מבני נתונים

תרגיל מעשי 1

תיעוד חיצוני

איוב אטרש – 318476611

מוחמד עאמר - 208644963

|  |  |  |
| --- | --- | --- |
| **Complexity** | **Explanation** | **Function** |
| O(1) | returns true if and only if the tree is empty. | public boolean empty() |
| O(log(n)) | returns the info of an item with key k if it exists in the tree otherwise, returns null . | public String search(int k) |
| O(log(n)) | inserts an item with key k and info i to the WAVL tree. the tree must remain valid (keep its invariants) so we call I\_Rebalance. returns the number of rebalancing operations, or 0 if no rebalancing operations were necessary. returns -1 if an item with key k already exists in the tree | public int insert(int k, String i) |
| O(log(n)) | deletes an item with key k from the binary tree, if it is there; the tree must remain valid (keep its invariants).  returns the number of rebalancing operations, or 0 if no rebalancing operations were needed.  returns -1 if an item with key k was not found in the tree .  Every delete function (deleteUnary , deleteBinary , deleteLeaf) calls D\_Rebalance , therefore there complexity is O(log(n)) . | public int delete(int k) |
| O(log(n)) | Returns the info of the item with the smallest key in the tree, or null if the tree is empty. | public String min() |
| O(log(n)) | Returns the info of the item with the largest key in the tree, or null if the tree is empty. | public String max() |
| O(n) | Returns a sorted array which contains all keys in the tree or an empty array if the tree is empty | public int[] keysToArray() |
| O(n) | Returns a sorted array which contains all value in the tree or an empty array if the tree is empty | public String[] infoToArray() |
| O(1) | Returns the number of nodes in the tree | public int size() |
| O(n) | This function returns the info of the i-th largest key in the tree otherwise it returns null . | public String select(int i) |
| O(1) | Return the root of the tree or null if the tree is empty | public WAVLNode getRoot() |
| O(log(n)) | Rebalances the tree starting from “node” after an insert is accomplished  and then returns the number of rebalancing steps . | Public int I\_Rebalance(WAVLNode node) |
| O(1) | Rotates the given node left or right depending on the string s. | Public void Rotate(WAVLNode node,String s) |
| O(log(n)) | Return the node with key k if it exists in the tree, otherwise return the right position to insert | Public WAVLNode Position(int k) |
| O(log(n)) | This function only activates the in order traversal of the tree. Returns the sorted array of nodes . | public WAVLNode[] activate\_in\_order(WAVLNode r) |
| O(log(n)) | In order traversal of the tree. Prepares the array that the above function returns . | public int in\_order(WAVLNode r,int i,WAVLNode[] arr) |
| O(log(n)) | Returns the successor of the node | public WAVLNode Successor(WAVLNode node) |
| O(log(n)) | Starts with node and resets its size depending on its children and moves up doing the same thing until it reaches the root | private void reSetTreeSize(WAVLNode node) |
| O(1) | A constructor for external node that initializes a virtual node by initializing a rank to -1 , size to be 0. | public WAVLNode() |
| O(1) | A constructor that initializes a real node with a : left child, right child , parent , key and value of the node. | public WAVLNode(WAVLNode left,WAVLNode parent ,WAVLNode right, int k,String v) |
| O(1) | Returns the node key by accessing the key field . | public int getKey() |
| O(1) | Returns the value of the node by accessing the value field. | public String getValue() |
| O(1) | Returns the left child of the node by accessing the left field . | public WAVLNode getLeft() |
| O(1) | Returns the right child of the node by accessing the right field . | public WAVLNode getRight() |
| O(1) | Checks whether the node is inner or external by checking its rank. | public boolean isInnerNode() |
| O(1) | Returns the sub-tree size of the node by accessing the size field. | public int getSubtreeSize() |
| O(1) | Sets the sub-tree size of the node by accessing the childs size field. | private void setSubtreeSize() |
| O(1) | Checks whether a node is a leaf or not a leaf . | public boolean isLeaf() |
| O(1) | Checks whether a node is a unary node or not a unary node . | public boolean isUnary() |
| O(1) | Returns the rank of the node by accessing the rank field. | public int getRank() |
| O(log(n)) | Deletes a given binary node from the tree and returns the number of rebalancing steps that occurred :  1)Calls the Successor(WAVLNode) with the deleted node to find its successor  2)Replace the deleted node to be the successor using Replace function .  3) delete the node using deleteLeaf if it is a leaf after replacement or deleteUnary if it's an unary node . | private int deleteBinary(WAVLNode node) |
| O(1) | Receives two nodes : WAVLNode node and WAVLNode suc ; replaces node with suc by changing the indicators of these nodes . | private void Replace(WAVLNode node, WAVLNode suc) |
| O(log(n)) | Deletes a given unary node from the tree and returns the number of rebalancing steps that occurred . | private int deleteUnary(WAVLNode node) |
| O(log(n)) | Deletes a given leaf node from the tree and returns the number of rebalancing steps that occurred . | private int deleteLeaf(WAVLNode node) |
| O(log(n)) | Rebalances the tree starting from “node” after an delete is accomplished  and returns the number of rebalancing steps that occurred . | private int D\_Rebalance(WAVLNode node) |

**שדות ב – WAVLTree :**

|  |  |
| --- | --- |
| **Details** | **Parameters** |
| Static node that used for external nodes. | final WAVLNode ext\_leaf |
| The root of the tree . | WAVLNode root |

**שדות ב – WAVLNode :**

|  |  |
| --- | --- |
| **Details** | **Parameters** |
| The key of the node. | int key |
| The info of the node. | String value |
| The left child of the node. | WAVLNode left |
| The right child of the node. | WAVLNode right |
| The parent of the node. | WAVLNode parent |
| The rank of the node. | int rank |
| The size of the node. | int size |

**מדידות :**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| מספר סידורי | מספר פעולות | מספר פעולות האיזון הממוצע לפעולת  insert | מספר פעולות האיזון הממוצע לפעולת delete | מספר פעולות האיזון המקסימלי לפעולת insert | מספר פעולות האיזון המקסימלי לפעולת delete |
| 1 | 10,000 | 3.4061 | 2.7505 | 17 | 10 |
| 2 | 20,000 | 3.4040 | 2.7494 | 16 | 11 |
| 3 | 30,000 | 3.4125 | 2.7284 | 17 | 12 |
| 4 | 40,000 | 3.4356 | 2.7396 | 18 | 12 |
| 5 | 50,000 | 3.4009 | 2.7371 | 18 | 12 |
| 6 | 60,000 | 3.4039 | 2.7368 | 18 | 12 |
| 7 | 70,000 | 3.4157 | 2.7383 | 18 | 12 |
| 8 | 80,000 | 3.4231 | 2.7408 | 18 | 11 |
| 9 | 90,000 | 3.4109 | 2.7452 | 19 | 12 |
| 10 | 100,000 | 3.4145 | 2.7386 | 19 | 12 |

ציפינו לקבל בטבלה שמספר פעולות האיזון הממוצע לפעולות ההכנסה והמחיקה (amort(insert/delete)) יהיה קבוע ושמספר פעולות האיזון המקסימלי והמינימלי יהיה חסום על ידי גובה מקסימלי של עץ WAVL מאוזן שזה שווה בערך 2\* כאשר n הוא מספר האיברים בעץ שלנו .

התוצאות שקיבלנו בטבלה הנ"ל **תואמות** למה שצפינו כי מספר פעולות האיזון הממוצע לפעולת insert ו-delete נשאר קבוע וקל לראות ששמספר פעולות האיזון המקסימלי לפעולת ההכנסה והמחיקה חסום על ידי 2\* בערך .

משמעות המדידות שביצענו היא וודוי שבאמת מתקיים amort(insert)=amort(delete)=O(1) וכי מספר פעולות האיזון איננו חורג מגובה העץ . זה מוכיח שעבור הרבה פעולות מחיקה, עץ WAVL עדיף יותר מאשר עץ AVL שלא מקיים התכונה הזו .