Names: Tom Lapid, Paz Reingold
User Names: tomlapid, pazreingold
Id's: 203135256, 208209734

Documentation

public class RBNode

Class RBNode's objects represent the Red-Black tree nodes.
Class RBNode is an inner class of class RBTree.

Class fields:

private int key – contains the key of the node. Can only be a non-negative integer. Initiated by the user in the Class's constructor.

private String value - contains the value of the node. Initiated by the user in the Class's constructor.

private RBNode parent - contains the parent of the node. Default value: null.

private RBNode rightChild - contains the right child of the node. Default value: null.

private RBNode leftChild - contains the left child of the node. Default value: null.

private String Color - contains the color of the node.
Optional values: Red/Black**.** Default value: Black.

Class methods:

Every single method of the followings has complexity O(1).

public RBNode (int key, String value) – RBNode class constructor. Initiating the key & value fields to the given parameters.

Getters –

public getRight() – returns the node ' s right child .

public getLeft() – returns the node ' s left child.

public getParent() – returns the node ' s parent.

public getColor() – returns the node ' s color.

public getValue() – returns the node ' s value.

public getKey() – returns the node ' s key.

Setters –

private setRight (RBNode rightChild) – setting the new right child of the node to the given node.

private setLeft (RBNode leftChild) – setting the new left child of the node to the given node.

private setParent(RBNode parent) – setting the new parent of the node to the given node.

private setColor(String color) – setting the new color of the node to the given color. Can only set to Red/Black.

private setValue(String value) – setting the new value of the node to the given value.

private setKey(int key) – setting the new key of the node to the given key.

Additional-

public String Print() – printing the node (by key value and color). Used for internal checking.

public class RBTree

Class RBTree's objects represent the Red-Black trees.

Class fields:

private RBNode root - contains the root of the tree, refers to RBNode object. Must be Black.
Default value: null.

private RBNode Max_Value - contains the RBNode with maximal key in the tree.
Default value: null.

private RBNode Min_Value - contains the RBNode with minimal key in the tree.
Default value: null.

private int Tree_size - contains the number of the nodes in the tree . Default value: 0.

private RBNode leaf – contains the leaf node, if a node in the tree doesn't have a child, the leaf node become his child. The leaf node children both nulls, his parent is changing, the key has constant value of -1, and the value has constant value of null.

Class methods:

public RBTree() - empty constructor , initialize an empty tree.

public RBTree(int key , String value) – a constructor which initialize a tree and his root by the given parameters using the insert method.

public RBNode getRoot() - returns the root of the red black tree.
Complexity - **O (1),** since it's using the root field.

public boolean empty() - returns true if and only if the tree is empty (checks if the root field equals null).
Complexity - **O (1)**, since it's using the root field.

public String search(int k) - returns the value of an item with key k if it exists in the tree. Otherwise, returns null.
Calls to: inner search.
Complexity - **O (logn)** – since it has a helper method with O (logn) Complexity.

private RBNode inner_search(RBNode x,int k) – returns the node with the key k if it exists in the tree. Otherwise, returns the leaf node.
Complexity - **O (logn)** – since it's a recursive function running through the height of a balance binary tree.

private void replace_leftChild (RBNode x,RBNode y) - replacing the left child of x with y.
Complexity - **O (1).**

private void replace_rightChild (RBNode x,RBNode y) - replacing the right child of x with y.
Complexity - **O (1).**

private void Transplant(RBNode x,RBNode y) - transplanting y instead of x in the tree .
Complexity - **O (1).**

private void Left_Rotate(RBNode x) - rotates the subtree of x to the left by the rules we learned in the lecture.
Calls to: Transplant, replace_leftChild , replace_rightChild.
Complexity - **O (1)** since all the helper methods are complexity O (1).

private void right_Rotate(RBNode x) - rotates the subtree of x to the right by the rules we learned in the lecture.
Calls to: Transplant, replace_leftChild , replace_rightChild.
Complexity - **O (1)** since all the helper methods are complexity O (1).

private RBNode RBTree_Position(RBNode T,int z) - returns the node with the key z if it exists in the tree. Otherwise, returns z's future parent.
Complexity - **O (logn)** - since it's a recursive function running through the height of a balance binary tree.

private int RB_Tree_InsertFixup(RBNode root,RBNode z) - returns the number of color switches that have been done during repairing the tree after inserting a node.
The repairing of the tree is done according to the algorithm taught in lecture goes as following:
As long as z's parent exits and red the method deals with three possible cases, symmetrical according to "z"'s parent (right/left child) -
1. z's uncle w is red.
2. z's uncle w is black, z is a right child.
3. z's uncle w is black, z is a left child.

With each case we perform the appropriated color switches, counting, and rotations to keep a valid RBTree.
Calls to: Left_Rotate, right_Rotate.
Complexity - **O (logn)** - since it's an iterated function running through the height of a balance binary tree.

public int insert(int k, String v) - inserts an item with key k and value v to the red black tree.
The tree must remain valid (keep its invariants).
Returns the number of color switches, or 0 if no color switches were necessary.
First, if the tree is empty we create (through the given parameters) the inserted RBNode ("z") and put it as the tree's root (returns 1).
Else, the method finds z's parent ("y"), using the RBTree_Position helper method, (returns -1 if an item with key k already exists in the tree).
Now, the method inserts "z" as "y"'s child (left/right child) and sends "z" and the root to the RB_Tree_InsertFixup helper method in order to keep a valid RBTree.
During the method we update the Tree's relevant fields.
Calls to: RBTree_Position, RB_Tree_InsertFixup.
Complexity - **O (logn)** - since it has O(logn) helper methods.

private int RB_Tree_DeleteFixup(RBNode x) – returns the number of color switches that have been done during repairing the tree after deleting a node.
The repairing of the tree is done according to the algorithm taught in lecture goes as following:
As long as the given parameter "x" is not black and is not the Tree's root the method deals with four possible cases, symmetrical according to x(right/left child) -
1. x's sibling w is red.
2. x's sibling w is black, and both children of w are black.
3. x's sibling w is black, w's left child is red, and w's right child is black.
4. x's sibling w is black, and w's right child is red.

With each case we perform the appropriated color switches, counting, and rotations to keep a valid RBTree.
Calls to: Left_Rotate, Right_Rotate.
Complexity - **O (logn)** - since it's an iterated function running through the height of a balance binary tree.

public int delete(int k) - deletes an item with key k from the binary tree, if it is there.
The tree must remain valid (keep its invariants).
First, the method find the RBNode ("z") contains the key k using the inner_search helper method, and returns -1 if the key k was not found in the tree.
If the key was found, we got two main cases:
1. "z" has two children – in this case we find its successor ("y") using the Successor helper method, then we transfer his data to "z" ,delete "y" and if "y" was black we send his child ("x") to the RB_Tree_ DeleteFixup helper method in order to keep a valid RBTree.
2. "z" has at most one child - in this case we define y=z, delete "y" and if "y" was black we send his child ("x") to the RB_Tree_ DeleteFixup helper method in order to keep a valid RBTree.
During the method we update the Tree's relevant fields.
Calls to: inner_search, RB_Tree_ DeleteFixup, Successor, PreDecessor.
Complexity - **O (logn)** - since it has logn helper methods.

public RBNode getLeaf() - returns the leaf node, for internal testing.
Complexity - **O (1)**

public int getMinKey() - - returns minimal key in the RBTree, for internal testing.
Complexity - **O (1)**

public String min() – returns the value of the item with the smallest key in the tree. Using Min_Value field and because of that has complexity O (1).

public String max() – returns the value of the item with the largest key in the tree. Using Max_Value field and because of that has complexity O (1).

public int[] keysToArray() - returns a sorted array which contains all keys in the tree, or an empty array if the tree is empty **.**
Complexity is O(n) since its using the next detailed method which has complexity is O(n)

private int keysToArray_inner(RBNode x,int[]arr,int index) – recursive method that returns the current index in the array which is going to be updated with the relevant key and updates the key array while going through the RBTree.
complexity is O(n) because it runs recursively through every single node in the tree once.

public String[] valuesToArray() - returns an array which contains all values in the tree **,** sorted by their respective keys **,** or an empty array if the tree is empty.
Complexity is O(n) since its using the next detailed method which has complexity is O(n)

private int valuesToArray_inner(RBNode x,String[]arr,int index) - – recursive method that returns the current index in the array which is going to be updated with the relevant value and updates the value array while going through the RBTree.
complexity is O(n) because it runs recursively through every single node in the tree once.

public int size() -returns the number of nodes in the tree. Complexity is **O (1)** since its using the field Tree_size

מספר סידורי	מספר פעולות	מספר החלפות צבע ממוצע לפעולת insert	מספר החלפות צבע ממוצע לפעולת delete
1	10,000	2.3219	2.4645
2	20,000	2.3193	2.47125
3	30,000	2.2995	2.4645666666666667
4	40,000	2.31735	2.464825
5	50,000	2.32228	2.46226
6	60,000	2.3142166666666667	2.4623166666666667
7	70,000	2.3105714285714285	2.468057142857143
8	80,000	2.3199375	2.4690875
9	90,000	2.312977777777778	2.4628111111111113
10	100,000	2.31453	2.46916

מכיוון שלמדנו בכיתה שזמן האמורטייזד לפעולת מחיקה ופעולת הוספה בעץ אדום שחור הוא O(1) ומכיוון שמספר שינוי הצבע כמובן חסום ע'י מספר הפעולות הכולל, אז גם הוא O(1) ולכן ציפינו לראות שאין תלות בין גודל העץ למספר שינויי הצבע הממוצע.

הציפייה שלנו אכן התממשה. בתוצאות ,כפי שניתן לראות בטבלה, ממוצעי החלפות הצבע בהכנסות ובמחיקות הינו דומה (בין 2 ל-3) ואינו תלוי בגודל העץ.