МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

 Институт радиоэлектроники и информационных технологий

Кафедра «Вычислительные системы и технологии»

ОТЧЁТ

По лабораторной работе №1

по дисциплине «Тестирование ПО»

Вариант №20 «Выборка из красно-черного дерева»

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_            Скорынин С.С.

СТУДЕНТ:

  Смирнова С.В.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

        21-ПО

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород

2025

**Задание:**

Изучить средства тестирования, доступные в используемом вами стеке(.net C#, Java, Python, Golang и.т.д). .

· Разработать набор unit-тестов. Алгоритм или задачу необходимо согласовать с преподавателем.

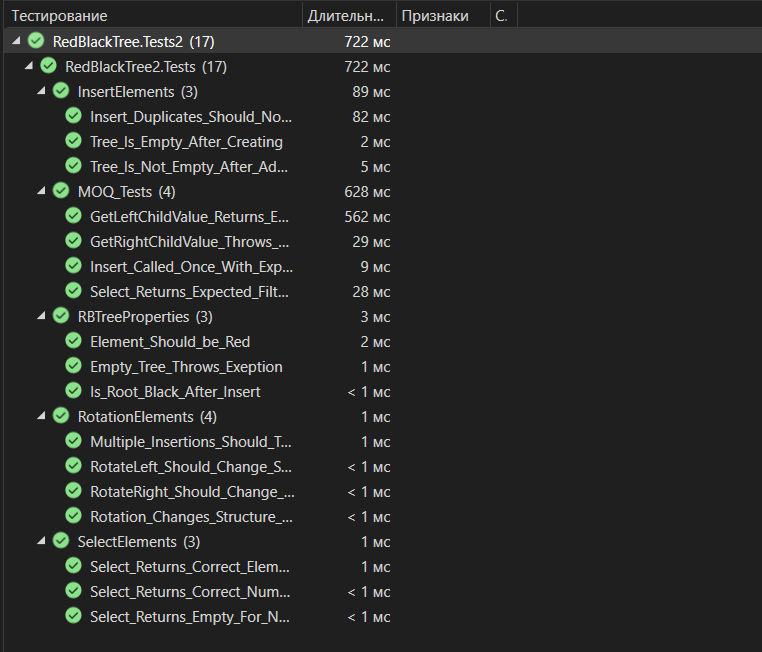
· Реализовать алгоритм или задач на языке С#, поддерживаемом платформой .NET. Использование других языков необходимо заранее согласовать с преподавателем.

· Обеспечить максимально возможное покрытие кода тестами. При написании задачи писать слабосвязанный код. Использовать mock- объекты. Использовать утилиты покрытия кода.

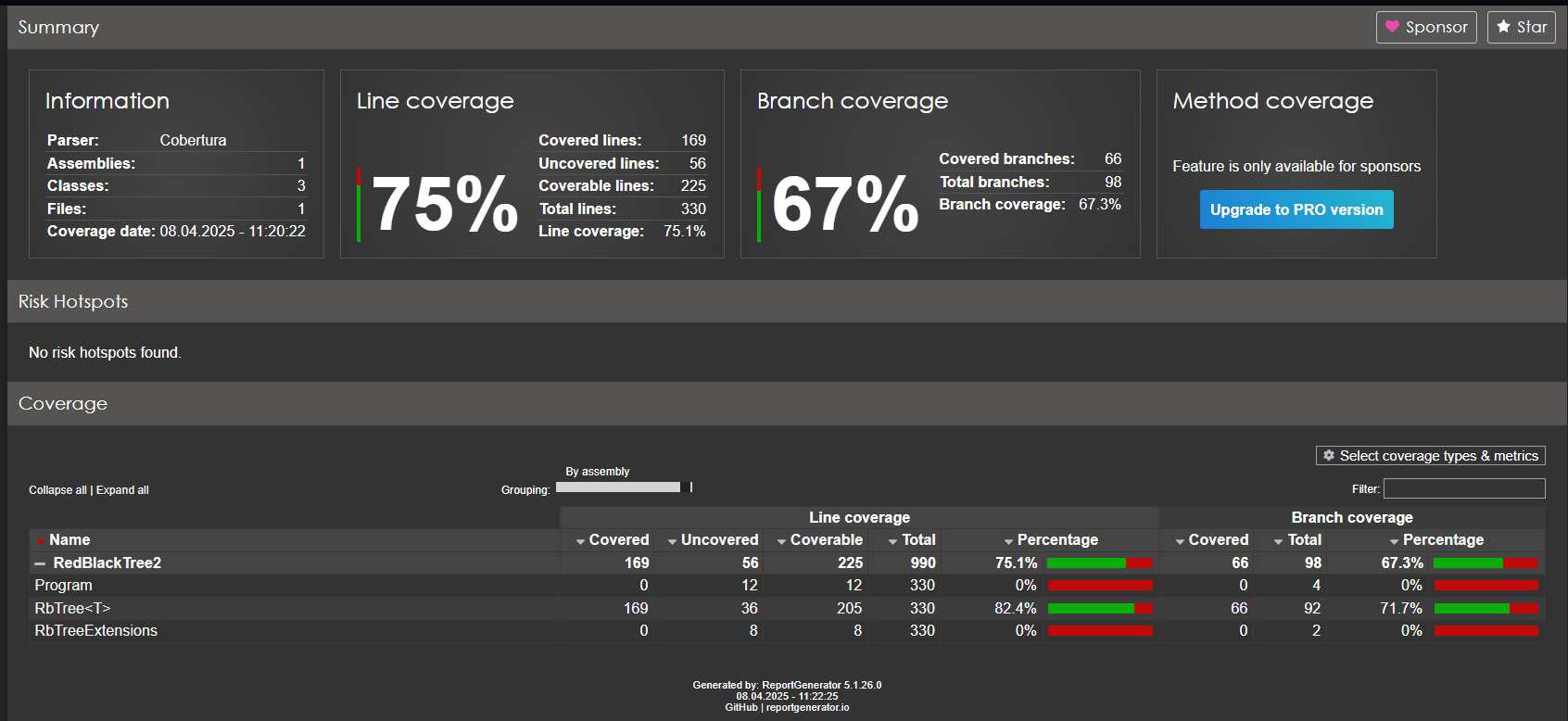
· Продемонстрировать: Работающую функциональность в соответствии с заданием o Проходящие unit-тесты. Показать с помощью утилиты процент покрытия кода unit-тестами.

· Сделать отчет по лабораторной работе.

**Ход работы:**

Обозреватель тестов:

В качестве утилиты для покрытия кода использовался Coverlet:



**Реализация красно-черного дерева:**

using System;

using System.Collections.Generic;

using System.Linq;

public interface IRbTree<T> where T : IComparable<T>

{

void Insert(T item);

IEnumerable<T> Select(Func<T, bool> condition);

T GetLeftChildValue(T parent);

T GetRightChildValue(T parent);

}

public class RbTree<T> : IRbTree<T> where T : IComparable<T>

{

public enum NodeColor { Red, Black }

public T GetLeftChildValue(T parent)

{

var node = FindNode(parent);

return node.Left != null ? node.Left.Data : throw new KeyNotFoundException("Left child not found");

}

public T GetRightChildValue(T parent)

{

var node = FindNode(parent);

return node.Right != null ? node.Right.Data : throw new KeyNotFoundException("Right child not found");

}

private Node FindNode(T value)

{

var current = \_root;

while (current != null)

{

int cmp = value.CompareTo(current.Data);

if (cmp == 0) return current;

current = cmp < 0 ? current.Left : current.Right;

}

throw new KeyNotFoundException("Value not found in tree");

}

public NodeColor GetRootColor()

{

if (\_root == null)

{

throw new InvalidOperationException("Tree is empty");

}

return \_root.Color;

}

public T GetRootValue()

{

if (\_root == null)

{

throw new InvalidOperationException("Tree is empty");

}

return \_root.Data;

}

public NodeColor GetNodeColor(T item)

{

if (\_root == null)

{

throw new InvalidOperationException("Tree is empty");

}

// Ищем узел с указанным значением

Node current = \_root;

while (current != null)

{

int comparison = item.CompareTo(current.Data);

if (comparison == 0)

{

return current.Color; // Нашли узел - возвращаем его цвет

}

current = comparison < 0 ? current.Left : current.Right;

}

throw new KeyNotFoundException($"Item {item} not found in tree");

}

public T GetNodeValue(T item)

{

if (\_root == null)

{

throw new InvalidOperationException("Tree is empty");

}

// Ищем узел с указанным значением

Node current = \_root;

while (current != null)

{

int comparison = item.CompareTo(current.Data);

if (comparison == 0)

{

return current.Data; // Нашли узел - возвращаем его value

}

current = comparison < 0 ? current.Left : current.Right;

}

throw new KeyNotFoundException($"Item {item} not found in tree");

}

public class Node

{

public T Data { get; set; }

public Node Left { get; set; }

public Node Right { get; set; }

public Node Parent { get; set; }

public NodeColor Color { get; set; }

public Node(T data)

{

Data = data;

Color = NodeColor.Red;

}

}

private Node \_root;

private int \_count;

public void Insert(T item)

{

var newNode = new Node(item);

\_root = Insert(\_root, newNode);

if (newNode.Parent != null || \_root == newNode)

{

FixViolations(newNode);

\_count++;

}

}

private Node Insert(Node root, Node newNode)

{

if (root == null)

return newNode;

int comparison = newNode.Data.CompareTo(root.Data);

if (comparison < 0)

{

root.Left = Insert(root.Left, newNode);

if (root.Left != null) // Добавляем проверку на null

root.Left.Parent = root;

}

else if (comparison > 0)

{

root.Right = Insert(root.Right, newNode);

if (root.Right != null)

root.Right.Parent = root;

}

// Если comparison == 0 значит элемент дубликат

return root;

}

private void FixViolations(Node node)

{

Node parent = null;

Node grandParent = null;

while (node != \_root && node.Color == NodeColor.Red && node.Parent.Color == NodeColor.Red)

{

parent = node.Parent;

grandParent = parent.Parent;

if (parent == grandParent.Left)

{

Node uncle = grandParent.Right;

if (uncle != null && uncle.Color == NodeColor.Red)

{

grandParent.Color = NodeColor.Red;

parent.Color = NodeColor.Black;

uncle.Color = NodeColor.Black;

node = grandParent;

}

else

{

if (node == parent.Right)

{

RotateLeft(parent);

node = parent;

parent = node.Parent;

}

RotateRight(grandParent);

SwapColors(parent, grandParent);

node = parent;

}

}

else

{

Node uncle = grandParent.Left;

if (uncle != null && uncle.Color == NodeColor.Red)

{

grandParent.Color = NodeColor.Red;

parent.Color = NodeColor.Black;

uncle.Color = NodeColor.Black;

node = grandParent;

}

else

{

if (node == parent.Left)

{

RotateRight(parent);

node = parent;

parent = node.Parent;

}

RotateLeft(grandParent);

SwapColors(parent, grandParent);

node = parent;

}

}

}

\_root.Color = NodeColor.Black;

}

private void RotateLeft(Node node)

{

Node rightChild = node.Right;

node.Right = rightChild.Left;

if (node.Right != null)

node.Right.Parent = node;

rightChild.Parent = node.Parent;

if (node.Parent == null)

\_root = rightChild;

else if (node == node.Parent.Left)

node.Parent.Left = rightChild;

else

node.Parent.Right = rightChild;

rightChild.Left = node;

node.Parent = rightChild;

}

private void RotateRight(Node node)

{

Node leftChild = node.Left;

node.Left = leftChild.Right;

if (node.Left != null)

node.Left.Parent = node;

leftChild.Parent = node.Parent;

if (node.Parent == null)

\_root = leftChild;

else if (node == node.Parent.Left)

node.Parent.Left = leftChild;

else

node.Parent.Right = leftChild;

leftChild.Right = node;

node.Parent = leftChild;

}

private void SwapColors(Node node1, Node node2)

{

var temp = node1.Color;

node1.Color = node2.Color;

node2.Color = temp;

}

public IEnumerable<T> Select(Func<T, bool> condition)

{

if (\_root == null)

yield break;

var stack = new Stack<Node>();

var current = \_root;

while (stack.Count > 0 || current != null)

{

if (current != null)

{

stack.Push(current);

current = current.Left;

}

else

{

current = stack.Pop();

if (condition(current.Data))

{

yield return current.Data;

}

current = current.Right;

}

}

}

}

public static class RbTreeExtensions

{

public static IEnumerable<T> Select<T>(this IEnumerable<T> sequence, Func<T, bool> condition) where T : IComparable<T>

{

var tree = new RbTree<T>();

// Построение дерева из последовательности

foreach (var item in sequence)

{

tree.Insert(item);

}

// Выборка из дерева

return tree.Select(condition);

}

}

class Program

{

static void Main(string[] args)

{

var testData = Enumerable.Range(0, 500000000).Select(x => x \* 2);

Console.WriteLine("Starting filtering...");

var filtered = testData.Select(x => x > 490000000 && x % 2 == 0);

Console.WriteLine("Results:");

foreach (var item in filtered.Take(10))

{

Console.WriteLine(item);

}

Console.WriteLine("Completed. Press any key to exit...");

Console.ReadKey();

}

}

**Юнит тесты:**

using NUnit.Framework;

using System.Linq;

using Moq;

using RedBlackTree2;

namespace RedBlackTree2.Tests

{

public class InsertElements

{

[SetUp]

public void Setup()

{

}

[Test]

public void Tree\_Is\_Empty\_After\_Creating()

{

var tree = new RbTree<int>();

var result = tree.Select(x => true);

//проверяем, что дерево пустое при созданиии

Assert.IsEmpty(result);

//Assert.Pass();

}

[Test]

public void Tree\_Is\_Not\_Empty\_After\_Add\_Elements()

{

var tree = new RbTree<int>();

tree.Insert(1);

var result = tree.Select(x => true);

//Проверяем, что дерево не пустое после добавления элемента

Assert.IsNotEmpty(result);

}

[Test]

public void Insert\_Duplicates\_Should\_Not\_Increase\_Count()

{

var tree = new RbTree<int>();

tree.Insert(1);

tree.Insert(1);

var result = tree.Select(x => true).Count();

Assert.AreEqual(1, result);

}

}

public class SelectElements

{

[Test]

public void Select\_Returns\_Correct\_Elements()

{

var tree = new RbTree<int>();

tree.Insert(1);

tree.Insert(2);

tree.Insert(3);

tree.Insert(4);

tree.Insert(5);

var result = tree.Select(x => x>3).ToList();

Assert.That(result[0],Is.EqualTo(4));

Assert.That(result[1], Is.EqualTo(5));

}

[Test]

public void Select\_Returns\_Correct\_Number\_Of\_Elements()

{

var tree = new RbTree<int>();

tree.Insert(1);

tree.Insert(2);

tree.Insert(3);

tree.Insert(4);

tree.Insert(5);

var result = tree.Select(x => x > 3).Count();

Assert.AreEqual(2, result);

}

[Test]

public void Select\_Returns\_Empty\_For\_Non\_Matching\_Condition()

{

var tree = new RbTree<int>();

tree.Insert(1);

tree.Insert(2);

tree.Insert(3);

var result = tree.Select(x => x > 3);

Assert.That(result, Is.Empty);

}

}

public class RBTreeProperties

{

[Test]

public void Is\_Root\_Black\_After\_Insert()

{

var tree = new RbTree<int>();

tree.Insert(1);

tree.Insert(2);

tree.Insert(3);

var result = tree.GetRootColor();

//Assert.That(result, Is.EqualTo(NodeColor.Black);

Assert.AreEqual(result, RbTree<int>.NodeColor.Black);

}

[Test]

public void Empty\_Tree\_Throws\_Exeption()

{

var tree = new RbTree<int>();

Assert.Throws<InvalidOperationException>(()=>tree.GetRootColor());

}

[Test]

public void Element\_Should\_be\_Red()

{

var tree = new RbTree<int>();

tree.Insert(1);

tree.Insert(2);

var selectRes = tree.Select(x => true).ToList();

var secondNodeColor = tree.GetNodeColor(selectRes[1]);

Assert.AreEqual(secondNodeColor, RbTree<int>.NodeColor.Red);

}

}

public class RotationElements

{

[Test]

public void Rotation\_Changes\_Structure\_Correctly()

{

var tree = new RbTree<int>();

tree.Insert(10);

tree.Insert(20);

tree.Insert(30);

var result = tree.Select(x=>true).ToList();

var rootColor = tree.GetNodeColor(result[1]);

// Проверяем, что корень изменился

Assert.AreEqual(20, tree.GetRootValue());

Assert.AreEqual(rootColor, RbTree<int>.NodeColor.Black);

}

[Test]

public void RotateLeft\_Should\_Change\_Structure\_Correctly()

{

var tree = new RbTree<int>();

tree.Insert(10); // Корень

tree.Insert(20); // Правая ветка

tree.Insert(30); // Вызовет RotateLeft

Assert.AreEqual(20, tree.GetRootValue());

Assert.AreEqual(10, tree.GetLeftChildValue(20));

Assert.AreEqual(30, tree.GetRightChildValue(20));

}

[Test]

public void RotateRight\_Should\_Change\_Structure\_Correctly()

{

var tree = new RbTree<int>();

tree.Insert(30); // Корень

tree.Insert(20); // Левая ветка

tree.Insert(10); // Вызовет RotateRight

Assert.AreEqual(20, tree.GetRootValue());

Assert.AreEqual(10, tree.GetLeftChildValue(20));

Assert.AreEqual(30, tree.GetRightChildValue(20));

}

[Test]

public void Multiple\_Insertions\_Should\_Trigger\_Correct\_Rotations()

{

var tree = new RbTree<int>();

var values = new[] { 20, 30, 25, 40, 22};

foreach (var value in values)

{

tree.Insert(value);

}

Assert.AreEqual(25, tree.GetRootValue());

Assert.AreEqual(20, tree.GetLeftChildValue(25));

Assert.AreEqual(30, tree.GetRightChildValue(25));

Assert.AreEqual(40, tree.GetRightChildValue(30));

Assert.AreEqual(22, tree.GetRightChildValue(20));

}

//[Test]

//public void After\_Rotation\_Tree\_Maintain\_Balance()

//{

// var tree = new RbTree<int>();

// tree.Insert(50);

// tree.Insert(30);

// tree.Insert(70);

// tree.Insert(60);

// tree.Insert(80);

// tree.Insert(90);

// Assert.AreEqual(70, tree.GetRootValue());

// Assert.AreEqual(50, tree.GetLeftChildValue(70));

// Assert.AreEqual(90, tree.GetRightChildValue(70));

// Assert.AreEqual(30, tree.GetLeftChildValue(50));

// Assert.AreEqual(60, tree.GetRightChildValue(50));

// Assert.AreEqual(90, tree.GetRightChildValue(80));

//}

}

public class MOQ\_Tests

{

[Test]

public void Insert\_Called\_Once\_With\_Expected\_Value()

{

var mock = new Mock<IRbTree<int>>();

mock.Object.Insert(42);

mock.Verify(tree => tree.Insert(42), Times.Once);

}

[Test]

public void Select\_Returns\_Expected\_Filtered\_Values()

{

var mock = new Mock<IRbTree<int>>();

mock.Setup(tree => tree.Select(It.IsAny<Func<int, bool>>()))

.Returns<Func<int, bool>>(predicate => new List<int> { 1, 2, 3, 4, 5 }.Where(predicate));

var result = mock.Object.Select(x => x > 2).ToList();

Assert.AreEqual(new List<int> { 3, 4, 5 }, result);

}

[Test]

public void GetLeftChildValue\_Returns\_Expected\_Value()

{

var mock = new Mock<IRbTree<string>>();

mock.Setup(tree => tree.GetLeftChildValue("parent")).Returns("left-child");

var result = mock.Object.GetLeftChildValue("parent");

Assert.AreEqual("left-child", result);

}

[Test]

public void GetRightChildValue\_Throws\_Exception\_When\_Not\_Found()

{

var mock = new Mock<IRbTree<int>>();

mock.Setup(tree => tree.GetRightChildValue(99)).Throws(new KeyNotFoundException("Right child not found"));

Assert.Throws<KeyNotFoundException>(() => mock.Object.GetRightChildValue(99));

}

}

}