כל אחת.
* מחזירה את מערך המפתחות.

**סיבוכיות**-

### nodeToArray(IAVLNode curr, IAVLNode[] orderedNodes, int i)

**תיאור כללי-** הפונקציה יוצרת מערך של צמתים הממוין לפי המפתחות שלהם.

**עקרון מימוש-** פונקציה רקורסיבית. מקבלת מצביע לצומת נוכחית curr, מערך צמתים ואינדקס i. הפונקציה רצה על העץ ב-inOrder ומכניסה לפי הסדר את הצמתים למערך הצמתים כך שלבסוף מתקבל מערך ממוין לפי המפתחות.

* תנאי עצירה – האינדקס שווה לגודל המערך. יקרה כשהמערך מלא.
* כל עוד curr לא עלה וירטואלי (isRealNode רצה ב-):
  + קריאה לפונקציה על הבן השמאלי של curr ושמירת הערך המוחזר באינדקס.
  + הכנסת curr למקום האינדקס ברשימה.
  + קריאה לפונקציה על הבן הימני של curr ושמירת הערך המוחזר באינדקס.
* החזרת האינדקס.

**סיבוכיות-** בכל צומת בעץ עוברים פעמיים לכל היותר.

### Size()

**תיאור כללי**- הפונקציה מחזירה את מספר האיברים בעץ.

**עקרון מימוש**- המימוש מחזיר את שדה ה-size של השורש שמהווה את גודל העץ כולו. (עדכון השדה מתבצע במידת הצורך כחלק מתהליך ה-rebalance).

**סיבוכיות**-

### getRoot()

**תיאור כללי**- הפונקציה מחזירה את השורש של העץ (אובייקט AVLNode ).

**עקרון מימוש**- הרשימה תמיד מחזיקה מצביע לשורש, הפונקציה מחזירה את השורש בהתאם למצביע זה. (עדכון השדה מתבצע במידת הצורך בהכנסה ומחיקה של צומת מהעץ).

**סיבוכיות**-

### setRoot(IAVLNode newRoot)

***תיאור כללי-*** *מאפשרת להגדיר את הצומת כשורש העץ (בשימוש בפונקציית split)*

***עקרון מימוש-*** *מגדירה את שדה root להצביע על* newRoot*.*

***סיבוכיות-***

### getVirtualLeaf()

**תיאור כללי**- הפונקציה מחזירה את העלה הווירטואלי של העץ (אובייקט AVLNode ).

**עקרון מימוש**- הרשימה תמיד מחזיקה מצביע לעלה הווירטואלי, הפונקציה מחזירה את העלה הווירטואלי בהתאם למצביע זה.

**סיבוכיות**-

### split(int x)

**תיאור כללי**- הפונקציה מקבלת מפתח x שנמצא בעץ. על הפונקציה להפריד את העץ ל- 2 עצי AVL כאשר המפתחות של האחד גדולים מ- x ושל השני קטנים מ- x יש לממש את הפונקציה על פי המימוש שנלמד בהרצאה בסיבוכיות𝑂(log𝑛)

**עקרון מימוש**- הפונקציה:

* מחפשת את הצומת שהמפתח שלה הוא x ע"י שימוש בפונקציה treePosition שרצה ב-.
* יוצרת מערך של שני עצי AVL – באינדקס 0 יהיה העץ המכיל את הצמתים הקטנים מ-x ובאינדקס 1 יהיה העץ המכיל את הצמתים הגדולים מ-x.
* אם לצומת עם המפתח x יש צאצאים, מוסיפה אותם בהתאמה – בן שמאלי לעץ הקטן (0), בן ימני לעץ הגדול (1):
  + מגדירה את שורש העצים במערך להצביע לשורש תת העץ השמאלי או הימני בהתאמה (setRoot שרצה ב-).
  + מעדכנת את מצביע ההורה של שורשים אלו להצביע לעלה ווירטואלי באמצעות הפונקציות getRight, getLeft, setParent שרצות כל אחת ב-.
* מנתקת את x מהעץ המקורי והילדים שלו באמצעות setLeft, setRight, setParent שרצות ב-.
* פועלת בהתאם לאלגוריתם שראינו בהרצאה-
  + עולה במעלה העץ להורה הקודם של x באמצעות getParent ו-isRealNode ( כל אחת)
  + יוצרת צומת עותק להורה שהגיעה אליו ()
  + יוצרת עץ חדש שיכיל את תת העץ שנרצה להוסיף.
  + אם x בתת העץ הימני של ההורה (משווה ל-getKey ב-)
    - מוסיפה את תת העץ השמאלי של ההורה עם setRoot ו-getLeft () לעץ החדש שיצרה. עץ זה כעת מכיל מפתחות שקטנים מ-x ונרצה להוסיף אותו לעץ באינדקס 0. מעדכנת את השורש של העץ החדש להצביע לעלה הווירטואלי של העץ (getRoot, setParent, getVirtualLeaf שכל אחת ).
    - קוראת ל-join לעץ באינדקס 0 עם צומת עותק ההורה והעץ החדש שיצרה.

היא רצה ב-

* + אם x בתת העץ השמאלי של ההורה (משווה ל-getKey ב-)
    - מוסיפה את תת העץ הימני של ההורה עם setRoot ו-getRight () לעץ החדש שיצרה. עץ זה כעת מכיל מפתחות שגדולים מ-x ונרצה להוסיף אותו לעץ באינדקס 1. מעדכנת את השורש של העץ החדש להצביע לעלה הווירטואלי של העץ (getRoot, setParent, getVirtualLeaf שכל אחת ).
    - קוראת ל-join לעץ באינדקס 1 עם צומת עותק ההורה והעץ החדש שיצרה.

היא רצה ב-

האלגוריתם הזה מתבצע לכל היותר פעמים, בכל פעם נקרא על הורה יותר עליון בעץ עד שמגיע לשורש. מכיוון ש-join מתבצע על תתי עצים ראינו בהרצאה כי סיבוכיות האלגוריתם נשארת .

* מחזירה את מערך העצים

**סיבוכיות**-

### join(IAVLNode x, AVLtree t)

**תיאור כללי**- הפונקציה מקבלת צומת x ועץ t שכל ה keys- שלהם קטנים, או שכולם גדולים, מה keys- של העץ הנוכחי שביחס אליו קראנו לjoin. על הפונקציה לאחד את x,t לעץ הנוכחי כפי שמומש בהרצאה. על הפעולה לרוץ בזמן 𝑂(log𝑛). על הפעולה להחזיר את העלות של פעולת ה -join (הפרש גבהי העצים + 1).

**עקרון מימוש**- הפונקציה:

* מחשבת את הערך המוחלט של הפרש הגבהים בין העצים + 1 (באמצעות getHeight, getRoot, abs שרצות ב-( ושומרת במשתנה result.
* במידה ושני העצים ריקים (קריאה ל-empty שרצה ב-(- מוסיפה את הצומת x לעץ עם insert על getKey ו-getValue של x (רצות ב-). מכיוון שהעץ ריק insert של הוספת שורש לוקחת פעולות. מחזירה את result ששווה במקרה הזה ל-1.
* אם רק אחד מהעצים ריק –
  + מוסיפה את x לעץ המלא באמצעות insert על getKey ו-getValue של x (רצות ב-). Insert במקרה זה רצה ב- כאשר n זה גודל העץ המלא (בפועל זה גובה העץ המלא).
  + אם העץ המלא זה לא העץ הנוכחי- מעדכנת את שדות העץ הנוכחי לשדות של t באמצעות getRoot, getMax, getMin, שפועלות ב-.
  + מחזירה את result

במקרה זה הפרש הגבהים בין העצים הוא גובה העץ המלא וזה זהה ל- של פעולת ה-insert, לכן סה"כ הסיבוכיות פה תהיה שזה בפועל הפרש הגבהים בין העצים + 1 (תוספת גובה שיכולה להיגרם מ-x).

* אחרת, במידה ושני העצים מלאים:
  + בודקת איזה עץ צריך להיות ימני ואיזה שמאלי באיחוד בהתאם לאם x גדול או קטן מהמפתח בשורש (getRoot, getKey שרצות ב-).
  + בודקת אם הפרשי הגבהים בין מינוס 1 ל 1 – אם כן, מגדירה את x להיות שורש שני העצים, ומגדירה אותם להיות הבן הימני והשמאלי בהתאם. מכך נקבל עץ מאוזן:
    - מעדכנת את המצביעים של ההורים והבנים באמצעות setRight, setLeft, setParent, getRoot, getRight, getLeft, setParnet שרצות ב-.
    - מעדכנת את size של x עם setSize ().
    - מגדירה את הגובה של x להיות המקסימלי מבין הגבהים של הבן הימני או השמאלי + 1 (getRight, getHeight, getRight, max שרצות ב-).
    - מעדכנת את שורש העץ להצביע ל-x.
  + אחרת- מבצעת את האלגוריתם שראינו בהרצאה:
    - יורדת לאורך העץ הגבוה יותר, שמאלה אם יש לו מפתחות גדולים יותר מהעץ השני או ימינה אם הם קטנים יותר, עד שמגיעה לצומת בגובה של השורש של העץ השני. (באמצעות getHeight, getRoot, getLeft, getRight שרצות ב-).
    - מחברת את x כהורה של העץ הנמוך והצומת שהגיעה אליה בעץ הגבוה, ומגדירה את ההורה של צומת זו כהורה של x (בהתאם לאלגוריתם שראינו בהרצאה). עדכון המצביעים מתבצע באמצעות קריאה ל- setParent, getParent, setLeft, setRight, getLeft, getRight, getRoot שרצות ב- כל אחת.
    - מגדירה את הגובה של x להיות המקסימלי מבין הגבהים של הבן הימני או השמאלי + 1 (getRight, getHeight, getRight, max שרצות ב-).
    - מעדכנת את מצביע השורש של העץ להצביע לשורש העץ הגבוה.
  + מאזנת את העץ באמצעות rebalance מ-x במעלה העץ. היא רצה בסיבוכיות ששווה למסלול מ-x לשורש העץ ששווה להפרש הגבהים בין העצים + 1.
  + מעדכנת את המצביעים למקסימום ולמינימום באמצעות getMax של העץ הימני ו-getMin של העץ השמאלי ב- כל אחת.

במקרה זה הסיבוכיות מושפעת מפעולת ה-rebalance שלוקחת כאמור הפרש הגבהים בין העצים + 1 וזו הסיבוכיות הרצויה.

* מחזירה את result

**סיבוכיות**-

## מחלקת AVLNode

### שדות AVLNode

שדות האובייקט- אתחול בהתאם לערכי ברירת המחדל הבאים:

* left- מצביע לבן השמאלי של הצומת. מאותחל כמצביע ל-null.
* right- מצביע לבן הימני של הצומת. מאותחל כמצביע ל-null.
* parent- מצביע להורה של הצומת. מאותחל כמצביע ל-null.
* height- שומר את גובה הצומת. מאותחל בערך -1 (רלוונטי לצומת ווירטואלית).
* key- שומר את מפתח הצומת. מאותחל בערך -1 (רלוונטי לצומת ווירטואלית).
* info- שומר את ערך הצומת. מאותחל כמצביע ל-null.
* size- שומר את גודל תת העץ שהצומת היא השורש שלו. מאותחל ב-0.

### בנאי AVLNode

1. בנאי ריק:
   1. רלוונטי לצומת ווירטואלית.
   2. לא מקבל ערכים בקריאה. משאירים את ערכי ברירת המחדל של השדות.
2. בנאי שמקבל key -value
   1. לצומת אמיתית- מעדכן את שדות key ו-info בהתאם לערכים שקיבל.
   2. מעדכן שדה size להיות 1.

**סיבוכיות**-

### getKey()

**תיאור כללי**- מחזיר את המפתח של הצומת, או 1- אם הצומת הוא וירטואלי.

**עקרון מימוש**- הפונקציה מחזירה את השדה key (אם הצומת וירטואלית ה-key שלה שווה ל-(-1))

**סיבוכיות**-

### getValue()

**תיאור כללי**- מחזיר את ה info של הצומת או null אם הצומת הוא וירטואלי.

**עקרון מימוש**- הפונקציה מחזירה את השדה info (אם הצומת וירטואלית ה-info שלה הוא null).

**סיבוכיות**-

### setLeft(IAVLNode node)

***תיאור כללי****- מגדיר מצביע לבן השמאלי של הצומת.*

***עקרון מימוש****- הפונקציה מגדירה את שדה left להצביע לצומת ל-node.*

***סיבוכיות****-*

### getLeft(IAVLNode node)

**תיאור כללי**- מחזיר את הבן השמאלי של הצומת, או null אם אין כזה.

**עקרון מימוש**- הפונקציה מחזירה את שדה left שמצביע לבן השמאלי של הצומת.

**סיבוכיות**-

### setRight(IAVLNode node)

***תיאור כללי****- מגדיר מצביע לבן הימני של הצומת.*

***עקרון מימוש****- הפונקציה מגדירה את שדה right להצביע ל- node.*

***סיבוכיות****-*

### getRight(IAVLNode node)

**תיאור כללי**- מחזיר את הבן הימני של הצומת, או null אם אין כזה.

**עקרון מימוש**- הפונקציה מחזירה את שדה right שמצביע לבן הימני של הצומת.

**סיבוכיות**-

### setParent(IAVLNode node)

***תיאור כללי****- מגדיר את ההורה של הצומת.*

***עקרון מימוש****- הפונקציה מגדירה את שדה parent להצביע ל-node.*

***סיבוכיות****-*

### getParent()

***תיאור כללי****- מחזיר את ההורה של הצומת או null אם אין כזה (במקרה שהצומת היא השורש של העץ).*

***עקרון מימוש****-* מחזירה את שדה parent שמצביע להורה של הצומת.

***סיבוכיות****-*

### isRealNode()

**תיאור כללי**- מחזיר TRUE אם הצומת מייצג צומת אמיתי בעץ (קרי: צומת שאינו וירטואלי).

**עקרון מימוש**- הפונקציה בודקת את שדה ה-key של הצומת, אם הערך שונה מ-(-1) מחזירים TRUE, אחרת FALSE.

**סיבוכיות**-

### setHeight(int height)

***תיאור כללי****- מעדכן את גובה הצומת.*

***עקרון מימוש****- הפונקציה מגדירה את שדה height של הצומת כשווה לגובה שהפונקציה קיבלה.*

***סיבוכיות****-*

### getHeight()

**תיאור כללי**- מחזיר את גובה הצומת, -1 עבור צומת וירטואלי. יש לממש בסיבוכיות .

**עקרון מימוש**- מחזיר את שדה height של הצומת.

**סיבוכיות**-

### getSize()

**תיאור כללי**- מחזיר את גודל תת העץ שהצומת היא השורש שלו.

**עקרון מימוש**- מחזיר את שדה size של הצומת.

**סיבוכיות**-

### setSize()

**תיאור כללי**- מעדכן את גודל תת העץ שהצומת היא השורש שלו.

**עקרון מימוש**- מגדיר את שדה size של הצומת להיות שווה לחיבור גודל תת העץ הימני עם גודל תת העץ השמאלי ועוד 1 שזו התוספת של השורש. קוראת ל-getSize שמתבצעת ב- ומבצעת חישוב אריתמטי של חיבור.

**סיבוכיות**-

# חלק ב- ניסויי/תיאורטי

## שאלה 1

### סעיף א

הטבלה שהתקבלה:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מספר סידורי i | 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 | 1999000 | 38884 | 982090 | 33230 |
| 2 | 7998000 | 85764 | 4083776 | 76098 |
| 3 | 31996000 | 187524 | 15920835 | 166653 |
| 4 | 127992000 | 407044 | 64038125 | 369990 |
| 5 | 511984000 | 878084 | 257790803 | 808751 |

### סעיף ב

ננתח את כמות החילופים שמבוצעים עבור מערך ממוין הפוך בגודל :

נשים לב כי במערך ממוין הפוך האיבר ה- יבצע חילופים (האיבר הראשון "צריך" להגיע למקום האחרון, האיבר השני למקום אחד לפני אחרון וכן הלאה) אז נקבל סכום סדרה חשבונית:

מבחינת סיבוכיות מדובר ב-

ננתח את עלות החיפושים עבור מערך ממוין הפוך בגודל :

במערך ממוין הפוך בכל הכנסה מכניסים את האיבר המינימלי לעץ, ולכן בכל חיפוש אורך המסלול שהאלגוריתם יעשה יהיה מאיבר המקסימום עד לשורש העץ וממנו מסלול עד לאיבר השמאלי ביותר. העץ יאזן את עצמו על ידי גלגולים ימינה. כלומר, סך הכל יעלה צעדים, ואז ירד צעדים. כאשר מכניסים איברים לעץ, בזכות תכונות האיזון והעובדה שעומק העץ חסום ע"י עלות החיפושים תהיה לכל הפחות תהיה סכום פעמיים log האיבר ה-i שמכניסים לעץ:

ועלות החיפושים תהיה לכל היותר n חיפושים בעץ בגובה , סה"כ

ולכן נקבל כחסם הדוק

### סעיף ג

הערכים בטבלה מסעיף א' מתאימים לניתוח בסעיף ב'.

נשים לב שבמידה ונפעיל על n את הפונקציה שמייצגת את הסיבוכיות של האלגוריתם- עבור מספר החילופים, ו- עבור עלות החיפושים - נקבל קשר ליניארי בין התוצאות שקיבלנו בסעיף א לבין תוצאת הפונקציה המתאימה. המתאם הליניארי שקיבלנו מראה שאכן הפונקציות מתנהגות בהתאם לאופן בו ניתחנו אותן.

### סעיף ד

מס' החילופים במערך באופן כללי הוא , נסמן את מספר החילופים שמבצע האיבר ה- ב-. כפי שראינו באופן דומה בשיעור, מספר הרמות בעץ שנצטרך לעלות עבור האיבר ה-, יהיה בסדר גודל של תודות לפעולות האיזון ששומרות על גובה העץ לוגריתמי במס' האיברים שיש בו. לכן סה"כ עלות insertion sort ב AVL היא סכום עלות החיפושים ועלות המעבר על איברי הרשימה שצריך להכניס.

העלות עבור החיפושים הינה:

כאשר המעבר הראשון נובע מחוקי לוגים והמעבר הרביעי נובע מאי שוויון הממוצעים. נוסיף את העלות על מעבר הרשימה ונקבל שהעלות הכוללת חסומה ע"י:

## שאלה 2

### סעיף א

הטבלה שהתקבלה:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| מספר סידורי i | 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 | 2.62 | 7 | 2.5 | 13 |
| 2 | 2.45 | 5 | 2.5 | 14 |
| 3 | 2.75 | 5 | 2.75 | 15 |
| 4 | 2.77 | 10 | 2.61 | 17 |
| 5 | 2.56 | 7 | 2.91 | 18 |
| 6 | 2.4 | 6 | 2.6 | 19 |
| 7 | 2.56 | 5 | 2.41 | 20 |
| 8 | 2.6 | 8 | 2.68 | 21 |
| 9 | 2.21 | 7 | 2.82 | 22 |
| 10 | 2.73 | 4 | 2.5 | 24 |

### סעיף ב

ננתח באופן תיאורטי את העלות של join ממוצע לשני התרחישים- splitאקראי או על האיבר המקסימלי בתת העץ השמאלי.

בהרצאה ראינו כי עלות ממוצעת של פעולת split שנתמך בפעולות join (שעלותן היא הפרש הגבהים של שני העצים בערך מוחלט ועוד 1), היא . זאת משום שגובה עץ AVL הוא , וכי split עוברת על מסלול מהצומת שעבורו מפצלים את העץ עד שורש העץ, ובכל צעד למעלה מבצעת join לאחד מהעצים – עץ המפתחות הקטנים או עץ המפתחות הגדולים. נשים לב כי עלות join היקרה ביותר שיכולה להתרחש היא זאת המתוארת בסעיף ג (join לאיבר המקסימלי בתת העץ השמאלי בבניית עץ המפתחות הגדולים ממנו), ובמקרה זה עלות הפעולה תהיה כגובה העץ (הרחבה בסעיף ג). אבל, נשים לב כי פעולות ה-join שבונות את עץ המפתחות הקטנים מצומת הפיצול יהיו בהכרח פעולות זולות- בכל צעד במעלה המסלול מצומת הפיצול ועד השורש (לא כולל) תתבצע פעולת join על שני עצים שהגבהים שלהם מאד קרובים מכיוון שמתכונות עץ AVL ההפרש בגבהים בין שני הבנים של צומת הוא לכל היותר 1, לכל אב עליון של צומת הפיצול לוקחים את תת העץ השמאלי ומאחדים עם מה שאיחדנו עד כה (שיכיל את הבנים השמאליים הקודמים ולכן עץ זה יהיה באותו הגובה פלוס מינוס 1 או 2). כלומר, במקרה זה פעולת join תהיה בסדר גודל של .

אם נסכום את עלות פעולות ה-join במקרה זה ונחלק בכמות הפעולות נקבל

כלומר, מדובר בממוצע בפעולה בסדר גודל קבוע. נשים לב שגם בטבלת התוצאות נראה כי גם כאשר העץ גדל עלות הפעולה הממוצעת נשארת דומה ויחסית נמוכה. הטרייד אוף הזה בין הפעולות היקרות והזולות מסביר את הביצועים בפועל. נשים לב כי במקרה של split אקראי איננו יודעים היכן ממוקמת צומת הפיצול. אם היא במיקום הגרוע שניתחנו (או במיקום הסימטרי לו) ראינו כי העלות הממוצעת היא קבועה. המקרים הבאים שיכולים להיות יקרים הם צומת אקראית שהיא עלה, זאת מכיוון שכמות הקריאות ל-join תהיה כגובה העץ. אבל נשים לב שמקרה של האיבר המקסימלי או המינימלי בעץ כמות פעולות ה-join תהיה כגובה העץ וכל פעולה תהיה פעולה זולה (בעצם נקבל מצב שעץ מפתחות אחד ריק ושני מלא בכל צמתי העץ מלבד צומת הפיצול) ולכן סיכום פעולות ה-join יהיה שיתבצעו ב- קריאות ל-join ושוב נקבל כי בממוצע העלות תהיה . בין שני המקרים שתיארנו יש טווח שבו הפרש הגבהים המקסימלי בין עצים הולך וקטן, ולכן גם פעולות ה-join הופכות להיות זולות (כאשר הזולה ביותר היא לעלה המקסימלי או המינמלי עבור הרמה הכי תחתונה של העץ) ונקבל כי הממוצע הפעולה לוקחת . נשים לב שאם אנחנו עולים מרמת העלים לרמה הבאה בעץ הניתוח דומה (ההפרשים בגבהים דומים). ככל שהצומת שהוגרלה יותר קרובה לשורש כך יהיו פחות פעולות join כך שלכל היותר עלות הפעולה הממוצעת נשארת . זה כמובן גם מסתדר עם התוצאות שבטבלה בדומה למקרה הנ"ל שניתחנו, העלות בין גדלי עצים שונים נשארת דומה וזה מתאים לכך שהעלות הממוצעת בניתוח התיאורטי קבועה.

### סעיף ג

ננתח באופן תיאורטי את העלות של join מקסימלי עבור splitעל האיבר המקסימלי בתת העץ השמאלי.

במקרה זה, עץ המפתחות הגדולים מצומת הפיצול יכיל את תת העץ הימני של שורש העץ ואת שורש העץ. פעולת ה-Join שתבנה את העץ הזה תתבצע על עץ ריק, העץ שהוא תת העץ הימני של שורש העץ, ועל השורש עצמו. פעולת ה-join תלויה בהפרש הגבהים בין העצים שהיא מאחדת ובמקרה הזה תעלה כגובה העץ כולו - האיבר המקסימלי בתת העץ השמאלי הוא או עלה או צומת אונרית עם בן שמאלי בלבד שהוא עלה (אחרת לא נשמר המבנה של עץ AVL), לכן אם נסמן ב-H את גובה העץ המקורי, הפרש הגבהים יהיה בין H+1 ל-H. זה התרחיש הגרוע ביותר לפעולת join כחלק מפעולת split מכיוון שזהו הפרש הגבהים המקסימלי שיכול להתקבל בעץ AVL בין הצמתים הנ"ל. לכן, ככל שגובה העץ גדל כך גדלה עלות ה-join המקסימלי. כלומר, נקבל כי עלות פעולת join מקסימלי במקרה זה תהיה . גדלי העצים איתם הרצנו את הניסוי גדלים יחסית בסדרי גודל שונים ולכן ניתן להבחין בהתייקרות הפעולה- ככל ש-n גדל משמעותית כך גדל עומק העץ והפרש הגבהים לפעולת ה-join גדל והיא בפועל יקרה יותר.