Assignment 2: Coding Question

● Graded

Student

Tyler NGUYEN

Total Points

100 / 100 pts

Autograder Score

30.0 / 30.0

Passed Tests

Test empty() (1/1)
Test randomly generated elements (8/8)
Test member() with delete() (2/2)
Whitebox test (16/16)
Test member() (1/1)
Test tiny tree { 2, 1, 3 } (2/2)

Question 2

Manual Grading

70 / 70 pts

✔ **– 0 pts** Correct

– 70 pts Late submission

– 70 pts Incorrect or can not be complied

– 33 pts Only pass 14/30 cases

– 42 pts Only pass 12/30 cases

– 56 pts Only pass 6/30 cases

– 60 pts Only pass 4/30 cases

– 63 pts Only pass 3/30 cases

– 65 pts Only pass 2/30 cases

– 68 pts Only pass 1/30 cases

+ 1 pt Provide a constructor that does not take any arguments and initialize the tree to an empty one.

+ 1 pt Implement empty

+ 1 pt Implement insert

+ 1 pt Implement delete

+ 1 pt Implement member

+ 1 pt Implement toString()

+ 1 pt Implement preorder tree walk

Autograder Results

Test empty() (1/1)

Test randomly generated elements (8/8)

Test member() with delete() (2/2)

Whitebox test (16/16)

Test member() (1/1)

Test tiny tree { 2, 1, 3 } (2/2)

Submitted Files

RedBlackTree.java

Download

```java
/*
author: Tyler Nguyen UCID: 30158563
date: October 28, 2023
description: RedBlackTree class that implements Dictionary and its methods as well as some helper functions
 */


package ca.ucalgary.cpsc331.a2;

public class RedBlackTree implements Dictionary {

    private static final boolean RED = true; //constant to represent the colour red
    private static final boolean BLACK = false; //constant to represent the colour black

    private class Node {
        Node parent;
        Node left;
        Node right;
        boolean red;
        int key;

        /**
         * constructor to create a new node
         * @param key, which is the value to be stored in the node
         */
        Node(int key) {
            this.key = key;
            this.red = RED;
            this.parent = null;
            this.left = NIL;
            this.right = NIL;
        }
    }
    private Node root;
    private Node NIL; //will act as sentinel for the tree

    /**
     * constructor for the Red Black Tree
     * initializes the tree with a NIL node
     */
    public RedBlackTree() {
        NIL = new Node(-1);
        NIL.red = BLACK;
        NIL.left = null;
        NIL.right = null;
        root = NIL;
    }

```

```
49      /**
50       * will check if the tree is empty
51       *
52       * @return true if the tree is indeed empty and false if not empty
53       */
54      @Override
55      public boolean empty() {
56          return root == NIL;
57      }
58
59      /**
60       * will search the tree to find the node with a given key through recursion
61       *
62       * @param node the node that is being searched
63       * @param key the value that is being searched
64       * @return the node that contains the that we are searching for if found and NIL if cant be found
65       */
66      private Node search(Node node, int key) {
67          if (node == NIL) { //checking if the node is empty/NULL
68              return NIL;
69          }
70          if (key < node.key) {
71              return search(node.left, key); //search in the left subtree of the current node
72          } else if (key > node.key) {
73              return search(node.right, key); //search in the right subtree of the current node
74          }
75          return node;  //current node's key is the key being searched
76      }
77
78      /**
79       * does a left rotation on the given node x in the tree
80       *
81       * @param x the that is going to be rotated
82       */
83      private void rotateLeft(Node x) {
84          Node y = x.right; //storing the nodes right child into y
85          x.right = y.left;
86          if (y.left != NIL) { //checking if y's left child is empty/NULL
87              y.left.parent = x;
88          }
89          y.parent = x.parent;
90          if (x.parent == null) {
91              root = y;
92          } else if (x == x.parent.left) { //if x is a left child of its parent then update x's parents
93                                           //left child to be y
94              x.parent.left = y;
95          } else {
96              //otherwise x is a right child of its parent then update x's parents right child to be y
97              x.parent.right = y;
98          }
99          y.left = x; //make x to be the left child of y
100         x.parent = y; //make x to have y as its parent
```

```
101     }
102
103     /**
104      * does a right rotation on the given node y in the tree
105      *
106      * @param y the that is going to be rotated
107      */
108     private void rotateRight(Node y) {
109         Node x = y.left; //storing the nodes left child into x
110         y.left = x.right;
111         if (x.right != NIL) { //checking if x's right child is empty/NULL
112             x.right.parent = y;
113         }
114         x.parent = y.parent;
115         if (y.parent == null) {
116             root = x;
117         } else if (y == y.parent.right) { //if y is a right child of its parent then update y's
118                                           //parents right child to be x
119             y.parent.right = x;
120         } else {
121             //otherwise y is a left child of its parent then update y's parents left child to be x
122             y.parent.left = x;
123         }
124         x.right = y; //make y to be the right child of x
125         y.parent = x; //make y to have x as its parent
126     }
127
128     /**
129      * fixes the possible red black tree property violations after the insertion of a new node k
130      *
131      * @param k the newly inserted node that may violate the red black tree properties
132      */
133     private void insertFixUp(Node k) {
134         Node u; //uncle node
135         while (k.parent != null && k.parent.red) { //fixing tree while the parent of k is a red colour
136             if (k.parent == k.parent.parent.right) { //if the parent of k's parent is the right child
137                                                      //of its parent
138                 u = k.parent.parent.left;
139                 if (u.red) { //case 1, where the uncle is red, thus violating the red black tree property
140                     u.red = BLACK;
141                     k.parent.red = BLACK;
142                     k.parent.parent.red = RED;
143                     k = k.parent.parent;
144                 } else { //case 2, where k is the left child of its parent
145                     if (k == k.parent.left) {
146                         k = k.parent;
147                         rotateRight(k);
148                     } //case 3, where k is the right child of its parent
149                     k.parent.red = BLACK;
150                     k.parent.parent.red = RED;
151                     rotateLeft(k.parent.parent);
152                 }
```

```
153             } else { //same as before cases but when k's parent is the left child of its parent
154                 u = k.parent.parent.right;
155 
156                 if (u.red) {
157                     u.red = BLACK;
158                     k.parent.red = BLACK;
159                     k.parent.parent.red = RED;
160                     k = k.parent.parent;
161                 } else {
162                     if (k == k.parent.right) {
163                         k = k.parent;
164                         rotateLeft(k);
165                     }
166                     k.parent.red = BLACK;
167                     k.parent.parent.red = RED;
168                     rotateRight(k.parent.parent);
169                 }
170             }
171             if (k == root) { //exit the loop if the root is reached
172                 break;
173             }
174         }
175         root.red = BLACK; //enforces the fact that the root of the tree is black to preserve the
176                           //red black tree property
177     }
178 
179     /**
180      * inserts a new node with the given key k into the tree
181      *
182      * @param k the key that is to be inserted
183      * @return true only if the insertion is successful and false the key trying to be inserted
184      *         already exists in the tree
185      */
186     @Override
187     public boolean insert(int k) {
188         Node node = new Node(k); //creates a new node for the input key
189         if (search(this.root, k) != NIL) { //checks if the input key already exists
190             return false; //key already exists so insertion fails
191         }
192 
193         //standard binary search tree insert procedure
194         Node temp = root;
195         Node parent = null;
196         while (temp != NIL) {
197             parent = temp;
198             if (node.key < temp.key) {
199                 temp = temp.left;
200             } else {
201                 temp = temp.right;
202             }
203         }
204         node.parent = parent; //setting the parent of the newly created node
```

```
205
206         if (parent == null) {
207             root = node; //now if tree is empty then make the new node the root
208         } else if (node.key < parent.key) {
209             parent.left = node; //if new nodes is less than the parent then insert it as the left child
210         } else {
211             parent.right = node; //otherwise insert it as the right child since it is greater than parent
212         }
213
214         node.left = NIL; //initializing the new nodes children to NIL
215         node.right = NIL;
216
217
218         insertFixUp(node); //fixing the possible violations of the red black tree properties
219                            //after the insertion
220         return true; //insertion is successful
221     }
222     /**
223      * fixes the possible red black tree property violations after the deletion of a new node x
224      *
225      * @param x the node for where the red black tree properties need to be fixed
226      */
227     private void deleteFix(Node x) {
228         Node s; //sibling node
229         while (x != root && x.red == BLACK) { //fixing tree while x is not the root and is black
230             if (x.parent != null && x == x.parent.left) {
231                 s = x.parent.right;
232                 //enforcing the sentinel is correct if needed
233                 if (s == null){
234                     s = NIL;
235                 }
236                 if (s.left == null){
237                     s.left = NIL;
238                 }
239                 if (s.right == null){
240                     s.right = NIL;
241                 }
242                 if (s.red == RED) { //case 1: the sibling is red so we need to rotate it left and
243                                     //recolour
244                     s.red = BLACK;
245                     x.parent.red = RED;
246                     rotateLeft(x.parent);
247                     s = x.parent.right;
248                 }
249
250                 if (s.left.red == BLACK && s.right.red == BLACK) {
251                     //case 2: both the left and right child of the sibling are black so we have
252                     //to move the violation up the tree
253                     s.red = RED;
254                     x = x.parent;
255                 } else {
256                     if (s.right.red == BLACK) {
```

```
                //case 3: right child of sibling is black so we need to rotate it right and
                //recolour
                s.left.red = BLACK;
                s.red = RED;
                rotateRight(s);
                s = x.parent.right;
            }
            //case 4: keep recolouring and rotate left
            s.red = x.parent.red;
            x.parent.red = BLACK;
            s.right.red = BLACK;
            rotateLeft(x.parent);
            x = root;
        }
    } else if (x.parent != null) { //same as before cases but when x is a right child of its parent
        s = x.parent.left;
        if (s == null){
            s = NIL;
        }
        if (s.left == null){
            s.left = NIL;
        }
        if (s.right == null){
            s.right = NIL;
        }
        if (s.red == RED) {
            s.red = BLACK;
            x.parent.red = RED;
            rotateRight(x.parent);
            s = x.parent.left;
        }

        if (s.right.red == BLACK && s.left.red == BLACK) {
            s.red = RED;
            x = x.parent;
        } else {
            if (s.left.red == BLACK) {
                s.right.red = BLACK;
                s.red = RED;
                rotateLeft(s);
                s = x.parent.left;
            }

            s.red = x.parent.red;
            x.parent.red = BLACK;
            s.left.red = BLACK;
            rotateRight(x.parent);
            x = root;
        }
    }
}
x.red = BLACK;
```

```
309     }
310
311     /**
312      * gets the node which has the minimum key in the tree at the given node
313      *
314      * @param node the root node of the subtree to find the minimum key
315      * @return the node that has the minimum key in the subtree
316      */
317     private Node minimum(Node node) {
318         while (node.left != NIL) { //traverses to the left most child
319             node = node.left;
320         }
321         return node; //returns the minimum key in the subtree
322     }
323
324     /**
325      * replaces the subtree that is rooted at given node u with other subtree routed at given node v
326      * in the red black tree
327      *
328      * @param u the node that has the subtree that will be replaced
329      * @param v the node that has the subtree that will replace u's subtree
330      */
331     private void transplant(Node u, Node v) {
332         if (u.parent == null) { //if the given node u is the root then set v to the root
333             root = v;
334         } else if (u == u.parent.left) { //if the given node u is a left child of its parent
335                                          //then make the left child of u's parent to v
336             u.parent.left = v;
337         } else { //if the given node u is a right child of its parent then make the right child of
338                  //u's parent to v
339             u.parent.right = v;
340         }
341         v.parent = u.parent; //now update the parent of v to be the parent of u
342     }
343
344     /**
345      * delete the node with key k from the red black tree
346      *
347      * @param k the key of the node that is to be deleted
348      * @return true if sucessful in deletion and otherwise false if the key does not exist
349      */
350     @Override
351     public boolean delete(int k) {
352         Node node = search(root, k); //searching for the given input k in the tree
353         if (node == NIL) {
354             return false; //key does not exist in the tree so deletion will fail
355         }
356
357         Node y = node;
358         boolean yOriginalColor = y.red;
359         Node x;
360
```

```
361
362         if (node.left == NIL) { //case of where the node has either one child either left or right
363             x = node.right;
364             transplant(node, node.right);
365         } else if (node.right == NIL) {
366             x = node.left;
367             transplant(node, node.left);
368         } else { //case of where the node has both its left and right children
369             y = minimum(node.right);
370             yOriginalColor = y.red; //need to keep the original colour of y before changes
371             x = y.right;
372             if (y.parent == node) {
373                 x.parent = y;
374             } else {
375                 transplant(y, y.right);
376                 y.right = node.right;
377                 y.right.parent = y;
378             }
379             transplant(node, y);
380             y.left = node.left; //connecting y to the left subtree of node
381             y.left.parent = y;
382             y.red = node.red; //preserving the colour of node in y
383         }
384         if (yOriginalColor == BLACK) { //now fix any red black tree properties if the orignal
385                                        //colour of y was black
386             deleteFix(x);
387         }
388         return true; //successful deletion
389     }
390
391     /**
392      * determines if the node that has the key k exists in the red black tree
393      *
394      * @param k the key to search for in the tree
395      * @return true if k was found in the tree and otherwise false if not found
396      */
397     @Override
398     public boolean member(int k) {
399         return search(root, k) != NIL; //uses the search helper method to find if key k exists
400     }
401
402     /**
403      * will create a string representation of the red black tree that uses preorder traversal
404      *
405      * @return the representation of the red black tree that is in a form of a string
406      */
407     @Override
408     public String toString() {
409         StringBuilder result = new StringBuilder(); //making a StringBuilder to represent the string
410         preOrder(root, "*", result); //uses the preOrder helper method to
411         return result.toString();
412     }
```

```
413
414     /**
415      * achives a preorder traversal of the red black tree and constructs a string representation of
416      * the structure of the tree with its node colours
417      *
418      * @param node the current node that is in the traversal
419      * @param address the address string which indicates the path to the current node
420      * @param result the result of the tree's string representation
421      */
422     private void preOrder(Node node, String address, StringBuilder result) {
423         if (node != NIL) { //if the node is not NIL then it appends the node's address, its colour
424                            //either red or black and the key to the result
425             result.append(address).append(":");
426             result.append(node.red ? "red:" : "black:").append(node.key).append("\n");
427             preOrder(node.left, address + "L", result); //traverses the left subtree
428             preOrder(node.right, address + "R", result); //traverses the right subtree
429         }
430     }
431 }
432
```