שם: ראזי עבד אלרחמאן

שם משתמש: razi1

תעודת זהות: 212185748

שם: מוחמד טאהא

שם משתמש: mohammedtaha

תעודת זהות: 211924196

Our classes:

We have two classes AVLNode and AVLTree we'll begin talking about the AVLNode class:

AVLNode

We added the following functions to the interface :

void setKey(int newKey): sets the current node's key to be the given newKey

void setValue(String NewValue): sets the current node's value to be the given newValue

int getRankDifference(): returns the rank difference of the current node, which is the height difference between the parent and the current node.

```
public int getRankDifference() {
        AVLNode parent = this.getParent();
        if (parent == null)
            return -1;
        else {
            int parentRank = parent.getHeight();
            int thisRank = this.getHeight();
            return parentRank - thisRank;
            }
        }
```

int getSize(): returns the size of current node

void setSize(int size):sets the size of current node

all these functions work in O(1) time complexity.

We also have the following fields and constructor:

```
private  AVLNode left: save the left node for current node
private  AVLNode right: save the right node for current node
private  AVLNode parent; save the parent node for current node
private int key: save key
private boolean info;
private int rank;
private int size;
private int numoftrue;
private AVLNode() {
}
public AVLNode(int key, boolean info) {
this.key = key;
this.info = info;
this.size = 1;
this.setLeft(EXTERNAL_LEAF);
this.setRight(EXTERNAL_LEAF);
EXTERNAL_LEAF.setSize(0); }
```

AVLTree

- We have the following class constant and the root field:

```
private final AVLNode EXTERNAL_LEAF = new AVLNode(); // we now have an external
leaf that we can always point at.
private IAVLNode root;// the root of the tree
```

- The constructor below works in this fashion:

  We add the appropriate rank, size key values to match the external leaf. All these actions require O(1)

```
public AVLTree() {
        // When generating a new empty tree, we make sure it only contains an
external leaf.
        // We also need to modify EXTERNAL_LEAF fields so they match the
requirements.
        this.EXTERNAL_LEAF.rank = -1;
        this.EXTERNAL_LEAF.key = -1;
        this.EXTERNAL_LEAF.size = 0;
        this.root = EXTERNAL_LEAF;
    }
```

  We create the root and initialize it to be equal to the external leaf.

```
private AVLTree(IAVLNode root) {
  this();
  this.root = root;
    }
```

- We added the following functions:
  - Private static int updateHeight(AVLNode node): to update the height of node, it updates the height according to its children's height, it runs in O(1) time complexity.
  - Private static int BalanceFactor(AVLNode node): calculate the BF of a given node.
  - Private static void attachNodes(AVLNode parent,IAVLNode child): we overwrite a one of parent's child nodes using child, if the child is an external leaf the function does nothing. At the end we update size of parent.
  - Private static void updateSize(AVLNode node): updates size of a given node according to its children's sizes.
  - Int rightRotation/leftRotation(AVLTree T, AVLNode node): we rotate left or right the given node, we update the size of each effected node by the rotation we update also their height and update the root if it was changed we return the number of operations. Time complexity O(1).
  - THE FOLLOWING FUNCTIONS ARE USED FOR INSERTIONS:
  - Private IAVLNode searchByKey(int k): return a node if said node is in the tree, if not returns null. Time complexity O(logn) when n is the size of the tree, since it works by the Binary Search principle.

- Private IAVLNode findInsertionPoint(int key): returns the last real node encountered while looking for the placement of k, if the key exists or the tree is empty we return null. Time complexity O(logn) since it is based on Binary search principle.
- Private int rebalanceFromNode(AVLNode): given a node we calculate the BF for this node and whether or not the height of this node matches the expected height which is the max of the two childern's height plus one. If it doen't match we rebalance according to the cases by calculating the bf of the appropriate child and then sending nodes to rotation functions. Time complexity O(logn).Even though, we only change a couple of pointers in each rotation if even needed, the worst case is us going all the way up to the root to promote the rank. Returns the number of operations of rotations promotions and demote.
- THE FOLLOWING FUNCTIONS ARE USED FOR DELETIONS:
- Private void swap(AVLNode node1, AVLNode node2): swaps two nodes with each other. Each one gets the key and value of the other. Time complexity O(1).
- Private IAVLNode sucessor(AVLNode node): returns the successor of given node, time complexity of O(logn)
- Private Boolean isLeaf(AVLNode Node): checks if given node is a leaf
- Private Boolean isUnary(AVLNode Node): checks if given node is an unary
- public int demote(AVLNode Node): demote node by one
- public int promote(AVLNode Node): promote node by one
- Private int deleteByNode(AVLNode node): This function deals with the three states of nodes, if the given node is leaf we replace it by an external leaf, if it is unary we replace it by it is child and attach it to the parent. This function never receives a binary node. Returns the number of operations needed to change the height. O(1) time complexity
- Private int rebalanceFromNode3(AVLNode node): rebalances tree after deletionm total time complexity is o(logn) in worst case, going all the way to the root.
- End of helping functions for deletion and insertions.
- Public Iterator< AVLNod> getInOrderIterator(): returns an iterator that goes over the tree nodes in ascending order. Works in O(logn) time complexity since it needs to find the minimal node first.

Time complexity for the original functions

- Public insert(int k,String i): calls upon the functions below one in the following order

  findInsertionpoint(..)➔ updateHeight➔ rebalanceFromNode3()

  the overall time complexty is O(logn)+O(1) +O(logn)=O(logn) in worst case.

- Public delete(int k): calls upon the functions below one in the following order

  searchByKey() ➔ isUnary(..) ➔ isleaf(..)➔deleteByNode➔ rebalanceFromNode3 or

  searchByKey() ➔ isUnary(..) ➔ isleaf(..)➔successor➔swap➔ deleteByNode➔ rebalanceFromNode3.

  the overall time complexty is O(logn)+O(1)+O(1) +O(logn)+O(1)+O(1)+ O(logn)+=O(logn) in worst case.

- Public int[] keysToArray,public String[] infoToArray: calls for the getInOrderIterator function then calls for the iterator members O(logn).

prefixXOR(int k) : we start from the node it self, after we search for it,going up to the root, with the help of the field that we called numoftrue which saved the number of true values in a sub tree of this root (the root included)

so it goes up til the root with log(n) complexity

succPrefixXor(int k) : we start from the smallest key,after we search for it, and we do successor every time till we reach the node we want to calcite its prefix, so this might take O(nlogn) if we do to the largest value for instance

מדידות

שאלה 1:

מספר סידורי	עלות prefixXor ממוצעת (כל הקריאות)	עלות succPrefixXor ממוצעת (כל הקריאות)	עלות prefixXor ממוצעת (100 קריאות ראשונות)	עלות succPrefixXor ממוצעת (100 קריאות ראשונות)
1	1892	9031	3560	9288
2	1301	18293	2693	8498
3	966	20496	2027	7635
4	887	14281	2577	8733
5	700	15214	2198	7531

לדעתנו הניתוח התיאורטי כן מתיישב עם התוצאות שקיבלנו בטבלה שמלמעלה, נוכל לראות שבעמוד הראשון והשני, ככל שהתקדמנו יותר מבחינת איברים הפער הולך וגדל, כלומר זה מראה לנו את ההבדל בין

O(nlogn) שזה לפונקציית succ לבין ה prefixXor שזה O(log n)

ולגבי העמוד השלישי והרביעי זה אכן המצב,

כלומר בסוף אחרי שהרצנו את התוכנית היה לנו ברור מבחינה מעשית מה ההבדל בין השתי סוביכיות השונות, כי ב succ צריך להתחיל מהערך הקטן וכשכל שמתקדמים זמן הריצה הולך וגדל, לעומת pre שפועלת ב לוג אן כי מתחילים מהצומת ועולים לכל היותר עד השורש שזה לוג אן ,

ולגבי העמוד השני והרביעי, ברור שזמן ה 100 קריאות ב succ יהיה יותר טוב מזה של כל האיברים, כי כמו שהסברתי שאם לוקחים את ה 100 איברים קטנים אנחנו יחסית לא עושים הרבה עבודה כמו אם שלוקחים את כל האיברים ומתחילים מהקטן, זה למה הזמנים בעמוד הרביעי יותר טוב מהעמוד השני.

שאלה 2 :

	עץ AVL סדרה חשבונית	עץ ללא מנגנון איזון סדרה חשבונית	עץ AVL סדרה מאוזנת	סדרה AVL מאוזנת עץ ללא מנגנון איזון סדרה מאוזנת	עץ AVL סדרה אקראית	עץ ללא מנגנון איזון סדרה אקראית
1	4104	8853	8762	9615	5091	2004
2	2290	6629	2234	1336	2663	1455
3	1566	7660	2132	1300	2169	1224
4	1318	6816	2041	1213	1767	1056
5	1236	10144	1920	1021	1622	888

הסדרה החשבונית:

זה כן מתיישב עם הניתוח התיאורטי, אנחנו רואים בטבלה שהעץ בלי איזון עובד עם זמן פחות טוב מ avl כי אין מנגנון איזון ובסדרה חשבונית למשל מ 1 עד 1000, אורך העץ יהיה אחרי ההכנסה 1000, לעמות לוג 1000 ב avl , אז ככל שמכניסים יותר איברים זמן ההכנס בממוצע של העץ בלי איזון הולך ונהיה יותר ויותר גרוע בפער יותר גדול.

הסדרה המאוזנת:

נשים לב שפה אין צורך במנגנון איזון, כי בכל מקרה העץ יהיה מאוזן, ובעצם עץ ה avl עושה עבודה קצת נוספת, זה למה בטבלה אנו רואים שזמן העץ בלי מנגנון איזון קצת יותר טוב, אז גם פה תוצואות הטבלה מתיישבות עם הניתוח התיואורטי

סדרה אקראית:

אנו רואים פה שהעץ בלי מנגנון האיזון מנצח את ה avl , וזה אכן מתיישב עם הניתוח התיאוריטי, כי כמו שהוכחנו בכיתה שאם נכניס מפתחות באקראי ונעשה הכנסה זמן ההכנסה נהיה קצת יותר טוב מהכנסה ב avl , נשים לב שאין הבדל כל כך גדול, אבל עדיין צריך להגיד שבאקראי מנצח לפי התוצאות, ייתכן גם מצבים ש ה avl ינצח כמו שראינו בטבלה ובכיתה,