```cpp
//Red black Tree
#include <iostream>

using namespace std;

// Enum representing the color of a node in the red-black tree
enum Color {
    RED, BLACK,
};

// Node structure for the red-black tree
typedef struct RBNode* Ref;
struct RBNode {
    int key;
    Color color;
    Ref parent, left, right;
};

// Global variable representing the nil (sentinel) node
Ref nil;

// Function to create a new node with given key, color, and nil reference
Ref createNode(int key, Color color, Ref nil) {
    Ref p = new RBNode{ key, color, nil, nil, nil };
    return p;
}

// Left rotation operation in the red-black tree
void leftRotate(Ref& root, Ref x) {
    Ref y = x->right;
    x->right = y->left;

    if (y->left != nil) {
        y->left->parent = x;
    }
    y->parent = x->parent;

    if (x->parent == nil) {
        root = y;
    }
    else {
        if (x == x->parent->left) {
            x->parent->left = y;
        }
        else {
            x->parent->right = y;
        }
    }

```

```cpp
    y->left = x;
    x->parent = y;
}

// Right rotation operation in the red-black tree
void rightRotate(Ref& root, Ref x) {
    Ref y = x->left;
    x->left = y->right;

    if (y->right != nil) {
        y->right->parent = x;
    }
    y->parent = x->parent;

    if (x->parent == nil) {
        root = y;
    }
    else {
        if (x == x->parent->right) {
            x->parent->right = y;
        }
        else {
            x->parent->left = y;
        }
    }

    y->right = x;
    x->parent = y;
}

// Binary Search Tree (BST) insertion operation
void BST_Insert(Ref& root, Ref x) {
    Ref y = nil, z = root;
    while (z != nil) {
        y = z;

        if (x->key < z->key) z = z->left;
        else if (x->key > z->key) z = z->right;
        else return; // Key already exists, do nothing
    }

    x->parent = y;
    if (y == nil) root = x;
    else {
        if (x->key < y->key) y->left = x;
        else y->right = x;
    }
}

```

```cpp
// Adjustments after left child insertion in the red-black tree
void insertionLeftAdjust(Ref& root, Ref& x) {
    Ref u = x->parent->parent->right;
    if (u->color == RED) {
        x->parent->color = u->color = BLACK;
        x->parent->parent->color = RED;
        x = x->parent->parent;
    }
    else {
        if (x == x->parent->right) {
            x = x->parent;
            leftRotate(root, x);
        }
        x->parent->color = BLACK;
        x->parent->color = BLACK;
        x->parent->parent->color = RED;
        rightRotate(root, x->parent->parent);
    }
}

// Adjustments after right child insertion in the red-black tree
void insertionRightAdjust(Ref& root, Ref& x) {
    Ref u = x->parent->parent->left;
    if (u->color == RED) {
        x->parent->color = u->color = BLACK;
        x->parent->parent->color = RED;
        x = x->parent->parent;
    }
    else {
        if (x == x->parent->left) {
            x = x->parent;
            rightRotate(root, x);
        }
        x->parent->color = BLACK;
        x->parent->color = BLACK;
        x->parent->parent->color = RED;
        leftRotate(root, x->parent->parent);
    }
}

// Fix-up routine after insertion in the red-black tree
void insertionFixUp(Ref& root, Ref x) {
    while (x->parent->color == RED) {
        if (x->parent == x->parent->parent->left) {
            insertionLeftAdjust(root, x);
        }
        else {
            insertionRightAdjust(root, x);
        }
```

```
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 }
285
286 struct Node {
287     int key;
288     Node* left, * right;
289     int height;
290 };
291
292 bool isGreater(int a, int b) {
```

```
293     return a > b;
294 }
295
296 bool isSmaller(int a, int b) {
297     return a < b;
298 }
299
300 bool isEqual(int a, int b) {
301     return a == b;
302 }
303
304 int getHeight(Node* root) {
305     if (!root) return 0;
306     return root->height;
307 }
308
309 void fixHeight(Node*& root) {
310     root->height = 1 + std::max(getHeight(root->left), getHeight(root->right));
311 }
312
313 void leftRotate(Node*& root) {
314     Node* B = root->right;
315     Node* Y = B->left;
316
317     B->left = root;
318     root->right = Y;
319
320     fixHeight(root);
321     fixHeight(B);
322
323     root = B;
324 }
325
326 void rightRotate(Node*& root) {
327     Node* B = root->left;
328     Node* Y = B->right;
329
330     B->right = root;
331     root->left = Y;
332
333     fixHeight(root);
334     fixHeight(B);
335
336     root = B;
337 }
338
339 int getBalanceFactor(Node* node) {
340     return getHeight(node->left) - getHeight(node->right);
```

```
341 }
342
343 void insertNode(Node*& root, int key) {
344     if (!root) {
345         root = new Node{ key, nullptr, nullptr, 1 };
346         return;
347     }
348
349     if (isSmaller(key, root->key)) {
350         insertNode(root->left, key);
351     }
352     else if (isGreater(key, root->key)) {
353         insertNode(root->right, key);
354     }
355     else return;
356
357     fixHeight(root);
358
359     int bf = getBalanceFactor(root);
360
361     if (bf > 1 && isSmaller(key, root->left->key)) {
362         rightRotate(root);
363         return;
364     }
365
366     if (bf < -1 && isGreater(key, root->right->key)) {
367         leftRotate(root);
368         return;
369     }
370
371     if (bf > 1 && isGreater(key, root->left->key)) {
372         leftRotate(root->left);
373         rightRotate(root);
374         return;
375     }
376
377     if (bf < -1 && isSmaller(key, root->right->key)) {
378         rightRotate(root->right);
379         leftRotate(root);
380         return;
381     }
382 }
383
384 Node* findPredecessor(Node* node) {
385     if (!node->left) {
386         std::cout << "This node does not have predecessor!";
387         return nullptr;
388     }
389
```

```
390     Node* y = node->left;
391     while (y->right) y = y->right;
392     return y;
393 }
394 
395 void deleteNode(Node*& root, int key) {
396     if (!root) return;
397     else if (isSmaller(key, root->key)) deleteNode(root->left, key);
398     else if (isGreater(key, root->key)) deleteNode(root->right, key);
399     else {
400         if (!root->left) {
401             Node* temp = root->right;
402             delete root;
403             root = temp;
404             return;
405         }
406         else if (!root->right) {
407             Node* temp = root->left;
408             delete root;
409             root = temp;
410             return;
411         }
412         else {
413             Node* pred = findPredecessor(root);
414             root->key = pred->key;
415 
416             deleteNode(root->left, pred->key);
417         }
418     }
419 
420     fixHeight(root);
421 
422     int bf = getBalanceFactor(root);
423 
424     if (bf > 1 && getBalanceFactor(root->left) >= 0) {
425         rightRotate(root);
426     }
427 
428     else if (bf < -1 && getBalanceFactor(root->right) <= 0) {
429         leftRotate(root);
430     }
431 
432     else if (bf > 1 && getBalanceFactor(root->left) < 0) {
433         leftRotate(root->left);
434         rightRotate(root);
435     }
436 
437     else if (bf < -1 && getBalanceFactor(root->right) > 0) {
438         rightRotate(root->right);
```

```
439             leftRotate(root);
440         }
441     }
```