Week 9-LAB A

Q1. Write a C/C++ program to insert elements into a Red-Black Tree and ensure the tree maintains its balancing properties.
Sample Input: Insert the following elements in sequence: 10, 20, 30, 15, 25, 5.
Output: Display the Red-Black Tree after each insertion.

Ans:

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

enum Color { RED, BLACK };

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

Node(int 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;
}

void inorderHelper(Node *root) {
if (root == nullptr)
return;

inorderHelper(root->left);
cout << root->data << " (" << (root->color == RED ? "R" : "B") << ") ";
inorderHelper(root->right);
}

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;
}

public:
RedBlackTree() { root = nullptr; }

void insert(const int &data) {
Node *pt = new Node(data);
root = BSTInsert(root, pt);
fixViolation(pt);
cout << "Tree after inserting " << data << ": ";
display();
cout << "\n";
}

void display() { inorderHelper(root); }
};

int main() {
RedBlackTree tree;

int arr[] = {10, 20, 30, 15, 25, 5};
```

```
for (int i = 0; i < 6; i++) {
tree.insert(arr[i]);
}
return 0;
}
```

```
Tree after inserting 10: 10 (B)
Tree after inserting 20: 10 (B) 20 (R)
Tree after inserting 30: 10 (R) 20 (B) 30 (R)
Tree after inserting 15: 10 (B) 15 (R) 20 (B) 30 (B)
Tree after inserting 25: 10 (B) 15 (R) 20 (B) 25 (R) 30 (B)
Tree after inserting 5: 5 (R) 10 (B) 15 (R) 20 (B) 25 (R) 30 (B)
```

Q2. Write a program to calculate the height of a Red-Black Tree.
Sample Input: Insert elements [20, 15, 30, 10, 25, 35], then compute the height of the tree.
Output: Display the height of the Red-Black Tree

Ans:

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

enum Color { RED, BLACK };

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

Node(int 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;
}

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

public:
RedBlackTree() { root = nullptr; }

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

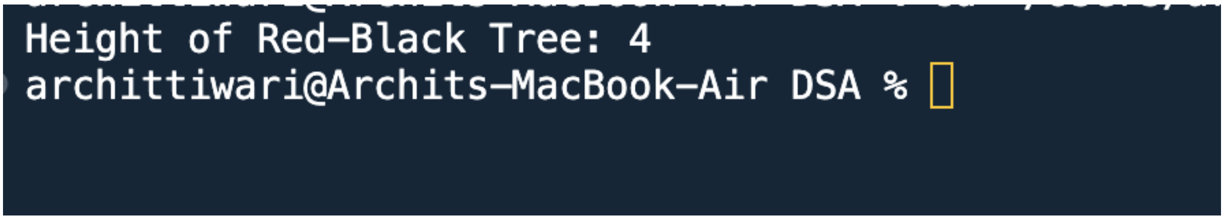
```
int height() {
return heightHelper(root);
}
};

int main() {
RedBlackTree tree;

int arr[] = {20, 15, 30, 10, 25,44,36,4,7,11, 35};
for (int i = 0; i < 6; i++) {
tree.insert(arr[i]);
}

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

return 0;
}
```



Q3. Create an RB Tree by inserting the nodes in following sequence: 20,30,40,50,60,70,80,90,100,110,120,130.

Ans:

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

enum Color { RED, BLACK };

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

Node(int 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;
}

public:
RedBlackTree() { root = nullptr; }

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

```
fixViolation(pt);
}
void displayInOrder(Node *root) {
if (root == nullptr)
return;

displayInOrder(root->left);
cout << root->data << " (" << (root->color == RED ? "R" : "B") << ") ";
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, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130}; for
(int i = 0; i < 12; i++) {
tree.insert(arr[i]);
tree.display();
}

return 0;
}
```

```
Red-Black Tree In-Order: 20 (B)
Red-Black Tree In-Order: 20 (B) 30 (R)
Red-Black Tree In-Order: 20 (R) 30 (B) 40 (R)
Red-Black Tree In-Order: 20 (B) 30 (B) 40 (B) 50 (R)
Red-Black Tree In-Order: 20 (B) 30 (B) 40 (R) 50 (B) 60 (R)
Red-Black Tree In-Order: 20 (B) 30 (B) 40 (B) 50 (R) 60 (B) 70 (R)
Red-Black Tree In-Order: 20 (B) 30 (B) 40 (B) 50 (R) 60 (R) 70 (B) 80 (R)
Red-Black Tree In-Order: 20 (B) 30 (R) 40 (B) 50 (B) 60 (B) 70 (R) 80 (B) 90 (R)
Red-Black Tree In-Order: 20 (B) 30 (R) 40 (B) 50 (B) 60 (B) 70 (R) 80 (R) 90 (B) 100 (R)
Red-Black Tree In-Order: 20 (B) 30 (B) 40 (B) 50 (B) 60 (B) 70 (B) 80 (B) 90 (R) 100 (B) 110 (R)
Red-Black Tree In-Order: 20 (B) 30 (B) 40 (B) 50 (B) 60 (B) 70 (B) 80 (B) 90 (R) 100 (R) 110 (B) 120 (R)
Red-Black Tree In-Order: 20 (B) 30 (B) 40 (B) 50 (B) 60 (B) 70 (R) 80 (B) 90 (B) 100 (B) 110 (R) 120 (B) 130 (R)
```

Q4. You are tasked with building a spell checker using a Red-Black Tree to efficiently store and search for valid words in a dictionary. Write a program that performs the following tasks:

a) Create a Red-Black Tree to store a dictionary of valid words. Each node in the tree should represent a word from the dictionary.

b) Tokenize the inputted String into words (ignoring punctuation and whitespace), and for each word:

c) Check if it exists in the Red-Black Tree (i.e., if it's a valid word). If it's not found, print the word as a potential misspelling.

Ans:

```
#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!


=== Code Execution Successful ===
```

Q5. Implement a data structure that supports dynamic order statistics using a Red Black Tree. The data structure should support the following operations: Insert(x): Insert an integer x into the data structure.
Select(k): Find the k^{th} smallest element in the data structure.
Rank(x): Find the rank of integer x in the data structure (i.e., the number of elements less than or equal to x).
Your program should implement the Red-Black Tree as the underlying data structure to achieve efficient operations. Make sure to maintain the Red-Black Tree properties after insertions and deletions.

Ans:

```
#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 (int i = 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 4th smallest: 20
Rank of 25: 5
archittiwari@Archits-MacBook-Air DSA %
```