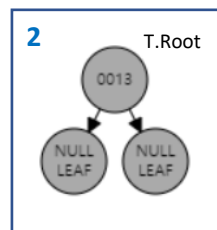
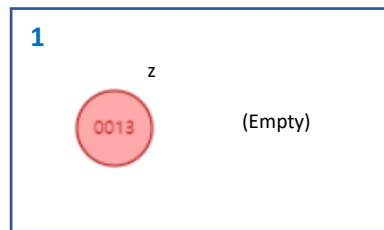


Assignment 9

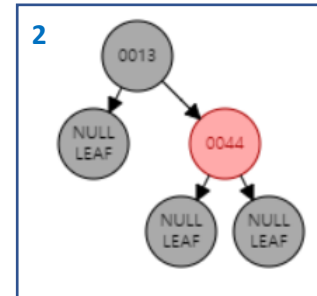
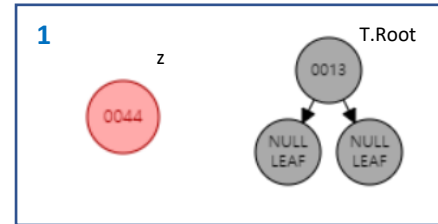
Name: DONGWOOK LEE

Problem 9.1 Understanding Red Black Trees

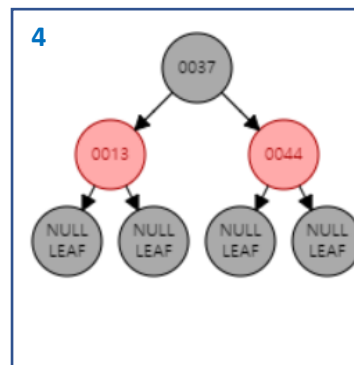
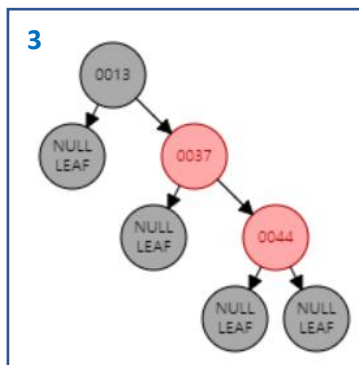
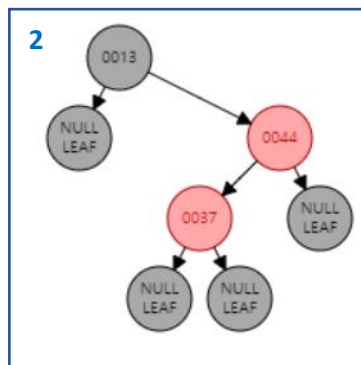
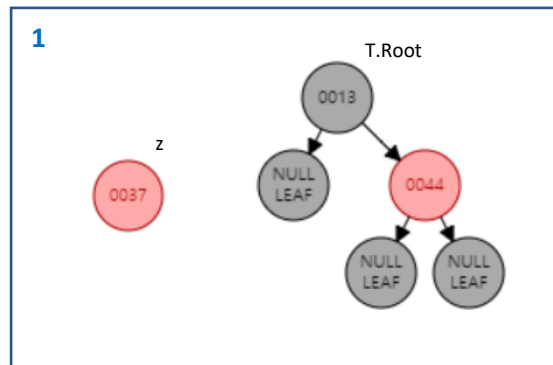
(a) Insert {13, 44, 37, 7, 22, 16} into an empty RB Tree in order



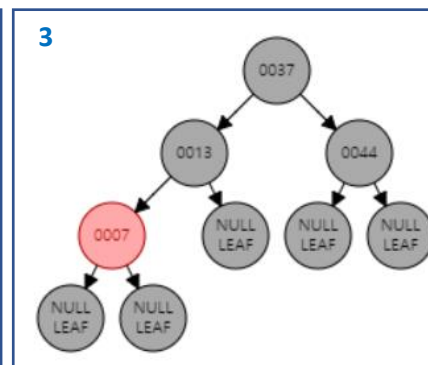
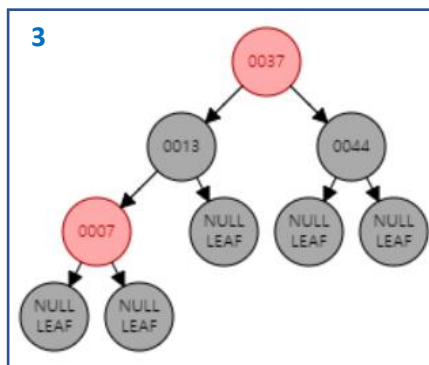
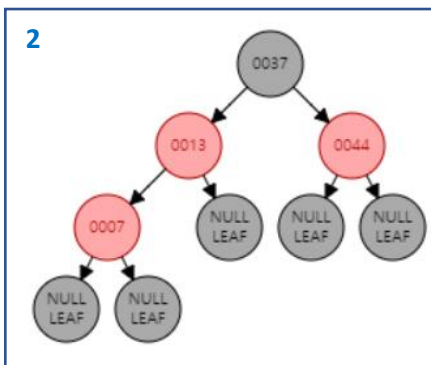
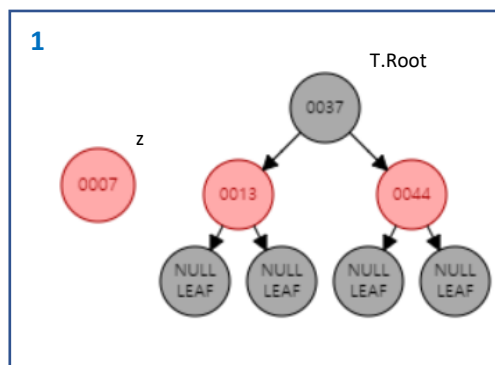
1. Insert 13 into an empty tree T.
Violates RooB
→ T.root.color = black;
2. Complete



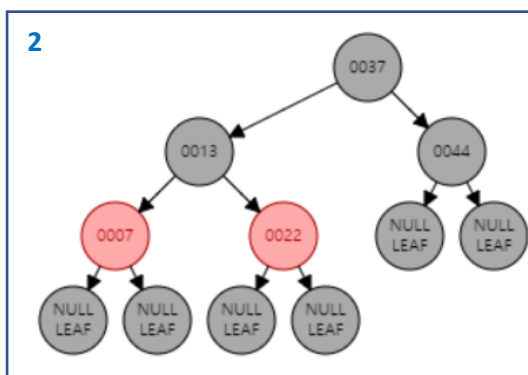
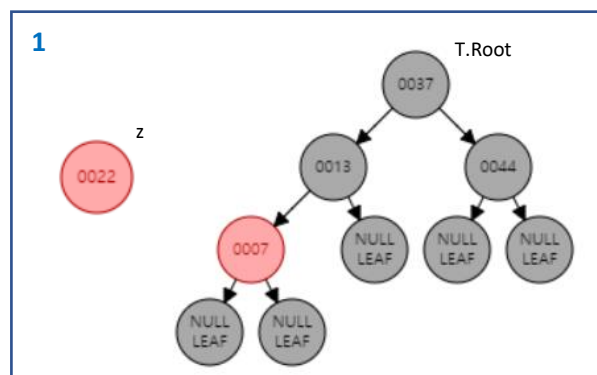
1. Insert 44 into the tree.
Compare keys of z and T.Root. → z.key > T.Root.key
Right child of T.Root is NULL, replace with z
2. Complete



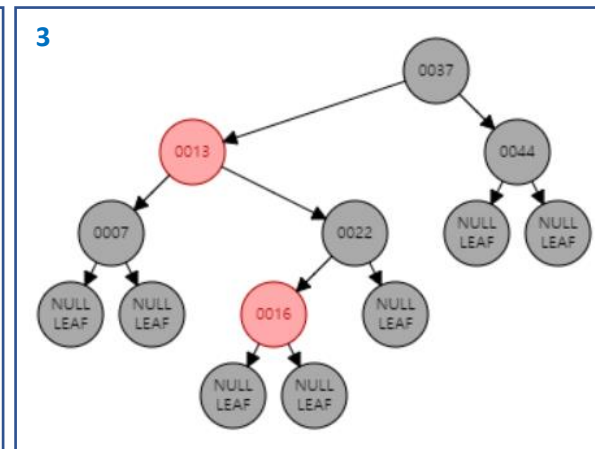
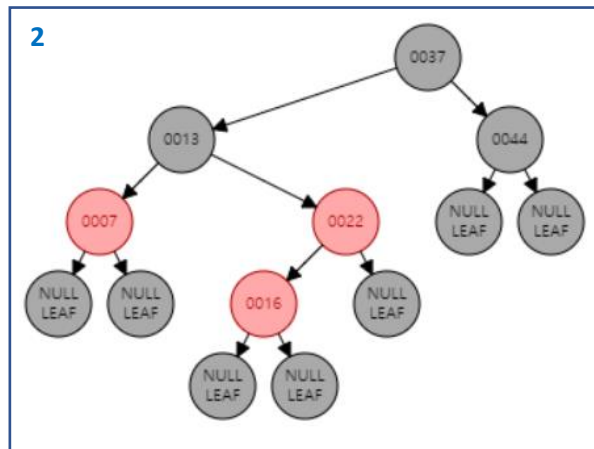
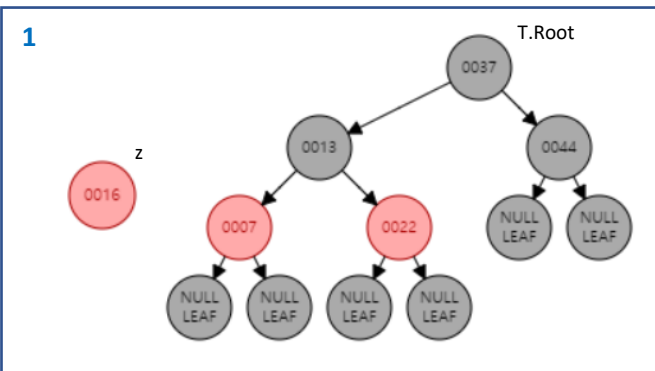
1. Insert 37 into the tree.
Compare keys of z and T.Root. → z.key > T.Root.key
Compare keys of z and T.root.right → z.key < T.root.right.key
2. T.Root.right.left is NULL, replace with z. Violates BredB
z = z.p.left & z.p = z.p.p.right & z.p.p.left = black (Symmetric to Case 2)
→ Right rotation (T, z.p)
3. z = z.p.right & z.p = z.p.p.right & z.p.p.left = black (Symmetric to Case 3)
→ Left rotation (T, z.p.p)
4. Complete



1. Insert 7 into the tree.
Compare keys of z and T.root. → z.key < T.root.key
Compare keys of z and T.root.left → z.key < T.root.left.key
2. T.root.left.left is NULL, replace with z. Violates BredB
z.p = z.p.p.left & z.p.p.right = red (Case 1)
→ z.p.color = black; z.p.p.right.color = black; z.p.p.color = red;
3. Violates RooB
→ T.root.color = black;
4. Complete

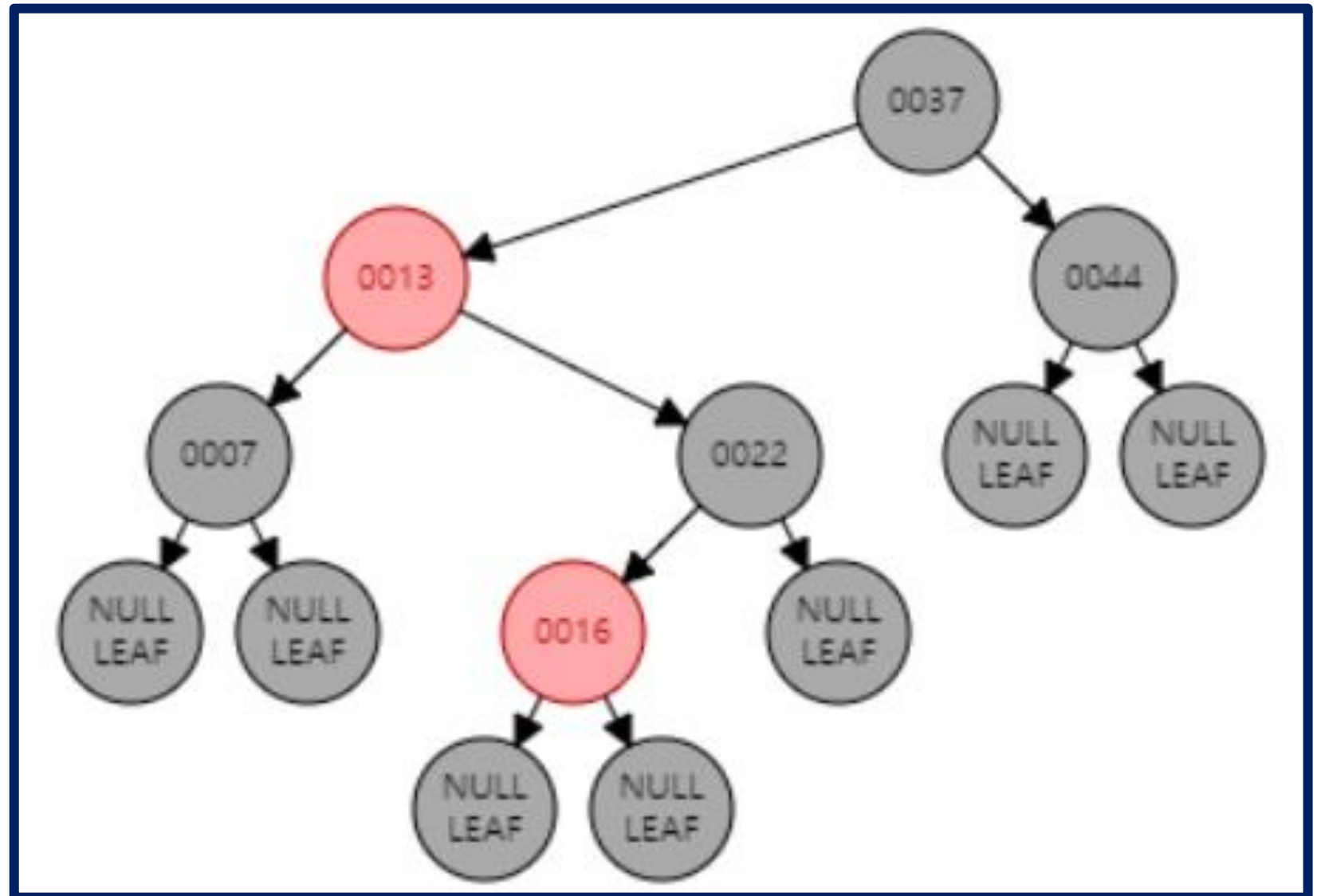


1. Insert 22 into the tree.
Compare keys of z and T.root. → z.key < T.root.key
Compare keys of z and T.root.left → z.key > T.root.left.key
2. T.root.left.right is NULL, replace with z.
Complete



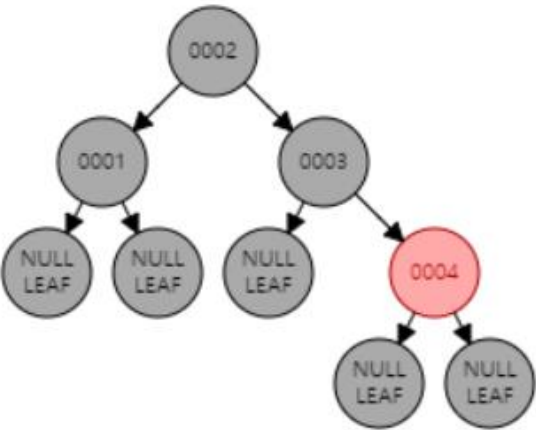
1. Insert 16 into the tree.
 Compare keys of z and T.root. $\rightarrow z.key < T.root.key$
 Compare keys of z and T.root.left $\rightarrow z.key > T.root.left.key$
 Compare keys of z and T.root.left.right $\rightarrow z.key < T.root.left.right.key$
2. T.root.left.right.left is NULL, replace with z. Violates BredB
 $z.p = z.p.p.right$ & $z.p.p.left = red$ (Symmetric to Case 1)
 $\rightarrow z.p.color = black; z.p.p.left.color = black; z.p.p.color = red;$
3. Complete

Result:

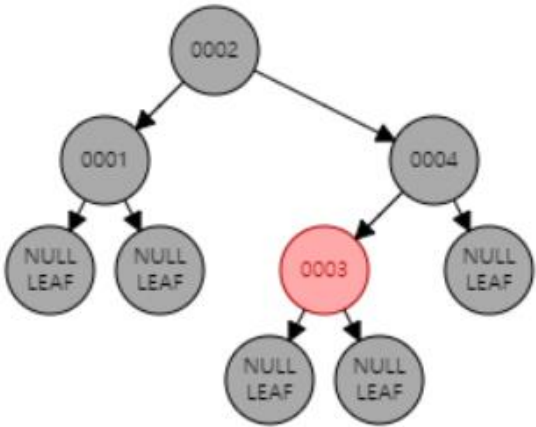


(b) Make all valid RB Tree with the elements {1, 2, 3, 4}.

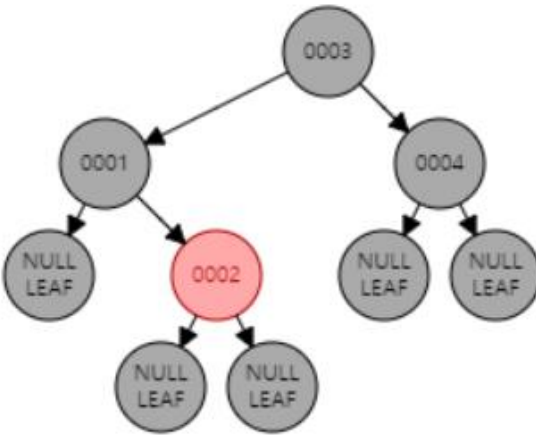
1 → 2 → 3 → 4
1 → 3 → 2 → 4
2 → 1 → 3 → 4
2 → 3 → 1 → 4
3 → 1 → 2 → 4
3 → 2 → 1 → 4



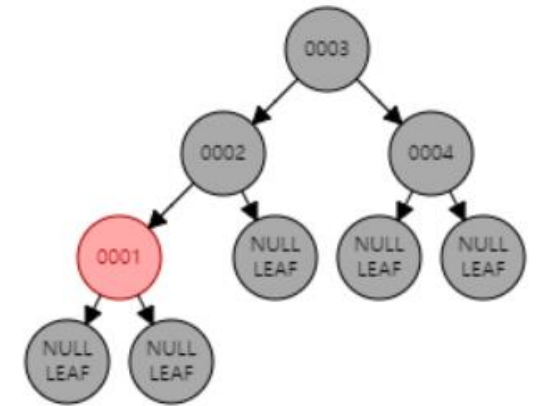
1 → 2 → 4 → 3
1 → 4 → 2 → 3
2 → 1 → 4 → 3
2 → 4 → 1 → 3
4 → 1 → 2 → 3
4 → 2 → 1 → 3



1 → 3 → 4 → 2
1 → 4 → 3 → 2
3 → 1 → 4 → 2
3 → 4 → 1 → 2
4 → 1 → 3 → 2
4 → 3 → 1 → 2



2 → 4 → 3 → 1
2 → 3 → 4 → 1
3 → 2 → 4 → 1
3 → 4 → 2 → 1
4 → 2 → 3 → 1
4 → 3 → 2 → 1



```
RBTree_Class.h  Implementing_RedBlackTree.cpp
Implementing_RedBlackTree (Global Scope)

1  #pragma once
2  enum Color { RED, BLACK };
3  struct Node {
4      Node(Color colo) : color(colo) {}
5      int data;
6      Color color;
7      Node* left, * right, * parent;
8  };
9
10 class RedBlackTree {
11 private:
12     Node* root;
13     Node* nil = new Node(BLACK);
14 protected:
15     void rotateLeft(Node* z);
16     void rotateRight(Node* z);
17 public:
18     RedBlackTree();
19     void insertNode(int data);
20     void deleteNode(Node* tobeDeleted);
21     Node* predecessor(const Node* z);
22     Node* successor(const Node* z);
23     Node* getMinimum();
24     Node* getMaximum();
25     Node* search(int data);
26
27     void RB_Insert_FixUp(Node *z);
28     void Transplant(Node* z, Node* x);
29     void RB_Delete_FixUp(Node* x);
30 };
```

```

RBTree_Class.h
Implementing_RedBlackTree
RedBlackTree
insertNode(int data)

1 // Implementing_RedBlackTree.cpp :
2 // This file contains the 'main' function. Program execution begins and ends there.
3 // Run program: Ctrl + F5 or Debug > Start Without Debugging menu
4 // Debug program: F5 or Debug > Start Debugging menu
5
6 #include "RBTree_Class.h"
7 #include <iostream>
8
9 void RedBlackTree::rotateLeft(Node* z) {
10     Node* newz = z->right;
11
12     z->right = newz->left;
13     if (newz->left != nil) {
14         newz->left->parent = z;
15     }
16
17     newz->left = z;
18
19     newz->parent = z->parent;
20     if (z->parent == nil) {
21         root = newz;
22     }
23
24     z->parent = newz;
25 }
26
27 void RedBlackTree::rotateRight(Node* z) {
28     Node* newz = z->left;
29
30     z->left = newz->right;
31     if (newz->right != nil) {
32         newz->right->parent = z;
33     }
34
35     newz->right = z;
36
37     newz->parent = z->parent;
38     if (z->parent == nil) {
39         root = newz;
40     }
41
42     z->parent = newz;
43 }
44
45 RedBlackTree::RedBlackTree() {
46     root = nil;
47 }
48
49 void RedBlackTree::insertNode(int data) {
50     Node* z = new Node(RED);
51     z->data = data;
52
53     Node* y = nil;
54     Node* x = root;
55     while (x != nil) {
56         y = x;
57         if (z->data < x->data) {
58             x = x->left;
59         }
60         else {
61             x = x->right;
62         }
63     }
64     z->parent = y;
65     if (y == nil) {
66         root = z;
67     }
68     else if (z->data < y->data) {
69         y->left = z;

```

```

70     }
71     else {
72         y->right = z;
73     }
74     z->left = nil;
75     z->right = nil;
76     RB_Insert_FixUp(z);
77 }
78
79 void RedBlackTree::RB_Insert_FixUp(Node* z){
80     Node* uncle = NULL;
81
82     while (z->parent->color == RED) {
83         if (z->parent->parent->left == z->parent) {
84             uncle = z->parent->parent->right;
85             if (uncle->color == RED) {
86                 z->parent->color = BLACK;
87                 uncle->color = BLACK;
88                 uncle->parent->color = RED;
89                 z = uncle->parent;
90             }
91             else if (z == z->parent->right) {
92                 z = z->parent;
93                 rotateLeft(z);
94                 uncle->color = BLACK;
95                 uncle->parent->color = RED;
96                 rotateRight(z->parent->parent);
97             }
98         }
99         else {
100             uncle = z->parent->parent->left;
101             if (uncle->color == RED) {
102                 z->parent->color = BLACK;
103                 uncle->color = BLACK;
104                 uncle->parent->color = RED;
105                 z = uncle->parent;
106             }
107             else {
108                 if (z == z->parent->left) {
109                     z = z->parent;
110                     rotateRight(z);
111                 }
112                 uncle->color = BLACK;
113                 uncle->parent->color = RED;
114                 rotateLeft(z->parent->parent);
115             }
116         }
117     }
118     root->color = BLACK;
119 }
120
121 void RedBlackTree::RB_Delete_FixUp(Node* x) {
122     Node* w;
123
124     while (x != root && x->color == BLACK) {
125         if (x == x->parent->left) {
126             w = x->parent->right;
127             if (w->color == RED) {
128                 w->color = BLACK;
129                 x->parent->color = RED;
130                 rotateLeft(x->parent);
131             }
132             if (w->left->color == BLACK && w->right->color == BLACK) {
133                 w->color = RED;
134                 x = x->parent;
135             }
136         }
137         else {
138             if (w->right->color == BLACK) {
139                 w->left->color = BLACK;

```



```

139         w->color = RED;
140         rotateRight(w);
141         w = x->parent->right;
142     }
143     w->color = x->parent->color;
144     x->parent->color = BLACK;
145     w->right->color = BLACK;
146     rotateLeft(x->parent);
147     x = root;
148 }
149 }
150 else {
151     w = x->parent->left;
152     if (w->color == RED) {
153         w->color = BLACK;
154         x->parent->color = RED;
155         rotateRight(x->parent);
156     }
157     if (w->right->color == BLACK && w->left->color == BLACK) {
158         w->color = RED;
159         x = x->parent;
160     }
161     else {
162         if (w->left->color == BLACK) {
163             w->right->color = BLACK;
164             w->color = RED;
165             rotateLeft(w);
166             w = x->parent->left;
167         }
168         w->color = x->parent->color;
169         x->parent->color = BLACK;
170         w->left->color = BLACK;
171         rotateRight(x->parent);
172         x = root;
173     }
174 }

```

```

175 }
176 x->color = BLACK;
177 }
178
179 void RedBlackTree::Transplant(Node* z, Node* x) {
180     if (z->parent == nil) {
181         root = x;
182     }
183     else if (z == z->parent->left) {
184         z->parent->left = x;
185     }
186     else {
187         z->parent->right = x;
188     }
189
190     if (x) {
191         x->parent = z->parent;
192     }
193 }
194
195
196 void RedBlackTree::deleteNode(Node* z) {
197     Node* y = z;
198     Node* x;
199     Color Yoriginal = y->color;
200     if (z->left == nil) {
201         x = z->right;
202         //RB_Transplant(z, x);
203     }
204     else if (z->right == nil) {
205         x = z->left;
206         //RB_Transplant(z, x);
207     }
208     else {
209         y = successor(z);
210         Yoriginal = y->color;

```

```

211         x = y->right;
212         if (y->parent == z) {
213             x->parent = y;
214         }
215         else {
216             //RB_Transplant(y, x);
217             y->right = z->right;
218             x->parent = y;
219         }
220         //RB_Transplant(z, y);
221         y->left = z->left;
222         y->left->parent = y;
223         y->color = z->color;
224     }
225     if (Yoriginal == BLACK) {
226         RB_Delete_FixUp(x);
227     }
228 }
229
230 Node* RedBlackTree::predecessor(const Node* z) {
231     // maximum from its left child
232     if (z == getMinimum()) {
233         return NULL;
234     }
235     Node* y = z->left;
236     while (y != nil) {
237         y = y->right;
238     }
239     return y->parent;
240 }
241
242
243 Node* RedBlackTree::successor(const Node* z) {
244     // minimum from its right child
245     if (z == getMaximum()) {
246         return NULL;
247     }
248     Node* y = z->right;
249     while (y != nil) {
250         y = y->left;
251     }
252     return y->parent;
253 }
254
255 Node* RedBlackTree::getMinimum() {
256     Node* y = root;
257     if (y == nil) {
258         return NULL;
259     }
260     while (y != nil) {
261         y = y->left;
262     }
263     return y->parent;
264 }
265
266 Node* RedBlackTree::getMaximum() {
267     Node* y = root;
268     if (y == nil) {
269         return NULL;
270     }
271     while (y != nil) {
272         y = y->right;
273     }
274     return y->parent;
275 }
276
277 Node* RedBlackTree::search(int data) {
278     Node* y = root;
279     while (y != nil) {
280         if (y->data == data) {
281             return y;
282         }
283         else if (data > y->data) {
284             y = y->right;
285         }
286         else {
287             y = y->left;
288         }
289     }
290     return NULL;
291 }

```