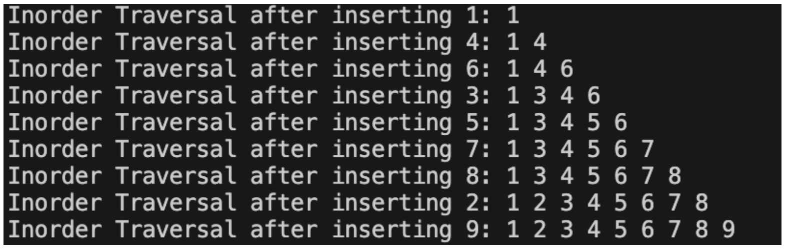
1.

```
#include <iostream>

class RedBlackTree {
private:
    struct Node {
        int data;
        Node* left;
        Node* right;
        char colour; // 'R' for Red, 'B' for Black
        Node* parent;

        Node(int data) : data(data), left(nullptr), right(nullptr), colour('R'), parent(nullptr) {}
    };

    Node* root;

    Node* rotateLeft(Node* node) {
        Node* x = node->right;
        node->right = x->left;
        if (x->left != nullptr)
            x->left->parent = node;

        x->parent = node->parent;
        if (node->parent == nullptr) {
            root = x;
        } else if (node == node->parent->left) {
            node->parent->left = x;
        } else {
            node->parent->right = x;
        }
        x->left = node;
        node->parent = x;
        return x;
    }

    Node* rotateRight(Node* node) {
        Node* x = node->left;
        node->left = x->right;
        if (x->right != nullptr)
            x->right->parent = node;

        x->parent = node->parent;
    if (node->parent == nullptr) {
        root = x;
    } else if (node == node->parent->right) {
        node->parent->right = x;
    } else {
        node->parent->left = x;
    }
    x->right = node;
    node->parent = x;
    return x;
}

void fixViolation(Node* node) {
    while (node != root && node->parent->colour == 'R') {
        if (node->parent == node->parent->parent->left) {
            Node* uncle = node->parent->parent->right;
            if (uncle != nullptr && uncle->colour == 'R') {
                node->parent->colour = 'B';
                uncle->colour = 'B';
                node->parent->parent->colour = 'R';
                node = node->parent->parent;
            } else {
                if (node == node->parent->right) {
                    node = node->parent;
                    rotateLeft(node);
                }
                node->parent->colour = 'B';
                node->parent->parent->colour = 'R';
                rotateRight(node->parent->parent);
            }
        } else {
            Node* uncle = node->parent->parent->left;
            if (uncle != nullptr && uncle->colour == 'R') {
                node->parent->colour = 'B';
                uncle->colour = 'B';
                node->parent->parent->colour = 'R';
                node = node->parent->parent;
            } else {
                if (node == node->parent->left) {
                    node = node->parent;
                    rotateRight(node);
                }
                node->parent->colour = 'B';
                node->parent->parent->colour = 'R';
```

```
                    rotateLeft(node->parent->parent);
                }
            }
        }
        root->colour = 'B';
    }

    Node* insertHelp(Node* root, int data) {
        if (root == nullptr)
            return new Node(data);

        if (data < root->data) {
            root->left = insertHelp(root->left, data);
            root->left->parent = root;
        } else if (data > root->data) {
            root->right = insertHelp(root->right, data);
            root->right->parent = root;
        }
        return root;
    }

    void inorderTraversalHelper(Node* node) {
        if (node != nullptr) {
            inorderTraversalHelper(node->left);
            std::cout << node->data << " ";
            inorderTraversalHelper(node->right);
        }
    }

    void printTreeHelper(Node* node, int space) {
        if (node != nullptr) {
            space += 10;
            printTreeHelper(node->right, space);
            std::cout << std::endl;
            for (int i = 10; i < space; i++)
                std::cout << " ";
            std::cout << node->data << " (" << node->colour << ")" << std::endl;
            printTreeHelper(node->left, space);
        }
    }

public:
    RedBlackTree() : root(nullptr) {}
    void insert(int data) {
        if (root == nullptr) {
            root = new Node(data);
            root->colour = 'B'; // Root is always black
        } else {
            Node* newNode = insertHelp(root, data);
            fixViolation(newNode);
        }
    }

    void inorderTraversal() {
        inorderTraversalHelper(root);
        std::cout << std::endl;
    }

    void printTree() {
        printTreeHelper(root, 0);
    }

    ~RedBlackTree() {
        // Destructor to free memory (implement if needed)
    }
};

int main() {
    RedBlackTree tree;
    int arr[] = {1, 4, 6, 3, 5, 7, 8, 2, 9};
    for (int i : arr) {
        tree.insert(i);
        std::cout << "Inorder Traversal after inserting " << i << ": ";
        tree.inorderTraversal();
    }
    std::cout << "Final Tree Structure:" << std::endl;
    tree.printTree();
    return 0;
}
```

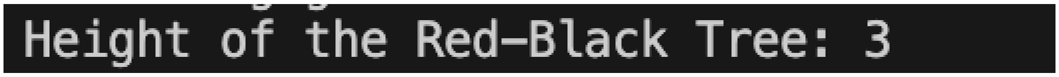


```
Inorder Traversal after inserting 1: 1
Inorder Traversal after inserting 4: 1 4
Inorder Traversal after inserting 6: 1 4 6
Inorder Traversal after inserting 3: 1 3 4 6
Inorder Traversal after inserting 5: 1 3 4 5 6
Inorder Traversal after inserting 7: 1 3 4 5 6 7
Inorder Traversal after inserting 8: 1 3 4 5 6 7 8
Inorder Traversal after inserting 2: 1 2 3 4 5 6 7 8
Inorder Traversal after inserting 9: 1 2 3 4 5 6 7 8 9
```

2.

```
#include <iostream>

class RedBlackTree {
private:
    struct Node {
        int data;
        Node* left;
        Node* right;
        char colour; // 'R' for Red, 'B' for Black
        Node* parent;

        Node(int data) : data(data), left(nullptr), right(nullptr), colour('R'), parent(nullptr) {}
    };

    Node* root;

    Node* rotateLeft(Node* node) {
        Node* x = node->right;
        node->right = x->left;
        if (x->left != nullptr)
            x->left->parent = node;

        x->parent = node->parent;
        if (node->parent == nullptr) {
            root = x;
        } else if (node == node->parent->left) {
            node->parent->left = x;
        } else {
            node->parent->right = x;
        }
        x->left = node;
        node->parent = x;
        return x;
    }

    Node* rotateRight(Node* node) {
        Node* x = node->left;
        node->left = x->right;
        if (x->right != nullptr)
            x->right->parent = node;

        x->parent = node->parent;
        if (node->parent == nullptr) {
```

```
        root = x;
    } else if (node == node->parent->right) {
        node->parent->right = x;
    } else {
        node->parent->left = x;
    }
    x->right = node;
    node->parent = x;
    return x;
}

void fixViolation(Node* node) {
    while (node != root && node->parent->colour == 'R') {
        if (node->parent == node->parent->parent->left) {
            Node* uncle = node->parent->parent->right;
            if (uncle != nullptr && uncle->colour == 'R') {
                node->parent->colour = 'B';
                uncle->colour = 'B';
                node->parent->parent->colour = 'R';
                node = node->parent->parent;
            } else {
                if (node == node->parent->right) {
                    node = node->parent;
                    rotateLeft(node);
                }
                node->parent->colour = 'B';
                node->parent->parent->colour = 'R';
                rotateRight(node->parent->parent);
            }
        } else {
            Node* uncle = node->parent->parent->left;
            if (uncle != nullptr && uncle->colour == 'R') {
                node->parent->colour = 'B';
                uncle->colour = 'B';
                node->parent->parent->colour = 'R';
                node = node->parent->parent;
            } else {
                if (node == node->parent->left) {
                    node = node->parent;
                    rotateRight(node);
                }
                node->parent->colour = 'B';
                node->parent->parent->colour = 'R';
                rotateLeft(node->parent->parent);
                }
            }
        }
        root->colour = 'B';
    }

    Node* insertHelp(Node* root, int data) {
        if (root == nullptr)
            return new Node(data);

        if (data < root->data) {
            root->left = insertHelp(root->left, data);
            root->left->parent = root;
        } else if (data > root->data) {
            root->right = insertHelp(root->right, data);
            root->right->parent = root;
        }
        return root;
    }

    int heightHelper(Node* node) {
        if (node == nullptr) {
            return 0;
        }
        int leftHeight = heightHelper(node->left);
        int rightHeight = heightHelper(node->right);
        return std::max(leftHeight, rightHeight) + 1;
    }

public:
    RedBlackTree() : root(nullptr) {}

    void insert(int data) {
        if (root == nullptr) {
            root = new Node(data);
            root->colour = 'B'; // Root is always black
        } else {
            Node* newNode = insertHelp(root, data);
            fixViolation(newNode);
        }
    }

    int height() {
        return heightHelper(root);
    }
};

int main() {
    RedBlackTree tree;
    int elements[] = {20, 15, 30, 10, 25, 35};

    for (int data : elements) {
        tree.insert(data);
    }

    std::cout << "Height of the Red-Black Tree: " << tree.height() << std::endl;

    return 0;
}
```

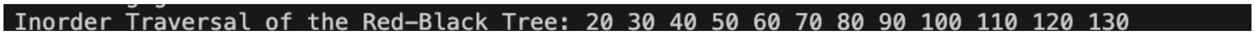


```
Height of the Red-Black Tree: 3
```

3.

```
#include <iostream>

using namespace std;

class RedBlackTree {
private:
    struct Node {
        int data;
        Node* left;
        Node* right;
        char colour; // 'R' for Red, 'B' for Black
        Node* parent;

        Node(int data) : data(data), left(nullptr), right(nullptr), colour('R'), parent(nullptr) {}
    };

    Node* root;

    Node* rotateLeft(Node* node) {
        Node* x = node->right;
        node->right = x->left;
        if (x->left != nullptr)
        x->left->parent = node;

    x->parent = node->parent;
    if (node->parent == nullptr) {
        root = x;
    } else if (node == node->parent->left) {
        node->parent->left = x;
    } else {
        node->parent->right = x;
    }
    x->left = node;
    node->parent = x;
    return x;
}

Node* rotateRight(Node* node) {
    Node* x = node->left;
    node->left = x->right;
    if (x->right != nullptr)
        x->right->parent = node;

    x->parent = node->parent;
    if (node->parent == nullptr) {
        root = x;
    } else if (node == node->parent->right) {
        node->parent->right = x;
    } else {
        node->parent->left = x;
    }
    x->right = node;
    node->parent = x;
    return x;
}

void fixViolation(Node* node) {
    while (node != root && node->parent->colour == 'R') {
        if (node->parent == node->parent->parent->left) {
            Node* uncle = node->parent->parent->right;
            if (uncle != nullptr && uncle->colour == 'R') {
                // Case 1: Uncle is red
                node->parent->colour = 'B';
                uncle->colour = 'B';
                node->parent->parent->colour = 'R';
                node = node->parent->parent; // Move up
            } else {
                // Case 2: Uncle is black
                if (node == node->parent->right) {
                    // Left rotation needed
                    node = node->parent;
                    rotateLeft(node);
                }
                node->parent->colour = 'B';
                node->parent->parent->colour = 'R';
                rotateRight(node->parent->parent);
            }
        } else {
            Node* uncle = node->parent->parent->left;
            if (uncle != nullptr && uncle->colour == 'R') {
                // Case 1: Uncle is red
                node->parent->colour = 'B';
                uncle->colour = 'B';
                node->parent->parent->colour = 'R';
                node = node->parent->parent; // Move up
            } else {
                // Case 2: Uncle is black
                if (node == node->parent->left) {
                    // Right rotation needed
                    node = node->parent;
                    rotateRight(node);
                }
                node->parent->colour = 'B';
                node->parent->parent->colour = 'R';
                rotateLeft(node->parent->parent);
            }
        }
    }
    root->colour = 'B'; // Ensure root is always black
}

Node* insertHelp(Node* root, int data) {
    if (root == nullptr)
        return new Node(data);

    if (data < root->data) {
        root->left = insertHelp(root->left, data);
        root->left->parent = root;
    } else if (data > root->data) {
        root->right = insertHelp(root->right, data);
```

```
            root->right->parent = root;
        }
        return root;
    }

    void inorderTraversalHelper(Node* node) {
        if (node != nullptr) {
            inorderTraversalHelper(node->left);
            cout << node->data << " ";
            inorderTraversalHelper(node->right);
        }
    }

public:
    RedBlackTree() : root(nullptr) {}

    void insert(int data) {
        if (root == nullptr) {
            root = new Node(data);
            root->colour = 'B'; // Root is always black
        } else {
            Node* newNode = insertHelp(root, data);
            fixViolation(newNode);
        }
    }

    void inorderTraversal() {
        inorderTraversalHelper(root);
        cout << endl;
    }

    ~RedBlackTree() {
        // Destructor to free memory (implement if needed)
    }
};

int main() {
    RedBlackTree tree;
    int elements[] = {20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130};

    for (int data : elements) {
        tree.insert(data);
    }
    cout << "Inorder Traversal of the Red-Black Tree: ";
    tree.inorderTraversal();

    return 0;
}
```



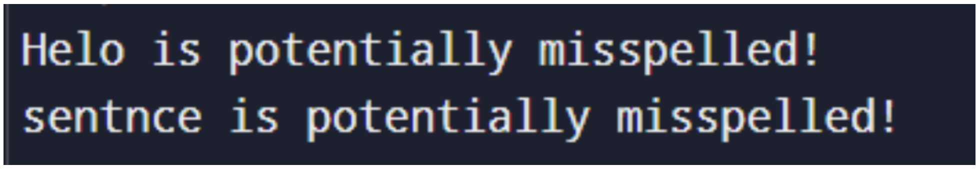
```
Inorder Traversal of the Red-Black Tree: 20 30 40 50 60 70 80 90 100 110 120 130
```

4.

```
#include <iostream> #include <sstream> #include <cctype> #include <set>
using namespace std;

enum Color { RED, BLACK };

struct Node {
string data;
Color color;
Node *left, *right, *parent;

Node(string data) {
this->data = data;
left = right = parent = nullptr; this->color = RED;

} };

class RedBlackTree { private:

Node *root;

void rotateLeft(Node *&pt) { Node *pt_right = pt->right; pt->right = pt_right->left;

if (pt->right != nullptr) pt->right->parent = pt;

pt_right->parent = pt->parent;

if (pt->parent == nullptr) root = pt_right;

else if (pt == pt->parent->left) pt->parent->left = pt_right;

else
pt->parent->right = pt_right;

pt_right->left = pt;

pt->parent = pt_right; }
```

```
void rotateRight(Node *&pt) { Node *pt_left = pt->left; pt->left = pt_left->right;

if (pt->left != nullptr) pt->left->parent = pt;

pt_left->parent = pt->parent;

if (pt->parent == nullptr) root = pt_left;

else if (pt == pt->parent->left) pt->parent->left = pt_left;

else
pt->parent->right = pt_left;

pt_left->right = pt;

pt->parent = pt_left; }

void fixViolation(Node *&pt) { Node *parent_pt = nullptr;
Node *grand_parent_pt = nullptr;

while ((pt != root) && (pt->color != BLACK) && (pt->parent->color == RED)) { parent_pt =
pt->parent;
grand_parent_pt = pt->parent->parent;

if (parent_pt == grand_parent_pt->left) { Node *uncle_pt = grand_parent_pt->right;

if (uncle_pt != nullptr && uncle_pt->color == RED) { grand_parent_pt->color = RED;
parent_pt->color = BLACK;
uncle_pt->color = BLACK;

pt = grand_parent_pt; }else{

if (pt == parent_pt->right) {

rotateLeft(parent_pt); pt = parent_pt; parent_pt = pt->parent;

}

rotateRight(grand_parent_pt); swap(parent_pt->color, grand_parent_pt->color); pt = parent_pt;

} }else{

Node *uncle_pt = grand_parent_pt->left;

if (uncle_pt != nullptr && uncle_pt->color == RED) { grand_parent_pt->color = RED;
parent_pt->color = BLACK;
uncle_pt->color = BLACK;
pt = grand_parent_pt; }else{

if (pt == parent_pt->left) { rotateRight(parent_pt); pt = parent_pt; parent_pt = pt->parent;

}

rotateLeft(grand_parent_pt); swap(parent_pt->color, grand_parent_pt->color); pt = parent_pt;

} }

}

root->color = BLACK; }

Node* BSTInsert(Node* root, Node *pt) { if (root == nullptr)

return pt;

if (pt->data < root->data) {
root->left = BSTInsert(root->left, pt); root->left->parent = root;

} else if (pt->data > root->data) { root->right = BSTInsert(root->right, pt); root->right->parent =
root;

}

return root; }

bool searchHelper(Node *root, string word) {
if (root == nullptr) return false;
if (word < root->data) return searchHelper(root->left, word);
else if (word > root->data) return searchHelper(root->right, word); else return true;

}

public:
RedBlackTree() { root = nullptr; }

void insert(const string &data) { Node *pt = new Node(data); root = BSTInsert(root, pt);
fixViolation(pt);

}

bool search(const string &word) { return searchHelper(root, word);

} };
```

```
void tokenizeAndCheck(RedBlackTree &dictTree, const string &sentence) { stringstream
ss(sentence);
string word;
while (ss >> word) {

for (int i = 0; i < word.length(); i++) { if (ispunct(word[i])) {

word.erase(i--, 1); }

}
if (!dictTree.search(word)) {

cout << word << " is potentially misspelled!" << endl; }

} }

int main() {
RedBlackTree dictTree;
set<string> dictionary = {"hello", "world", "this", "is", "a", "test", "dictionary"};

for (auto word : dictionary) { dictTree.insert(word);

}

string input = "Helo, this is a test sentnce!"; tokenizeAndCheck(dictTree, input);

return 0; }
```



```
Helo is potentially misspelled!
sentnce is potentially misspelled!
```

5.

```
#include <iostream> using namespace std;

enum Color { RED, BLACK };

struct Node { int data;
```

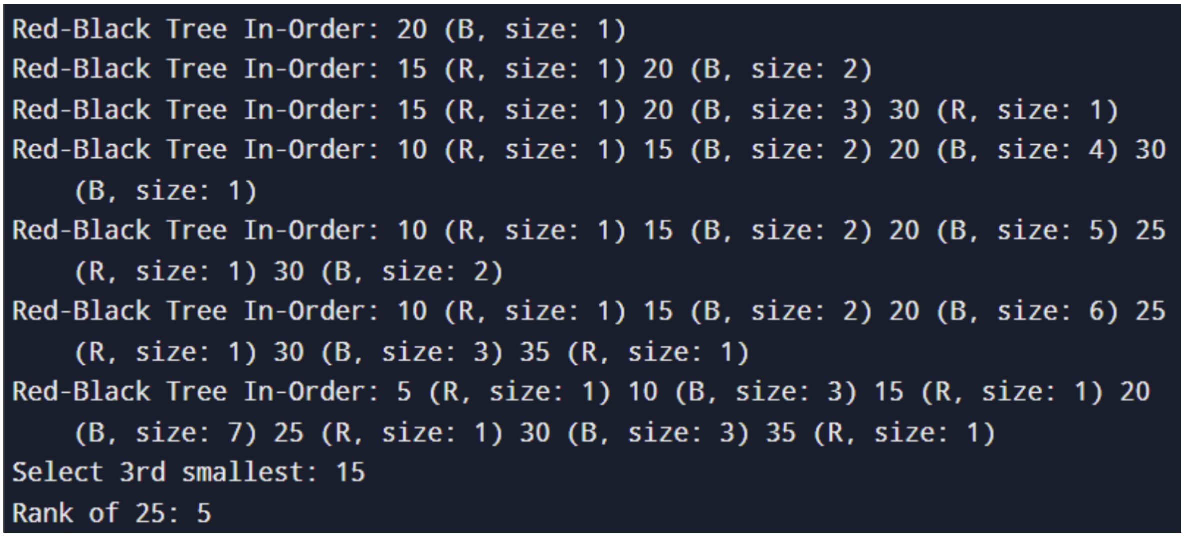
```
Color color;
Node *left, *right, *parent;
int size; // To store the size of the subtree rooted at this node

Node(int data) {
this->data = data;
left = right = parent = nullptr;
this->color = RED;
this->size = 1; // Initially, the size of a single node subtree is 1

} };

class RedBlackTree { private:

Node *root;

void rotateLeft(Node *&pt) { Node *pt_right = pt->right; pt->right = pt_right->left;

if (pt->right != nullptr) pt->right->parent = pt;

pt_right->parent = pt->parent;

if (pt->parent == nullptr) root = pt_right;

else if (pt == pt->parent->left) pt->parent->left = pt_right;

else
pt->parent->right = pt_right;

pt_right->left = pt; pt->parent = pt_right;

pt->size = 1 + size(pt->left) + size(pt->right);

pt_right->size = 1 + size(pt_right->left) + size(pt_right->right); }

void rotateRight(Node *&pt) { Node *pt_left = pt->left; pt->left = pt_left->right;

if (pt->left != nullptr) pt->left->parent = pt;

pt_left->parent = pt->parent;

if (pt->parent == nullptr) root = pt_left;

else if (pt == pt->parent->left) pt->parent->left = pt_left;

else
pt->parent->right = pt_left;
```

```
pt_left->right = pt; pt->parent = pt_left;

pt->size = 1 + size(pt->left) + size(pt->right);

pt_left->size = 1 + size(pt_left->left) + size(pt_left->right); }

void fixViolation(Node *&pt) { Node *parent_pt = nullptr;
Node *grand_parent_pt = nullptr;

while ((pt != root) && (pt->color != BLACK) && (pt->parent->color == RED)) { parent_pt =
pt->parent;
grand_parent_pt = pt->parent->parent;

if (parent_pt == grand_parent_pt->left) { Node *uncle_pt = grand_parent_pt->right;

if (uncle_pt != nullptr && uncle_pt->color == RED) { grand_parent_pt->color = RED;
parent_pt->color = BLACK;
uncle_pt->color = BLACK;

pt = grand_parent_pt; }else{

if (pt == parent_pt->right) { rotateLeft(parent_pt); pt = parent_pt; parent_pt = pt->parent;

}

rotateRight(grand_parent_pt); swap(parent_pt->color, grand_parent_pt->color); pt = parent_pt;

} }else{

Node *uncle_pt = grand_parent_pt->left;

if (uncle_pt != nullptr && uncle_pt->color == RED) { grand_parent_pt->color = RED;
parent_pt->color = BLACK;
uncle_pt->color = BLACK;

pt = grand_parent_pt; }else{

if (pt == parent_pt->left) { rotateRight(parent_pt);

pt = parent_pt;

parent_pt = pt->parent; }

rotateLeft(grand_parent_pt); swap(parent_pt->color, grand_parent_pt->color); pt = parent_pt;

} }

}
```

```
root->color = BLACK; }

Node* BSTInsert(Node* root, Node *pt) { if (root == nullptr)

return pt;

if (pt->data < root->data) {
root->left = BSTInsert(root->left, pt); root->left->parent = root;

} else if (pt->data > root->data) { root->right = BSTInsert(root->right, pt); root->right->parent =
root;

}

root->size = 1 + size(root->left) + size(root->right);

return root; }

int size(Node *node) {
return node == nullptr ? 0 : node->size;

}

Node* selectHelper(Node *node, int k) { if (node == nullptr) return nullptr;

int leftSize = size(node->left); if (k == leftSize + 1)

return node;
else if (k <= leftSize)

return selectHelper(node->left, k); else

return selectHelper(node->right, k - leftSize - 1); }

int rankHelper(Node *node, int x) { if (node == nullptr) return 0;

if (x < node->data)
return rankHelper(node->left, x);

else if (x > node->data)
return 1 + size(node->left) + rankHelper(node->right, x);

else
return size(node->left) + 1;

}
```

```
public:
RedBlackTree() { root = nullptr; }

void insert(const int &data) { Node *pt = new Node(data); root = BSTInsert(root, pt);
fixViolation(pt);

}

int select(int k) {
Node* result = selectHelper(root, k); if (result != nullptr)

return result->data; else

return -1; // Return -1 if k is out of bounds }

int rank(int x) {
return rankHelper(root, x);

}

void displayInOrder(Node *root) { if (root == nullptr)

return;

displayInOrder(root->left);
cout << root->data << " (" << (root->color == RED ? "R" : "B") << ", size: " << root->size << ") ";
displayInOrder(root->right);

}

void display() {
cout << "Red-Black Tree In-Order: "; displayInOrder(root);
cout << endl;

}

Node* getRoot() { return root;

} };

int main() { RedBlackTree tree;

int arr[] = {20, 15, 30, 10, 25, 35, 5}; for(inti=0;i<7;i++){

tree.insert(arr[i]);

tree.display(); }
```

```
cout << "Select 3rd smallest: " << tree.select(3) << endl; cout << "Rank of 25: " << tree.rank(25)
<< endl;

return 0; }
```



```
Red-Black Tree In-Order: 20 (B, size: 1)
Red-Black Tree In-Order: 15 (R, size: 1) 20 (B, size: 2)
Red-Black Tree In-Order: 15 (R, size: 1) 20 (B, size: 3) 30 (R, size: 1)
Red-Black Tree In-Order: 10 (R, size: 1) 15 (B, size: 2) 20 (B, size: 4) 30
    (B, size: 1)
Red-Black Tree In-Order: 10 (R, size: 1) 15 (B, size: 2) 20 (B, size: 5) 25
    (R, size: 1) 30 (B, size: 2)
Red-Black Tree In-Order: 10 (R, size: 1) 15 (B, size: 2) 20 (B, size: 6) 25
    (R, size: 1) 30 (B, size: 3) 35 (R, size: 1)
Red-Black Tree In-Order: 5 (R, size: 1) 10 (B, size: 3) 15 (R, size: 1) 20
    (B, size: 7) 25 (R, size: 1) 30 (B, size: 3) 35 (R, size: 1)
Select 3rd smallest: 15
Rank of 25: 5
```