Практическая работа № 12.
Классы-коллекции, создаваемые пользователем

Цель работы: Получить практические навыки создания классов, реализующих коллекции.

1. Постановка задачи

2.1. Задание 1.

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. Удалить список из памяти.

2.2. Задание 2.

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.
5. Распечатать полученное дерево.
6. Удалить дерево из памяти.

2.3. Задание 3

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).

2.4. Задание 4

Реализовать обобщенную коллекцию, указанную в варианте. Для этого:

1. Реализовать конструкторы:
    - public MyCollection() - предназначен для создания пустой коллекции.
    - public MyCollection (int capacity*)* - создает пустую коллекцию с начальной емкостью, заданной параметром capacity.
    - public MyCollection (MyCollection c) - служит для создания коллекции, которая инициализируется элементами и емкостью коллекции, заданной параметром c.
2. Для всех коллекций реализовать:
    - свойство Count, позволяющее получить количество элементов в коллекции;
    - методы для добавления одного или нескольких элементов в коллекцию;
    - методы для удаления одного или нескольких элементов из коллекции (кроме деревьев);[1]
    - метод для поиска элемента по значению;

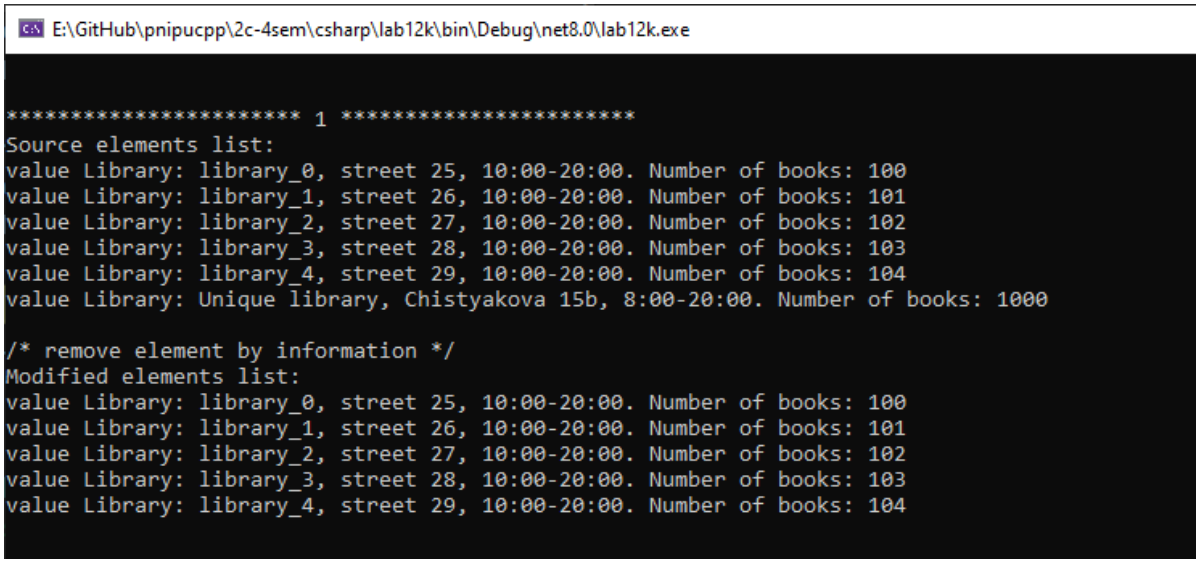
- метод для клонирования коллекции;
- метод для поверхностного копирования;
- метод для удаления коллекции из памяти.

3. Реализовать интерфейсы IEnumerable и IEnumerator (если это необходимо).
4. Написать демонстрационную программу, в которой создаются коллекции, и демонстрируется работа всех реализованных методов, в том числе, перебор коллекции циклом foreach.

При работе с коллекцией использовать объекты из иерархии классов, разработанной в работе №10.

№ варианта	Двунаправленный список	Бинарное дерево	Хеш-таблица	Коллекция
7	Удалить из списка первый элемент с заданным информационным полем (например, с заданным именем).	Найти количество элементов дерева, у которых поле (например, имя) начинается с заданного символа.	Открытая адресация, поиск и удаление по ключу	Очередь на базе однонаправленного списка

Задание 1.



Задание 2.

```
********************** 2 **********************
0Library (0)
  8Library (1)
    9Library (2)
      10Library (3)
      11Library (3)
    12Library (2)
      13Library (3)
      14Library (3)
  1Library (1)
    2Library (2)
      3Library (3)
      4Library (3)
    5Library (2)
      6Library (3)
      7Library (3)
6 tree elements starts with '8' first symbol

In binary tree searched 2 pathes
walkPath.WalkDirections.Count = 4
Search tree added. Trees count 1
walkPath.WalkDirections.Count = 0
Search tree added. Trees count 2
Num Pathes 2
-----------------------------------------
0Library (0)
-----------------------------------------
0Library (0)
```

Задание 3

```
********************** 3 **********************
----- Add 8 elements to collection ------
[36] Obj: org_3
[42] Obj: org_0
[74] Obj: org_4
[79] Obj: org_7
[122] Obj: org_6
[130] Obj: org_1
[146] Obj: org_2
[155] Obj: org_5
[203] Obj: ObjectForDeleting
----- find object and delete from collection ------
object found
object deleted
[36] Obj: org_3
[42] Obj: org_0
[74] Obj: org_4
[79] Obj: org_7
[122] Obj: org_6
[130] Obj: org_1
[146] Obj: org_2
[155] Obj: org_5
```

```
----- Full fill collection (collisions demo) ------
Collision on 20 index!
Collision on 36 index!
Collision on 39 index!
Collision on 46 index!
Collision on 49 index!
Collision on 50 index!
Collision on 51 index!
Collision on 54 index!
Collision on 63 index!
Collision on 68 index!
Collision on 69 index!
Collision on 70 index!
Collision on 71 index!
Collision on 74 index!
Collision on 76 index!
Collision on 81 index!
Collision on 85 index!
Collision on 92 index!
Collision on 95 index!
Collision on 107 index!
Collision on 109 index!
Collision on 110 index!
Collision on 118 index!
Collision on 124 index!
```

Задание 4

```
************************ 4 ************************
collection is empty

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue

Pushed lab12k.Organization data to queue
```

```
--- printing by foreach ---
Data in myQueue: org_0, address, time --
Data in myQueue: org_1, address, time --
Data in myQueue: org_2, address, time --
Data in myQueue: org_3, address, time --
Data in myQueue: org_4, address, time --
Data in myQueue: org_5, address, time --
Data in myQueue: org_6, address, time --
Data in myQueue: org_7, address, time --
Data in myQueue: org_8, address, time --
Data in myQueue: org_9, address, time --

--- read from queue ---
Readed from queue: org_0, address, time --
Readed from queue: org_1, address, time --
Readed from queue: org_2, address, time --
Readed from queue: org_3, address, time --
Readed from queue: org_4, address, time --
Readed from queue: org_5, address, time --
Readed from queue: org_6, address, time --
Readed from queue: org_7, address, time --
Readed from queue: org_8, address, time --
Readed from queue: org_9, address, time --
```

Исходный код

Lab121.cs

```
namespace lab12k.labs
{
    public class Lab121
    {
        private static void DraftPrint(MyDCLinkedList<Library>? linkedList)
        {
            if (linkedList == null)
            {
                Console.WriteLine("linkedList is null");
                return;
            }

            MyDCLinkedListNode<Library>? node = linkedList.Tail;
            while (node != null)
            {
                Console.WriteLine(node.Data.GetFullInfo());
                node = node.Next;
            }
        }

        public void Start()
        {
            MyDCLinkedList<Library> linkedList = new MyDCLinkedList<Library>();
            for (int i = 0; i < 5; i++)
                linkedList.InsertLast(new Library($"library_{i}", $"street {i +
25}", "10:00-20:00", i + 100));

            /* object for deleting from LL */
            Library libForDeleteFromLL = new Library("Unique library",
"Chistyakova 15b", "8:00-20:00", 1000);
            linkedList.InsertLast(libForDeleteFromLL);

            Console.WriteLine("Source elements list:");
            //DraftPrint(linkedList);
            foreach (var value in linkedList)
                Console.WriteLine("value {0}", value.Data.GetFullInfo());

            Console.Write("\n");

            /* remove element by information */
            Console.WriteLine("/* remove element by information */");
            if (!linkedList.RemoveElem(libForDeleteFromLL))
                Console.WriteLine("failed to delete object from linked list");

            Console.WriteLine("Modified elements list:");
            foreach (var value in linkedList)
                Console.WriteLine("value {0}", value.Data.GetFullInfo());
        }
    }
}
```

Lab122.cs

```
namespace lab12k.labs
{
    /* node creator impl for create and set data in tree node */
    public class TreeNodeCreatorImpl : ITreeBuilderNodeCreator<Library>
    {
        private int nodeNum;

        public TreeNodeCreatorImpl() {
            nodeNum = 0;
        }

        public MyBinaryTreeNode<Library>? CreateNode(MyBinaryTreeNode<Library>
?rootNode, int level) {
            Library lib = new Library($"{nodeNum}Library", $"street {nodeNum +
100}", $"8:00-20:00", nodeNum + 222);
            MyBinaryTreeNode<Library>? node = new MyBinaryTreeNode<Library>(lib,
rootNode);
            nodeNum++;
            return node;
        }
    }

    /* tree node printer implementation for print data in node graph */
    public class TreeNodePrinterImpl : ITreeNodePrinter<Library>
    {
        public string? Print(MyBinaryTreeNode<Library>? node, int level) {
            if (node == null)
                return "node was null";

            if (node.data == null)
                return "node data is null";

            return $"{node.data.GetOrgName()} ({level})";
        }
    }

    public class Lab122
    {
        public void Start()
        {
            const int numTreeLevels = 3;

            TreeNodeCreatorImpl creatorImpl = new TreeNodeCreatorImpl(); // create
NodeCreator implementation
            IdealBinaryTreeBuilder<Library> treeBuilder = new
IdealBinaryTreeBuilder<Library>(creatorImpl, numTreeLevels); // create
IdealBinaryTreeBuilder
            MyBinaryTreeNode<Library>? idealTreeRootNode =
treeBuilder.GetRootNode(); // get ref to root ideal tree node

            TreeNodePrinterImpl printerImpl = new TreeNodePrinterImpl(); // create
NodePrinter implementation
            new MyBinaryTreePrinter<Library>(printerImpl, idealTreeRootNode,
numTreeLevels); // print tree

            MyBinaryTreeNodesCounter treeElementsCounter = new
MyBinaryTreeNodesCounter(idealTreeRootNode, '1'); //create tree elements counter
            Console.WriteLine("{0} tree elements starts with '8' first symbol",
treeElementsCounter.GetNumNodes()); //print count elements starts with 'L' sym

            MyBSTBuilder binaryTreeSearchTreeBuilder = new
MyBSTBuilder(idealTreeRootNode, '1');
            Console.WriteLine("Num Pathes {0}",
binaryTreeSearchTreeBuilder.walkPathes.Count);
```

```
            foreach(MyBinaryTreeNode<Library> ?rootNode in
binaryTreeSearchTreeBuilder.searchTreeRootNodes) {
                Console.WriteLine("-----------------------------------------");
                new MyBinaryTreePrinter<Library>(printerImpl, rootNode,
numTreeLevels); // print BST
            }
        }
    }
}
```

Lab123.cs

```
namespace lab12k
{
    public class Lab123
    {
        MyHashTable<Organization, string> ?ht;

        public void Start()
        {
            ht = new MyHashTable<Organization, string>(256);
            Organization forDeleting = new Organization("For deleting", "",
"10:00-20:00");
            ht.Add(forDeleting, "ObjectForDeleting");

            Console.WriteLine("----- Add 8 elements to collection ------");
            for (int i = 0; i < 8; i++) {
                string str = $"org_{i}";
                Organization org = new Organization(str, $"address {i}", "10:00-
20:00");
                if(-1 == ht.Add(org, str)) {
                    Console.WriteLine("Collision!\n");
                }
            }

            ht.Print();

            Console.WriteLine("----- find object and delete from collection ------
");
            if (ht.Find(forDeleting) != null) {
                Console.WriteLine("object found");
                if(ht.Remove(forDeleting)) {
                    if(ht.Find(forDeleting) == null) {
                        Console.WriteLine("object deleted");
                        ht.Print();
                    }
                }
            }

            // show collisions
            Console.WriteLine("----- Full fill collection (collisions demo) ------
");
            for (int i = 8; i < ht.Count; i++) {
                string str = $"next_{i}";
                Organization org = new Organization(str, $"address {i}", "12:00-
22:00");
                if (-1 == ht.Add(org, str)) {
                    Console.WriteLine("Collision on {0} index!", i);
                }
            }
        }
    }
}
```

IdealBinaryTreeBuilder.cs

```
namespace lab12k
{
    public interface ITreeBuilderNodeCreator<_Ty>
    {
        /* for create node and set data to default constructor */
        public MyBinaryTreeNode<_Ty>? CreateNode(MyBinaryTreeNode<_Ty>? rootNode,
int level);
    }

    public class IdealBinaryTreeBuilder<_Ty>
    {
        private MyBinaryTreeNode<_Ty> root;
        ITreeBuilderNodeCreator<_Ty> creator;

        private void BuildTreeBranchRecursive(MyBinaryTreeNode<_Ty>? root, int
level, ref MyBinaryTreeNode<_Ty>? node) {
            /* end node no have child nodes */
            if (level == 0) {
                node = null;
                return;
            }
            level--;
            node = creator.CreateNode(root, level);
            BuildTreeBranchRecursive(root, level, ref node.left);
            BuildTreeBranchRecursive(root, level, ref node.right);
        }

        public IdealBinaryTreeBuilder(ITreeBuilderNodeCreator<_Ty> nodeCreator,
int levels) {
            creator = nodeCreator;
            root = creator.CreateNode(root, levels);
            BuildTreeBranchRecursive(root, levels, ref root.right);
            BuildTreeBranchRecursive(root, levels, ref root.left);
        }

        public MyBinaryTreeNode<_Ty> GetRootNode() { return root; }
    }
}
```

MyBinaryTree.cs

```
namespace lab12k
{
    public class MyBinaryTreeNode<_Ty>
    {
        public MyBinaryTreeNode<_Ty>? root;
        public MyBinaryTreeNode<_Ty>? left;
        public MyBinaryTreeNode<_Ty>? right;
        public _Ty data;

        public MyBinaryTreeNode(_Ty Data) {
            data = Data;
            root = null;
            left = null;
            right = null;
        }

        public MyBinaryTreeNode(_Ty Data, MyBinaryTreeNode<_Ty> rootNode) {
            data = Data;
            root = rootNode;
            left = null;
            right = null;
        }

        public MyBinaryTreeNode<_Ty>? AddLeftNode(_Ty Data)
```

```
        {
            return left = new MyBinaryTreeNode<_Ty>(Data, this);
        }

        public MyBinaryTreeNode<_Ty>? AddRightNode(_Ty Data)
        {
            return right = new MyBinaryTreeNode<_Ty>(Data, this);
        }
    }
}
```

MyBinaryTreeNodesCounter.cs

```
namespace lab12k
{
    public class MyBinaryTreeNodesCounter
    {
        int counter;
        char firstChar;
        int charIdx;

        MyBinaryTreeNodesCounter() {
            counter = 0;
            firstChar ='\0';
            charIdx = 0;
        }

        private void SearchRecursive(MyBinaryTreeNode<Library>? node)
        {
            if(node == null)
                return;

            string? str = node.data.GetOrgName();
            if (charIdx < str.Length && str[charIdx] == firstChar)
               counter++;

            SearchRecursive(node.left);
            SearchRecursive(node.right);
        }

        public MyBinaryTreeNodesCounter(MyBinaryTreeNode<Library> ?rootNode, char
firstCharForFind, int charIndex = 0) {
            counter = 0;
            firstChar = firstCharForFind;
            charIdx = charIndex;
            SearchRecursive(rootNode.left);
            SearchRecursive(rootNode.right);
        }

        public int GetNumNodes() { return counter; }
    }
}
```

MyBinaryTreePrinter.cs

```
namespace lab12k
{
    public interface ITreeNodePrinter<_Ty>
    {
        public string? Print(MyBinaryTreeNode<_Ty>? node, int level);
    }

    public class MyBinaryTreePrinter<_Ty>
    {
        private int padWidth;
```

```
        private int maxLevel;
        private ITreeNodePrinter<_Ty> printer;

        private void PrintTreeBranch(int level, MyBinaryTreeNode<_Ty>? node)
        {
            if (null == node /*|| level > maxLevel*/)
                return;

            int spacesCount = level * padWidth;
            Console.WriteLine(String.Format("{0," + spacesCount + "}{1}", "",
printer.Print(node, level)));
            level++;
            PrintTreeBranch(level, node.left);
            PrintTreeBranch(level, node.right);
        }

        public MyBinaryTreePrinter(ITreeNodePrinter<_Ty> nodePrinter,
MyBinaryTreeNode<_Ty> ?rootnode, int numlevels, int padwith = 2)
        {
            if (rootnode == null) {
                Console.WriteLine("root node is null");
                return;
            }
            printer = nodePrinter;
            padWidth = padwith;
            maxLevel = numlevels;
            Console.WriteLine(printer.Print(rootnode, 0));
            PrintTreeBranch(1, rootnode.left);
            PrintTreeBranch(1, rootnode.right);
        }
    }
}
```

MyBSTBuilder.cs

```
namespace lab12k
{
    /* Binary Search Tree (BST) */
    public class MyBSTBuilder
    {
        public class NodeSearchContext {
            public char firstChar;
            public int  charIdx;

            public NodeSearchContext(char chr, int idx) {
                firstChar = chr;
                charIdx = idx;
            }
        }

        /* this class store all pathes to needed data elem */
        public class NodesWalkPath
        {
            public enum TREE_PATH_DIR
            {
                DIR_LEFT_NODE = 0,
                DIR_RIGHT_NODE = 1
            };

            private List<TREE_PATH_DIR>              nodesWalkDirections;
            private MyBinaryTreeNode<Library>?       endNodeRef;
            private NodeSearchContext                seatrchData;
            public  List<MyBinaryTreeNode<Library>?> ?skipRefs;

            public MyBinaryTreeNode<Library>? EndNodeRef {
                get {  return endNodeRef; }
```

```
            }

            public List<TREE_PATH_DIR> WalkDirections {
                get { return nodesWalkDirections; }
            }

            public NodesWalkPath(NodeSearchContext search) {
                nodesWalkDirections = new List<TREE_PATH_DIR>();
                seatrchData = search;
            }


            private bool NodeInSkipList(MyBinaryTreeNode<Library>? node)
            {
                for (int i = 0; i < skipRefs.Count; i++)
                    if (node == skipRefs[i])
                        return true;

                return false;
            }
            // returns true if needed element found in tree branch
            private bool SearchRecursive(MyBinaryTreeNode<Library>? node) {
                bool b_left_node_found;
                bool b_right_node_found;

                // node is not exists (end tree node)
                if (node == null) {
                    nodesWalkDirections.Clear(); //clear directions
                    return false; //break searching
                }

                string? str = node.data.GetOrgName();
                if (seatrchData.charIdx < str.Length && str[seatrchData.charIdx]
== seatrchData.firstChar) {
                    if(!NodeInSkipList(node)) {
                        endNodeRef = node; // save found node ref0
                        return true; //break searching
                    }
                }

                // recursive search in left branch and in right branch
                // if needed value found in one on the branches, return true
                b_left_node_found = SearchRecursive(node.left);
                if (b_left_node_found)
                    nodesWalkDirections.Add(TREE_PATH_DIR.DIR_LEFT_NODE); // elem
found in left node

                b_right_node_found = SearchRecursive(node.right);
                if(b_right_node_found)
                    nodesWalkDirections.Add(TREE_PATH_DIR.DIR_RIGHT_NODE); // elem
found in right node

                Debug.Assert(!(b_left_node_found && b_right_node_found),
"Impossible to find elements in both trees at once!");
                return b_left_node_found || b_right_node_found;
            }

            public bool Search(MyBinaryTreeNode<Library>? rootNode,
List<MyBinaryTreeNode<Library>?> ?skipList) {
                // recursive search in left branch and in right branch
                skipRefs = skipList;
                Debug.Assert(rootNode != null, "rootNode must be not null");
                return SearchRecursive(rootNode.left) ||
SearchRecursive(rootNode.right);
            }
        }
```

```
        private NodeSearchContext                           search;
        public readonly List<NodesWalkPath?>                walkPathes;
        public readonly List<MyBinaryTreeNode<Library>?>    searchTreeRootNodes;

        private bool SearchAllPaths(MyBinaryTreeNode<Library>? rootNode,
NodeSearchContext search)
        {
            List<MyBinaryTreeNode<Library>?> skipNodesList = new
List<MyBinaryTreeNode<Library>?>();

            // 1-st search
            // if this call walked on all tree and not found needed elem, break
searching
            NodesWalkPath firstWalk = new NodesWalkPath(search);
            if (!firstWalk.Search(rootNode, skipNodesList)) {
                //There are links in the sheet but the function returned false
(not found)
                Debug.Assert(firstWalk.EndNodeRef == null, "Ooops! Something went
wrong! Search returned false (elements not found) but firstWalk.EndNodeRef !=
null");
                return false; // Eelement with needed info not found in all tree.
Return false
            }
            walkPathes.Add(firstWalk);

            // next searches
            // Needed element found in tree and this node reference saved in the
list. It is one of the pathes :)
            // Continue searching from current node next...
            for (int i = 0; i < walkPathes.Count; i++) {
                NodesWalkPath newWalk = new NodesWalkPath(search);
                if(walkPathes[i].EndNodeRef != null) {
                    if (newWalk.Search(walkPathes[i].EndNodeRef, skipNodesList)) {
                        walkPathes.Add(newWalk);
                        Console.WriteLine("");
                        i = 0; // repeat cycle from start and break current
iteration
                        break;
                    }
                }
            }
            Console.WriteLine("In binary tree searched {0} pathes",
walkPathes.Count);
            return true;
        }

        private bool SearchPathWithIgnoreExistsRefs(MyBinaryTreeNode<Library>?
rootNode, List<MyBinaryTreeNode<Library>?> ? skipNodesList)
        {
            NodesWalkPath firstWalk = new NodesWalkPath(search);
            if (firstWalk.Search(rootNode, skipNodesList)) {
                //There are links in the sheet but the function returned false
(not found)
                Debug.Assert(firstWalk.EndNodeRef == null, "Ooops! Something went
wrong! Search returned false (elements not found) but firstWalk.EndNodeRef !=
null");
                return false; // Eelement with needed info not found in all tree.
Return false
            }
            walkPathes.Add(firstWalk);
            skipNodesList.Add(firstWalk.EndNodeRef);
            return true;
        }

        private bool SearchAllPaths2(MyBinaryTreeNode<Library>? rootNode,
NodeSearchContext search)
        {
```

```
            List<MyBinaryTreeNode<Library>?> skipNodesList = new
List<MyBinaryTreeNode<Library>?>();
            while (SearchPathWithIgnoreExistsRefs(rootNode, skipNodesList));
            Console.WriteLine("In binary tree searched {0} pathes",
walkPathes.Count);
            return true;
        }

        public void BuildSearchTrees() {
            /* search path from end node to root node and generate new search tree
*/
            for (int i = 0; i < walkPathes.Count; i++) {
                NodesWalkPath? walkPath = walkPathes[i];

                Debug.Assert(walkPath != null, "walkPath was null");
                MyBinaryTreeNode<Library>? endSrcNode = walkPath.EndNodeRef;
                Debug.Assert(endSrcNode != null, "endSrcNode was null");
                Console.WriteLine("walkPath.WalkDirections.Count = {0}",
walkPath.WalkDirections.Count);

                //TODO: generate seatch tree by direction info and root nodes info
                MyBinaryTreeNode<Library>? mySrcRoot = null;
                MyBinaryTreeNode<Library>? newNode = null;
                MyBinaryTreeNode<Library>? previousRootNode = null;

                if(endSrcNode != null) {
                    newNode = new MyBinaryTreeNode<Library>(endSrcNode.data);
                    while (endSrcNode != null) {

                        // this node is chlid?!
                        mySrcRoot = endSrcNode.root;
                        if (mySrcRoot != null) {
                            // this node is child
                            // create parent for me and set needed link to my ref
                            MyBinaryTreeNode<Library>? myNewRoot = new
MyBinaryTreeNode<Library>(mySrcRoot.data);

                            /* solve node connection */
                            if (mySrcRoot.left == endSrcNode)
                            {
                                myNewRoot.left = newNode;
                            }
                            else if (mySrcRoot.right == endSrcNode)
                            {
                                myNewRoot.right = newNode;
                            }
                            else
                            {
                                Debug.Assert(false, "Unexpected end of tree
branch");
                            }
                            newNode = myNewRoot;
                            previousRootNode = newNode;
                        }
                        else {
                            // first root node (no parents)
                            searchTreeRootNodes.Add(previousRootNode);
                            Console.WriteLine("Search tree added. Trees count
{0}", searchTreeRootNodes.Count);
                            break;
                        }
                        endSrcNode = mySrcRoot;
                    }
                }

            }
        }
```

```
        public MyBSTBuilder(MyBinaryTreeNode<Library>? rootNode, char firstSym,
int charIdx = 0) {
            search = new NodeSearchContext(firstSym, charIdx);
            walkPathes = new List<NodesWalkPath?>();
            searchTreeRootNodes = new List<MyBinaryTreeNode<Library>?>();
            if (SearchAllPaths(rootNode, search))
                //if (SearchAllPaths2(rootNode, search))
                BuildSearchTrees();
        }
    }
}
```

MyCollectionQueue.cs

```
namespace lab12k
{
    public class MyCollectionNode<_Ty>
    {
        MyCollectionNode<_Ty>? next_node;
        _Ty node_data;

        public MyCollectionNode(MyCollectionNode<_Ty>? next, _Ty data) {
            next_node = next;
            node_data = data;
        }

        public _Ty Data {
            get { return node_data; }
            set { node_data = value; }
        }

        public MyCollectionNode<_Ty>? Next {
            get { return next_node; }
            set { next_node = value; }
        }
    };

    public class MyCollectionQueue<_Ty> : IEnumerable<_Ty>
    {

        public class MyCollectionQueueEnumeraror<_Ty2> : IEnumerator<_Ty2>
        {
            private MyCollectionNode<_Ty2>? tail;

            public MyCollectionQueueEnumeraror(MyCollectionNode<_Ty2>? tailref) {
                tail = tailref;
            }

            public _Ty2 Current {
                get { return tail.Data; }
            }
            object IEnumerator.Current {
                get { return Current; }
            }
            public bool MoveNext()
            {
                if (tail != null)
                    tail = tail.Next;

                return tail != null;
            }
            public void Reset()
            {
                tail = null;
            }
            public void Dispose() { }
```

```
        };

        public class MyCollectionQueueEnumeraror2<_Ty2> : IEnumerator<_Ty2>
        {
            _Ty2 data;
            MyCollectionQueue<_Ty2> collectionQueue;

            public MyCollectionQueueEnumeraror2(MyCollectionQueue<_Ty2>?
thisQueue) {
                collectionQueue = thisQueue.Copy2();
            }

            public _Ty2 Current
            {
                get { return data; }
            }
            object IEnumerator.Current
            {
                get { return Current; }
            }
            public bool MoveNext()
            {
                if (!collectionQueue.IsEmpty()) {
                    data = collectionQueue.Front();
                    return true;
                }
                return false;
            }
            public void Reset() { }
            public void Dispose() { }
        };

        public IEnumerator<_Ty> GetEnumerator() {
            //return new MyCollectionQueueEnumeraror<_Ty>(tail);
            return new MyCollectionQueueEnumeraror2<_Ty>(this);
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }

        int count;
        MyCollectionNode<_Ty>? head;
        MyCollectionNode<_Ty>? tail;

        public int Count {
            get { return count; }
        }

        public MyCollectionQueue() {
            head = null;
            tail = null;
            count = 0;
        }

        public MyCollectionQueue(int capacity) {
            head = null;
            tail = null;
            count = capacity; // define queue size

            // if number of elements greater 0
            if(count > 0) {
                _Ty data = default(_Ty);
                MyCollectionNode<_Ty>? newNode = new MyCollectionNode<_Ty>(null,
data);
                // create new empty nodes
```

```
                for (int i = 0; i < count; i++) {
                    if (head != null)  // if head exists element
                        head.Next = newNode; // set next ref to exists element

                    head = newNode;
                }
            }
        }

        public MyCollectionQueue(MyCollectionQueue<_Ty> ?queueWithInit) {
            if(queueWithInit != null) {
                MyCollectionQueue<_Ty> copy = queueWithInit.Copy2();
                while (!copy.IsEmpty()) {
                    PushBack(copy.Front());
                }
            }
        }

        public void PushBack(_Ty data) {
            MyCollectionNode<_Ty>? newNode = null;
            newNode = new MyCollectionNode<_Ty>(null, data);
            if (head != null) // if previous node exists
                head.Next = newNode; // next node for previous - this new node

            head = newNode; //set new node to head ref
            if (tail == null)
                tail = head; // queue is empty or not initialized. Set tail to
head ref

            count++; // increment count elements in queue
        }

        public void PushBackMultiple(_Ty[] dataArray, int count) {
            if(count > 0) {
                for (int i = 0; i < count; i++) {
                    PushBack(dataArray[i]);
                }
            }
        }

        public MyCollectionNode<_Ty>? Find(_Ty dataForFind) {
            MyCollectionNode<_Ty>? nodeRef = tail;
            if (nodeRef != null) {
                while(nodeRef != null) {
                    if(nodeRef.GetHashCode() == dataForFind.GetHashCode()) {
                        return nodeRef; // element found
                    }
                    nodeRef = nodeRef.Next;
                }
            }
            return null; // not found
        }

        public bool Remove(MyCollectionNode<_Ty>? nodeRefForDel) {
            if (nodeRefForDel == null)
                return false;

            MyCollectionNode<_Ty>? nodeRef = tail; // tail is start
            while (nodeRef != null) { // if start node is not null
                MyCollectionNode<_Ty>? nextRef = nodeRef.Next; // save ref to next
node
                if(nextRef != null) { // if ref to next node is not null
                    if (nodeRefForDel == nextRef) { // if ref to next node equals
ref node for delete
                        nodeRef.Next = nodeRefForDel.Next; // set 'next' this node
ref to 'next' node ref in deleting
                        return true;
```

```
                    }
                }
                nodeRef = nextRef;
            }
            return false;
        }

        public bool RemoveMultiple(MyCollectionNode<_Ty>?[] nodesRefForDel, int
count) {
            bool bSuccess = true; // return is OK
            for (int i = 0; i < count; i++) // for each element
                bSuccess &= Remove(nodesRefForDel[i]); // change bSuccess to false
if one of function failed

            return bSuccess; // return bSuccess
        }

        public bool IsEmpty() {
            return tail == null; // tail ref is null. queue is empty
        }

        public _Ty Front() {
            _Ty data = default(_Ty); // init new empty object instance data
            if (tail != null) { // if tail not null
                data = tail.Data; // copy data from queue node
                tail = tail.Next; // move to next ref and set tail to this ref
                count--; // element readed from queue, decrement count
            }
            return data; // return copied data
        }

        public MyCollectionQueue<_Ty> Copy() {  // DEPTH copy
            return new MyCollectionQueue<_Ty>(this);
        }

        public MyCollectionQueue<_Ty> Copy2() {
            MyCollectionQueue<_Ty> queueCopy = new MyCollectionQueue<_Ty>();
            queueCopy.head = head;
            queueCopy.tail = tail;
            queueCopy.count = count;
            return queueCopy;
        }


        // free memory
        public void Free() {
            head = null;
            tail = null;
        }
    }
}
```

MyDCLinkedList.cs

```
/* Double Connected Linked List */
namespace lab12k
{
    public class MyDCLinkedListNode<_Ty>
    {
        public MyDCLinkedListNode<_Ty>? Last;
        public MyDCLinkedListNode<_Ty>? Next;
        public _Ty Data;

        public MyDCLinkedListNode() {
            Last = null;
            Next = null;
        }
```

```
        public MyDCLinkedListNode(_Ty data) {
            Data = data;
            Last = null;
            Next = null;
        }

        public MyDCLinkedListNode(MyDCLinkedListNode<_Ty>? alast,
MyDCLinkedListNode<_Ty>? anext) {
            Last = alast;
            Next = anext;
        }

        public void InsertFirst(MyDCLinkedListNode<_Ty>? node) {
            // skip addition null ptr
            if (node == null)
                return;

            MyDCLinkedListNode<_Ty>? lastNode = Last; // save last ptr in temp
variable
            Last = node; // set last to new node
            node.Last = lastNode; // set last node to new node
            node.Next = this; // next node for this new node - this
        }

        public void InsertNext(MyDCLinkedListNode<_Ty>? node) {
            // skip addition null ptr
            if (node == null)
                return;

            MyDCLinkedListNode<_Ty>? nextNode = Next;
            Next = node; // set Next ptr to this node
            node.Next = nextNode; // set Next node ptr to new node
            node.Last = this; // set last node ptr to this node
        }

        /* default insert new node to next position */
        public MyDCLinkedListNode<_Ty>? NewNode() {
            MyDCLinkedListNode<_Ty>? newNode = new MyDCLinkedListNode<_Ty>();
            InsertNext(newNode);
            return newNode;
        }

        /* insert new node to last */
        public MyDCLinkedListNode<_Ty>? NewNode(bool bInsertFirst)
        {
            MyDCLinkedListNode<_Ty>? newNode = new MyDCLinkedListNode<_Ty>();
            InsertFirst(newNode);
            return newNode;
        }

        public void Unlink()
        {
            MyDCLinkedListNode<_Ty>? last = Last;
            MyDCLinkedListNode<_Ty>? next = Next;
            if(last != null)
                last.Next = next;
            if (next != null)
                next.Last = last;
        }
    }

    public class MyDCLinkedList<_Ty> : IEnumerable<MyDCLinkedListNode<_Ty>>
    {
        public MyDCLinkedListNode<_Ty> ?Tail; // first added element
        public MyDCLinkedListNode<_Ty> ?Head; // last added element
```

```
        public IEnumerator<MyDCLinkedListNode<_Ty>> GetEnumerator() {
            MyDCLinkedListNode<_Ty>? current = Tail;
            while (current != null) {
                yield return current;
                current = current.Next;
            }
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }

        public MyDCLinkedList() {
            Tail = null;
            Head = null;
        }

        public MyDCLinkedListNode<_Ty>? InsertFirst(_Ty data) {
            MyDCLinkedListNode < _Ty > ?newNode = new
MyDCLinkedListNode<_Ty>(data);

            /* add node to empty linked list */
            //TODO: use IsEmpty()
            if (Tail == null) {
                Tail = newNode;
                Head = newNode;
                return newNode;
            }

            /* add next node */
            Tail.InsertNext(newNode);
            Tail = newNode;
            return newNode;
        }

        public bool IsEmpty()
        {
            //Debug.Assert((Head != null && Tail == null) || (Head == null && Tail
!= null), "What happened?!"); //error state
            return Head == null && Tail == null;
        }

        public MyDCLinkedListNode<_Ty>? InsertLast(_Ty data) {
            MyDCLinkedListNode<_Ty>? newNode = new MyDCLinkedListNode<_Ty>(data);

            /* add node to empty linked list */
            //TODO: use IsEmpty()
            if (Head == null) {
                Head = newNode;
                Tail = newNode;
                return newNode;
            }

            /* add next node */
            Head.InsertNext(newNode);
            Head = newNode;
            return newNode;
        }

        public MyDCLinkedListNode<_Ty>? Find(_Ty data) {
            MyDCLinkedListNode<_Ty>? node = Tail;
            while(node != null) {
                if(node.Data != null && node.Data.Equals(data)) {
                    return node;
                }
                node = node.Next;
```

```
            }
            return null;
        }

        public bool RemoveElem(_Ty data) {
            MyDCLinkedListNode<_Ty>? node = Find(data);
            if (node == null)
                return false;

            node.Unlink();
            return true;
        }
    }
}
```

MyHashTable.cs

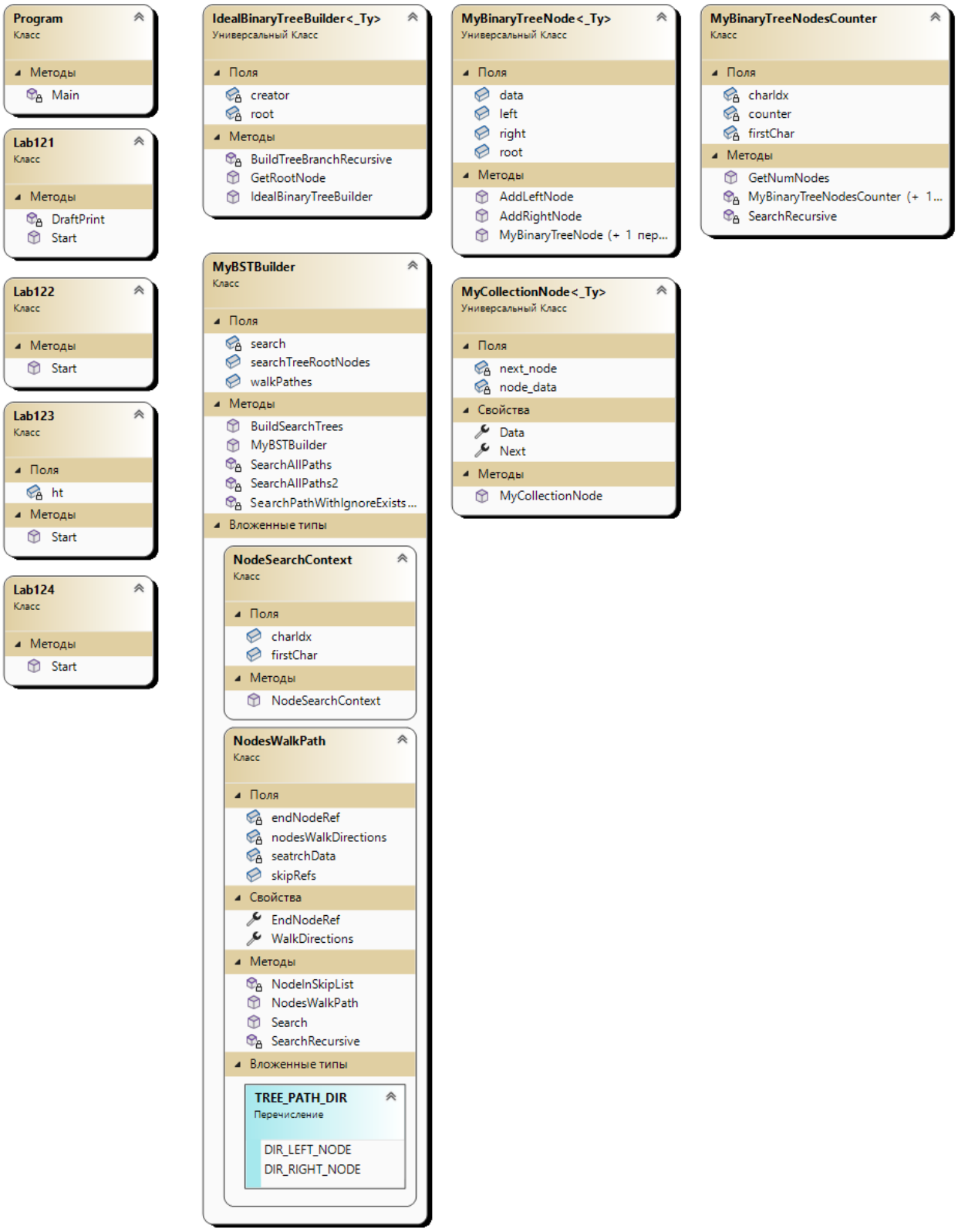
```
namespace lab12k
{
    public class MyHashTable<_TyKey, _TyVal>
    {
        int hashTableSize;
        _TyVal[] tbl;

        public int Count
        {
            get
            {
                return tbl.Length;
            }
        }

        static private int MyHashFunc(_TyKey data) {
            return data.GetHashCode();
        }

        public MyHashTable(int size) {
            hashTableSize = size;
            if (hashTableSize == 0)
                hashTableSize++;

            tbl = new _TyVal[hashTableSize];
        }

        private int GetIndexByKey(_TyKey key) {
            return Math.Abs(MyHashFunc(key)) % tbl.Length;
        }

        public int Add(_TyKey key, _TyVal data) {
            int idx = GetIndexByKey(key);
            if (tbl[idx] == null) {
                tbl[idx] = data;
                return idx;
            }
            return -1;
        }

        public void Print() {
            for (int i = 0; i < hashTableSize; i++) {
                if (tbl[i] != null) {
                    Console.WriteLine("[{0}] Obj: {1}", i, tbl[i]);
                }
            }
        }

        public bool Exists(_TyKey key) { return tbl[GetIndexByKey(key)] != null; }
        public _TyVal Find(_TyKey key) { return tbl[GetIndexByKey(key)]; }
```

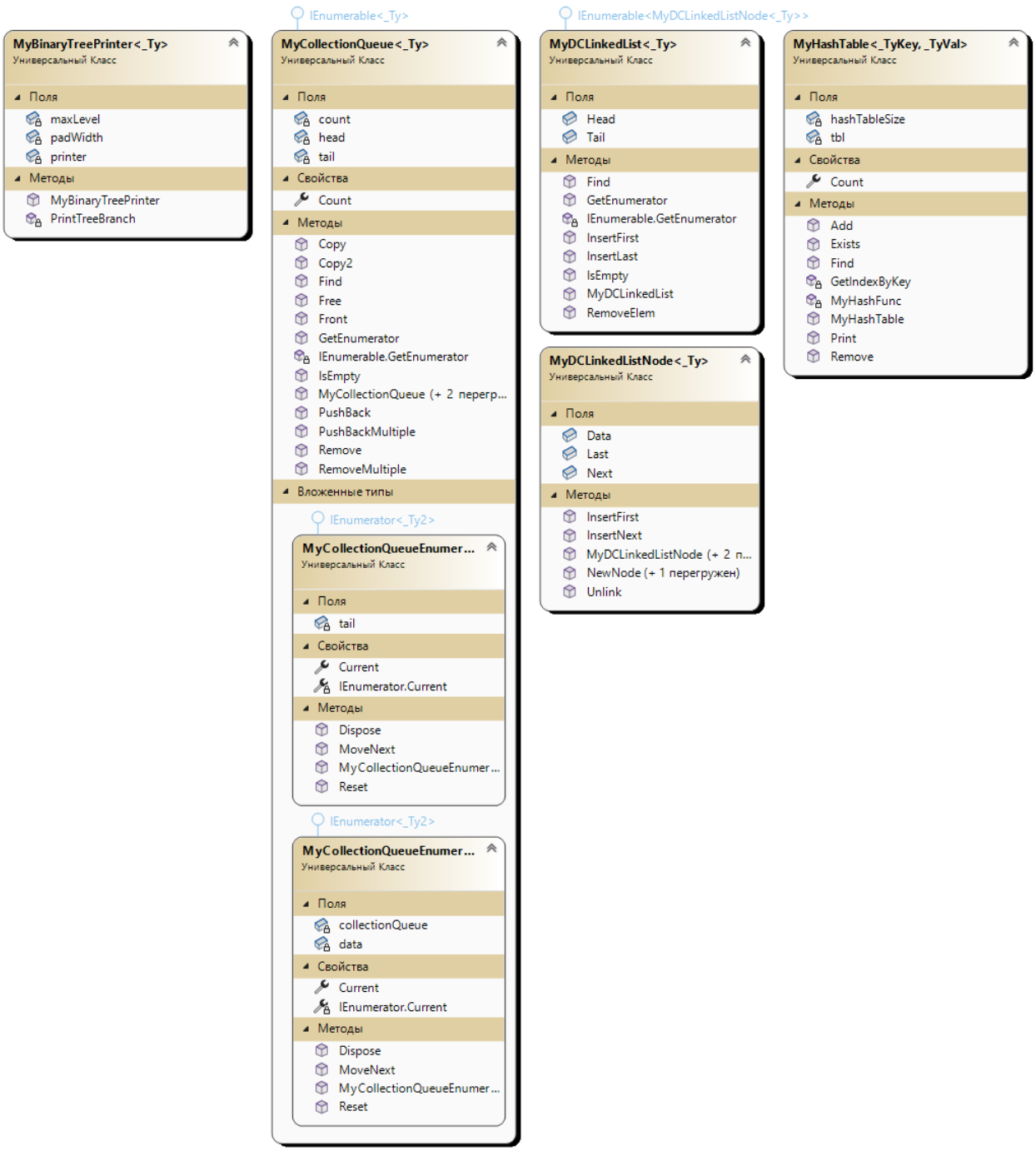
```
        public bool Remove(_TyKey key) {
            int idx = GetIndexByKey(key);
            if(tbl[idx] != null) {
                tbl[idx] = default(_TyVal);
                return true;
            }
            return false;
        }
    }
}
```

Program.cs

```
namespace lab12k
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\n\n********************** 1
**********************");
            new Lab121().Start();

            Console.WriteLine("\n\n********************** 2
**********************");
            new Lab122().Start();

            Console.WriteLine("\n\n********************** 3
**********************");
            new Lab123().Start();

            Console.WriteLine("\n\n********************** 4
**********************");
            new Lab124().Start();
        }
    }
}
```

Диаграмма классов




MyBinaryTreePrinter<_Ty>
Универсальный Класс
Поля
maxLevel
padWidth
printer
Методы
MyBinaryTreePrinter
PrintTreeBranch
IEnumerable<_Ty>
MyCollectionQueue<_Ty>
Универсальный Класс
Поля
count
head
tail
Свойства
Count
Методы
Copy
Copy2
Find
Free
Front
GetEnumerator
IEnumerable.GetEnumerator
IsEmpty
MyCollectionQueue (+ 2 перегр...
PushBack
PushBackMultiple
Remove
RemoveMultiple
Вложенные типы
IEnumerator<_Ty2>
MyCollectionQueueEnumer...
Универсальный Класс
Поля
tail
Свойства
Current
IEnumerator.Current
Методы
Dispose
MoveNext
MyCollectionQueueEnumer...
Reset
IEnumerator<_Ty2>
MyCollectionQueueEnumer...
Универсальный Класс
Поля
collectionQueue
data
Свойства
Current
IEnumerator.Current
Методы
Dispose
MoveNext
MyCollectionQueueEnumer...
Reset
IEnumerable<MyDCLinkedListNode<_Ty>>
MyDCLinkedList<_Ty>
Универсальный Класс
Поля
Head
Tail
Методы
Find
GetEnumerator
IEnumerable.GetEnumerator
InsertFirst
InsertLast
IsEmpty
MyDCLinkedList
RemoveElem
MyDCLinkedListNode<_Ty>
Универсальный Класс
Поля
Data
Last
Next
Методы
InsertFirst
InsertNext
MyDCLinkedListNode (+ 2 п...
NewNode (+ 1 перегружен)
Unlink
MyHashTable<_TyKey, _TyVal>
Универсальный Класс
Поля
hashTableSize
tbl
Свойства
Count
Методы
Add
Exists
Find
GetIndexByKey
MyHashFunc
MyHashTable
Print
Remove