|  |
| --- |
| Пермский филиал федерального государственного автономного образовательного учреждения высшего образования  «Национальный исследовательский университет  «Высшая школа экономики»  *Факультет социально-экономических и компьютерных наук* |
|  |
| Лузин Владимир Витальевич ЛАБОРАТОРНАЯ РАБОТА №12 КЛАССЫ-КОЛЛЕКЦИИ, СОЗДАВАЕМЫЕ ПОЛЬЗОВАТЕЛЕМ *Отчёт*  Студента образовательной программы «Разработка информационных систем для бизнеса» по направлению подготовки *09.03.04 Программная инженерия*   |  |  | | --- | --- | |  | Руководитель  Преподаватель кафедры ИТБ  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Марквирер В. Д. | |

# ОГЛАВЛЕНИЕ

[ЛАБОРАТОРНАЯ РАБОТА №12 КЛАССЫ-КОЛЛЕКЦИИ, СОЗДАВАЕМЫЕ ПОЛЬЗОВАТЕЛЕМ 1](#_Toc134836347)

[ОГЛАВЛЕНИЕ 2](#_Toc134836348)

[Постановка задачи 3](#_Toc134836349)

[Часть 1 3](#_Toc134836350)

[Часть 2 3](#_Toc134836351)

[Часть 3 4](#_Toc134836352)

[Часть 4 4](#_Toc134836353)

[Диаграмма классов 6](#_Toc134836354)

[Алгоритмы построения динамических структур данных 10](#_Toc134836355)

[Часть 1 10](#_Toc134836356)

[Часть 2 11](#_Toc134836357)

[Часть 3 12](#_Toc134836358)

[Программа (листинг) 13](#_Toc134836359)

[Код unit-тестов (листинг) 88](#_Toc134836360)

[Анализ покрытия тестов 115](#_Toc134836361)

# Постановка задачи

## Часть 1

Общая постановка задачи:

1. Сформировать двунаправленный список, в информационное поле записать объекты из иерархии классов лабораторной работы №10.
2. Распечатать полученный список.
3. Выполнить обработку списка в соответствии с заданием.
4. Распечатать полученный список.
5. Выполнить клонирование списка, показать что под объекты, хранящиеся в информационном поле выделена разная память
6. Удалить список из памяти.

Постановка задачи варианта №11 – Создать обобщённый двунаправленный список и дополнительно к общей задаче реализовать в нём добавление элемента с заданным номером. Всё взаимодействие с двунаправленным списком должно происходить через ошибкоустойчивый пользовательский интерфейс.

## Часть 2

Общая постановка задачи:

1. Сформировать идеально сбалансированное бинарное дерево, в информационное поле записать объекты из иерархии классов лабораторной работы №10.
2. Распечатать полученное дерево.
3. Выполнить обработку дерева в соответствии с заданием, вывести полученный результат.
4. Преобразовать идеально сбалансированное дерево в дерево поиска. Информационное поле склонировать.
5. Распечатать полученное дерево.
6. Удалить дерево из памяти.

Постановка задачи варианта №11 – Создать обобщённое идеально сбалансированное дерево поиска и дополнительно к общей задаче реализовать в нём поиск минимального элемента. Всё взаимодействие с идеально сбалансированным деревом поиска должно происходить через ошибкоустойчивый пользовательский интерфейс.

## Часть 3

Общая постановка задачи:

1. Создать хеш-таблицу и заполнить ее элементами.
2. Выполнить поиск элемента в хеш-таблице
3. Удалить найденный элемент из хеш-таблицы.
4. Выполнить поиск элемента в хеш-таблице
5. Показать, что будет при добавлении элемента в хеш-таблицу, если в таблице уже находится максимальное число элементов (для метода открытой адресации, для метода цепочек просто показать добавление в таблицу).

Постановка задачи варианта №11 – Создать хеш-таблицу через алгоритм открытой адресации и дополнительно к общей задаче реализовать в нём поиск и удаление по ключу. Всё взаимодействие с хеш-таблицей должно происходить через ошибкоустойчивый пользовательский интерфейс.

## Часть 4

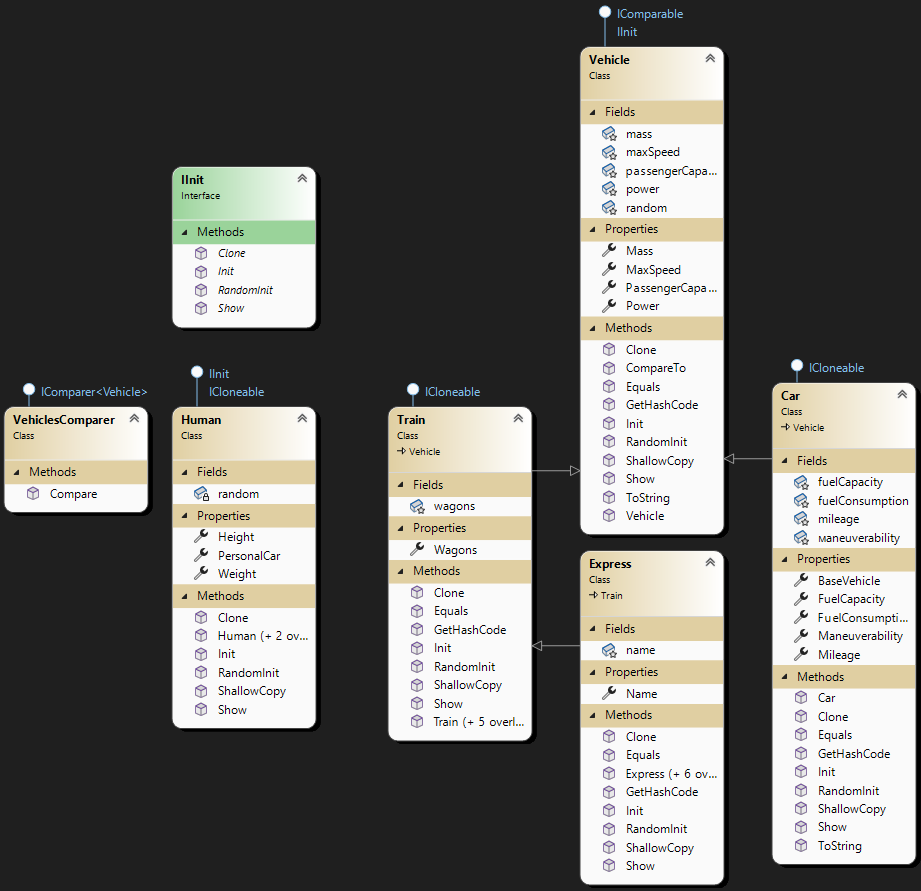
Общая постановка задачи:

1. Реализовать конструкторы MyCollection(), MyCollection (int capacity) и MyColletion(MyCollection c)
2. Для всех коллекций реализовать методы для добавления одного или нескольких элементов в коллекцию, методы для удаления одного или нескольких элементов из коллекции, метод для поиска элемента по ключу и/или по значению, метод для глубокого клонирования коллекции (вместе с элементами), метод для поверхностного копирования и индексатор для получения/изменения элемента по индексу/ключу.
3. Для всех коллекций реализовать интерфейсы IEnumerable, ICollection.
4. Написать демонстрационную программу, в которой создаются коллекции, и демонстрируется работа всех реализованных методов, в том числе, перебор коллекции циклом foreach.

Постановка задачи варианта №11 – Реализовать все задачи, описанные в общей постановке задачи, в идеально сбалансированном обобщённом дереве поиска. Всё взаимодействие с деревом должно происходить через ошибкоустойчивый пользовательский интерфейс.

# Диаграмма классов

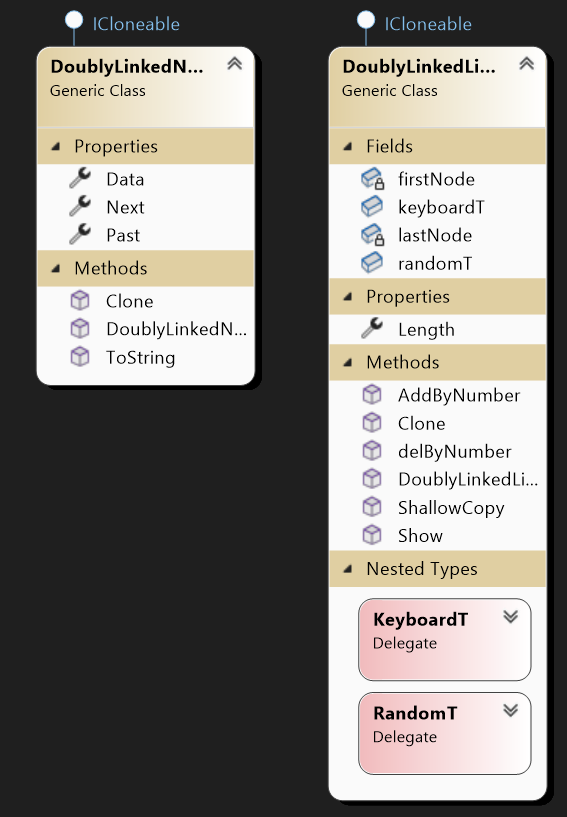
Диаграммы классов объектов иерархии представлена ниже (см. рисунок 1).



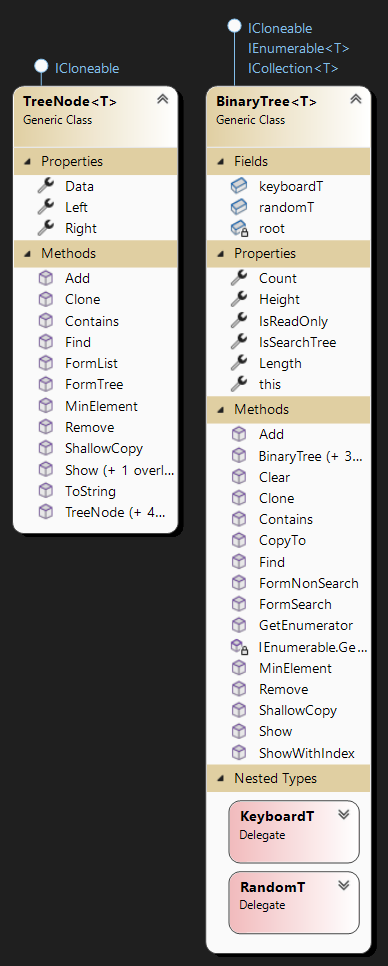
***Рис. 1. Диаграммы классов объектов иерархии***

В представленной диаграмме классов реализованы все необходимые классы. Дополнительно представлен интерфейс IInit.

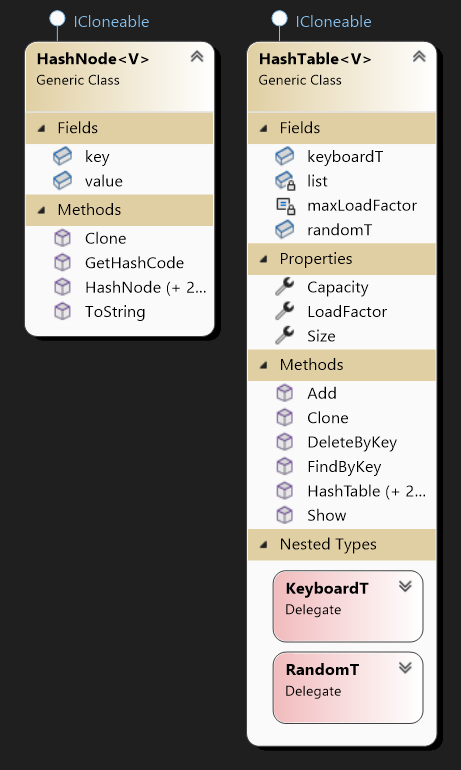
Диаграммы классов для всех 4 задач лабораторной работы №12 также представлены ниже (см. рисунки 2-4).



***Рис. 2. Диаграмма классов двунаправленного списка (задание 1)***



***Рис. 3. Диаграмма классов идеально сбалансированного дерева поиска (задания 2 и 4)***



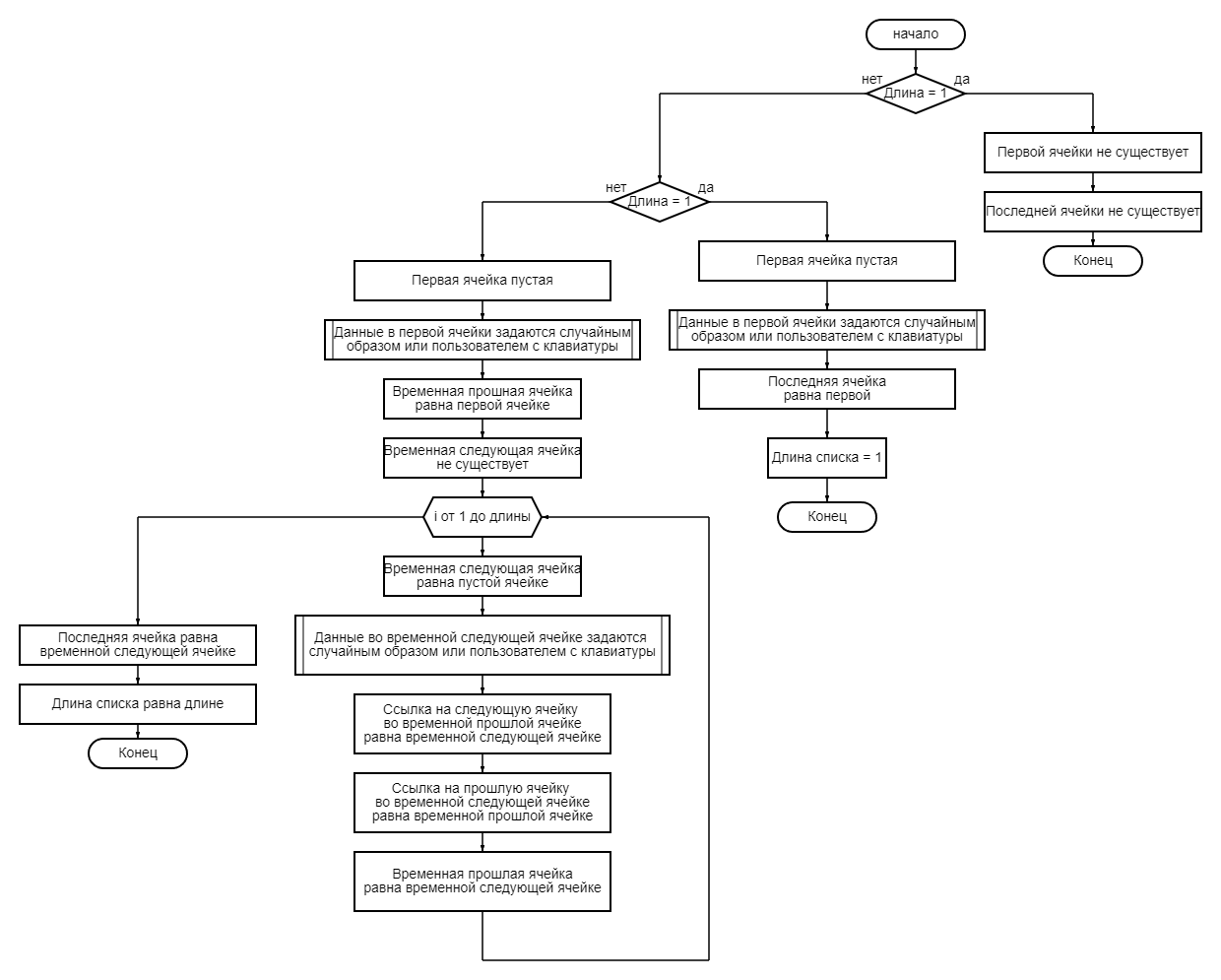
***Рис. 4. Диаграмма классов хеш-таблицы (задание 3)***

В представленных диаграммах классов реализован весь необходимый функционал. Дополнительно для внутренних классов использованы делегаты, с помощью который можно задать нужную генерацию обобщённого класса

# Алгоритмы построения динамических структур данных

## Часть 1

Алгоритм создания двунаправленного списка для заданной длины представлен ниже (см. рисунок 5).

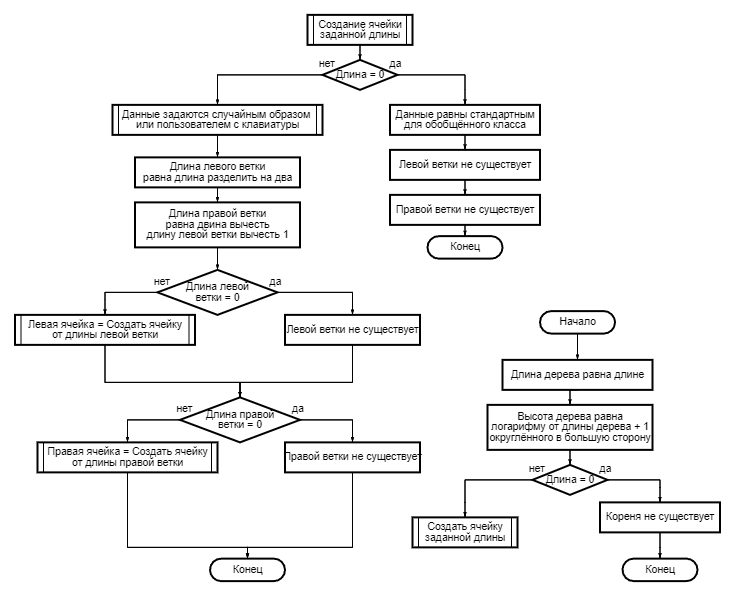


***Рис. 5. алгоритм создания двунаправленного списка***

Стоит отметить, что в представленном алгоритме подпрограммой является делегат создания значения обобщённого класса.

## Часть 2

Алгоритм создания идеально сбалансированного дерева поиска для заданной длины представлен ниже (см. рисунок 6).

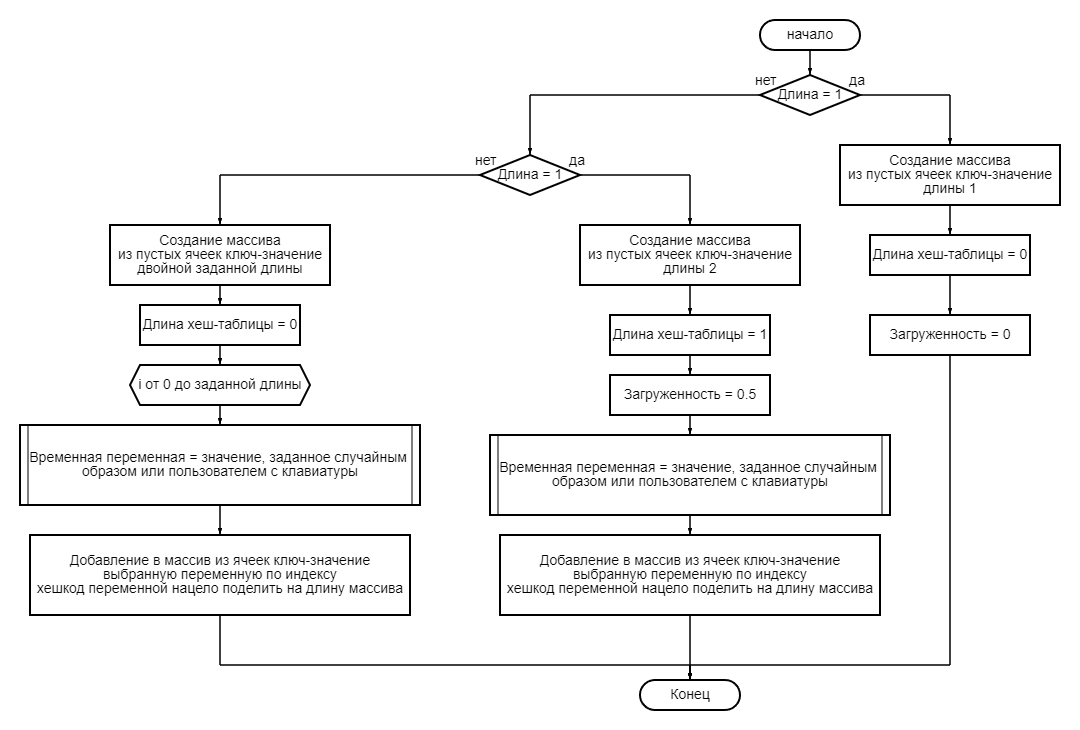


***Рис. 6. алгоритм создания идеально сбалансированного дерева***

Стоит отметить, что в представленном алгоритме подпрограммой является делегат создания значения обобщённого класса.

## Часть 3

Алгоритм создания Хеш-таблицы через открытую адресацию для заданной длины представлен ниже (см. рисунок 7).



***Рис. 7. алгоритм создания хеш-таблицы***

Стоит отметить, что в представленном алгоритме подпрограммой является делегат создания значения обобщённого класса.

# Программа (листинг)

Код программы и её классов представлен в листинге ниже.

Основной класс Program:

using System.Diagnostics.CodeAnalysis;

using ClassLibraryHSE1course;

namespace lab

{

[ExcludeFromCodeCoverage]

internal class Program

{

[ExcludeFromCodeCoverage]

public static void PrintLine(string str = "")

{

//Random rnd = new Random();

//ConsoleColor[] consoleColors = (ConsoleColor[]) ConsoleColor.GetValues(typeof(ConsoleColor));

//Console.ForegroundColor = consoleColors[rnd.Next(0,consoleColors.Length)];

Console.WriteLine(str);

//Console.ResetColor();

}

[ExcludeFromCodeCoverage]

public static void Print(string str = "")

{

//Random rnd = new Random();

//ConsoleColor[] consoleColors = (ConsoleColor[])ConsoleColor.GetValues(typeof(ConsoleColor));

//Console.ForegroundColor = consoleColors[rnd.Next(0, consoleColors.Length)];

Console.Write(str);

//Console.ResetColor();

}

[ExcludeFromCodeCoverage]

private static Vehicle RandomVehicle(int min, int max)

{

Vehicle vehicle = new Vehicle();

Random rnd = new Random();

int vehicleType = rnd.Next(1, 4);

switch (vehicleType)

{

case 1: // car

Car car = new Car();

car.RandomInit(min, max);

vehicle = car;

break;

case 2: // train

Train train = new Train();

train.RandomInit(min, max);

vehicle = train;

break;

case 3: // express

Express express = new Express();

express.RandomInit(min, max);

vehicle = express;

break;

}

return vehicle;

}

[ExcludeFromCodeCoverage]

private static Vehicle KeyboardVehicle(int i)

{

Vehicle vehicle = new Vehicle();

Console.Clear();

Output.PrintClasses(i);

int vehicleType = Input.ReadSwitch(1, 3);

switch (vehicleType)

{

case 1: // car

Car car = new Car();

car.Init();

vehicle = car;

break;

case 2: // train

Train train = new Train();

train.Init();

vehicle = train;

break;

case 3: // express

Express express = new Express();

express.Init();

vehicle = express;

break;

}

return vehicle;

}

[ExcludeFromCodeCoverage]

static void Main(string[] args)

{

Output.AddPrinterLine(PrintLine);

Output.AddPrinter(Print);

DoublyLinkedList<Vehicle> list = new DoublyLinkedList<Vehicle>();

DoublyLinkedList<Vehicle>.randomT = RandomVehicle;

DoublyLinkedList<Vehicle>.keyboardT = KeyboardVehicle;

BinaryTree<Vehicle> tree = new BinaryTree<Vehicle>();

BinaryTree<Vehicle>.randomT = RandomVehicle;

BinaryTree<Vehicle>.keyboardT = KeyboardVehicle;

HashTable<Vehicle> hashtable = new HashTable<Vehicle>();

HashTable<Vehicle>.randomT = RandomVehicle;

HashTable<Vehicle>.keyboardT = KeyboardVehicle;

int collectionNumber;

int menuNumber;

int fillType;

int min;

int max;

bool isCreated1 = false;

bool isCreated2 = false;

bool isCreated3 = false;

do

{

Output.Menu();

collectionNumber = Input.ReadSwitch(1,4); // 1 - Doubly linked list

switch (collectionNumber)

{

case 1:

do

{

Output.DataMenu(1);

menuNumber = Input.ReadSwitch(1, 6); // 1 - Form new collection, 2 - Print collection, 3 - Add collection element 4 - Сopy collection (demonstration) 5 delete

switch (menuNumber)

{

case 1: // Form new collection

int len;

do

{

len = Input.ReadInt("Enter list length", 2);

} while (Input.Warning(len, 1000));

if (len != 0)

{

Console.Clear();

Output.FillTypes();

fillType = Input.ReadSwitch(1, 2);

switch (fillType)

{

case 1: // Random

Input.RndNumbRange(out min, out max);

list = new DoublyLinkedList<Vehicle>(len, min, max);

break;

case 2: // Keyboard

list = new DoublyLinkedList<Vehicle>(len);

break;

}

}

else

{

list = new DoublyLinkedList<Vehicle>();

}

Console.Clear();

PrintLine("Doubly linked list successfully created");

Input.Cont();

isCreated1 = true;

break;

case 2: // Print

if (isCreated1)

{

list.Show();

}

else

{

PrintLine("Necessary to form new collection before printing");

}

Input.Cont();

break;

case 3: // Add element

if (isCreated1)

{

int pos = Input.ReadInt("Enter the position where the random vehicle will be added", 1);

Console.Clear();

Output.FillTypes(2);

fillType = Input.ReadSwitch(1, 2);

switch (fillType)

{

case 1: // Random

Input.RndNumbRange(out min, out max);

Console.Clear();

list.AddByNumber(pos-1, RandomVehicle(min, max));

break;

case 2: // Keyboard

list.AddByNumber(pos-1, KeyboardVehicle(0));

break;

}

Console.Clear();

PrintLine("Successfully added in Doubly linked list");

}

else

{

PrintLine("Necessary to form new collection before adding");

}

Input.Cont();

break;

case 4: // Сopy collection (demonstration)

// Create original list

DoublyLinkedList<IInit> testList = new DoublyLinkedList<IInit>();

PrintLine("Empty original list:");

testList.Show();

PrintLine();

DoublyLinkedList<IInit> listEmptyCopy = (DoublyLinkedList<IInit>)testList.Clone();

PrintLine("Empty copy list:");

listEmptyCopy.Show();

PrintLine();

listEmptyCopy.AddByNumber(1, RandomVehicle(3, 3));

PrintLine("Modified Empty copy:");

listEmptyCopy.Show();

PrintLine();

PrintLine("Empty original list:");

testList.Show();

PrintLine();

PrintLine();

PrintLine();

// Initialization

Human testHuman = new Human();

testHuman.RandomInit(100, 100);

Car TestCar = new Car();

TestCar.RandomInit(100, 100);

testList.AddByNumber(1, testHuman);

testList.AddByNumber(2, TestCar);

PrintLine("Original List"); // Print original hashtable

testList.Show();

DoublyLinkedList<IInit> shallowCopy = (DoublyLinkedList<IInit>)testList.ShallowCopy(); // Shallow copy

DoublyLinkedList<IInit> deepCopy = (DoublyLinkedList<IInit>)testList.Clone(); // Shallow copy

testHuman.RandomInit(1, 1);

TestCar.RandomInit(1, 1);

PrintLine("New original List:");

testList.Show();

PrintLine();

PrintLine("Shallow copy:");

shallowCopy.Show();

PrintLine();

shallowCopy.AddByNumber(3, RandomVehicle(3, 3));

PrintLine("Modified shallow copy:");

shallowCopy.Show();

PrintLine();

PrintLine("Deep copy:");

deepCopy.Show();

PrintLine();

deepCopy.AddByNumber(3, RandomVehicle(3, 3));

PrintLine("Modified deep copy:");

deepCopy.Show();

PrintLine();

PrintLine("New original List:");

testList.Show();

PrintLine();

Input.Cont();

break;

case 5: // delete

if (isCreated1)

{

list = new DoublyLinkedList<Vehicle>();

PrintLine("Doubly linked list successfully deleted");

isCreated1 = false;

}

else

{

PrintLine("Necessary to form new collection before deleting");

}

Input.Cont();

break;

}

} while (menuNumber != 6);

break;

case 2:

do

{

Output.DataMenu(2, tree.IsSearchTree);

menuNumber = Input.ReadSwitch(1, 13); // 1 - Form new collection, 2 - Print collection, 3 - Add collection element 4 - Сopy collection (demonstration) 5 delete

switch (menuNumber)

{

case 1: // Form new collection

int len;

do

{

len = Input.ReadInt("Enter tree length", 2);

} while (Input.Warning(len, 100));

if (len != 0)

{

Console.Clear();

Output.FillTypes();

fillType = Input.ReadSwitch(1, 2);

switch (fillType)

{

case 1: // Random

Input.RndNumbRange(out min, out max);

tree = new BinaryTree<Vehicle>(len, min, max);

break;

case 2: // Keyboard

tree = new BinaryTree<Vehicle>(len);

break;

}

}

else

{

tree = new BinaryTree<Vehicle>();

}

Console.Clear();

PrintLine("Binary tree successfully created");

Input.Cont();

isCreated2 = true;

break;

case 2: // Print

if (isCreated2)

{

if (tree.Length != 0)

{

tree.ShowWithIndex();

}

else

{

PrintLine("Empty tree");

}

}

else

{

PrintLine("Necessary to form new collection before printing");

}

Input.Cont();

break;

case 3: // Add

if (isCreated2)

{

Output.FillTypes(2);

fillType = Input.ReadSwitch(1, 2);

switch (fillType)

{

case 1: // Random

Input.RndNumbRange(out min, out max);

Console.Clear();

tree.Add(RandomVehicle(min, max));

break;

case 2: // Keyboard

Console.Clear();

tree.Add(KeyboardVehicle(0));

break;

}

}

else

{

PrintLine("Necessary to form new collection before adding");

}

Input.Cont();

break;

case 4: // min

if (isCreated2)

{

if (tree.Length != 0)

{

PrintLine("Minimum tree value: ");

Vehicle mn = tree.MinElement();

if (mn != null)

{

mn.Show();

}

else

{

PrintLine("Empty element");

}

}

else

{

PrintLine("Empty tree");

}

}

else

{

PrintLine("Necessary to form new collection before searching");

}

Input.Cont();

break;

case 5: // transform

if (isCreated2)

{

if (tree.Length != 0)

{

if (tree.IsSearchTree)

{

tree.FormNonSearch();

PrintLine("Perfectly balanced tree successfully created");

}

else

{

tree.FormSearch();

PrintLine("Perfectly balanced search tree successfully created");

PrintLine("All duplicate elements removed");

}

}

else

{

PrintLine("Impossible to transform empty tree");

}

}

else

{

PrintLine("Necessary to form new collection before transforming");

}

Input.Cont();

break;

case 6: // Сopy collection (demonstration)

// Create original list

BinaryTree<IInit> testTree = new BinaryTree<IInit>();

PrintLine("Empty original Tree:");

testTree.Show();

PrintLine();

BinaryTree<IInit> testTreeCopy = (BinaryTree<IInit>)testTree.Clone();

PrintLine("Empty copy Tree:");

testTreeCopy.Show();

PrintLine();

testTreeCopy.Add(RandomVehicle(3, 3));

PrintLine("Modified Empty copy:");

testTreeCopy.Show();

PrintLine();

PrintLine("Empty original Tree:");

testTree.Show();

PrintLine();

PrintLine();

PrintLine();

// Initialization

Human testHuman = new Human();

testHuman.RandomInit(100, 100);

Car TestCar = new Car();

TestCar.RandomInit(100, 100);

testTree.Add(testHuman);

testTree.Add(TestCar);

PrintLine("Original Tree"); // Print original hashtable

testTree.Show();

BinaryTree<IInit> shallowCopy = (BinaryTree<IInit>)testTree.ShallowCopy(); // Shallow copy

BinaryTree<IInit> deepCopy = (BinaryTree<IInit>)testTree.Clone(); // Shallow copy

testHuman.RandomInit(1, 1);

TestCar.RandomInit(1, 1);

PrintLine("New original Tree:");

testTree.Show();

PrintLine();

PrintLine("Shallow copy:");

shallowCopy.Show();

PrintLine();

shallowCopy.Add(RandomVehicle(3, 3));

PrintLine("Modified shallow copy:");

shallowCopy.Show();

PrintLine();

PrintLine("Deep copy:");

deepCopy.Show();

PrintLine();

deepCopy.Add(RandomVehicle(3, 3));

PrintLine("Modified deep copy:");

deepCopy.Show();

PrintLine();

PrintLine("New original Tree:");

testTree.Show();

PrintLine();

Input.Cont();

break;

case 7: // IEnumerable demonstration

if (isCreated2)

{

if (tree.Length == 0)

{

PrintLine("Empty tree");

}

else

{

int i = 0;

foreach (Vehicle veh in (BinaryTree<Vehicle>)tree)

{

i++;

Print($"{i}) ");

veh.Show();

}

}

}

else

{

PrintLine("Necessary to form new collection before handle");

}

Input.Cont();

break;

case 8: // Delete value

if (isCreated2)

{

if (tree.Length != 0)

{

bool isDeleted = tree.Remove(KeyboardVehicle(0));

if (isDeleted) PrintLine("Binary tree value successfully deleted");

else PrintLine("No such value in binary tree");

}

else

{

PrintLine("Impossible to delete elements from empty tree");

}

}

else

{

PrintLine("Necessary to form new collection before deleting elements");

}

Input.Cont();

break;

case 9: // Search value

if (isCreated2)

{

if (tree.Length != 0)

{

Vehicle veh = new Vehicle();

PrintLine("Enter values by which the element will be found");

veh.Init();

Console.Clear();

Vehicle foundedVal = tree.Find(veh);

if (foundedVal == default(Vehicle))

{

PrintLine("No such value");

}

else

{

PrintLine("Founded value:");

foundedVal.Show();

}

}

else

{

PrintLine("Impossible to search elements in empty tree");

}

}

else

{

PrintLine("Necessary to form new collection before searching elements");

}

Input.Cont();

break;

case 10: // indexer (Demonstration)

BinaryTree<IInit> indTree = new BinaryTree<IInit>();

for (int i = 1; i < 11; i++)

{

indTree.Add(RandomVehicle(i, i));

}

Console.Clear();

indTree.ShowWithIndex();

PrintLine();

PrintLine();

Print("First element = ");

((Vehicle)indTree[0]).Show();

Print("Third element = ");

((Vehicle)indTree[2]).Show();

Print("Last element = ");

((Vehicle)indTree[9]).Show();

PrintLine();

PrintLine();

indTree[0] = RandomVehicle(0, 0);

indTree[2] = RandomVehicle(0, 0);

indTree[9] = RandomVehicle(0, 0);

indTree.ShowWithIndex();

Input.Cont();

break;

case 11: // CopyTo (demonstration)

BinaryTree<IInit> copyToTree = new BinaryTree<IInit>();

for (int i = 1; i < 11; i++)

{

copyToTree.Add(RandomVehicle(i, i));

}

Console.Clear();

copyToTree.ShowWithIndex();

PrintLine();

PrintLine();

PrintLine();

Vehicle[] arr = new Vehicle[10];

copyToTree.CopyTo(arr,0);

int j = 0;

foreach (Vehicle item in arr)

{

j++;

Print($"{j}) ");

item.Show();

PrintLine();

}

Input.Cont();

break;

case 12: // delete

if (isCreated2)

{

tree.Clear();

PrintLine("Binary tree successfully deleted");

isCreated2 = false;

}

else

{

PrintLine("Necessary to form new collection before deleting");

}

Input.Cont();

break;

}

} while (menuNumber != 13);

break;

case 3:

do

{

Output.DataMenu(3);

menuNumber = Input.ReadSwitch(1, 7);

switch (menuNumber)

{

case 1: // Form new collection

int len;

do

{

len = Input.ReadInt("Enter hashtable length", 2);

} while (Input.Warning(len, 100));

if (len != 0)

{

Console.Clear();

Output.FillTypes();

fillType = Input.ReadSwitch(1, 2);

switch (fillType)

{

case 1: // Random

Input.RndNumbRange(out min, out max);

hashtable = new HashTable<Vehicle>(len, min, max);

break;

case 2: // Keyboard

hashtable = new HashTable<Vehicle>(len);

break;

}

}

else

{

hashtable = new HashTable<Vehicle>();

}

Console.Clear();

PrintLine("Hashtable successfully created");

Input.Cont();

isCreated3 = true;

break;

case 2: // Print

if (isCreated3)

{

hashtable.Show();

}

else

{

PrintLine("Necessary to form new collection before printing");

}

Input.Cont();

break;

case 3: // Add element

if (isCreated3)

{

Console.Clear();

Output.FillTypes(2);

fillType = Input.ReadSwitch(1, 2);

switch (fillType)

{

case 1: // Random

Input.RndNumbRange(out min, out max);

Console.Clear();

hashtable.Add(RandomVehicle(min, max));

break;

case 2: // Keyboard

hashtable.Add(KeyboardVehicle(0));

break;

}

Console.Clear();

PrintLine("Successfully added in Hashtable");

}

else

{

PrintLine("Necessary to form new collection before adding");

}

Input.Cont();

break;

case 4: // Search by key

if (isCreated3)

{

int lkey = Input.ReadInt("Enter key");

Vehicle? founded = hashtable.FindByKey(lkey);

if (founded != null)

{

PrintLine("Value found by key:");

founded.Show();

}

else

{

PrintLine("Value not found");

}

}

else

{

PrintLine("Necessary to form new collection before searching");

}

Input.Cont();

break;

case 5: // Delete by key

if (isCreated3)

{

int lkey = Input.ReadInt("Enter key");

bool isDeleted = hashtable.DeleteByKey(lkey);

if (isDeleted)

{

PrintLine("Value sucessfully deleted");

}

else

{

PrintLine("Value not found");

}

}

else

{

PrintLine("Necessary to form new collection before searching");

}

Input.Cont();

break;

case 6: // Delete hashtable

if (isCreated3)

{

hashtable = new HashTable<Vehicle>();

PrintLine("HashTable successfully deleted");

isCreated3 = false;

}

else

{

PrintLine("Necessary to form new collection before deleting");

}

Input.Cont();

break;

}

} while (menuNumber != 7);

break;

}

} while (collectionNumber != 4);

}

}

}

Класс Vehicle:

using System.Diagnostics.CodeAnalysis;

using System.Numerics;

namespace ClassLibraryHSE1course

{

public class Vehicle : IComparable,IInit

{

// Fields

protected double mass = 0; // Kg

protected double maxSpeed = 0; // Km per hour

protected int power = 0; // In horsepowers

protected int passengerCapacity = 0; // People

static protected Random random = new Random();

// Properties

public double Mass

{

get

{

return this.mass;

}

set

{

if (value >= 0)

{

this.mass = value;

}

else

{

this.mass = 0;

}

}

}

public double MaxSpeed

{

get

{

return this.maxSpeed;

}

set

{

if (value >= 0)

{

this.maxSpeed = value;

}

else

{

this.maxSpeed = 0;

}

}

}

public int Power

{

get

{

return this.power;

}

set

{

if (value >= 0)

{

this.power = value;

}

else

{

this.power = 0;

}

}

}

public int PassengerCapacity

{

get

{

return this.passengerCapacity;

}

set

{

if (value >= 0)

{

this.passengerCapacity = value;

}

else

{

this.passengerCapacity = 0;

}

}

}

// Constructors

[ExcludeFromCodeCoverage]

public Vehicle(double mass = 0, double speed = 0, int power = 0, int passengerCapacity= 0)

{

Mass = mass;

MaxSpeed = speed;

Power= power;

PassengerCapacity = passengerCapacity;

}

// Methods

[ExcludeFromCodeCoverage] // abstract class

public override bool Equals(object? obj)

{

if (obj != null)

{

if (obj is Vehicle temp)

{

return this.Mass == temp.Mass && this.MaxSpeed == temp.MaxSpeed && this.Power == temp.Power && this.PassengerCapacity == temp.PassengerCapacity;

}

}

return false;

}

// Method (virtual)

[ExcludeFromCodeCoverage]

public virtual void Show()

{

Console.Write($"Mass = {Mass}, maximum speed = {MaxSpeed}, power = {Power}, passenger capacity = {PassengerCapacity}");

}

[ExcludeFromCodeCoverage]

public virtual void Init()

{

Console.Write("Enter vehicle mass: ");

bool fl;

double numd;

do

{

fl = double.TryParse(Console.ReadLine(), out numd);

if (numd < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle mass again: ");

} while (!fl);

Mass = numd;

Console.Write("Enter vehicle maximum speed: ");

do

{

fl = double.TryParse(Console.ReadLine(), out numd);

if (numd < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle maximum speed again: ");

} while (!fl);

MaxSpeed = numd;

int numi;

Console.Write("Enter vehicle power: ");

do

{

fl = int.TryParse(Console.ReadLine(), out numi);

if (numi < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle power again: ");

} while (!fl);

Power = numi;

Console.Write("Enter vehicle passenger capacity: ");

do

{

fl = int.TryParse(Console.ReadLine(), out numi);

if (numi < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle passenger capacity again: ");

} while (!fl);

PassengerCapacity = numi;

}

public virtual void RandomInit(int leftBound, int rightBound)

{

if (leftBound < 0)

{

leftBound= 0;

}

if (rightBound < 0)

{

rightBound= 0;

}

if (rightBound < leftBound)

{

rightBound= leftBound;

}

Mass = random.Next(leftBound, rightBound+1);

MaxSpeed = random.Next(leftBound, rightBound+1);

Power = random.Next(leftBound, rightBound+1);

PassengerCapacity = random.Next(leftBound, rightBound+1);

}

public int CompareTo(object? obj)

{

if (obj is double dbl) return (int)(this.Mass - dbl);

else if (obj is int ibl) return (int)(this.Mass - ibl);

else if (obj is Vehicle vehicle)

{

if (this.Mass != vehicle.Mass) return (int)(this.Mass - vehicle.Mass);

if (this.MaxSpeed != vehicle.MaxSpeed) return (int)(this.MaxSpeed - vehicle.MaxSpeed);

if (this.Power != vehicle.Power) return this.Power - vehicle.Power;

return this.PassengerCapacity - vehicle.PassengerCapacity;

}

else throw new ArgumentException("Incorrect parameter value");

}

public virtual Vehicle ShallowCopy()

{

return (Vehicle)this.MemberwiseClone();

}

public virtual object Clone()

{

return new Vehicle(mass, MaxSpeed, Power, PassengerCapacity);

}

public override int GetHashCode()

{

return Mass.GetHashCode()+MaxSpeed.GetHashCode()+Power.GetHashCode()+PassengerCapacity.GetHashCode();

}

[ExcludeFromCodeCoverage]

public override string ToString()

{

return $"Vehicle: Mass = {Mass}, maximum speed = {MaxSpeed}, power = {Power}, passenger capacity = {PassengerCapacity}";

}

}

}

Класс Car:

using System.Diagnostics.CodeAnalysis;

namespace ClassLibraryHSE1course

{

public class Car : Vehicle, ICloneable

{

// Fields

protected double fuelCapacity = 0; // Litres

protected double fuelConsumption = 0; // Per km

protected double мaneuverability = 0; // Turning speed

protected double mileage = 0; // Km

// Properties

public double FuelCapacity

{

get

{

return this.fuelCapacity;

}

set

{

if (value >= 0)

{

this.fuelCapacity = value;

}

else

{

this.fuelCapacity = 0;

}

}

}

public double FuelConsumption

{

get

{

return this.fuelConsumption;

}

set

{

if (value >= 0)

{

this.fuelConsumption = value;

}

else

{

this.fuelConsumption = 0;

}

}

}

public double Maneuverability

{

get

{

return this.мaneuverability;

}

set

{

if (value >= 0)

{

this.мaneuverability = value;

}

else

{

this.мaneuverability = 0;

}

}

}

public double Mileage

{

get

{

return this.mileage;

}

set

{

if (value >= 0)

{

this.mileage = value;

}

else

{

this.mileage = 0;

}

}

}

[ExcludeFromCodeCoverage]

public Vehicle BaseVehicle

{

get

{

return new Vehicle(Mass, MaxSpeed, Power, PassengerCapacity);

}

}

// Constructors

[ExcludeFromCodeCoverage]

public Car(double mass = 0, double maxSpeed = 0, int power = 0, int passengerCapacity = 0, double fuelCapacity= 0, double fuelConsumption = 0, double мaneuverability = 0, double mileage = 0) : base(mass, maxSpeed, power, passengerCapacity)

{

FuelCapacity = fuelCapacity;

FuelConsumption = fuelConsumption;

Maneuverability = мaneuverability;

Mileage = mileage;

}

// Methods

public override bool Equals(object? obj)

{

if (obj != null)

{

if (obj is Car temp)

{

return this.Mass == temp.Mass && this.MaxSpeed == temp.MaxSpeed && this.Power == temp.Power && this.PassengerCapacity == temp.PassengerCapacity && this.FuelCapacity == temp.FuelCapacity && this.FuelConsumption == temp.FuelConsumption && this.Maneuverability == temp.Maneuverability && this.Mileage == temp.Mileage;

}

}

return false;

}

// Methods (virtual)

[ExcludeFromCodeCoverage]

public override void Show()

{

base.Show();

Console.WriteLine($", fuel capacity = {FuelCapacity}, fuel consumption = {FuelConsumption}, maneuverability = {Maneuverability}, mileage = {Mileage}");

}

[ExcludeFromCodeCoverage]

public override void Init()

{

base.Init();

Console.Write("Enter vehicle fuel capacity: ");

bool fl;

double num;

do

{

fl = double.TryParse(Console.ReadLine(), out num);

if (num < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle fuel capacity again: ");

} while (!fl);

FuelCapacity = num;

Console.Write("Enter vehicle maximum fuel consumption: ");

do

{

fl = double.TryParse(Console.ReadLine(), out num);

if (num < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle fuel consumption again: ");

} while (!fl);

FuelConsumption = num;

Console.Write("Enter vehicle maneuverability: ");

do

{

fl = double.TryParse(Console.ReadLine(), out num);

if (num < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle maneuverability again: ");

} while (!fl);

Maneuverability = num;

Console.Write("Enter vehicle passenger mileage: ");

do

{

fl = double.TryParse(Console.ReadLine(), out num);

if (num < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle mileage again: ");

} while (!fl);

Mileage = num;

}

public override void RandomInit(int leftBound, int rightBound)

{

base.RandomInit(leftBound, rightBound);

if (leftBound < 0)

{

leftBound= 0;

}

if (rightBound < 0)

{

rightBound= 0;

}

if (rightBound < leftBound)

{

rightBound= leftBound;

}

FuelCapacity = random.Next(leftBound, rightBound+1);

FuelConsumption = random.Next(leftBound, rightBound+1);

Maneuverability = random.Next(leftBound, rightBound+1);

Mileage = random.Next(leftBound, rightBound+1);

}

public override Car ShallowCopy()

{

return (Car) this.MemberwiseClone();

}

public override object Clone()

{

return new Car(this.Mass, this.MaxSpeed, this.Power, this.PassengerCapacity, this.fuelCapacity, this.FuelConsumption, this.Maneuverability, this.mileage);

}

public override int GetHashCode()

{

return base.GetHashCode()+fuelCapacity.GetHashCode()+FuelConsumption.GetHashCode()+Maneuverability.GetHashCode()+mileage.GetHashCode();

}

[ExcludeFromCodeCoverage]

public override string ToString()

{

return $"Car: Mass = {Mass}, maximum speed = {MaxSpeed}, power = {Power}, passenger capacity = {PassengerCapacity}, fuel capacity = {FuelCapacity}, fuel consumption = {FuelConsumption}, maneuverability = {Maneuverability}, mileage = {Mileage}";

}

}

}

Класс Train:

using System.Diagnostics.CodeAnalysis;

namespace ClassLibraryHSE1course

{

public class Train:Vehicle,ICloneable

{

// Fields

protected int[] wagons = new int[0]; // Number of passengers in each wagon

// Properties

[ExcludeFromCodeCoverage]

public int[] Wagons { get; set; }

// Constructors

[ExcludeFromCodeCoverage]

public Train() : base()

{

Wagons = new int[0];

}

[ExcludeFromCodeCoverage]

public Train(double mass) : base(mass)

{

Wagons = new int[0];

}

[ExcludeFromCodeCoverage]

public Train(double mass, double maxSpeed) : base(mass, maxSpeed)

{

Wagons = new int[0];

}

[ExcludeFromCodeCoverage]

public Train(double mass, double maxSpeed, int power) : base(mass, maxSpeed, power)

{

Wagons = new int[0];

}

[ExcludeFromCodeCoverage]

public Train(double mass, double maxSpeed, int power, int passengerCapacity) : base(mass, maxSpeed, power, passengerCapacity)

{

Wagons = new int[0];

}

[ExcludeFromCodeCoverage]

public Train(double mass, double maxSpeed, int power, int passengerCapacity, int[] wagons) : base(mass, maxSpeed, power, passengerCapacity)

{

Wagons = wagons;

}

// Methods

public override bool Equals(object? obj)

{

if (obj != null)

{

if (obj is Train temp)

{

if (this.Mass == temp.Mass && this.MaxSpeed == temp.MaxSpeed && this.Power == temp.Power && this.PassengerCapacity == temp.PassengerCapacity && this.Wagons.Length == temp.Wagons.Length)

{

if (this.Wagons.Length == 0)

{

return true;

}

else

{

bool flag = true;

for (int i = 0; i < this.Wagons.Length; i++)

{

if (this.Wagons[i] != temp.Wagons[i])

{

flag = false;

}

}

return flag;

}

}

}

}

return false;

}

//Methods (virtual)

[ExcludeFromCodeCoverage]

public override void Show()

{

base.Show();

Console.WriteLine($", number of wagons = {Wagons.Length}");

if (Wagons.Length > 0)

{

Console.Write("Wagons passengers: ");

foreach (int i in Wagons)

{

Console.Write($"{i} ");

}

Console.WriteLine();

}

else

{

Console.WriteLine("No Wagons");

}

}

[ExcludeFromCodeCoverage]

public override void Init()

{

base.Init();

int numi;

Console.Write("Enter vehicle number of wagons: ");

bool fl;

do

{

fl = int.TryParse(Console.ReadLine(), out numi);

if (numi < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter vehicle number of wagons again: ");

} while (!fl);

Wagons = new int[numi];

Console.WriteLine("Enter number of passengers in each wagon (integer number): ");

for (int i = 0; i < Wagons.Length; i++)

{

int num;

Console.Write($"{i+1}) ");

do

{

fl = int.TryParse(Console.ReadLine(), out num);

if (numi < 0) fl = false;

if (!fl)

{

Console.WriteLine("Error! repeat the wagon enter");

Console.Write($"{i+1}) ");

}

} while (!fl);

Wagons[i] = num;

}

}

public virtual void RandomInit(int leftBound, int rightBound)

{

base.RandomInit(leftBound, rightBound);

if (leftBound < 0)

{

leftBound= 0;

}

if (rightBound < 0)

{

rightBound= 0;

}

if (rightBound < leftBound)

{

rightBound= leftBound;

}

Wagons = new int[random.Next(0, 10)];

for (int i = 0; i < Wagons.Length; i++)

{

Wagons[i] = random.Next(leftBound, rightBound+1);

}

}

public override Train ShallowCopy()

{

return (Train)this.MemberwiseClone();

}

public override object Clone()

{

return new Train(this.Mass, this.MaxSpeed, this.Power, this.PassengerCapacity, this.Wagons);

}

public override int GetHashCode()

{

return base.GetHashCode()+Wagons.Length.GetHashCode();

}

}

}

Класс Express:

using System.Diagnostics.CodeAnalysis;

namespace ClassLibraryHSE1course

{

public class Express:Train

{

// Fields

protected string name = "Nameless";

// Properties

[ExcludeFromCodeCoverage]

public string Name { get; set; }

// Constructors

[ExcludeFromCodeCoverage]

public Express() : base()

{

Name = "Nameless";

}

[ExcludeFromCodeCoverage]

public Express(double mass) : base(mass)

{

Name = "Nameless";

}

[ExcludeFromCodeCoverage]

public Express(double mass, double maxSpeed) : base(mass, maxSpeed)

{

Name = "Nameless";

}

[ExcludeFromCodeCoverage]

public Express(double mass, double maxSpeed, int power) : base(mass, maxSpeed, power)

{

Name = "Nameless";

}

[ExcludeFromCodeCoverage]

public Express(double mass, double maxSpeed, int power, int passengerCapacity) : base(mass, maxSpeed, power, passengerCapacity)

{

Name = "Nameless";

}

[ExcludeFromCodeCoverage]

public Express(double mass, double maxSpeed, int power, int passengerCapacity, int[] wagons) : base(mass, maxSpeed, power, passengerCapacity, wagons)

{

Name = "Nameless";

}

[ExcludeFromCodeCoverage]

public Express(double mass, double maxSpeed, int power, int passengerCapacity, int[] wagons, string name) : base(mass, maxSpeed, power, passengerCapacity, wagons)

{

Name = name;

}

// Methods

public override bool Equals(object? obj)

{

if (obj != null)

{

if (obj is Express temp)

{

if (this.Mass == temp.Mass && this.MaxSpeed == temp.MaxSpeed && this.Power == temp.Power && this.PassengerCapacity == temp.PassengerCapacity && this.Wagons.Length == temp.Wagons.Length && this.Name == temp.Name)

{

if (this.Wagons.Length == 0)

{

return true;

}

else

{

bool flag = true;

for (int i = 0; i < this.Wagons.Length; i++)

{

if (this.Wagons[i] != temp.Wagons[i])

{

flag = false;

}

}

return flag;

}

}

}

}

return false;

}

// Methods (virtual)

[ExcludeFromCodeCoverage]

public override void Show()

{

Console.Write($"{Name}: ");

base.Show();

}

[ExcludeFromCodeCoverage]

public override void Init()

{

base.Init();

Console.Write("Enter vehicle name: ");

Name = Console.ReadLine();

if (Name == "")

{

Name = "Nameless";

}

}

public override void RandomInit(int leftBound, int rightBound)

{

base.RandomInit(leftBound, rightBound);

if (leftBound < 0)

{

leftBound= 0;

}

if (rightBound < 0)

{

rightBound= 0;

}

if (rightBound < leftBound)

{

rightBound= leftBound;

}

string str = "";

for (int i = 0; i < 10; i++)

{

str = str + Convert.ToChar(random.Next(70, 90));

}

Name = str;

}

public override Express ShallowCopy()

{

return (Express)this.MemberwiseClone();

}

public override object Clone()

{

return new Express(this.Mass, this.MaxSpeed, this.Power, this.PassengerCapacity, this.Wagons, this.Name);

}

public override int GetHashCode()

{

return base.GetHashCode()+Name.GetHashCode();

}

}

}

Класс Human:

using System.Diagnostics.CodeAnalysis;

namespace ClassLibraryHSE1course

{

public class Human:IInit, ICloneable

{

static Random random = new Random();

[ExcludeFromCodeCoverage]

public double Weight { get; set; }

[ExcludeFromCodeCoverage]

public double Height { get; set; }

[ExcludeFromCodeCoverage]

public Car PersonalCar { get; set; }

[ExcludeFromCodeCoverage]

public Human()

{

Weight = 0;

Height = 0;

PersonalCar= new Car();

}

[ExcludeFromCodeCoverage]

public Human(double w, double h)

{

Weight = w;

Height = h;

PersonalCar = new Car();

}

[ExcludeFromCodeCoverage]

public Human(double w, double h, Car personalCar)

{

Weight = w;

Height = h;

PersonalCar = personalCar;

}

[ExcludeFromCodeCoverage]

public void Show()

{

Console.WriteLine($"Weight = {Weight}, height = {Height}, Human perosnal car:");

PersonalCar.Show();

Console.WriteLine();

}

[ExcludeFromCodeCoverage]

public void Init()

{

Console.Write("Enter human weight: ");

bool fl;

double num;

do

{

fl = double.TryParse(Console.ReadLine(), out num);

if (num < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter human weight: ");

} while (!fl);

Weight = num;

Console.Write("Enter human height: ");

do

{

fl = double.TryParse(Console.ReadLine(), out num);

if (num < 0) fl = false;

if (!fl) Console.Write($"Input error! Enter human weight: ");

} while (!fl);

Height = num;

PersonalCar.Init();

}

public void RandomInit(int leftBound, int rightBound)

{

if (leftBound < 0)

{

leftBound= 0;

}

if (rightBound < 0)

{

rightBound= 0;

}

if (rightBound < leftBound)

{

rightBound= leftBound;

}

Weight = random.Next(leftBound, rightBound+1);

Height = random.Next(leftBound, rightBound+1);

PersonalCar.RandomInit(leftBound,rightBound);

}

public Human ShallowCopy()

{

return (Human)this.MemberwiseClone();

}

public object Clone()

{

return new Human(this.Weight, this.Height, (Car) this.PersonalCar.Clone());

}

}

}

Класс VehicleComparer:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ClassLibraryHSE1course

{

public class VehiclesComparer: IComparer<Vehicle>

{

public int Compare(Vehicle? x, Vehicle? y)

{

if (x == null || y == null) throw new ArgumentException("Incorrect parameter value");

if (x.PassengerCapacity != y.PassengerCapacity) return x.PassengerCapacity - y.PassengerCapacity;

if (x.Power != y.Power) return x.Power - y.Power;

if (x.MaxSpeed != y.MaxSpeed) return (int)(x.MaxSpeed - y.MaxSpeed);

return (int)(x.Mass - y.Mass);

}

}

}

Интерфейс IInit:

namespace ClassLibraryHSE1course

{

public interface IInit

{

void Show();

void Init();

void RandomInit(int x, int y);

}

}

Класс Output:

using System.Diagnostics.CodeAnalysis;

namespace lab

{

[ExcludeFromCodeCoverage]

public static class Output

{

public delegate void PrinterLine(string message);

private static PrinterLine printerLine;

public delegate void Printer(string message);

private static Printer printer;

public static void AddPrinterLine(PrinterLine prnt)

{

printerLine += prnt;

}

public static void ReducePrinterLine (PrinterLine prnt)

{

printerLine -= prnt;

}

public static void AddPrinter(Printer prnt)

{

printer += prnt;

}

public static void ReducePrinter(Printer prnt)

{

printer -= prnt;

}

/// <summary>

/// print line with line break

/// </summary>

/// <param name="str"></param>

public static void PrintLine(string str)

{

printerLine.Invoke(str);

}

/// <summary>

/// Print Line without line break

/// </summary>

/// <param name="str"></param>

public static void Print(string str)

{

printer.Invoke(str);

}

/// <summary>

/// Write the menu of array choosing in the consol

/// </summary>

public static void Menu()

{

PrintLine("Choose data type using arrows. Press enter to confirm your choice");

PrintLine("1 - Hashtable");

PrintLine("2 - Oueue<T>");

PrintLine("3 - List <T> and Dictionary <K,T> (exersice 3)");

PrintLine("0 - Shut down");

}

/// <summary>

/// Write the menu of choosing filling type

/// </summary>

/// <param name="tp">type of menu</param>

public static void FillTypes(int type = 1)

{

if (type == 1)

{

PrintLine("Choose how to fill the Collection using arrows. Press enter to confirm your choice");

PrintLine("1 - Fill with random vehicles");

PrintLine("2 - Enter vehicles from the keyboard");

}

else if (type == 2)

{

PrintLine("Choose how to fill the Collection using arrows. Press enter to confirm your choice");

PrintLine("1 - Add random vehicles");

PrintLine("2 - Add vehicle from the keyboard");

}

}

/// <summary>

/// Write the menu of command choosing in the consol

/// </summary>

/// <param name="arrType">type of datas</param>

public static void DataMenu(int type = 1)

{

if (type == 1)

{

PrintLine("Choose command using arrows. Press enter to confirm your choice");

PrintLine("1 - Form new collection");

PrintLine("2 - Print collection");

PrintLine("3 - Add collection element");

PrintLine("4 - Delete collection element");

PrintLine("5 - Select a query");

PrintLine("6 - Сopy collection (demonstration)");

PrintLine("7 - Sort");

PrintLine("8 - Search");

PrintLine("0 - Back");

}

else if (type == 2)

{

PrintLine("Choose command using arrows. Press enter to confirm your choice");

PrintLine("1 - Form new collections");

PrintLine("2 - Print collections");

PrintLine("3 - Add collections element");

PrintLine("4 - Delete collections element");

PrintLine("5 - Search (demonstration)");

PrintLine("0 - Back");

}

}

public static void PrintClasses(int i)

{

if (i > 0)

{

PrintLine($"Choose type of vehicle {i}:");

PrintLine("1 - Car");

PrintLine("2 - Train");

PrintLine("3 - Express");

}

else if (i == -1)

{

PrintLine($"Choose type of vehicle that need to be found:");

PrintLine("1 - Car");

PrintLine("2 - Train");

PrintLine("3 - Express");

}

else

{

PrintLine($"Choose type of vehicle:");

PrintLine("1 - Car");

PrintLine("2 - Train");

PrintLine("3 - Express");

}

}

public static void PrintRequests(int type = 1)

{

if (type == 1) // Hashtable

{

PrintLine("Choose request");

PrintLine("1 - Count the number of expresses");

PrintLine("2 - Count the number of passengers in all trains");

PrintLine("3 - Print all cars");

}

}

public static void PrintSortType()

{

PrintLine("Choose sorting type");

PrintLine("1 - Ascending sort");

PrintLine("2 - Descending sort");

}

public static void PrintSearchType()

{

PrintLine("Choose searching type");

PrintLine("1 - find by hashcode");

PrintLine("2 - find by value");

}

public static void PrintCollectionsType()

{

PrintLine("Choose collection type");

PrintLine("1 - List<Car>");

PrintLine("2 - List<string>");

PrintLine("3 - Dictionary<Vehicle,Car>");

PrintLine("4 - Dictionary<string,Car>");

}

public static void PrintElementType()

{

PrintLine("Choose elemnt type");

PrintLine("1 - first");

PrintLine("2 - middle");

PrintLine("3 - last");

PrintLine("4 - non-existent");

}

/// <summary>

/// Print that the data is not initialized

/// </summary>

/// <param name="type">type of data</param>

public static void NotInit(int type=0)

{

switch (type)

{

case 0:

PrintLine("Necessary to form new array before working with him");

break;

case 1:

PrintLine("Necessary to enter text before working with him");

break;

}

Input.Cont();

}

}

}

Класс Input:

using System.Diagnostics.CodeAnalysis;

namespace lab

{

[ExcludeFromCodeCoverage]

public static class Input

{

/// <summary>

/// Request to continue the program

/// </summary>

public static void Cont()

{

Output.PrintLine("press Enter to continue...");

Console.TreatControlCAsInput = true;

ConsoleKeyInfo key;

do

{

key = Console.ReadKey(true);

}

while (key.Key != ConsoleKey.Enter);

Console.TreatControlCAsInput = false;

Console.Clear();

}

/// <summary>

/// Enter an information by keyboard

/// </summary>

/// <param name="min">how much rows before firstcommand</param>

/// <param name="max"> Number of the last row</param>

/// <returns> Number choosen by a user</returns>

public static int ReadSwitch(int min, int max)

{

Console.TreatControlCAsInput = true;

ConsoleKeyInfo key;

int com = min;

int type = -1;

do

{

Console.SetCursorPosition(0, com);

key = Console.ReadKey(true);

switch (key.Key)

{

case ConsoleKey.UpArrow:

if (com != min) com--;

break;

case ConsoleKey.W:

if (com != min) com--;

break;

case ConsoleKey.DownArrow:

if (com != max) com++;

break;

case ConsoleKey.S:

if (com != max) com++;

break;

case ConsoleKey.Enter:

type = com;

break;

}

} while (type == -1);

Console.TreatControlCAsInput = false;

Console.Clear();

return type;

}

/// <summary>

/// Number enter by the user

/// incorrect enter causes an error message and repeated enter

/// </summary>

/// <param name="tx">input message</param>

/// <param name="type">filtr type

/// 0 - any numbers

/// 1 - (>0)

/// 2 - (>=0)</param>

/// <returns>Bumber enterd by the user</returns>

public static int ReadInt(string tx, int type = 0)

{

Output.Print($"{tx}(");

if (type == 1) Output.Print("positive ");

if (type == 2) Output.Print("non-negative ");

Output.Print("integer number): ");

bool fl;

int num;

do

{

fl = int.TryParse(Console.ReadLine(), out num);

if (type == 1 && num <= 0 || type == 2 && num < 0) fl = false;

if (!fl) Output.Print($"Input error! {tx} again: ");

} while (!fl);

return num;

}

/// <summary>

/// Random Number's range enter by the user

/// incorrect enter causes an error message and repeated enter

/// </summary>

/// <param name="min">minimal random number</param>

/// <param name="max">maximal random number</param>

public static void RndNumbRange(out int min, out int max, int typeMes = 1)

{

bool fl;

max = 0;

do

{

fl = true;

if (typeMes == 2)

{

min = ReadInt("Enter the minimum length of array(>=0)");

if (min < 0)

{

Output.PrintLine("Error! the length cannot be negative number. Repeat enter");

fl = false;

}

else

{

max = ReadInt("Enter the maximum length of array");

}

}

else

{

min = ReadInt("Enter the minimum value of array class elements");

max = ReadInt("Enter the maximum value of array class elements");

}

if (max<min)

{

if (typeMes == 2)

{

Output.PrintLine("Error! the maximum length cannot be less than the minimum. Repeat enter");

}

else

{

Output.PrintLine("Error! the maximum value cannot be less than the minimum. Repeat enter");

}

fl = false;

}

} while (!fl);

Console.Clear();

}

/// <summary>

/// read number and filter it from minimum to maximum value

/// </summary>

/// <param name="min">minimal filtr value</param>

/// <param name="max">maximum filtr value</param>

/// <param name="message">output message before input</param>

/// <returns>filtered number entered by user</returns>

public static int ReadFromTo(int min, int max, string message = "number")

{

bool fl;

int k;

Output.Print($"Enter a {message}(integer number from {min} to {max}): ");

do

{

fl = int.TryParse(Console.ReadLine(), out k);

if (k<min || k>max) fl = false;

if (!fl) Output.Print($"Input error! Enter a {message} again: ");

} while (!fl);

return k;

}

/// <summary>

/// warning message if len > lenMax

/// user need to confirm the action

/// </summary>

/// <param name="len">length of object</param>

/// <param name="lenMax">maximal length</param>

/// <returns>true if user don't confirm the action

/// false if user confirmed the action

/// </returns>

public static bool Warning(int len, int lenMax)

{

if (len > lenMax)

{

Console.Clear();

Output.PrintLine("This action may cause the program to slow down, terminate prematurely or it can be just unpleasant for you.");

Output.PrintLine("Are you sure you want to interact with large collection?");

Output.PrintLine("1 - Yes");

Output.PrintLine("2 - No");

int temp = ReadSwitch(2, 3);

if (temp == 2)

{

return false;

}

else

{

return true;

}

}

return false;

}

}

}

Класс Output:

using System.Diagnostics;

using System.Diagnostics.CodeAnalysis;

using ClassLibraryHSE1course;

namespace lab

{

[ExcludeFromCodeCoverage]

public static class TestCollection

{

private static Stopwatch sw = new Stopwatch();

public static List<Car> col1 = new List<Car>();

public static List<string> col2 = new List<String>();

public static Dictionary<Vehicle, Car> col3 = new Dictionary<Vehicle, Car>();

public static Dictionary<string, Car> col4 = new Dictionary<string, Car>();

public static Car first;

public static Car middle;

public static Car last;

static public void InitCollections(int length)

{

List<Car> col1 = new List<Car>();

List<string> col2 = new List<String>();

Dictionary<Vehicle, Car> col3 = new Dictionary<Vehicle, Car>();

Dictionary<string, Car> col4 = new Dictionary<string, Car>();

for (int i = 0; i < length; i++)

{

try

{

Car car = new Car();

car.RandomInit(1,10000);

Vehicle vehicle = car.BaseVehicle;

col1.Add(car);

col2.Add(car.ToString());

col3.Add(vehicle, car);

col4.Add(vehicle.ToString(), car);

if (i == 0) first = (Car)car.Clone();

if (i == length / 2) middle = (Car)car.Clone();

if (i == length - 1) last = (Car)car.Clone();

}

catch (Exception)

{

i--;

}

}

}

[ExcludeFromCodeCoverage]

public static void PrintListOfCar()

{

Output.PrintLine("List<Car>:");

int i = 1;

foreach (Car car in col1)

{

Output.Print($"{i}) ");

Output.PrintLine(car.ToString());

i++;

}

}

[ExcludeFromCodeCoverage]

public static void PrintListOfString()

{

Output.PrintLine("List<String>:");

int i = 1;

foreach (string str in col2)

{

Output.Print($"{i}) ");

Output.PrintLine(str);

i++;

}

}

[ExcludeFromCodeCoverage]

public static void PrintDictionaryOfCar()

{

Output.PrintLine("Dictionary<Vehicle, Car>:");

int i = 1;

foreach (Vehicle vehicle in col3.Keys)

{

Output.PrintLine($"{i}) ");

Output.PrintLine($"Key) {vehicle.ToString()}");

Output.PrintLine($"Value) {col3[vehicle].ToString()}");

i++;

}

}

[ExcludeFromCodeCoverage]

public static void PrintDictionaryOfString()

{

Output.PrintLine("Dictionary<Vehicle, Car>:");

int i = 1;

foreach (string str in col4.Keys)

{

Output.PrintLine($"{i}) ");

Output.PrintLine($"Key) {str}");

Output.PrintLine($"Value) {col4[str].ToString()}");

i++;

}

}

[ExcludeFromCodeCoverage]

public static void FindElementsTime(Car element, string message)

{

// List<Car>

Output.PrintLine($"{message} element: ");

sw.Restart();

bool isFinded = col1.Contains(element);

sw.Stop();

if (isFinded)

{

Output.PrintLine($"In collection List<Car> element was found in {sw.ElapsedTicks} ticks");

}

else

{

Output.PrintLine($"No element in collections List<Car> in {sw.ElapsedTicks} ticks");

}

// List<String>

sw.Restart();

isFinded = col2.Contains(element.ToString());

sw.Stop();

if (isFinded)

{

Output.PrintLine($"In collection List<String> element was found in {sw.ElapsedTicks} ticks");

}

else

{

Output.PrintLine($"No element in collections List<String> in {sw.ElapsedTicks} ticks");

}

// Dictionary<Vehicle, Car> Key

sw.Restart();

isFinded = col3.ContainsKey(element.BaseVehicle);

sw.Stop();

if (isFinded)

{

Output.PrintLine($"In collection Dictionary<Vehicle, Car> element key was found in {sw.ElapsedTicks} ticks");

}

else

{

Output.PrintLine($"No element key in collections Dictionary<Vehicle, Car> in {sw.ElapsedTicks} ticks");

}

// Dictionary<Vehicle, Car> Value

sw.Restart();

isFinded = col3.ContainsValue(element);

sw.Stop();

if (isFinded)

{

Output.PrintLine($"In collection Dictionary<Vehicle, Car> element value was found in {sw.ElapsedTicks} ticks");

}

else

{

Output.PrintLine($"No element value in collections Dictionary<Vehicle, Car> in {sw.ElapsedTicks} ticks");

}

// Dictionary<String, Car> Key

sw.Restart();

isFinded = col4.ContainsKey(element.BaseVehicle.ToString());

sw.Stop();

if (isFinded)

{

Output.PrintLine($"In collection Dictionary<String, Car> element key was found in {sw.ElapsedTicks} ticks");

}

else

{

Output.PrintLine($"No element key in collections Dictionary<String, Car> in {sw.ElapsedTicks} ticks");

}

// Dictionary<String, Car> Value

sw.Restart();

isFinded = col4.ContainsValue(element);

sw.Stop();

if (isFinded)

{

Output.PrintLine($"In collection Dictionary<String, Car> element value was found in {sw.ElapsedTicks} ticks");

}

else

{

Output.PrintLine($"No element value in collections Dictionary<String, Car> in {sw.ElapsedTicks} ticks");

}

}

public static void AddElement(Car car)

{

col1.Add(car);

col2.Add(car.ToString());

col3.Add(car.BaseVehicle, car);

col4.Add(car.BaseVehicle.ToString(), car);

}

public static void DeleteElement(Car car)

{

col1.Remove(car);

col2.Remove(car.ToString());

col3.Remove(car.BaseVehicle);

col4.Remove(car.BaseVehicle.ToString());

}

}

}

Класс DoublyLinkedListNode:

using System.Diagnostics.CodeAnalysis;

using lab;

namespace ClassLibraryHSE1course

{

public class DoublyLinkedNode<T>: ICloneable where T:ICloneable

{

public T Data { get; set; }

public DoublyLinkedNode<T> ?Next { get; set; }

public DoublyLinkedNode<T> ?Past { get; set; }

[ExcludeFromCodeCoverage]

public DoublyLinkedNode()

{

Data = default(T);

Next = null;

Past = null;

}

[ExcludeFromCodeCoverage]

public DoublyLinkedNode(T data)

{

Data = data;

Next = null;

Past = null;

}

[ExcludeFromCodeCoverage]

public DoublyLinkedNode(DoublyLinkedNode<T> node)

{

Data = (T)node.Data.Clone();

Next = null;

Past = null;

}

public override string ToString()

{

return Data.ToString();

}

public object Clone()

{

return new DoublyLinkedNode<T>((T)Data.Clone());

}

}

}

Класс DoublyLinkedList:

using System.Collections;

using System.Diagnostics.CodeAnalysis;

using System.Reflection;

using lab;

using static lab.Output;

namespace ClassLibraryHSE1course

{

public class DoublyLinkedList<T>: ICloneable where T:ICloneable

{

private DoublyLinkedNode<T>? firstNode;

private DoublyLinkedNode<T>? lastNode;

public int Length { get; private set; }

public delegate T RandomT(int min, int max);

public static RandomT? randomT;

public delegate T KeyboardT(int i);

public static KeyboardT? keyboardT;

[ExcludeFromCodeCoverage]

public DoublyLinkedList()

{

firstNode = null;

lastNode = null;

}

[ExcludeFromCodeCoverage]

public DoublyLinkedList(int size) // keyboard

{

if (size == 0)

{

firstNode = null;

lastNode = null;

return;

}

if (size == 1)

{

firstNode = new DoublyLinkedNode<T>();

// Randomly choose one of 3 vehicles

firstNode.Data = keyboardT.Invoke(0);

lastNode = firstNode;

Length = size;

return;

}

// size > 1

firstNode = new DoublyLinkedNode<T>(keyboardT.Invoke(1));

DoublyLinkedNode<T>? pastNode = firstNode;

DoublyLinkedNode<T>? nextNode = null;

for (int i = 1; i < size; i++)

{

nextNode = new DoublyLinkedNode<T>();

// Randomly init one of 3 vehicles

nextNode.Data = keyboardT.Invoke(i+1);

pastNode.Next = nextNode;

nextNode.Past = pastNode;

pastNode = nextNode;

}

lastNode = nextNode;

Length = size;

}

[ExcludeFromCodeCoverage]

// Random vehicle constructor

public DoublyLinkedList(int size, int min, int max)

{

if (size == 0)

{

firstNode = null;

lastNode = null;

return;

}

if (size == 1)

{

firstNode = new DoublyLinkedNode<T>();

// Randomly choose one of 3 vehicles

firstNode.Data = randomT.Invoke(min, max);

lastNode = firstNode;

Length = size;

return;

}

// size > 1

firstNode = new DoublyLinkedNode<T>(randomT.Invoke(min, max));

DoublyLinkedNode<T>? pastNode = firstNode;

DoublyLinkedNode<T>? nextNode = null;

for (int i = 1; i < size; i++)

{

nextNode = new DoublyLinkedNode<T>();

// Randomly init one of 3 vehicles

nextNode.Data = randomT.Invoke(min, max);

pastNode.Next = nextNode;

nextNode.Past = pastNode;

pastNode = nextNode;

}

lastNode = nextNode;

Length = size;

}

public DoublyLinkedList(DoublyLinkedList<T> list) // copy

{

DoublyLinkedList<T> lt = (DoublyLinkedList<T>)list.Clone();

firstNode = lt.firstNode;

lastNode = lt.lastNode;

Length = lt.Length;

}

[ExcludeFromCodeCoverage]

public void Show(bool isFromStart = true)

{

if (Length == 0)

{

Output.PrintLine("Empty list");

}

else

{

Output.PrintLine("Doubly linked list: ");

DoublyLinkedNode<T>? currentNode;

if (isFromStart) // start from the first

{

currentNode = firstNode;

}

else // start from the last

{

currentNode = lastNode;

}

for (int i = 0; i < Length; i++)

{

Output.Print($"{i+1}) ");

if (currentNode == null)

{

Output.PrintLine("Empty Node");

}

else

{

MethodInfo methodInfo = currentNode.Data.GetType().GetMethod("Show");

if (methodInfo != null)

{

methodInfo.Invoke(currentNode.Data, new object[] { 0 });

}

else

{

throw new Exception("No showing method in data type");

}

}

if (isFromStart)

{

if (currentNode?.Next != null)

{

currentNode = currentNode.Next;

}

}

else

{

if (currentNode?.Past != null)

{

currentNode = currentNode.Past;

}

}

Output.PrintLine("");

}

}

}

// Add the item with the given number to the list

// num = 0 - in first element

// num >= length - in last element

public void AddByNumber(int num, T value)

{

if (Length == 0) // empty

{

firstNode = new DoublyLinkedNode<T>(value);

lastNode = firstNode;

}

else if (num >= Length) // last

{

DoublyLinkedNode<T> addedNode = new DoublyLinkedNode<T>(value);

addedNode.Past = lastNode;

lastNode.Next = addedNode;

lastNode = addedNode;

}

else if (num == 0) // first

{

DoublyLinkedNode<T> addedNode = new DoublyLinkedNode<T>(value);

addedNode.Next = firstNode;

firstNode.Past = addedNode;

firstNode = addedNode;

}

else

{

DoublyLinkedNode<T>? currentNode = firstNode;

for (int i = 1; i < Length; i++)

{

if (i == num)

{

DoublyLinkedNode<T> node = new DoublyLinkedNode<T>(value);

if (currentNode.Next != null)

{

node.Next = currentNode.Next;

currentNode.Next.Past = node;

}

node.Past = currentNode;

currentNode.Next = node;

break;

}

if (currentNode.Next != null)

{

currentNode = currentNode.Next;

}

}

}

Length += 1;

}

public void delByNumber(int num)

{

if (Length == 0 || num > Length) // empty

{

throw new IndexOutOfRangeException();

}

else if (Length == 1) // only one

{

firstNode = null;

lastNode = null;

}

else if (num == Length) // last

{

DoublyLinkedNode<T> preLastNode = lastNode.Past;

lastNode.Past.Next = null;

lastNode.Past = null;

lastNode = preLastNode;

}

else if (num == 0) // first

{

DoublyLinkedNode<T> preFirstNode = firstNode.Next;

firstNode.Next.Past = null;

firstNode.Next = null;

firstNode = preFirstNode;

}

else

{

DoublyLinkedNode<T>? currentNode = firstNode;

for (int i = 1; i < Length; i++)

{

if (i == num)

{

currentNode.Data = default;

if (currentNode.Next != null)

{

currentNode.Next.Past = currentNode.Past;

currentNode.Past.Next = currentNode.Next;

}

else

{

currentNode.Past.Next = null;

}

break;

}

if (currentNode.Next != null)

{

currentNode = currentNode.Next;

}

}

}

}

public object ShallowCopy()

{

DoublyLinkedList<T> newList = new DoublyLinkedList<T>();

DoublyLinkedNode<T>? tempNode = firstNode;

for (int i = 0; i < Length; i++)

{

newList.AddByNumber(Length, tempNode.Data);

if (tempNode.Next != null)

{

tempNode = tempNode.Next;

}

}

return newList;

}

public object Clone()

{

DoublyLinkedList<T> newList = new DoublyLinkedList<T>();

DoublyLinkedNode<T>? tempNode = firstNode;

for (int i = 0; i < Length; i++)

{

newList.AddByNumber(Length, (T)tempNode.Data.Clone()); // remove .Clone() to make shallow copy

if (tempNode.Next != null)

{

tempNode = tempNode.Next;

}

}

return newList;

}

}

}

Класс TreeNode:

using System;

using System.Collections.Generic;

using System.Diagnostics.CodeAnalysis;

using System.Linq;

using System.Text;

using lab;

using System.Threading.Tasks;

using System.Reflection;

using System.Collections;

using System.Drawing;

namespace ClassLibraryHSE1course

{

[ExcludeFromCodeCoverage]

public class TreeNode<T> : ICloneable where T : ICloneable,IComparable

{

public T Data { get; set; }

public TreeNode<T>? Right { get; set; }

public TreeNode<T>? Left { get; set; }

public TreeNode()

{

Data = default(T);

Left = null;

Right = null;

}

[ExcludeFromCodeCoverage]

public TreeNode(T data)

{

Data = data;

Left = null;

Right = null;

}

[ExcludeFromCodeCoverage]

public TreeNode(int size, int ind)

{

if (size == 0)

{

Data = default(T);

Left = null;

Right = null;

}

Data = BinaryTree<T>.keyboardT.Invoke(ind);

int leftSize = size/2;

int rightSize = size - leftSize - 1;

if (leftSize == 0) Left = null;

else Left = new TreeNode<T>(leftSize, ind + 1);

if (rightSize == 0) Right = null;

else Right = new TreeNode<T>(rightSize, size/2+ind+1);

}

[ExcludeFromCodeCoverage]

public TreeNode(int size, int min, int max)

{

if (size == 0)

{

Data = default(T);

Left = null;

Right = null;

}

Data = BinaryTree<T>.randomT.Invoke(min, max);

int leftSize = size/2;

int rightSize = size - leftSize - 1;

if (leftSize == 0) Left = null;

else Left = new TreeNode<T>(leftSize, min,max);

if (rightSize == 0) Right = null;

else Right = new TreeNode<T>(rightSize, min, max);

}

[ExcludeFromCodeCoverage]

public TreeNode(TreeNode<T> node)

{

Data = (T)node.Data.Clone();

Left = null;

Right = null;

}

public override string ToString()

{

return Data.ToString();

}

[ExcludeFromCodeCoverage]

public void Show(int len=0)

{

if (Right != null) Right.Show(len+5);

MethodInfo methodInfo = Data.GetType().GetMethod("Show");

if (methodInfo != null)

{

methodInfo.Invoke(Data, new object[] { len});

}

else

{

throw new Exception("No showing method in data type");

}

Output.PrintLine("");

if (Left != null) Left.Show(len+5);

}

[ExcludeFromCodeCoverage]

public void Show(int ind, int size, int len = 0)

{

if (Right != null) Right.Show(size/2+ind+1, size - size/2 - 1, len+5);

MethodInfo methodInfo = Data.GetType().GetMethod("Show");

if (methodInfo != null)

{

Output.Print($"{ind})");

methodInfo.Invoke(Data, new object[] { len });

}

else

{

throw new Exception("No showing method in data type");

}

Output.PrintLine("");

if (Left != null) Left.Show(ind + 1, size/2, len+5);

}

public T MinElement()

{

if (this != null)

{

T? lmin = default;

T? rmin = default;

T? min = default;

if (this.Data == null)

{

return min;

}

else

{

min = this.Data;

}

if (this.Left != null)

{

lmin = this.Left.MinElement();

if (lmin.CompareTo(min) < 0) min = lmin;

}

if (this.Right != null)

{

rmin = this.Right.MinElement();

if (rmin.CompareTo(min) < 0) min = rmin;

}

return min;

}

return default;

}

public List<T> FormList()

{

if (this != null)

{

List<T> list = new List<T>();

if (this.Data != null)

{

list.Add((T)this.Data.Clone());

}

if (this.Left != null)

{

List<T> llist = this.Left.FormList();

foreach (T i in llist)

{

list.Add(i);

}

}

if (this.Right != null)

{

List<T> rlist = this.Right.FormList();

foreach (T i in rlist)

{

list.Add(i);

}

}

return list;

}

return new List<T>();

}

public void FormTree(List<T> list)

{

int len = list.Count;

if (len == 0)

{

Data = default;

Left = null;

Right = null;

}

Data = list[len/2];

List<T> leftlist = list.GetRange(0, len/2);

List<T> rightlist = list.GetRange(len/2+1, (len-1)/2);

if (leftlist.Count == 0) Left = null;

else

{

Left = new TreeNode<T>();

Left.FormTree(leftlist);

}

if (rightlist.Count == 0) Right = null;

else

{

Right = new TreeNode<T>();

Right.FormTree(rightlist);

}

}

public void Add(T val, int len)

{

if (len == 0)

{

Data = val;

Left = null;

Right = null;

}

else if (len == 1)

{

Left = new TreeNode<T>(val);

}

else if (len == 2)

{

Right = new TreeNode<T>(val);

}

else

{

int llen = len / 2;

int rlen = len - llen - 1;

if (llen <= rlen)

{

Left.Add(val,llen);

}

else

{

Right.Add(val, rlen);

}

}

}

public T Find(T val)

{

if (Data == null) return default;

T lv = default;

T rv = default;

if (val.CompareTo(Data) == 0) return Data;

if (Left != null) lv = Left.Find(val);

if (lv is not null && val.CompareTo(lv) == 0) return lv;

if (Right != null) rv = Right.Find(val);

if (rv is not null && val.CompareTo(rv) == 0) return rv;

return default;

}

public bool Remove(T val, int len)

{

if (this != null)

{

return true;

}

return true;

}

public object ShallowCopy()

{

if (this == null)

{

return null;

}

TreeNode<T> treeClone = new TreeNode<T>(this.Data);

if (this.Left != null) treeClone.Left = (TreeNode<T>)this.Left.ShallowCopy();

if (this.Right != null) treeClone.Right = (TreeNode<T>)this.Right.ShallowCopy();

return treeClone;

}

public object Clone()

{

if (this == null)

{

return null;

}

TreeNode<T> treeClone = new TreeNode<T>((T)this.Data.Clone());

if (this.Left != null) treeClone.Left = (TreeNode<T>)this.Left.Clone();

if (this.Right != null) treeClone.Right = (TreeNode<T>)this.Right.Clone();

return treeClone;

}

public bool Contains(T item)

{

if (this == null)

{

return false;

}

bool isLeftContains = false;

bool isRightContains = false;

if (Left != null) isLeftContains = Left.Contains(item);

if (Right != null) isRightContains = Right.Contains(item);

if (EqualityComparer<T>.Default.Equals(this.Data, item) || isLeftContains || isRightContains) return true;

else return false;

}

}

}

Класс BinartTree:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Diagnostics.CodeAnalysis;

using System.Reflection;

using lab;

using static lab.Output;

using System.Drawing;

using System.Collections;

namespace ClassLibraryHSE1course

{

public class BinaryTree<T> : ICloneable, IEnumerable<T>, ICollection<T> where T : ICloneable, IComparable

{

private TreeNode<T>? root;

public int Length { get; private set; }

public int Height { get; private set; }

public bool IsSearchTree { get; private set; }

public int Count => Length;

public bool IsReadOnly => false;

public delegate T RandomT(int min, int max);

public static RandomT? randomT;

public delegate T KeyboardT(int i);

public static KeyboardT? keyboardT;

public BinaryTree()

{

root = null;

Length = 0;

Height = 0;

IsSearchTree = false;

}

[ExcludeFromCodeCoverage]

// zero vehicles constructor

public BinaryTree(int size) // keyboard

{

Length = size;

Height = (int)MathF.Ceiling(MathF.Log2(size + 1));

IsSearchTree = false;

if (size == 0)

{

root = null;

return;

}

// size > 1

root = new TreeNode<T>(size, 1);

}

[ExcludeFromCodeCoverage]

// Random vehicle constructor

public BinaryTree(int size, int min, int max)

{

Length = size;

Height = (int)MathF.Ceiling(MathF.Log2(size + 1));

IsSearchTree = false;

if (size == 0)

{

root = null;

return;

}

// size > 1

root = new TreeNode<T>(size, min, max);

}

[ExcludeFromCodeCoverage]

public BinaryTree(BinaryTree<T> tree) // copy

{

BinaryTree<T> tr = (BinaryTree<T>)tree.Clone();

root = tr.root;

Length = tr.Length;

Height = tr.Height;

IsSearchTree = tr.IsSearchTree;

}

[ExcludeFromCodeCoverage]

public void Show()

{

if (root == null)

{

Output.PrintLine("Empty Tree");

}

else

{

root.Show();

}

}

[ExcludeFromCodeCoverage]

public void ShowWithIndex()

{

if (root == null)

{

Output.PrintLine("Empty Tree");

}

else

{

root.Show(1, Length, 0);

}

}

public T MinElement()

{

if (IsSearchTree)

{

TreeNode<T>? tempRoot;

if (root != null) tempRoot = root;

else tempRoot = null;

while (tempRoot.Left != null)

{

tempRoot = tempRoot.Left;

}

return tempRoot.Data;

}

else

{

return root.MinElement();

}

}

public T Find(T val)

{

if (IsSearchTree) // Search tree

{

TreeNode<T>? tempRoot;

if (root != null)

{

tempRoot = root;

while (tempRoot.Left != null && tempRoot.Right != null) //|| ((tempRoot.Left != null && tempRoot.Left.Data )))

{

if (EqualityComparer<T>.Default.Equals(root.Data, val)) return root.Data;

else

{

if (val.CompareTo(tempRoot.Data) > 0)

{

if (tempRoot.Right != null)

{

tempRoot = tempRoot.Right;

}

else

{

return default;

}

}

else if (val.CompareTo(tempRoot.Data) < 0)

{

if (tempRoot.Left != null)

{

tempRoot = tempRoot.Left;

}

else

{

return default;

}

}

else

{

return tempRoot.Data;

}

}

}

if (val.CompareTo(tempRoot.Data) == 0) return tempRoot.Data;

return default;

}

else return default;

}

else // Non-search tree

{

return root.Find(val);

}

}

public void FormSearch(int addType = 0, T val = default) // 0 - no adding, 1 - add, 2 - remove

{

List<T> list = root.FormList();

if (addType == 1)

{

if (!EqualityComparer<T>.Default.Equals(val, default))

{

list.Add(val);

}

}

if (addType == 2)

{

if (!EqualityComparer<T>.Default.Equals(val, default))

{

if (list.Contains(val))

{

list.Remove(val);

}

}

}

list.Sort();

// Remove all duplicate elements

T temp = list[0];

List<T> delList = new List<T>();

for (int i = 1; i < list.Count; i++)

{

if (EqualityComparer<T>.Default.Equals(temp, list[i]))

{

delList.Add(temp);

}

else

{

temp = list[i];

}

}

foreach (T del in delList)

{

list.Remove(del);

}

root.FormTree(list);

Height = (int)MathF.Ceiling(MathF.Log2(Length + 1));

IsSearchTree = true;

}

public void FormNonSearch(T val = default)

{

IsSearchTree = false;

}

public void Add(T val)

{

if (IsSearchTree)

{

if (Contains(val))

{

PrintLine("Unable to add duplicate element to search tree");

return;

}

else

{

FormSearch(1, val);

}

}

else

{

if (root == null) root = new TreeNode<T>(val);

else root.Add(val, Length);

}

Length += 1;

Height = (int)MathF.Ceiling(MathF.Log2(Length + 1));

PrintLine("Value successfully added in tree");

}

public object ShallowCopy()

{

if (this == null)

{

return null;

}

BinaryTree<T> tree = new BinaryTree<T>();

tree.Height = this.Height;

tree.Length = this.Length;

tree.root = (TreeNode<T>)this.root.ShallowCopy();

tree.IsSearchTree = this.IsSearchTree;

return tree;

}

public object Clone()

{

if (this == null)

{

return null;

}

BinaryTree<T> tree = new BinaryTree<T>();

tree.Height = this.Height;

tree.Length = this.Length;

tree.IsSearchTree = this.IsSearchTree;

if (this.root != null) tree.root = (TreeNode<T>)this.root.Clone();

else this.root = null;

return tree;

}

public IEnumerator<T> GetEnumerator()

{

List<T> list = root.FormList();

foreach (T i in list)

{

yield return i;

}

}

IEnumerator IEnumerable.GetEnumerator()

{

throw new NotImplementedException();

}

public void Clear()

{

root = new TreeNode<T>();

}

public bool Contains(T item)

{

if (root.Contains(item))

{

return true;

}

else

{

return false;

}

}

public void CopyTo(T[] array, int arrayIndex)

{

if (array == null)

throw new ArgumentNullException("The array cannot be null");

if (arrayIndex < 0)

throw new ArgumentOutOfRangeException("The starting array index cannot be negative");

if (Count > array.Length - arrayIndex)

throw new ArgumentException("The destination array has fewer elements than the collection");

List<T> list = root.FormList();

for (int i = 0; i < list.Count; i++)

{

array[i + arrayIndex] = list[i];

}

}

public bool Remove(T item)

{

if (IsSearchTree)

{

if (!Contains(item))

{

return false;

}

else

{

FormSearch(2, item);

}

}

else

{

if (root == null || !Contains(item)) return false;

else

{

if (Length == 1 && EqualityComparer<T>.Default.Equals(root.Data, item))

{

root = new TreeNode<T>();

}

else

{

bool isDeleted = false;

List<T> list = root.FormList();

if (!EqualityComparer<T>.Default.Equals(item, default))

{

if (list.Contains(item))

{

isDeleted = list.Remove(item);

}

}

root.FormTree(list);

if (!isDeleted) return false;

}

}

}

Length -= 1;

Height = (int)MathF.Ceiling(MathF.Log2(Length + 1));

return true;

}

public T this[int index]

{

get

{

List<T> list = root.FormList();

return list[index];

}

set

{

List<T> list = root.FormList();

list[index] = value;

if (IsSearchTree)

{

list.Sort();

// Remove all duplicate elements

T temp = list[0];

List<T> delList = new List<T>();

for (int i = 1; i < list.Count; i++)

{

if (EqualityComparer<T>.Default.Equals(temp, list[i]))

{

delList.Add(temp);

}

else

{

temp = list[i];

}

}

foreach (T del in delList)

{

list.Remove(del);

}

Height = (int)MathF.Ceiling(MathF.Log2(Length + 1));

}

root.FormTree(list);

}

}

}

}

Класс HashNode:

using System;

using System.Collections.Generic;

using System.Diagnostics.CodeAnalysis;

using System.Linq;

using System.Reflection;

using System.Security.Cryptography;

using System.Text;

using System.Threading.Tasks;

using System.Xml.Linq;

using static System.Net.Mime.MediaTypeNames;

namespace ClassLibraryHSE1course

{

public class HashNode<V>: ICloneable where V : ICloneable

{

public int ?key;

public V ?value;

public HashNode()

{

key = null;

value = default(V);

}

[ExcludeFromCodeCoverage]

public HashNode(int ?key, V ?value)

{

this.key = key;

this.value = value;

}

[ExcludeFromCodeCoverage]

public HashNode(HashNode<V> node)

{

key = node.key;

value = (V)node.value.Clone();

}

public override string ToString()

{

return key.ToString() + ": " + value.ToString();

}

public object Clone()

{

return new HashNode<V>(key, (V)value.Clone());

}

public override int GetHashCode()

{

if (value != null)

{

return value.GetHashCode();

}

else

{

return 0;

}

}

}

}

Класс HashNode:

using lab;

using System.Diagnostics.CodeAnalysis;

using System.Reflection;

namespace ClassLibraryHSE1course

{

public class HashTable<V> : ICloneable where V : ICloneable

{

private HashNode<V>[] list;

public int Size { get; private set; }

public int Capacity

{

get

{

return list.Length;

}

}

public float LoadFactor { get; private set; }

private const float maxLoadFactor = 0.7f;

public delegate V RandomT(int min, int max);

public static RandomT? randomT;

public delegate V KeyboardT(int i);

public static KeyboardT? keyboardT;

[ExcludeFromCodeCoverage]

public HashTable()

{

list = new HashNode<V>[1];

Size = 0;

LoadFactor = 0;

}

[ExcludeFromCodeCoverage]

public HashTable(int size) // keyboard

{

if (size == 0)

{

list = new HashNode<V>[1];

Size = 0;

LoadFactor = 0;

}

else if (size == 1)

{

list = new HashNode<V>[2];

Size = 1;

LoadFactor = 0.5f;

V val = keyboardT.Invoke(0);

list[Math.Abs(val.GetHashCode() % Capacity)] = new HashNode<V>(val.GetHashCode(), val);

}

else

{

list = new HashNode<V>[size \* 2];

Size = 0;

for (int i = 0; i < size; i++)

{

V val = keyboardT.Invoke(0);

Add(val);

}

}

}

[ExcludeFromCodeCoverage]

public HashTable(int size, int min, int max) // random

{

if (size == 0)

{

list = new HashNode<V>[1];

Size = 0;

LoadFactor = 0;

}

else if (size == 1)

{

list = new HashNode<V>[2];

Size = 1;

LoadFactor = 0.5f;

V val = randomT.Invoke(min, max);

list[Math.Abs(val.GetHashCode() % Capacity)] = new HashNode<V>(val.GetHashCode(), val);

}

else

{

list = new HashNode<V>[size \* 2];

Size = 0;

for (int i = 0; i < size; i++)

{

V val = randomT.Invoke(min,max);

Add(val);

}

}

}

public void Add(V val)

{

Size += 1;

LoadFactor = Size / (float)Capacity;

do

{

List<V> tempList = new List<V>();

if (LoadFactor > maxLoadFactor)

{

for (int i = 0; i < Capacity; i++)

{

if (list[i] != null)

{

tempList.Add(list[i].value);

}

}

list = new HashNode<V>[Capacity \* 2];

LoadFactor = Size / (float)Capacity;

}

tempList.Add(val);

// place 1/all value(s)

for (int j = 0; j < tempList.Count; j++)

{

// from hash to end

bool isPlased = false;

for (int i = Math.Abs(val.GetHashCode() % Capacity); i < Capacity; i++)

{

if (list[i] == null)

{

list[i] = new HashNode<V>(tempList[j].GetHashCode(), tempList[j]);

isPlased = true;

break;

}

}

//from start to hash

if (!isPlased)

{

for (int i = 0; i < Capacity; i++)

{

if (list[i] == null)

{

list[i] = new HashNode<V>(tempList[j].GetHashCode(), tempList[j]);

break;

}

}

}

}

} while (LoadFactor > maxLoadFactor);

}

public V FindByKey(int k)

{

for (int i = Math.Abs(k % Capacity); i < Capacity; i++)

{

if (list[i] != null && k == list[i].key)

{

return list[i].value;

}

}

//from start to hash

for (int i = 0; i < Math.Abs(k % Capacity); i++)

{

if (list[i] != null && k == list[i].key)

{

return list[i].value;

}

}

return default(V);

}

public bool DeleteByKey(int k) // deleted - true, not found - false

{

for (int i = Math.Abs(k % Capacity); i < Capacity; i++)

{

if (list[i] != null && k == list[i].key)

{

list[i] = null;

return true;

}

}

//from start to hash

for (int i = 0; i < Math.Abs(k % Capacity); i++)

{

if (list[i] != null && k == list[i].key)

{

list[i] = null;

return true;

}

}

return false;

}

[ExcludeFromCodeCoverage]

public void Show()

{

if (list == null || Capacity == 0)

{

Output.PrintLine("Empty list");

}

else

{

Output.PrintLine("Hashtable: ");

for (int i = 0; i < Capacity; i++)

{

if (list[i] == null || list[i].key == null || list[i].value == null)

{

Output.PrintLine($"{i+1}) no value");

}

else

{

Output.Print($"{i+1}) key = {list[i].key}: ");

MethodInfo methodInfo = list[i].value.GetType().GetMethod("Show");

if (methodInfo != null)

{

methodInfo.Invoke(list[i].value, new object[] { 0 });

}

}

}

}

}

public object Clone()

{

throw new NotImplementedException();

}

}

}

## Код unit-тестов (листинг)

Код unit-тестов представлен в листинге ниже.

Класс UnitTestVehicle:

using ClassLibraryHSE1course;

using System.Diagnostics.CodeAnalysis;

namespace lab.Tests

{

[ExcludeFromCodeCoverage]

[TestClass]

public class UnitTests

{

// Vehicle

[TestMethod]

public void TestVehicleEquals1()

{

Vehicle vehicle1 = new Vehicle();

Vehicle vehicle2 = new Vehicle();

Assert.IsTrue(vehicle1.Equals(vehicle2));

}

[TestMethod]

public void TestVehicleEquals2()

{

Vehicle vehicle1 = new Vehicle();

Assert.IsFalse(vehicle1.Equals(null));

}

[TestMethod]

public void TestVehicleEquals3()

{

Vehicle vehicle1 = new Vehicle(1,2,3,4);

Vehicle vehicle2 = new Vehicle(1,2,3,4);

Assert.IsTrue(vehicle1.Equals(vehicle2));

}

[TestMethod]

public void TestVehicleEquals4()

{

Vehicle vehicle1 = new Vehicle(1, 2, 3);

Vehicle vehicle2 = new Vehicle(1, 2, 3);

Assert.IsTrue(vehicle1.Equals(vehicle2));

}

[TestMethod]

public void TestVehicleEquals5()

{

Vehicle vehicle1 = new Vehicle(1, 2);

Vehicle vehicle2 = new Vehicle(1, 2);

Assert.IsTrue(vehicle1.Equals(vehicle2));

}

[TestMethod]

public void TestVehicleEquals6()

{

Vehicle vehicle1 = new Vehicle(1);

Vehicle vehicle2 = new Vehicle(1);

Assert.IsTrue(vehicle1.Equals(vehicle2));

}

[TestMethod]

public void TestVehicleEquals7()

{

Vehicle vehicle1 = new Vehicle(1, 2, 3, 4);

Vehicle vehicle2 = new Vehicle(4, 3, 2, 1);

Assert.IsFalse(vehicle1.Equals(vehicle2));

}

[TestMethod]

public void TestVehicleEquals8()

{

Vehicle vehicle1 = new Vehicle();

Assert.IsFalse(vehicle1.Equals(4));

}

[TestMethod]

public void TestVehiclePropertyMass1()

{

int expected = 5;

Car car = new Car();

car.Mass = 5;

Assert.AreEqual(expected, car.Mass);

}

[TestMethod]

public void TestVehiclePropertyMass2()

{

int expected = 0;

Car car = new Car();

car.Mass = -5;

Assert.AreEqual(expected, car.Mass);

}

[TestMethod]

public void TestVehiclePropertyMaxSpeed1()

{

int expected = 5;

Car car = new Car();

car.MaxSpeed = 5;

Assert.AreEqual(expected, car.MaxSpeed);

}

[TestMethod]

public void TestVehiclePropertyMaxSpeed2()

{

int expected = 0;

Car car = new Car();

car.MaxSpeed = -5;

Assert.AreEqual(expected, car.MaxSpeed);

}

[TestMethod]

public void TestVehiclePropertyPower1()

{

int expected = 5;

Car car = new Car();

car.Power = 5;

Assert.AreEqual(expected, car.Power);

}

[TestMethod]

public void TestVehiclePropertyPower2()

{

int expected = 0;

Car car = new Car();

car.Power = -5;

Assert.AreEqual(expected, car.Power);

}

[TestMethod]

public void TestVehiclePropertyPassengerCapacity1()

{

int expected = 5;

Car car = new Car();

car.PassengerCapacity = 5;

Assert.AreEqual(expected, car.PassengerCapacity);

}

[TestMethod]

public void TestVehiclePropertyPassengerCapacity2()

{

int expected = 0;

Car car = new Car();

car.PassengerCapacity = -5;

Assert.AreEqual(expected, car.PassengerCapacity);

}

[TestMethod]

public void TestCompareTo1()

{

int expected = 2;

Car car = new Car(10);

Assert.AreEqual(expected, car.CompareTo(8));

}

[TestMethod]

public void TestCompareTo2()

{

double expected = 2;

Car car = new Car(10);

Assert.AreEqual(expected, car.CompareTo(8));

}

[TestMethod]

public void TestCompareTo3()

{

int expected = -2;

Car car = new Car(6);

Assert.AreEqual(expected, car.CompareTo(8));

}

[TestMethod]

public void TestCompareTo4()

{

double expected = -2;

Car car = new Car(6);

Assert.AreEqual(expected, car.CompareTo(8));

}

[TestMethod]

public void TestCompareTo5()

{

int expected = 0;

Car car = new Car(8);

Assert.AreEqual(expected, car.CompareTo(8));

}

[TestMethod]

public void TestCompareTo6()

{

double expected = 0;

Car car = new Car(8);

Assert.AreEqual(expected, car.CompareTo(8));

}

[TestMethod]

public void TestCompareTo7()

{

double expected = 2;

Car car1 = new Car(8);

Car car2 = new Car(6);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo8()

{

double expected = -2;

Car car1 = new Car(6);

Car car2 = new Car(8);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo9()

{

double expected = 2;

Car car1 = new Car(8, 8);

Car car2 = new Car(8, 6);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo10()

{

double expected = -2;

Car car1 = new Car(8, 6);

Car car2 = new Car(8, 8);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo11()

{

double expected = 2;

Car car1 = new Car(8, 8, 8);

Car car2 = new Car(8, 8, 6);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo12()

{

double expected = -2;

Car car1 = new Car(8, 8, 6);

Car car2 = new Car(8, 8, 8);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo13()

{

double expected = 2;

Car car1 = new Car(8, 8, 8, 8);

Car car2 = new Car(8, 8, 8, 6);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo14()

{

double expected = -2;

Car car1 = new Car(8, 8, 8, 6);

Car car2 = new Car(8, 8, 8, 8);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo15()

{

double expected = 0;

Car car1 = new Car(8, 8, 8, 8);

Car car2 = new Car(8, 8, 8, 8);

Assert.AreEqual(expected, car1.CompareTo(car2));

}

[TestMethod]

public void TestCompareTo16()

{

bool expected = true;

Car car = new Car(8, 8, 8, 8);

string str = "8, 8, 8, 8";

bool isException = false;

try

{

car.CompareTo(str);

}

catch (Exception)

{

isException = true;

}

Assert.AreEqual(expected, isException);

}

[TestMethod]

public void TestCompareTo17()

{

double expected = 18;

Car car = new Car(10);

Assert.AreEqual(expected, car.CompareTo(-8));

}

[TestMethod]

public void TestCompareTo18()

{

int expected = 18;

Car car = new Car(10);

Assert.AreEqual(expected, car.CompareTo(-8));

}

[TestMethod]

public void TestCompareTo19()

{

double expected = 8;

Car car = new Car(-8);

Assert.AreEqual(expected, car.CompareTo(-8));

}

[TestMethod]

public void TestCompareTo20()

{

int expected = 8;

Car car = new Car(-8);

Assert.AreEqual(expected, car.CompareTo(-8));

}

[TestMethod]

public void TestCompareTo21()

{

int expected = 16;

Car car = new Car(8.5);

Assert.AreEqual(expected, car.CompareTo(-7.5));

}

[TestMethod]

public void TestVehicleShallowCopy()

{

Vehicle vehicle1 = new Vehicle(1, 2, 3, 4);

Vehicle vehicle2 = vehicle1.ShallowCopy();

Assert.IsTrue(vehicle1.Equals(vehicle2));

vehicle2.Mass = 2;

Assert.AreNotEqual(vehicle1.Mass, vehicle2.Mass);

}

[TestMethod]

public void TestVehicleClone()

{

Vehicle vehicle1 = new Vehicle(1, 2, 3, 4);

Vehicle vehicle2 = (Vehicle)vehicle1.Clone();

Assert.IsTrue(vehicle1.Equals(vehicle2));

vehicle2.Mass = 2;

Assert.AreNotEqual(vehicle1.Mass, vehicle2.Mass);

}

[TestMethod]

public void TestVehicleGetHashCode()

{

Vehicle vehicle = new Vehicle(1, 2, 3, 4);

int expected = (vehicle.Mass.GetHashCode() + vehicle.MaxSpeed.GetHashCode() + vehicle.Power.GetHashCode() + vehicle.PassengerCapacity.GetHashCode());

Assert.AreEqual(expected, vehicle.GetHashCode());

}

// Car

[TestMethod]

public void TestCarPropertyFuelCapacity1()

{

int expected = 5;

Car car = new Car();

car.FuelCapacity = 5;

Assert.AreEqual(expected, car.FuelCapacity);

}

[TestMethod]

public void TestCarPropertyFuelCapacity2()

{

int expected = 0;

Car car = new Car();

car.FuelCapacity = -5;

Assert.AreEqual(expected, car.FuelCapacity);

}

[TestMethod]

public void TestCarPropertyFuelConsumption1()

{

int expected = 5;

Car car = new Car();

car.FuelConsumption = 5;

Assert.AreEqual(expected, car.FuelConsumption);

}

[TestMethod]

public void TestCarPropertyFuelConsumption2()

{

int expected = 0;

Car car = new Car();

car.FuelConsumption = -5;

Assert.AreEqual(expected, car.FuelConsumption);

}

[TestMethod]

public void TestCarPropertyManeuverability1()

{

int expected = 5;

Car car = new Car();

car.Maneuverability = 5;

Assert.AreEqual(expected, car.Maneuverability);

}

[TestMethod]

public void TestCarPropertyManeuverability2()

{

int expected = 0;

Car car = new Car();

car.Maneuverability = -5;

Assert.AreEqual(expected, car.Maneuverability);

}

[TestMethod]

public void TestCarPropertyMileage1()

{

int expected = 5;

Car car = new Car();

car.Mileage = 5;

Assert.AreEqual(expected, car.Mileage);

}

[TestMethod]

public void TestCarPropertyMileage2()

{

int expected = 0;

Car car = new Car();

car.Mileage = -5;

Assert.AreEqual(expected, car.Mileage);

}

[TestMethod]

public void TestCarRandomInit1()

{

int expected = 1;

Car car = new Car();

car.RandomInit(1, 1);

Assert.AreEqual(expected, car.Mass);

}

[TestMethod]

public void TestCarRandomInit2()

{

int expected = 0;

Car car = new Car();

car.RandomInit(-1, 0);

Assert.AreEqual(expected, car.Mass);

}

[TestMethod]

public void TestCarRandomInit3()

{

int expected = 0;

Car car = new Car();

car.RandomInit(-1, -2);

Assert.AreEqual(expected, car.Mass);

}

[TestMethod]

public void TestCarRandomInit4()

{

int expected = 2;

Car car = new Car();

car.RandomInit(2, 0);

Assert.AreEqual(expected, car.Mass);

}

[TestMethod]

public void TestCarEquals1()

{

Car car1 = new Car();

Car car2 = new Car();

Assert.IsTrue(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals2()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Assert.IsTrue(car2.Equals(car1));

}

[TestMethod]

public void TestCarEquals3()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(8, 7, 6, 5, 4, 3, 2, 1);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals4()

{

Train car1 = new Train(1, 2, 3, 4);

Car car2 = new Car(1, 2, 3, 4);

Assert.IsFalse(car2.Equals(car1));

}

[TestMethod]

public void TestCarEquals5()

{

Car car1 = new Car(1, 2, 3, 4);

Assert.IsFalse(car1.Equals(null));

}

[TestMethod]

public void TestCarEquals6()

{

Car car1 = new Car(1, 2, 3, 4);

Assert.IsFalse(car1.Equals(5));

}

[TestMethod]

public void TestCarEquals7()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(0, 2, 3, 4, 5, 6, 7, 8);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals8()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(1, 0, 3, 4, 5, 6, 7, 8);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals9()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(1, 2, 0, 4, 5, 6, 7, 8);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals10()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(1, 2, 3, 0, 5, 6, 7, 8);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals11()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(1, 2, 3, 4, 0, 6, 7, 8);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals12()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(1, 2, 3, 4, 5, 0, 7, 8);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals13()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(1, 2, 3, 4, 5, 6, 0, 8);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarEquals14()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = new Car(1, 2, 3, 4, 5, 6, 7, 0);

Assert.IsFalse(car1.Equals(car2));

}

[TestMethod]

public void TestCarShallowCopy()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = car1.ShallowCopy();

Assert.IsTrue(car1.Equals(car2));

car2.Mass = 2;

Assert.AreNotEqual(car1.Mass, car2.Mass);

}

[TestMethod]

public void TestCarClone()

{

Car car1 = new Car(1, 2, 3, 4, 5, 6, 7, 8);

Car car2 = (Car)car1.Clone();

Assert.IsTrue(car1.Equals(car2));

car2.Mass = 2;

Assert.AreNotEqual(car1.Mass, car2.Mass);

}

[TestMethod]

public void TestCarGetHashCode()

{

Car car = new Car(1, 2, 3, 4,5,6,7,8);

int expected = (car.Mass.GetHashCode() + car.MaxSpeed.GetHashCode() + car.Power.GetHashCode() + car.PassengerCapacity.GetHashCode() + car.Maneuverability.GetHashCode() + car.FuelConsumption.GetHashCode() + car.FuelCapacity.GetHashCode() + car.Mileage.GetHashCode()); ;

Assert.AreEqual(expected, car.GetHashCode());

}

// Train

[TestMethod]

public void TestTrainEquals1()

{

Train train1 = new Train();

Train train2 = new Train();

Assert.IsTrue(train1.Equals(train2));

}

[TestMethod]

public void TestTrainEquals2()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = new Train(1, 2, 3, 4, new int[1] { 5 });

Assert.IsTrue(train1.Equals(train2));

}

[TestMethod]

public void TestTrainEquals3()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = new Train(4, 3, 2, 1, new int[1] { 5 });

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestTrainEquals4()

{

Train train1 = new Train(1, 2, 3, 4);

Car train2 = new Car(1, 2, 3, 4);

Assert.IsFalse(train2.Equals(train1));

}

[TestMethod]

public void TestTrainEquals5()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Assert.IsFalse(train1.Equals(null));

}

[TestMethod]

public void TestTrainEquals6()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Assert.IsFalse(train1.Equals(5));

}

[TestMethod]

public void TestTrainEquals7()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = new Train(0, 2, 3, 4, new int[1] { 5 });

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestTrainEquals8()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = new Train(1, 0, 3, 4, new int[1] { 5 });

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestTrainEquals9()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = new Train(1, 2, 0, 4, new int[1] { 5 });

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestTrainEquals10()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = new Train(1, 2, 3, 0, new int[1] { 5 });

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestTrainEquals11()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = new Train(1, 2, 3, 4, new int[1] { 0 });

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestTrainEquals12()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = new Train(1, 2, 3, 4, new int[0] { });

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestTrainShallowCopy()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = train1.ShallowCopy();

Assert.IsTrue(train1.Equals(train2));

train2.Mass = 2;

Assert.AreNotEqual(train1.Mass, train2.Mass);

train2.Wagons = new int[2];

Assert.AreNotEqual(train1.Wagons, train2.Wagons);

}

[TestMethod]

public void TestTrainClone()

{

Train train1 = new Train(1, 2, 3, 4, new int[1] { 5 });

Train train2 = (Train)train1.Clone();

Assert.IsTrue(train1.Equals(train2));

train2.Mass = 2;

Assert.AreNotEqual(train1.Mass, train2.Mass);

train2.Wagons = new int[2];

Assert.AreNotEqual(train1.Wagons, train2.Wagons);

}

[TestMethod]

public void TestTrainGetHashCode()

{

Train train = new Train(1, 2, 3, 4, new int[1]);

int expected = (train.Mass.GetHashCode() + train.MaxSpeed.GetHashCode() + train.Power.GetHashCode() + train.PassengerCapacity.GetHashCode() + train.Wagons.Length.GetHashCode());

Assert.AreEqual(expected, train.GetHashCode());

}

// Express

[TestMethod]

public void TestExpressEquals1()

{

Express train1 = new Express();

Express train2 = new Express();

Assert.IsTrue(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals2()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Assert.IsTrue(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals3()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(4, 3, 2, 1, new int[1] { 5 }, "name");

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals4()

{

Express train1 = new Express(1, 2, 3, 4);

Car train2 = new Car(1, 2, 3, 4);

Assert.IsFalse(train2.Equals(train1));

}

[TestMethod]

public void TestExpressEquals5()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Assert.IsFalse(train1.Equals(null));

}

[TestMethod]

public void TestExpressEquals6()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Assert.IsFalse(train1.Equals(5));

}

[TestMethod]

public void TestExpressEquals7()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(0, 2, 3, 4, new int[1] { 5 }, "name");

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals8()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(1, 0, 3, 4, new int[1] { 5 }, "name");

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals9()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(1, 2, 0, 4, new int[1] { 5 }, "name");

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals10()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(1, 2, 3, 0, new int[1] { 5 }, "name");

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals11()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(1, 2, 3, 4, new int[1] { 0 }, "name");

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals12()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(1, 2, 3, 4, new int[0] { }, "name");

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestExpressEquals13()

{

Express train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Express train2 = new Express(1, 2, 3, 4, new int[1] { 5 }, "eman");

Assert.IsFalse(train1.Equals(train2));

}

[TestMethod]

public void TestExpressRandomInit1()

{

int expected = 1;

Express express = new Express();

express.RandomInit(1, 1);

Assert.AreEqual(expected, express.Mass);

}

[TestMethod]

public void TestExpressRandomInit2()

{

int expected = 0;

Express express = new Express();

express.RandomInit(-1, 0);

Assert.AreEqual(expected, express.Mass);

}

[TestMethod]

public void TestExpressRandomInit3()

{

int expected = 0;

Express express = new Express();

express.RandomInit(-1, -2);

Assert.AreEqual(expected, express.Mass);

}

[TestMethod]

public void TestExpressRandomInit4()

{

int expected = 2;

Express express = new Express();

express.RandomInit(2, 0);

Assert.AreEqual(expected, express.Mass);

}

[TestMethod]

public void TestExpressShallowCopy()

{

Train train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Train train2 = train1.ShallowCopy();

Assert.IsTrue(train1.Equals(train2));

train2.Mass = 2;

Assert.AreNotEqual(train1.Mass, train2.Mass);

train2.Wagons = new int[2];

Assert.AreNotEqual(train1.Wagons, train2.Wagons);

}

[TestMethod]

public void TestExpressClone()

{

Train train1 = new Express(1, 2, 3, 4, new int[1] { 5 }, "name");

Train train2 = (Express)train1.Clone();

Assert.IsTrue(train1.Equals(train2));

train2.Mass = 2;

Assert.AreNotEqual(train1.Mass, train2.Mass);

train2.Wagons = new int[2];

Assert.AreNotEqual(train1.Wagons, train2.Wagons);

}

[TestMethod]

public void TestExpressGetHashCode()

{

Express express = new Express(1, 2, 3, 4, new int[1],"123");

int expected = (express.Mass.GetHashCode() + express.MaxSpeed.GetHashCode() + express.Power.GetHashCode() + express.PassengerCapacity.GetHashCode() + express.Wagons.Length.GetHashCode() + express.Name.GetHashCode());

Assert.AreEqual(expected, express.GetHashCode());

}

// Comparer

[TestMethod]

public void TestComparer1()

{

double expected = 2;

Car car1 = new Car(0, 0, 0, 8);

Car car2 = new Car(0, 0, 0, 6);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer2()

{

double expected = -2;

Car car2 = new Car(0, 0, 0, 8);

Car car1 = new Car(0, 0, 0, 6);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer3()

{

double expected = 2;

Car car1 = new Car(0, 0, 8, 8);

Car car2 = new Car(0, 0, 6, 8);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer4()

{

double expected = -2;

Car car2 = new Car(0, 0, 8, 8);

Car car1 = new Car(0, 0, 6, 8);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer5()

{

double expected = 2;

Car car1 = new Car(0, 8, 8, 8);

Car car2 = new Car(0, 6, 8, 8);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer6()

{

double expected = -2;

Car car2 = new Car(0, 8, 8, 8);

Car car1 = new Car(0, 6, 8, 8);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer7()

{

double expected = 2;

Car car1 = new Car(8, 8, 8, 8);

Car car2 = new Car(6, 8, 8, 8);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer8()

{

double expected = -2;

Car car2 = new Car(8, 8, 8, 8);

Car car1 = new Car(6, 8, 8, 8);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer9()

{

double expected = 0;

Car car1 = new Car(8, 8, 8, 8);

Car car2 = new Car(8, 8, 8, 8);

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

[TestMethod]

public void TestComparer10()

{

bool expected = true;

Car car1 = new Car(8, 8, 8, 8);

Car car2 = null;

bool isException = false;

try

{

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

catch (Exception)

{

isException = true;

}

Assert.AreEqual(expected, isException);

}

[TestMethod]

public void TestComparer11()

{

bool expected = true;

Car car2 = new Car(8, 8, 8, 8);

Car car1 = null;

bool isException = false;

try

{

VehiclesComparer Comparer = new VehiclesComparer();

Assert.AreEqual(expected, Comparer.Compare(car1, car2));

}

catch (Exception)

{

isException = true;

}

Assert.AreEqual(expected, isException);

}

// Human

[TestMethod]

public void TestHumanRandomInit1()

{

int expected = 1;

Human human = new Human();

human.RandomInit(1, 1);

Assert.AreEqual(expected, human.Weight);

}

[TestMethod]

public void TestHumanRandomInit2()

{

int expected = 0;

Human human = new Human();

human.RandomInit(-1, 0);

Assert.AreEqual(expected, human.Weight);

}

[TestMethod]

public void TestHumanRandomInit3()

{

int expected = 0;

Human human = new Human();

human.RandomInit(-1, -2);

Assert.AreEqual(expected, human.Weight);

}

[TestMethod]

public void TestHumanRandomInit4()

{

int expected = 2;

Human human = new Human();

human.RandomInit(2, 0);

Assert.AreEqual(expected, human.Weight);

}

[TestMethod]

public void TestHumanShallowCopy()

{

Human human1 = new Human(1, 2, new Car(1, 2, 3, 4, 5, 6, 7, 8));

Human human2 = human1.ShallowCopy();

Assert.AreEqual(human1.Weight, human2.Weight);

Assert.AreEqual(human1.Height, human2.Height);

Assert.IsTrue(human1.PersonalCar.Equals(human2.PersonalCar));

human2.Weight = 10;

Assert.AreNotEqual(human1.Weight, human2.Weight);

human2.PersonalCar.Mass = 10;

Assert.AreEqual(human1.PersonalCar.Mass, human2.PersonalCar.Mass);

}

[TestMethod]

public void TestHumanClone()

{

Human human1 = new Human(1, 2, new Car(1, 2, 3, 4, 5, 6, 7, 8));

Human human2 = (Human)human1.Clone();

Assert.AreEqual(human1.Weight, human2.Weight);

Assert.AreEqual(human1.Height, human2.Height);

Assert.IsTrue(human1.PersonalCar.Equals(human2.PersonalCar));

human2.Weight = 10;

Assert.AreNotEqual(human1.Weight, human2.Weight);

human2.PersonalCar.Mass = 10;

Assert.AreNotEqual(human1.PersonalCar.Mass, human2.PersonalCar.Mass);

}

}

}

Класс UnitTestCollectionMethods:

using ClassLibraryHSE1course;

using System.Collections;

namespace lab.Tests

{

[TestClass]

public class UnitTestCollectionMethods

{

// Hashtable

[TestMethod]

public void TestHashtableRandomFill()

{

Car expected = new Car(1, 1, 1, 1, 1, 1, 1, 1);

Hashtable hashtable = new Hashtable(100);

hashtable = HashtableMethods.FillRandom(hashtable, 22, 1, 1);

foreach (Vehicle vehicle in hashtable.Values)

{

if (vehicle is Car car)

{

Assert.AreEqual(expected, car);

}

}

}

[TestMethod]

public void TestHashtableAdd()

{

Car expected = new Car(1, 1, 1, 1, 1, 1, 1, 1);

Hashtable hashtable = new Hashtable();

for (int i = 0; i < 22; i++)

{

hashtable = HashtableMethods.AddRandom(hashtable, 1, 1);

}

foreach (Vehicle vehicle in hashtable.Values)

{

if (vehicle is Car car)

{

Assert.AreEqual(expected.CompareTo(car), 0);

}

}

}

[TestMethod]

public void TestHashtableExpressNumber()

{

int expected = 2;

Hashtable hashtable = new Hashtable(3);

hashtable.Add(1, new Car(1, 2, 3, 4, 5, 6, 7, 8));

hashtable.Add(2, new Car(1, 3, 3, 4, 5, 6, 7, 8));

hashtable.Add(3, new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

hashtable.Add(4, new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

hashtable.Add(5, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

hashtable.Add(6, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, HashtableMethods.ExpressNumber(hashtable));

}

[TestMethod]

public void TestHashtablePassengersNumber()

{

int expected = 60;

Hashtable hashtable = new Hashtable();

hashtable.Add(1, new Car(1, 2, 3, 4, 5, 6, 7, 8));

hashtable.Add(2, new Car(1, 3, 3, 4, 5, 6, 7, 8));

hashtable.Add(3, new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

hashtable.Add(4, new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

hashtable.Add(5, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

hashtable.Add(6, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, HashtableMethods.PassengersNumber(hashtable));

}

[TestMethod]

public void TestHashtableFindByCar1()

{

int expected = 2;

Hashtable hashtable = new Hashtable();

hashtable.Add(1, new Car(1, 2, 3, 4, 5, 6, 7, 8));

hashtable.Add(2, new Car(1, 3, 3, 4, 5, 6, 7, 8));

hashtable.Add(3, new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

hashtable.Add(4, new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

hashtable.Add(5, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

hashtable.Add(6, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, HashtableMethods.FindByCar(hashtable, new Car(1, 3, 3, 4, 5, 6, 7, 8)));

}

[TestMethod]

public void TestHashtableFindByCar2()

{

int expected = -1;

Hashtable hashtable = new Hashtable();

hashtable.Add(1, new Car(1, 2, 3, 4, 5, 6, 7, 8));

hashtable.Add(2, new Car(1, 3, 3, 4, 5, 6, 7, 8));

hashtable.Add(3, new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

hashtable.Add(4, new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

hashtable.Add(5, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

hashtable.Add(6, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, HashtableMethods.FindByCar(hashtable, new Car(1, 3, 1, 4, 1, 6, 7, 1)));

}

[TestMethod]

public void TestHashtableFindByTrain1()

{

int expected = 5;

Hashtable hashtable = new Hashtable();

hashtable.Add(1, new Car(1, 2, 3, 4, 5, 6, 7, 8));

hashtable.Add(2, new Car(1, 3, 3, 4, 5, 6, 7, 8));

hashtable.Add(3, new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

hashtable.Add(4, new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

hashtable.Add(5, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

hashtable.Add(6, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, HashtableMethods.FindByTrain(hashtable, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 })));

}

[TestMethod]

public void TestHashtableFindByTrain2()

{

int expected = -1;

Hashtable hashtable = new Hashtable();

hashtable.Add(1, new Car(1, 2, 3, 4, 5, 6, 7, 8));

hashtable.Add(2, new Car(1, 3, 3, 4, 5, 6, 7, 8));

hashtable.Add(3, new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

hashtable.Add(4, new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

hashtable.Add(5, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

hashtable.Add(6, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, HashtableMethods.FindByTrain(hashtable, new Train(3, 3, 2, 3, new int[5] { 5, 3, 3, 3, 1 })));

}

[TestMethod]

public void TestHashtableFindByExpress1()

{

int expected = 6;

Hashtable hashtable = new Hashtable();

hashtable.Add(1, new Car(1, 2, 3, 4, 5, 6, 7, 8));

hashtable.Add(2, new Car(1, 3, 3, 4, 5, 6, 7, 8));

hashtable.Add(3, new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

hashtable.Add(4, new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

hashtable.Add(5, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

hashtable.Add(6, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, HashtableMethods.FindByExpress(hashtable, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2")));

}

[TestMethod]

public void TestHashtableFindByExpress2()

{

int expected = -1;

Hashtable hashtable = new Hashtable();

hashtable.Add(1, new Car(1, 2, 3, 4, 5, 6, 7, 8));

hashtable.Add(2, new Car(1, 3, 3, 4, 5, 6, 7, 8));

hashtable.Add(3, new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

hashtable.Add(4, new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

hashtable.Add(5, new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

hashtable.Add(6, new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, HashtableMethods.FindByExpress(hashtable, new Express(2, 3, 2, 1, new int[5] { 2, 2, 2, 2, 1 }, "express2")));

}

// Queue<T>

[TestMethod]

public void TestQueueRandomFill()

{

Car expected = new Car(1, 1, 1, 1, 1, 1, 1, 1);

Queue<Vehicle> queue = new Queue<Vehicle>(100);

queue = QueueMethods<Vehicle>.FillRandom(queue, 100, 1, 1);

foreach (Vehicle vehicle in queue)

{

if (vehicle is Car car)

{

Assert.AreEqual(expected, car);

}

}

}

[TestMethod]

public void TestQueueAdd1()

{

Car expected = new Car(1, 1, 1, 1, 1, 1, 1, 1);

Queue<Vehicle> queue = new Queue<Vehicle>(100);

for (int i = 0; i < 100; i++)

{

queue = QueueMethods<Vehicle>.AddRandom(queue, 1, 1, 1);

}

foreach (Vehicle vehicle in queue)

{

if (vehicle is Car car)

{

Assert.AreEqual(expected.CompareTo(car), 0);

}

}

}

[TestMethod]

public void TestQueueAdd2()

{

Train expected = new Train(1, 1, 1, 1, new int[1] {1});

Queue<Vehicle> queue = new Queue<Vehicle>(100);

for (int i = 0; i < 100; i++)

{

queue = QueueMethods<Vehicle>.AddRandom(queue, 2, 1, 1);

}

foreach (Vehicle vehicle in queue)

{

if (vehicle is Train train)

{

Assert.AreEqual(expected.CompareTo(train), 0);

}

}

}

[TestMethod]

public void TestQueueAdd3()

{

Express expected = new Express(1, 1, 1, 1, new int[1] { 1 });

Queue<Vehicle> queue = new Queue<Vehicle>(100);

for (int i = 0; i < 100; i++)

{

queue = QueueMethods<Vehicle>.AddRandom(queue, 3, 1, 1);

}

foreach (Vehicle vehicle in queue)

{

if (vehicle is Express express)

{

Assert.AreEqual(expected.CompareTo(express), 0);

}

}

}

[TestMethod]

public void TestQueueExpressNumber()

{

int expected = 2;

Queue<Vehicle> queue = new Queue<Vehicle>(6);

queue.Enqueue(new Car(1, 2, 3, 4, 5, 6, 7, 8));

queue.Enqueue(new Car(1, 3, 3, 4, 5, 6, 7, 8));

queue.Enqueue(new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

queue.Enqueue(new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

queue.Enqueue(new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

queue.Enqueue(new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, QueueMethods<Vehicle>.ExpressNumber(queue));

}

[TestMethod]

public void TestQueuePassengersNumber()

{

int expected = 60;

Queue<Vehicle> queue = new Queue<Vehicle>(6);

queue.Enqueue(new Car(1, 2, 3, 4, 5, 6, 7, 8));

queue.Enqueue(new Car(1, 3, 3, 4, 5, 6, 7, 8));

queue.Enqueue(new Train(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

queue.Enqueue(new Express(1, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

queue.Enqueue(new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

queue.Enqueue(new Express(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

Assert.AreEqual(expected, QueueMethods<Vehicle>.PassengersNumber(queue));

}

[TestMethod]

public void TestQueueSort1()

{

Car expected = new Car(1, 3, 3, 4, 5, 6, 7, 8);

Queue<Vehicle> queue = new Queue<Vehicle>(6);

queue.Enqueue(new Car(6, 2, 3, 4, 5, 6, 7, 8));

queue.Enqueue(new Car(1, 3, 3, 4, 5, 6, 7, 8));

queue.Enqueue(new Train(5, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

queue.Enqueue(new Express(2, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

queue.Enqueue(new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

queue.Enqueue(new Express(3, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

queue = QueueMethods<Vehicle>.Sort(queue, 1);

Assert.AreEqual(expected, queue.Dequeue());

}

[TestMethod]

public void TestQueueSort2()

{

Car expected = new Car(6, 2, 3, 4, 5, 6, 7, 8);

Queue<Vehicle> queue = new Queue<Vehicle>(6);

queue.Enqueue(new Car(6, 2, 3, 4, 5, 6, 7, 8));

queue.Enqueue(new Car(1, 3, 3, 4, 5, 6, 7, 8));

queue.Enqueue(new Train(5, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }));

queue.Enqueue(new Express(2, 2, 3, 4, new int[5] { 1, 2, 3, 4, 5 }, "express1"));

queue.Enqueue(new Train(4, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }));

queue.Enqueue(new Express(3, 3, 2, 1, new int[5] { 5, 4, 3, 2, 1 }, "express2"));

queue = QueueMethods<Vehicle>.Sort(queue, 2);

Assert.AreEqual(expected, queue.Dequeue());

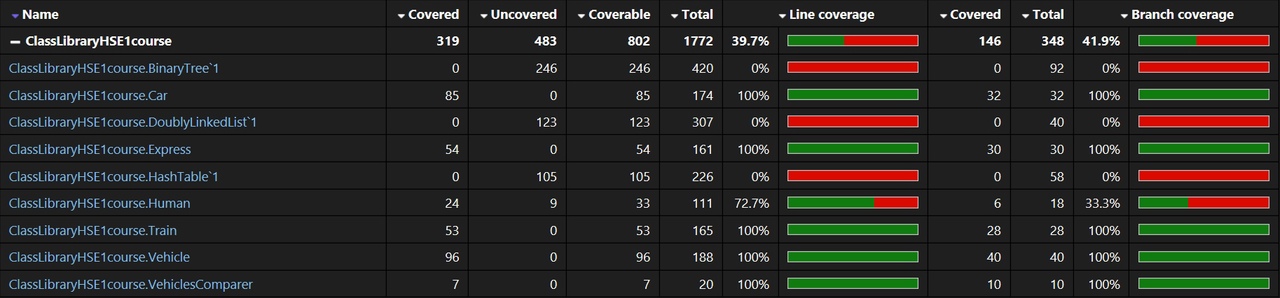
}

}

}

# Анализ покрытия тестов

Анализ покрытия кода тестами представлен ниже (см. рисунок 8).



***Рис. 8. Анализ покрытия тестов***

Из тестирования иерархии классов были исключены конструкторы без обработки входных значений, свойства без обработки входных и выходных значение, методы Show (Интерфейс вывода), методы Init (Интерфейс ввода с клавиатуры), методы ToString (Интерфейс вывода).

Классы Input и Output не тестируются, так как это интерфейсы ввода и вывода соответственно.

Во всех классах коллекций не тестируются только конструкторы (так как заполняют данные случайно или с клавиатуры).

Ячейки также не тестируются, так как хранят в себе только поля и конструкторы, но не хранят никакой логики и алгоритмов.

Основная программа (Program) не тестируется, так как тестирование иерархии классов и методов классов коллекций полностью покрывает критерии тестирования. Функции в основной программе так же не тестируются, так как генерируют случайное значение или значение с клавиатуры.

Таким образом, все элементы программы полностью протестированы как с точки зрения покрытия строчек кода, так и с точки зрения покрытия всех его ветвей.