

Is It A Red-Black Tree

黄文杰

3210103379

Date: 2023-11-1

Chapter 1: Introduction

A balanced binary search tree known as a red-black tree is found in data structures. It possesses the following five characteristics:

- Each node is either red or black.
- The root is black.
- Every leaf (NULL) is black.
- If a node is red, both of its children are black.
- For any given node, all straightforward paths from that node to its descendant leaves have an equal number of black nodes.

In this context, our challenge is to determine whether a given binary search tree adheres to the properties of a Red-Black Tree. This is a critical problem, as the validity of Red-Black Trees directly impacts their applications in computer science, including databases, operating systems, compilers, and more. This report will comprehensively describe how our program determines whether a given binary search tree is a valid Red-Black Tree, highlighting the underlying principles and importance of this process.

Chapter 2: Algorithm Specification

- Structure of a red-black tree node

```
1 // Structure for a Red-Black Tree node
2 typedef struct RBTreeNode {
3     int value;           // Node's key
4     int color;           // Node's color, can be red (RED) or black (BLACK)
5     struct RBTreeNode* left; // Left child
6     struct RBTreeNode* right; // Right child
7 } Node;
```

- createNode(int value, int color)

```
1 Node* createNode(int value, int color)
2     newNode <- allocate memory for a new RBTreeNode
3     newNode.value <- value
4     newNode.color <- color
5     newNode.left <- NULL
6     newNode.right <- NULL
7     return newNode
```

- Parameter Description :

- Input Parameters: An integer `value` representing the node key, an integer `color` representing the node color.
- Output: Returns a pointer to a new node.

- Function's purpose: Creates and returns a red-black tree node.
- **Algorithm Description :**
 1. Create a new node `newNode`.
 2. Set the `value` of `newNode` to the input parameter `value` and the `color` to the input parameter `color`.
 3. Initialize the `left` and `right` pointers of `newNode` to NULL.
 4. Return `newNode`.

- **insertNode(Node* root, Node* node)**

```

1 Node* insertNode(Node* root, Node* node)
2     if root == NULL
3         return node
4     if node.value < root.value and root.left == NULL
5         root.left <- node
6         return root
7     if node.value > root.value and root.right == NULL
8         root.right <- node
9         return root
10    if node.value < root.value
11        root.left <- insertNode(root.left, node)
12        return root
13    if node.value > root.value
14        root.right <- insertNode(root.right, node)
15        return root

```

- **Parameter Description :**

- Input Parameters: `root` represents the root of the current subtree, `node` represents the node to be inserted.
- Output: Returns the updated root of the subtree.
- Function's purpose: Inserts a new node into the red-black tree.

- **Algorithm Description :**

1. If `root` is NULL, return `node`.
2. If the value of `node` is less than the value of `root` and the left subtree of `root` is empty, insert `node` as the left child of `root` and return `root`.
3. If the value of `node` is greater than the value of `root` and the right subtree of `root` is empty, insert `node` as the right child of `root` and return `root`.
4. If the value of `node` is less than the value of `root`, recursively call `insertNode` with `root->left` and `node`, and set the result as `root->left` and return `root`.
5. If the value of `node` is greater than the value of `root`, recursively call `insertNode` with `root->right` and `node`, and set the result as `root->right` and return `root`.

- **buildRBTree(int *node, int size)**

```

1 Node* buildRBTTree(int *node, int size)
2     root <- NULL
3     for i <- 0 to size - 1
4         if node[i] < 0
5             color <- RED
6             value <- -node[i]
7         else
8             color <- BLACK
9             value <- node[i]
10        newNode <- createNode(value, color)
11        root <- insertNode(root, newNode)
12    return root

```

- **Parameter Description :**

- Input Parameters: An integer array `node` representing node colors and keys, an integer `size` representing the array size.
- Output: Returns a pointer to the root node.
- Function's purpose: Builds and returns a red-black tree.

- **Algorithm Description :**

1. Initialize `color` and `value` to 0.
2. Initialize `root` as NULL.
3. Iterate through elements of the array `node` :
 - If `node[i]` is negative, set `color` to RED and `value` to `-node[i]`; otherwise, set `color` to BLACK and `value` to `node[i]`.
 - Create a new node `newNode` using the `createNode` function, passing `value` and `color`.
 - Call the `insertNode` function to insert `newNode` into the red-black tree.
4. Return the root node `root`.

- **checkRBTTree(Node* root, int* blackHeight)**

```

1 int checkRBTTree(Node* root, int* blackHeight)
2     if root == NULL
3         blackHeight <- 0
4         return 1
5     if root.color == RED
6         if root.left != NULL and root.left.color == RED
7             return 0
8         if root.right != NULL and root.right.color == RED
9             return 0
10    leftBlackHeight <- 0
11    rightBlackHeight <- 0
12    if not checkRBTTree(root.left, leftBlackHeight)
13        return 0
14    if not checkRBTTree(root.right, rightBlackHeight)
15        return 0
16    if leftBlackHeight != rightBlackHeight
17        return 0
18    if root.color == BLACK

```

```

19     blackHeight <- leftBlackHeight + 1
20     else
21         blackHeight <- leftBlackHeight
22     return 1

```

- **Parameter Description :**

- Input Parameters: `root` represents the root of the current subtree, `blackHeight` represents the black height of the subtree.
- Output: Returns 1 (true) if it is a red-black tree, 0 (false) otherwise.
- Function's purpose: Checks if a subtree satisfies red-black tree properties and updates the black height of the subtree.

- **Algorithm Description :**

1. If `root` is NULL, set `blackHeight` to 0 and return 1.
2. If the color of `root` is RED, check its left and right child nodes, and return 0 if either child is RED.
3. Initialize `leftBlackHeight` and `rightBlackHeight` to 0.
4. Recursively call `checkRBTree` with `root->left` and `leftBlackHeight`, and store the result as `leftResult`.
5. Recursively call `checkRBTree` with `root->right` and `rightBlackHeight`, and store the result as `rightResult`.
6. If `leftBlackHeight` and `rightBlackHeight` are not equal, return 0.
7. If the color of `root` is BLACK, update `blackHeight` to `leftBlackHeight + 1`; otherwise, set `blackHeight` to `leftBlackHeight`.
8. Return 1 (Because all the preceding conditions have been met).

- **freeNodeSpace(Node* root)**

```

1 void freeNodeSpace(Node* root)
2     if root == NULL
3         return
4     if root.left == NULL
5         freeNodeSpace(root.right)
6         free(root)
7         return
8     if root.right == NULL
9         freeNodeSpace(root.left)
10        free(root)
11        return
12    freeNodeSpace(root.left)
13    freeNodeSpace(root.right)
14    free(root)
15    return

```

- **Parameter Description :**

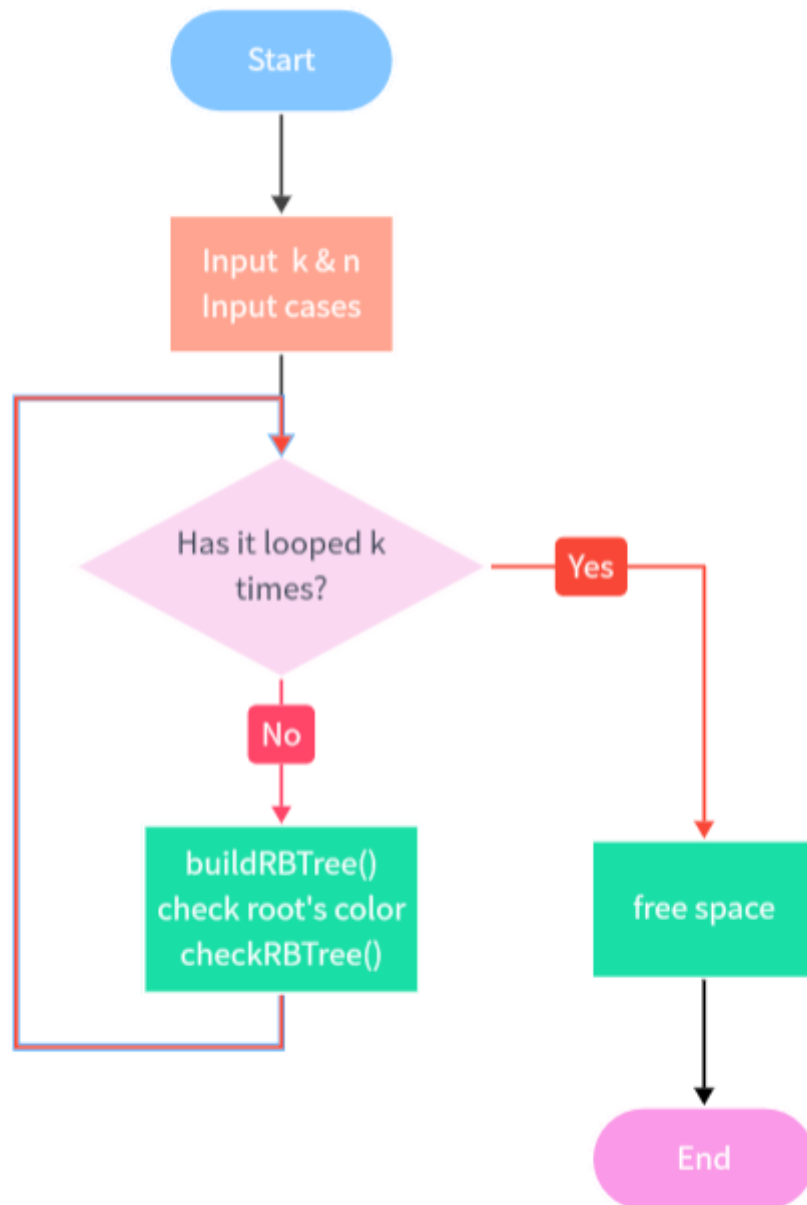
- Input Parameters: `root` represents the root of the subtree to be freed.
- Output: None.
- Function's purpose: Releases memory for the red-black tree.

- **Algorithm Description :**

1. If `root` is NULL, return.

2. If the left subtree of `root` is empty, recursively call `freeNodeSpace` with `root->right`, then free the memory for `root`.
3. If the right subtree of `root` is empty, recursively call `freeNodeSpace` with `root->left`, then free the memory for `root`.
4. Otherwise, recursively call `freeNodeSpace` with `root->left`, then recursively call `freeNodeSpace` with `root`

- a sketch of the main program



- the main program

```
1 int main(void){
```

```

2      int k,n; // k:total number of cases; n: the total number of nodes
    in the binary tree
3      int i,j;
4      printf("Input k: ");
5      scanf("%d",&k);
6      for(i=0;i<k;i++){
7          // Input n
8          printf("Input n: ");
9          scanf("%d",&n);
10         // Allocate a dynamic array of integers using malloc
11         int* nodeArray = (int*)malloc(n * sizeof(int));
12         if (nodeArray == NULL) {
13             printf("Memory allocation failed.\n");
14             return 1;
15         }
16         // Define the root of a RBTree
17         Node* root = NULL;
18         // Accept user input to initialize the array.
19         printf("Input n integers: ");
20         for(j=0;j<n;j++){
21             scanf("%d",&nodeArray[j]);
22         }
23         root=buildRBTree(nodeArray,n);
24         int blackHeight=0;
25         // First check if the root is RED
26         if(root!=NULL&&root->color==RED){
27             printf("Results: \n");
28             printf("No\n");
29             printf("\n");
30         }else if(checkRBTree(root,&blackHeight)){
31             printf("Results: \n");
32             printf("Yes\n");
33             printf("\n");
34         }else{
35             printf("Results: \n");
36             printf("No\n");
37             printf("\n");
38         }
39
40         // Free the allocated memory
41         freeNodeSpace(root);
42         free(nodeArray);
43     }
44 }

```

Chapter 3: Testing Results

1. Test Case 1: Comprehensive Test

Case1(question-provided test cases) :

- **Input :**

```
1 | 3
2 | 9
3 | 7 -2 1 5 -4 -11 8 14 -15
4 | 9
5 | 11 -2 1 -7 5 -4 8 14 -15
6 | 8
7 | 10 -7 5 -6 8 15 -11 17
```

- **Output & Expected Result :**

```
1 | Output:
2 | Yes
3 | No
4 | No
5 |
6 | Expected Result:
7 | Yes
8 | No
9 | No
```

- **Testing Purpose :**

This test case includes a moderately complex Red-Black Tree with both red and black nodes and two wrong cases. It tests the program's ability to handle various node values and validate a balanced Red-Black Tree.

Case2(other test cases) :

- **Input :**

```
1 | 1
2 | 11
3 | 53 -34 -80 18 46 74 88 -17 -33 -50 -72
```

- **Output & Expected Result :**

```
1 | Output:
2 | Yes
3 |
4 | Expected Result:
5 | Yes
```

- **Testing Purpose :**

This test case includes a moderately complex Red-Black Tree with both red and black nodes . It tests the program's ability to handle various node values and validate a balanced Red-Black Tree.

2. Test Case 2: Smallest & Largest Input Size

Small Size

Black root :

- Input :

1	1
2	1
3	5

- Output & Expected Result :

1	Output:
2	Yes
3	
4	Expected Result:
5	Yes

- Testing Purpose :

This test case focuses on the smallest possible Red-Black Tree, having just one node. It checks if the program can correctly identify a valid Red-Black Tree with the minimum size (black root node).

Red root:

- Input :

1	1
2	1
3	-6

- Output & Expected Result :

1	Output:
2	No
3	
4	Expected Result:
5	No

- Testing Purpose :

This test case focuses on the smallest possible Red-Black Tree, having just one node. It checks if the program can correctly identify a valid Red-Black Tree with the minimum size (red root node).

Large size:

- **Input :**

```
1 2
2 30
3 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
  27 28 29 30
4 100
5 1 -2 3 -4 5 -6 7 -8 9 -10 11 -12 13 -14 15 -16 17 -18 19 -20 21 -22
  23 -24 25 -26 27 -28 29 -30 -31 -32 -33 -34 -35 -36 -37 -38 -39 -40
  -41 -42 -43 -44 -45 -46 -47 -48 -49 -50 -51 -52 -53 -54 -55 -56 -57
  -58 -59 -60 -61 -62 -63 -64 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74
  -75 -76 -77 -78 -79 -80 -81 -82 -83 -84 -85 -86 -87 -88 -89 -90 -91
  -92 -93 -94 -95 -96 -97 -98 -99 -100
```

- **Output & Expected Result :**

```
1 Output:
2 No
3 No
4
5 Expected Result:
6 No
7 No
```

- **Testing Purpose :**

Test whether the program can handle cases with a large size.

3. Test Case 3: Extreme Case Test

Empty test case :

- **Input :**

```
1 | 0
```

- **Output & Expected Result :**

```
1 Output:
2
3 Expected Result:
4
```

- **Testing Purpose :**

Test whether the program can handle zero input cases

An empty tree :

- **Input :**

```
1 | 1
2 | 0
3 | (null)
```

- **Output & Expected Result :**

```
1 | Output:
2 | Yes
3 |
4 | Expected Result:
5 | Yes
```

- **Testing Purpose :**

Test whether the program can handle the case of an empty tree and return the correct result.

Large Input Size with Unbalanced Red Nodes :

- **Input :**

```
1 | 1
2 | 10
3 | -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
```

- **Output & Expected Result :**

```
1 | Output:
2 | No
3 |
4 | Expected Result:
5 | No
```

- **Testing Purpose :**

Confirm that the program detects an unbalanced tree with all red nodes.

Chapter 4: Analysis and Comments

- **Time complexity analysis**

- buildRBTree Function:**

- Iterating through the `node` array to create nodes and insert them into the Red-Black Tree: This operation takes $O(n)$ time, where n is the number of nodes in the tree.
 - The `createNode` function takes constant time to allocate memory and initialize the node's attributes.

- The `insertNode` function, when called for insertion, takes $O(h)$ time for each node insertion, where h is the height of the tree. In the worst case, when the tree is highly unbalanced, it can take $O(n)$ time.
- So, the total time complexity of building the Red-Black Tree is $O(n * h)$, where h can range from $\log(n)$ to n .

checkRBTree Function:

- The function is checking the properties of a Red-Black Tree recursively. In the worst case, it will visit every node in the tree, so the time complexity is $O(n)$.

freeNodeSpace Function:

- This function recursively traverses and deallocates the memory for each node. Similar to the `checkRBTree` function, it can visit every node in the tree, resulting in $O(n)$ time complexity.

• Space complexity analysis

Memory for nodeArray:

- In the `main` function, it allocates memory for an integer array `nodeArray` of size `n`. So, the space complexity here is $O(n)$.

Memory for Red-Black Tree:

- The memory used by the Red-Black Tree itself is mainly for the nodes and their attributes. In the worst case, if the tree is highly unbalanced, the space complexity can be $O(n)$.
- Additionally, for the recursive calls in `checkRBTree` and `freeNodeSpace`, they have function call stack space. In the worst case, the stack space can be $O(h)$, where h is the height of the tree. For a balanced tree, this is $O(\log(n))$, but for an unbalanced tree, it can be $O(n)$.
- The `createNode` function also allocates memory for each node, but the space required is proportional to the number of nodes, so it's also $O(n)$.

Total Space Complexity:

- The sum of the above space complexities is $O(n)$ for `nodeArray` + $O(n)$ for the Red-Black Tree + $O(h)$ for the function call stack space. Therefore, the overall space complexity is $O(n + h)$, where h is the height of the tree.

- **Conclusion**

In conclusion, the time complexity of the program is $O(n * h)$, where n is the number of nodes in the tree, and h is the height of the tree. The space complexity is $O(n + h)$, where n is the number of nodes and h is the height of the tree. The actual time and space complexity can vary depending on the balance of the Red-Black Tree.

Appendix: Source Code (in C)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Define colors for the Red-Black Tree
5  #define RED 0
6  #define BLACK 1
7
8  // Structure for a Red-Black Tree node
9  typedef struct RBTreeNode {
10     int value;           // Node's key
11     int color;           // Node's color, can be red (RED) or black
12                          // (BLACK)
13     struct RBTreeNode* left; // Left child
14     struct RBTreeNode* right; // Right child
15 } Node;
16
17 // Build Red-Black Tree
18 Node* buildRBTree(int *node, int size);
19 // Create a new node
20 Node* createNode(int key, int color);
21 // Insert a node into a RBTree and return the root of the RBTree
22 Node* insertNode(Node* root, Node* node);
23 // Check if a Binary-Tree is RBTree (1:true 0:false)
24 int checkRBTree(Node* root, int* blackHeight);
25 // Free the memory allocated for Red-Black tree nodes
26 void freeNodeSpace(Node* root);
27
28 int main(void){
29     int k,n; // k:total number of cases; n: the total number of nodes in
30             // the binary tree
31     int i,j;
32     printf("Input k: ");
33     scanf("%d",&k);
34     for(i=0;i<k;i++){
35         // Input n
36         printf("Input n: ");
37         scanf("%d",&n);
38         // Allocate a dynamic array of integers using malloc
39         int* nodeArray = (int*)malloc(n * sizeof(int));
40         if (nodeArray == NULL) {
41             printf("Memory allocation failed.\n");
42         }
43     }
44 }
```

```

40         return 1;
41     }
42     // Define the root of a RBTree
43     Node* root = NULL;
44     // Accept user input to initialize the array.
45     printf("Input n integers: ");
46     for(j=0;j<n;j++){
47         scanf("%d",&nodeArray[j]);
48     }
49     root=buildRBTree(nodeArray,n);
50     int blackHeight=0;
51     // First check if the root is RED
52     if(root!=NULL&&root->color==RED){
53         printf("Results: \n");
54         printf("No\n");
55         printf("\n");
56     }else if(checkRBTree(root,&blackHeight)){
57         printf("Results: \n");
58         printf("Yes\n");
59         printf("\n");
60     }else{
61         printf("Results: \n");
62         printf("No\n");
63         printf("\n");
64     }
65
66     // Free the allocated memory
67     freeNodeSpace(root);
68     free(nodeArray);
69 }
70 }
71
72 // Build Red-Black Tree
73 Node* buildRBTree(int *node, int size){
74     int color = 0;
75     int value = 0;
76     int i;
77     Node* root=NULL;
78     for(i=0;i<size;i++){
79         // Determine the color of the node based on its sign form.
80         if(node[i]<0){
81             color = RED;
82             value=-node[i];
83         }else{
84             color=BLACK;
85             value=node[i];
86         }
87         // Call functions to create nodes and insert nodes to build a Red-
88         // Black tree.
89         Node* node = createNode(value,color);
90         root=insertNode(root,node);
91     }
92     return root;
93 }

```

```

94 // Create a new node
95 Node* createNode(int value, int color) {
96     Node* newNode = (Node*)malloc(sizeof(struct RBTreeNode));
97     newNode->value = value;
98     newNode->color = color;
99     newNode->left = NULL;
100    newNode->right = NULL;
101    return newNode;
102 }
103
104 // Insert node into a RBTree and return the root of the RBTree
105 Node* insertNode(Node* root, Node* node){
106     if(root==NULL){
107         return node;
108     }
109     if((node->value<root->value)&&(root->left==NULL)){
110         root->left=node;
111         return root;
112     }
113     if((node->value>root->value)&&(root->right==NULL)){
114         root->right=node;
115         return root;
116     }
117     // recursive insertion
118     if(node->value<root->value){
119         root->left=insertNode(root->left,node);
120         return root;
121     }
122     if(node->value>root->value){
123         root->right=insertNode(root->right,node);
124         return root;
125     }
126 }
127
128 // Check if a Binary-Tree is RBTree (1:true 0:false)
129 /* Tips:
130 The root node here is not the actual root node of the Red-Black tree;
131 the root node for any subtree can be the parameter 'root' for this
132 function.
133 */
134 int checkRBTree(Node* root, int* blackHeight){
135     // 1. First, check if the node is empty.
136     if(root==NULL){
137         *blackHeight=0;
138         return 1;
139     }
140     // 2. Second, Second, if the current root node's color is red at this
141     point,
142     // then check if the colors of its two child nodes are also red.
143     if(root->color==RED){
144         if((root->left!=NULL)&&((root->left)->color==RED)){
145             return 0;
146         }
147         if((root->right!=NULL)&&((root->right)->color==RED)){
148             return 0;
149         }
150     }
151     // 3. Third, check if the black height is correct.
152     // 4. Fourth, check if the root node is black.
153     // 5. Fifth, check if the root node is not null.
154     // 6. Sixth, check if the root node is not null.
155     // 7. Seventh, check if the root node is not null.
156     // 8. Eighth, check if the root node is not null.
157     // 9. Ninth, check if the root node is not null.
158     // 10. Tenth, check if the root node is not null.
159     // 11. Eleventh, check if the root node is not null.
160     // 12. Twelfth, check if the root node is not null.
161     // 13. Thirteenth, check if the root node is not null.
162     // 14. Fourteenth, check if the root node is not null.
163     // 15. Fifteenth, check if the root node is not null.
164     // 16. Sixteenth, check if the root node is not null.
165     // 17. Seventeenth, check if the root node is not null.
166     // 18. Eighteenth, check if the root node is not null.
167     // 19. Nineteenth, check if the root node is not null.
168     // 20. Twentieth, check if the root node is not null.
169     // 21. Twenty-first, check if the root node is not null.
170     // 22. Twenty-second, check if the root node is not null.
171     // 23. Twenty-third, check if the root node is not null.
172     // 24. Twenty-fourth, check if the root node is not null.
173     // 25. Twenty-fifth, check if the root node is not null.
174     // 26. Twenty-sixth, check if the root node is not null.
175     // 27. Twenty-seventh, check if the root node is not null.
176     // 28. Twenty-eighth, check if the root node is not null.
177     // 29. Twenty-ninth, check if the root node is not null.
178     // 30. Thirtieth, check if the root node is not null.
179     // 31. Thirty-first, check if the root node is not null.
180     // 32. Thirty-second, check if the root node is not null.
181     // 33. Thirty-third, check if the root node is not null.
182     // 34. Thirty-fourth, check if the root node is not null.
183     // 35. Thirty-fifth, check if the root node is not null.
184     // 36. Thirty-sixth, check if the root node is not null.
185     // 37. Thirty-seventh, check if the root node is not null.
186     // 38. Thirty-eighth, check if the root node is not null.
187     // 39. Thirty-ninth, check if the root node is not null.
188     // 40. Fortieth, check if the root node is not null.
189     // 41. Forty-first, check if the root node is not null.
190     // 42. Forty-second, check if the root node is not null.
191     // 43. Forty-third, check if the root node is not null.
192     // 44. Forty-fourth, check if the root node is not null.
193     // 45. Forty-fifth, check if the root node is not null.
194     // 46. Forty-sixth, check if the root node is not null.
195     // 47. Forty-seventh, check if the root node is not null.
196     // 48. Forty-eighth, check if the root node is not null.
197     // 49. Forty-ninth, check if the root node is not null.
198     // 50. Fiftieth, check if the root node is not null.
199     // 51. Fifty-first, check if the root node is not null.
200     // 52. Fifty-second, check if the root node is not null.
201     // 53. Fifty-third, check if the root node is not null.
202     // 54. Fifty-fourth, check if the root node is not null.
203     // 55. Fifty-fifth, check if the root node is not null.
204     // 56. Fifty-sixth, check if the root node is not null.
205     // 57. Fifty-seventh, check if the root node is not null.
206     // 58. Fifty-eighth, check if the root node is not null.
207     // 59. Fifty-ninth, check if the root node is not null.
208     // 60. Sixtieth, check if the root node is not null.
209     // 61. Sixty-first, check if the root node is not null.
210     // 62. Sixty-second, check if the root node is not null.
211     // 63. Sixty-third, check if the root node is not null.
212     // 64. Sixty-fourth, check if the root node is not null.
213     // 65. Sixty-fifth, check if the root node is not null.
214     // 66. Sixty-sixth, check if the root node is not null.
215     // 67. Sixty-seventh, check if the root node is not null.
216     // 68. Sixty-eighth, check if the root node is not null.
217     // 69. Sixty-ninth, check if the root node is not null.
218     // 70. Seventieth, check if the root node is not null.
219     // 71. Seventy-first, check if the root node is not null.
220     // 72. Seventy-second, check if the root node is not null.
221     // 73. Seventy-third, check if the root node is not null.
222     // 74. Seventy-fourth, check if the root node is not null.
223     // 75. Seventy-fifth, check if the root node is not null.
224     // 76. Seventy-sixth, check if the root node is not null.
225     // 77. Seventy-seventh, check if the root node is not null.
226     // 78. Seventy-eighth, check if the root node is not null.
227     // 79. Seventy-ninth, check if the root node is not null.
228     // 80. Eightieth, check if the root node is not null.
229     // 81. Eighty-first, check if the root node is not null.
230     // 82. Eighty-second, check if the root node is not null.
231     // 83. Eighty-third, check if the root node is not null.
232     // 84. Eighty-fourth, check if the root node is not null.
233     // 85. Eighty-fifth, check if the root node is not null.
234     // 86. Eighty-sixth, check if the root node is not null.
235     // 87. Eighty-seventh, check if the root node is not null.
236     // 88. Eighty-eighth, check if the root node is not null.
237     // 89. Eighty-ninth, check if the root node is not null.
238     // 90. Ninetieth, check if the root node is not null.
239     // 91. Ninety-first, check if the root node is not null.
240     // 92. Ninety-second, check if the root node is not null.
241     // 93. Ninety-third, check if the root node is not null.
242     // 94. Ninety-fourth, check if the root node is not null.
243     // 95. Ninety-fifth, check if the root node is not null.
244     // 96. Ninety-sixth, check if the root node is not null.
245     // 97. Ninety-seventh, check if the root node is not null.
246     // 98. Ninety-eighth, check if the root node is not null.
247     // 99. Ninety-ninth, check if the root node is not null.
248     // 100. Check if the root node is not null.

```

```

147     }
148 }
149 // 3. Third, recursively check whether the left and right subtrees also
satisfy
150 // the properties of a Red-Black tree and pass the black height of the
left and
151 // right subtrees to the calling function using pointer parameters.
152 int leftBlackHeight=0;
153 int rightBlackHeight=0;
154 if(!checkRBTree(root->left,&leftBlackHeight)){
155     return 0;
156 }
157 if(!checkRBTree(root->right,&rightBlackHeight)){
158     return 0;
159 }
160 // Check if the black heights of the left and right subtrees are equal.
161 if(leftBlackHeight!=rightBlackHeight){
162     return 0;
163 }
164 // 4. Pass the black height of the current node to the parent node
using a pointer.
165 if(root->color==BLACK){
166     *blackHeight+=leftBlackHeight+1;
167 }else{
168     *blackHeight=leftBlackHeight;
169 }
170 return 1;
171 }
172
173
174 // Free the memory allocated for Red-Black tree nodes
175 void freeNodeSpace(Node* root){
176     if(root==NULL){
177         return;
178     }
179     // Add conditional checks, reduce additional stack overhead, and save
memory space
180     if(root->left==NULL){
181         freeNodeSpace(root->right);
182         free(root);
183         return;
184     }
185     // Add conditional checks, reduce additional stack overhead, and save
memory space
186     if(root->right==NULL){
187         freeNodeSpace(root->left);
188         free(root);
189         return;
190     }
191     // Recursively release the memory allocated to nodes
192     // (first left subtree, then right subtree, and finally the root node).
193     freeNodeSpace(root->left);
194     freeNodeSpace(root->right);
195     free(root);
196     return;

```


197 }
198
199