

```
1 //Red black Tree
2 #include <iostream>
3
4 using namespace std;
5
6 // Enum representing the color of a node in the red-black tree
7 enum Color {
8     RED, BLACK,
9 };
10
11 // Node structure for the red-black tree
12 typedef struct RBNode* Ref;
13 struct RBNode {
14     int key;
15     Color color;
16     Ref parent, left, right;
17 };
18
19 // Global variable representing the nil (sentinel) node
20 Ref nil;
21
22 // Function to create a new node with given key, color, and nil reference
23 Ref createNode(int key, Color color, Ref nil) {
24     Ref p = new RBNode{ key, color, nil, nil, nil };
25     return p;
26 }
27
28 // Left rotation operation in the red-black tree
29 void leftRotate(Ref& root, Ref x) {
30     Ref y = x->right;
31     x->right = y->left;
32
33     if (y->left != nil) {
34         y->left->parent = x;
35     }
36     y->parent = x->parent;
37
38     if (x->parent == nil) {
39         root = y;
40     }
41     else {
42         if (x == x->parent->left) {
43             x->parent->left = y;
44         }
45         else {
46             x->parent->right = y;
47         }
48     }
49 }
```

```
50     y->left = x;
51     x->parent = y;
52 }
53
54 // Right rotation operation in the red-black tree
55 void rightRotate(Ref& root, Ref x) {
56     Ref y = x->left;
57     x->left = y->right;
58
59     if (y->right != nil) {
60         y->right->parent = x;
61     }
62     y->parent = x->parent;
63
64     if (x->parent == nil) {
65         root = y;
66     }
67     else {
68         if (x == x->parent->right) {
69             x->parent->right = y;
70         }
71         else {
72             x->parent->left = y;
73         }
74     }
75
76     y->right = x;
77     x->parent = y;
78 }
79
80 // Binary Search Tree (BST) insertion operation
81 void BST_Insert(Ref& root, Ref x) {
82     Ref y = nil, z = root;
83     while (z != nil) {
84         y = z;
85
86         if (x->key < z->key) z = z->left;
87         else if (x->key > z->key) z = z->right;
88         else return; // Key already exists, do nothing
89     }
90
91     x->parent = y;
92     if (y == nil) root = x;
93     else {
94         if (x->key < y->key) y->left = x;
95         else y->right = x;
96     }
97 }
98
```

```
99 // Adjustments after left child insertion in the red-black tree
100 void insertionLeftAdjust(Ref& root, Ref& x) {
101     Ref u = x->parent->parent->right;
102     if (u->color == RED) {
103         x->parent->color = u->color = BLACK;
104         x->parent->parent->color = RED;
105         x = x->parent->parent;
106     }
107     else {
108         if (x == x->parent->right) {
109             x = x->parent;
110             leftRotate(root, x);
111         }
112         x->parent->color = BLACK;
113         x->parent->color = BLACK;
114         x->parent->parent->color = RED;
115         rightRotate(root, x->parent->parent);
116     }
117 }
118
119 // Adjustments after right child insertion in the red-black tree
120 void insertionRightAdjust(Ref& root, Ref& x) {
121     Ref u = x->parent->parent->left;
122     if (u->color == RED) {
123         x->parent->color = u->color = BLACK;
124         x->parent->parent->color = RED;
125         x = x->parent->parent;
126     }
127     else {
128         if (x == x->parent->left) {
129             x = x->parent;
130             rightRotate(root, x);
131         }
132         x->parent->color = BLACK;
133         x->parent->color = BLACK;
134         x->parent->parent->color = RED;
135         leftRotate(root, x->parent->parent);
136     }
137 }
138
139 // Fix-up routine after insertion in the red-black tree
140 void insertionFixUp(Ref& root, Ref x) {
141     while (x->parent->color == RED) {
142         if (x->parent == x->parent->parent->left) {
143             insertionLeftAdjust(root, x);
144         }
145         else {
146             insertionRightAdjust(root, x);
147         }
148     }
149 }
```

```
148     }
149     root->color = BLACK;
150 }
151
152 // Insert a key into the red-black tree
153 void Insert(Ref& root, int key) {
154     Ref x = createNode(key, RED, nil);
155     BST_Insert(root, x);
156     insertionFixUp(root, x);
157 }
158
159 // Create a red-black tree from an array of keys
160 Ref createTree(int a[], int n) {
161     Ref root = nil;
162
163     for (int i = 0; i < n; i++) {
164         Insert(root, a[i]);
165     }
166
167     return root;
168 }
169
170 // Adjustments after left child deletion in the red-black tree
171 void deleteLeftAdjust(Ref& root, Ref& x) {
172     Ref w = x->parent->right;
173
174     if (w->color == RED) {
175         w->color = BLACK;
176         x->parent->color = RED;
177         leftRotate(root, x->parent);
178         w = x->parent->right;
179     }
180
181     if ((w->right->color == BLACK) && (w->left->color == BLACK)) {
182         w->color = RED;
183         x = x->parent;
184     }
185     else {
186         if (w->right->color == BLACK) {
187             w->left->color = BLACK;
188             w->color = RED;
189             rightRotate(root, w);
190             w = x->parent->right;
191         }
192         w->color = x->parent->color;
193         x->parent->color = w->right->color = BLACK;
194         leftRotate(root, x->parent);
195         x = root;
196     }
```

```
197 }
198
199 // Adjustments after right child deletion in the red-black tree
200 void deleteRightAdjust(Ref& root, Ref& x) {
201     Ref w = x->parent->left;
202
203     if (w->color == RED) {
204         w->color = BLACK;
205         x->parent->color = RED;
206         leftRotate(root, x->parent);
207         w = x->parent->left;
208     }
209
210     if ((w->left->color == BLACK) && (w->right->color == BLACK)) {
211         w->color = RED;
212         x = x->parent;
213     }
214     else {
215         if (w->left->color == BLACK) {
216             w->right->color = BLACK;
217             w->color = RED;
218             leftRotate(root, w);
219             w = x->parent->left;
220         }
221         w->color = x->parent->color;
222         x->parent->color = w->left->color = BLACK;
223         rightRotate(root, x->parent);
224         x = root;
225     }
226 }
227
228 // Fix-up routine after deletion in the red-black tree
229 void deleteFixUp(Ref root, Ref x) {
230     while ((x->color == BLACK) && (x != root)) {
231         if (x == x->parent->left) deleteLeftAdjust(root, x);
232         else deleteRightAdjust(root, x);
233     }
234     x->color = BLACK;
235 }
236
237 // Search for a key in the red-black tree and return the corresponding node
238 Ref lookup(Ref root, int key) {
239     Ref p = root;
240     while (p != nil) {
241         if (key == p->key) return p;
242
243         if (key < p->key) p = p->left;
244         else p = p->right;
```

```
245     }
246
247     return nil;
248 }
249
250 // Find the predecessor of a given node in the red-black tree
251 Ref findPredecessor(Ref z) {
252     if (z->left == nil) {
253         std::cout << "This node does not have predecessor!";
254         return nullptr;
255     }
256
257     Ref y = z->left;
258     while (y->right != nil) y = y->right;
259     return y;
260 }
261
262 // Remove a key from the red-black tree
263 void Remove(Ref& root, int k) {
264     Ref z = lookup(root, k);
265     if (z == nil) return;
266
267     Ref y = (z->left == nil) || (z->right == nil) ? z : findPredecessor  ↗
        (z);
268
269     Ref x = (y->left == nil) ? y->right : y->left;
270
271     x->parent = y->parent;
272     if (y->parent == nil) root = x;
273     else {
274         if (y == y->parent->left) y->parent->left = x;
275         else y->parent->right = x;
276     }
277
278     if (y != z) z->key = y->key;
279     if (y->color == BLACK) {
280         deleteFixUp(root, x);
281     }
282
283     delete y;
284 }
```