Федеральное агентство связи

Государственное бюджетное образовательное учреждение высшего образование

Ордена Трудового Красного Знамени

«Московский технический университет связи и информатики»

Кафедра «МКиИТ»

дисциплина «СиАОД»

Отчет по Лабораторной работе №2

Подготовила студентка

группы БВТ1901: Нкурикийе Хафидати

Проверил: Мелехин А.

Москва 2021

Задание 1

Реализовать методы поиска в соответствии с заданием

Бинарный поиск	Бинарное дерево	Фибоначчиев	Интерполяционный

Выполнение заданий:

Задание 1

//**БИНАРНЫЙ ПОИСК O(logn)**

```
static int BinarySearch(int[] array, int searchedValue, int left, int right)
{
    Array.Sort(array);
    PrintArray(array);
    Stopwatch timer = Stopwatch.StartNew();
    //пока не сошлись границы массива
    while (left <= right)
    {
        //индекс среднего элемента
        var middle = (left + right) / 2;

        if (searchedValue == array[middle])
        {
            timer.Stop();
```

```
            return middle;
        }
        else if (searchedValue < array[middle])
        {
            //сужаем рабочую зону массива с правой стороны
            right = middle - 1;
        }
        else
        {
            //сужаем рабочую зону массива с левой стороны
            left = middle + 1;
        }
    }
    //ничего не нашли
    return -1;
}
```

Интерполяционный поиск:

```
//ИНТЕРПАЛЯЦИОННЫЙ ПОИСК O(loglogn)
    public static int InterpolationSearch(int[] array, int value)
    {
        int low = 0;
        int high = array.Length - 1;
        return InterpolationSearch(array, value, ref low, ref high);
    }
```

```
private static int InterpolationSearch(int[] array, int value, ref int low, ref int high)
{
   int index = -1;

   if (low <= high)
   {
      index = (int)(low + (((int)(high - low) / (array[high] - array[low])) * (value - array[low])));
      if (array[index] == value)
      {
         return index;
      }
      else
      {
         if (array[index] < value)
            low = index + 1;
         else
            high = index - 1;
      }

      return InterpolationSearch(array, value, ref low, ref high);
   }

   return index;
}
```

Фиббоначиев поиск:

```
private static bool FibbonachiSearch2(int[] arr, int x, int n)
    {
        int fibMMm2 = 0;
        int fibMMm1 = 1;
        int fibM = fibMMm2 + fibMMm1;



        while (fibM < n)
        {
            fibMMm2 = fibMMm1;
            fibMMm1 = fibM;
            fibM = fibMMm2 + fibMMm1;
        }


        int offset = -1;


        while (fibM > 1)
        {
            int i = Math.Min(offset + fibMMm2, n - 1);


            if (arr[i] < x)
            {
                fibM = fibMMm1;
                fibMMm1 = fibMMm2;
                fibMMm2 = fibM - fibMMm1;
                offset = i;
```

```
            }

            else if (arr[i] > x)
            {
                fibM = fibMMm2;
                fibMMm1 = fibMMm1 - fibMMm2;
                fibMMm2 = fibM - fibMMm1;
            }

            else
                return true;
        }

        if (fibMMm1 == 1 && arr[n - 1] == x)
            return true;

        return false;
    }
  }
}
```

Бинарное дерево

```
using System;

namespace Лабораторная_работа__2.Folder
{
    //представляет собой бинарное дерево
    class BinaryTree
    {
        public BinaryTreeNode RootNode { get; set; }

        // добавление узла в дерево
        public BinaryTreeNode Add(BinaryTreeNode node, BinaryTreeNode currentNode = null)
        {
            //если нет вершины то новый элемент будет вершиной
            if (RootNode == null)
            {
                node.ParentNode = null;
                return RootNode = node;
            }


            currentNode = currentNode ?? RootNode;

            node.ParentNode = currentNode;//текущий элемент станет родительским

            int result=node.Data.CompareTo(currentNode.Data);

            if (result == 0)
            {
                return currentNode;
```

```
    }
    else if( result < 0) // если новое значение меньше родительского то вставляем влево
    {
        if (currentNode.LeftNode == null)
            return currentNode.LeftNode = node;
        else return Add(node, currentNode.LeftNode);
    }
    else // если новое значение больше родительского то вставляем вправо
    {
        if (currentNode.RightNode == null)
            return currentNode.RightNode = node;
        else return Add(node, currentNode.RightNode);
    }
}

//удаление узла из дерева
public void Remove(BinaryTreeNode node)
{
    if (node == null)
    {
        return;
    }

    var currentNodeSide = node.NodeSide;
    //если у узла нет подузлов, можно его удалить
    if (node.LeftNode == null && node.RightNode == null)
    {
        if (currentNodeSide == Side.Left)
        {
```

```
            node.ParentNode.LeftNode = null;
        }
        else
        {
            node.ParentNode.RightNode = null;
        }
    }
    //если нет левого, то правый ставим на место удаляемого
    else if (node.LeftNode == null)
    {
        if (currentNodeSide == Side.Left)
        {
            node.ParentNode.LeftNode = node.RightNode;
        }
        else
        {
            node.ParentNode.RightNode = node.RightNode;
        }

        node.RightNode.ParentNode = node.ParentNode;
    }
    //если нет правого, то левый ставим на место удаляемого
    else if (node.RightNode == null)
    {
        if (currentNodeSide == Side.Left)
        {
            node.ParentNode.LeftNode = node.LeftNode;
        }
        else
```

```
        {
            node.ParentNode.RightNode = node.LeftNode;
        }

        node.LeftNode.ParentNode = node.ParentNode;
    }
    //если оба дочерних присутствуют,
    //то правый становится на место удаляемого,
    //а левый вставляется в правый
    else
    {
        switch (currentNodeSide)
        {
            case Side.Left:
                node.ParentNode.LeftNode = node.RightNode;
                node.RightNode.ParentNode = node.ParentNode;
                Add(node.LeftNode, node.RightNode);
                break;
            case Side.Right:
                node.ParentNode.RightNode = node.RightNode;
                node.RightNode.ParentNode = node.ParentNode;
                Add(node.LeftNode, node.RightNode);
                break;
        }
    }
}
//поиск по бинаному дереву
public BinaryTreeNode FindNode(int data, BinaryTreeNode startWithNode = null)
{
```

```
    startWithNode = startWithNode ?? RootNode;
    int result = data.CompareTo(startWithNode.Data);
    if (result == 0) {
        return startWithNode;
    }
    else if (result < 0) {
        if (startWithNode.LeftNode == null)
            return null;
        else return FindNode(data, startWithNode.LeftNode);
    }
    else
    {
        if(startWithNode.RightNode == null)
            return null;
        else return FindNode(data, startWithNode.RightNode);
    }
}
public void Remove(int data)
{
    var foundNode = FindNode(data);
    Remove(foundNode);
}

public void PrintTree(BinaryTreeNode startNode, string indent = "", Side? side = null)
{
    if (startNode != null)
    {
        //определяем сторону
        var nodeSide = side == null ? "+" : side == Side.Left ? "L" : "R";
```

```
            //выводим
            Console.WriteLine($"{indent} [{nodeSide}]- {startNode.Data}");
            //добавляем отступ
            indent += new string(' ', 3);
            //рекурсивный вызов для левой и правой веток
            PrintTree(startNode.LeftNode, indent, Side.Left);
            PrintTree(startNode.RightNode, indent, Side.Right);
        }
    }

    public void PrintTree()
    {
        PrintTree(RootNode);
    }


  }


}
```

//2

```
using System;

namespace Лабораторная_работа__2.Folder
{
    public enum Side
    {
        Left,
        Right
    }

    //класс представляет собой узел бинарного дерева
    class BinaryTreeNode
    {
```

```
        public BinaryTreeNode(int data)
        {
            Data = data;
        }

        public int Data { get; set; }

        public BinaryTreeNode LeftNode { get; set; }

        public BinaryTreeNode RightNode { get; set; }

        public BinaryTreeNode ParentNode { get; set; }

        public Side? NodeSide =>
            ParentNode == null
            ? (Side?)null
            : ParentNode.LeftNode == this
                ? Side.Left
                : Side.Right;


        public override string ToString() => Data.ToString();
    }
}
```

Задание 2

Простое рехэширование	Рехэширование с помощью псевдослучайных чисел	Метод цепочек

Хэш таблица с решением коллизий с помощью метода цепочек:

```
class ItemHashTable<T>
    {
        public int Key { get; set; }
```

```
        public List<T> Nodes { get; set; }

        public ItemHashTable(int key)
        {
            Key = key;
            Nodes = new List<T>();
        }
    }
}
```

```
using System;


namespace Лабораторная_работа__2.Folder
{
    class HashTable<T>
    {
        //хэш таблица в которой коллизии решаются методом цепочек

        private ItemHashTable<T>[] items;

        public HashTable(int size)
        {
            items = new ItemHashTable<T>[size];

            for (int i = 0; i < items.Length; i++)
            {
                items[i] = new ItemHashTable<T>(i);
            }
        }

        public void Add(T item)
        {
            var key = GetHash(item);
            items[key].Nodes.Add(item);
        }

        public void Delate(T item)
        {
            var key = GetHash(item);
            int index = items[key].Nodes.IndexOf(item);
            items[key].Nodes.RemoveAt(index);
        }

        public bool Search(T item)
        {
            var key = GetHash(item);
            return items[key].Nodes.Contains(item);
        }

        private int GetHash(T item)
        {
            return item.GetHashCode() % items.Length;
        }
```

```
        public void PrintHashTable()
        {
            for(int i = 0; i < items.Length; i++)
            {
                if (items[i].Nodes.Count != 0)
                {
                    Console.Write("Key : " + items[i].Key + " Values: ");
                    for (int j = 0; j < items[i].Nodes.Count; j++)
                    {
                        Console.Write(items[i].Nodes[j] + " ");
                    }
                    Console.WriteLine();
                }
            }
        }
    }
}
```

Map с решением с простым рехэшированием:

```
using System;


namespace Лабораторная_работа__2.Folder
{
    class ItemMap<TKey, TValue>
    {
        public TKey Key { get; set; }
        public TValue Value { get; set; }

        public ItemMap(TKey key, TValue value)
        {
            Key = key;
            Value = value;
        }

        public override int GetHashCode()
        {
            return Key.GetHashCode();
        }

        public override string ToString()
        {
            return "Ключ: "+ Key.ToString()+ " Значение: "+ Value.ToString();
        }
    }
}
```

//2

```
using System;
using System.Collections;
```

```
using System.Collections.Generic;

namespace Лабораторная_работа__2.Folder
{
    //Map  с простым рехэшированием
    class Map<TKey, TValue> : IEnumerable
    {
        private int size = 100;
        private ItemMap<TKey, TValue>[] Items;
        private List<TKey> Keys = new List<TKey>();

        public Map()
        {
            Items = new ItemMap<TKey, TValue>[size];
        }

        public void Add(ItemMap<TKey, TValue> item)
        {
            var hash = GetHash(item.Key);

            if (Keys.Contains(item.Key))
            {
                return;
            }

            if (Items[hash] == null)
            {
                Keys.Add(item.Key);
                Items[hash] = item;
            }
            else
            {
                var placed = false;
                for (var i = hash; i < size; i++)
                {
                    if (Items[i] == null)
                    {
                        Keys.Add(item.Key);
                        Items[i] = item;
                        placed = true;
                        break;
                    }

                    if (Items[i].Key.Equals(item.Key))
                    {
                        return;
                    }
                }

                if (!placed)
                {
                    for (var i = 0; i < hash; i++)
                    {
                        if (Items[i] == null)
                        {
                            Keys.Add(item.Key);
                            Items[i] = item;
                            placed = true;
```

```
                    break;
                }

                if (Items[i].Key.Equals(item.Key))
                {
                    return;
                }
            }
        }

        if (!placed)
        {
            throw new Exception("Словарь заполнен");
        }
    }
}

public IEnumerator GetEnumerator()
{
    foreach (var item in Items)
    {
        if (item != null)
        {
            yield return item;
        }
    }
}

public void Remove(TKey key)
{
    var hash = GetHash(key);

    if (!Keys.Contains(key))
    {
        return;
    }

    if (Items[hash] == null)
    {
        for (var i = 0; i < size; i++)
        {
            if (Items[i] != null && Items[i].Key.Equals(key))
            {
                Items[i] = null;
                Keys.Remove(key);
                return;
            }
        }

        return;
    }

    if (Items[hash].Key.Equals(key))
    {
        Items[hash] = null;
        Keys.Remove(key);
    }
    else
```

```
    {
        var rem = false;
        for (var i = hash; i < size; i++)
        {
            if (Items[i] == null)
            {
                return;
            }

            if (Items[i].Key.Equals(key))
            {
                Items[i] = null;
                Keys.Remove(key);
                return;
            }
        }

        if (!rem)
        {
            for (var i = 0; i < hash; i++)
            {
                if (Items[i] == null)
                {
                    return;
                }

                if (Items[i].Key.Equals(key))
                {
                    Items[i] = null;
                    Keys.Remove(key);
                    return;
                }
            }
        }
    }
}

public TValue Search(TKey key)
{

    var hash = GetHash(key);

    if (!Keys.Contains(key))
    {
        return default(TValue);
    }

    if (Items[hash] == null)
    {
        foreach (var item in Items)
        {
            if (item.Key.Equals(key))
            {
                return item.Value;
            }
        }

        return default(TValue);
```

```
            }

            if (Items[hash].Key.Equals(key))
            {
                return Items[hash].Value;
            }
            else
            {
                var finds = false;
                for (var i = hash; i < size; i++)
                {
                    if (Items[i] == null)
                    {
                        return default(TValue);
                    }

                    if (Items[i].Key.Equals(key))
                    {
                        return Items[i].Value;
                    }
                }

                if (!finds)
                {
                    for (var i = 0; i < hash; i++)
                    {
                        if (Items[i] == null)
                        {
                            return default(TValue);
                        }

                        if (Items[i].Key.Equals(key))
                        {
                            return Items[i].Value;
                        }
                    }
                }
            }

            return default(TValue);
        }

        private int GetHash(TKey key)
        {
            return key.GetHashCode() % size;
        }
    }
}
```

Мар с случайным рехэшированием:

```
using System;
using System.Collections;
using System.Collections.Generic;
```

```
namespace Лабораторная_работа__2.Folder
{
    //Map со случайным рехэшированием
    class Map2<TKey, TValue> : IEnumerable
    {
        private int size = 100;
        private ItemMap<TKey, TValue>[] Items;
        private List<TKey> Keys = new List<TKey>();

        public Map2()
        {
            Items = new ItemMap<TKey, TValue>[size];
        }

        public void Add(ItemMap<TKey, TValue> item)
        {
            var hash = GetHash(item.Key);

            if (Keys.Contains(item.Key))
            {
                return;
            }

            if (Items[hash] == null)
            {
                Keys.Add(item.Key);
                Items[hash] = item;
            }
            else
            {
                List<int> help = new List<int>();
                while (true)
                {
                    Random rand = new Random();
                    int NewHash = rand.Next(0, size-1);
                    if (!help.Contains(NewHash))
                    {
                        help.Add(NewHash);
                    }
                    if (Items[NewHash] == null)
                    {
                        Items[NewHash] = item;
                        Keys.Add(item.Key);
                        break;
                    }

                    if (help.Count >= size - Items.Length)
                    {
                        throw new Exception("Словарь заполнен");
                    }

                }
            }
        }

        public IEnumerator GetEnumerator()
        {
```

```
    foreach (var item in Items)
    {
        if (item != null)
        {
            yield return item;
        }
    }
}

public void Remove(TKey key)
{
    var hash = GetHash(key);

    if (!Keys.Contains(key))
    {
        return;
    }


    if (Items[hash] != null && Items[hash].Key.Equals(key))
    {
        Items[hash] = null;
        Keys.Remove(key);
    }
    else
    {
        for (var i = 0; i < size; i++)
        {
            if (Items[i] != null && Items[i].Key.Equals(key))
            {
                Items[i] = null;
                Keys.Remove(key);
                return;
            }
        }
    }
}

public TValue Search(TKey key)
{

    var hash = GetHash(key);

    if (!Keys.Contains(key))
    {
        return default(TValue);
    }


    if (Items[hash] != null && Items[hash].Key.Equals(key))
    {
        return Items[hash].Value;
    }
    else
    {
        foreach (var item in Items)
        {
            if (item.Key.Equals(key))
```

```
                    {
                        return item.Value;
                    }
                }

                return default(TValue);
            }
        }

        private int GetHash(TKey key)
        {
            return key.GetHashCode() % size;
        }
    }
}
```

Задание 3

Расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Подразумевается, что ферзь бьёт все клетки, расположенные по вертикалям, горизонталям и обеим диагоналям

Написать программу, которая находит хотя бы один способ решения задач

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Лабораторная_работа__2.Folder
{
     class Chess
     {
        private static  int Size = 8; //размер поля
        private static int[,] ChessField = new int[Size,Size]; //матрица представляющая
поле
        private static int QuantityFerz = 8; //количество нужных ферзей
```

```
        private void CreateChessField() // заполнение матрицы, где 0 - пустые клетки
        {
            for(int i = 0; i < Size; i++)
            {
                for (int j= 0; j < Size; j++)
                {
                    ChessField[i, j] = 0;
                }
            }
        }

        private bool Prov(int indexX,int indexY) // проверка, чтоб не выйти за границы
игрового поля
        {
            if (indexX > 7 || indexX < 0 || indexY > 7 || indexY < 0)
                return false;
            else return true;

        }


        //добавление нового ферзя на поле, при этом клетки по
диагоналя,вертикалям,горизонталям заполняются двойками
        //это значит, что на эти клетки нельзя постаить другого ферзя
        private void AddFerz(int indexX,int indexY)
        {
            for (int i = 0; i < Size; i++)
            {
                if (i != indexX)
                {
                    ChessField[i, indexY] = 2;
                }
            }

            for (int j = 0; j < Size; j++)
            {
                if (j != indexY)
                {
                    ChessField[indexX, j] = 2;
                }
            }

            int indexX1 = indexX;
            int indexY1 = indexY;


            for (int i = indexX1 + 1; i < Size; i++)
            {
                if (Prov(i, indexY1 + 1))
                    ChessField[i, indexY1 = indexY1 + 1] = 2;
                else break;
            }

            indexX1 = indexX;
            indexY1 = indexY;

            for (int i = indexX1-1 ; i >=0; i--)
```

```
            {
                indexY1 = indexY1 - 1;
                if (Prov(i, indexY1)){

                    ChessField[i, indexY1 ] = 2;

                }
                else break;
            }

            indexX1 = indexX;
            indexY1 = indexY;

            for (int i = indexY1 - 1; i >= 0; i--)
            {
                indexX1 = indexX1 + 1;
                if (Prov(indexX1, i))
                {

                    ChessField[indexX1, i] = 2;

                }
                else break;
            }

            indexX1 = indexX;
            indexY1 = indexY;


            for (int i = indexY1 + 1; i < Size; i++)
            {
                indexX1 = indexX1 - 1;
                if (Prov(indexX1, i))
                {
                    ChessField[indexX1, i] = 2;
                }
                else break;
            }

        }

        private void PrintMatrix()// функция для выводв игрового поля на экран
        {
            for (int i = 0; i < Size; i++)
            {
                for (int j = 0; j < Size; j++)
                {
                    Console.Write(ChessField[i, j] + " ");
                }
                Console.Write("\r\n");
            }
        }


        private bool FindPlace()//нахождение свободного места для нового ферзя(там где
нет 1 или 2, пустые места отмечены нулем)
        {
            for (int i = 0; i < Size; i++)
```

```
            {
                for (int j = 0; j < Size; j++)
                {
                   if (ChessField[i, j] == 0)
                   {
                        ChessField[i, j] = 1;
                        AddFerz(i, j);
                        return true;
                   }
                }
            }
            return false;
        }

        private void DelateTwo()//удаление всех 2 с поля перед выводом матрицы,для
наглядности
        {
            for (int i = 0; i < Size; i++)
            {
                for (int j = 0; j < Size; j++)
                {
                    if (ChessField[i, j] == 2)
                    {
                        ChessField[i, j] = 0;
                    }
                }
            }
        }

        public void Arrange()// расстановка ферзей
        {

            while (true)
            {
                CreateChessField();//создание игрового поля
                //генерируем позицию первого ферзя
                Random rand = new Random();
                int indexX = rand.Next(0, 8);
                int indexY = rand.Next(0, 8);
                //ставим его на поле как единицу
                ChessField[indexX, indexY] = 1;
                //заполняем все недступные места для будущих ферзей
                AddFerz(indexX, indexY);

                int count = 0;
                //ищем места для остальных ферзей

                for (int i = 0; i < QuantityFerz; i++)
                {
                    count++;
                    if (!FindPlace()) break;
                }
                //если все цспешно расставлены то выходим из цикла
                if (count == QuantityFerz) break;
                //если расставлены не все ферзи то делаем все заново
            }
            //когда найдена расстановка ферзей, удаляем все 2 и выводим матрицу
            DelateTwo();
```

```
            PrintMatrix();
        }
    }
}
```

Пример отображения результата:

```
Задание №1

Исходный массив: 4436 1919 4033 3049 4503 1085 2585 4292 3952 4965 1000
4.Интерпаляционный поиск
Индекс числа 4965 = 9

Исходный массив: 1000 1085 1919 2585 3049 3952 4033 4292 4436 4503 4965
1.Нахождение индекса элемента 4965 с помощью бинарного поиска: 10

2.Бинарное дерево, поиск элемента и его удаление
 [+]- 45
    [L]- 43
       [L]- 21
          [L]- 20
             [L]- 5
          [R]- 33
             [L]- 23
    [R]- 47
       [L]- 46
Добавим новый элемент
 [+]- 45
    [L]- 43
       [L]- 21
          [L]- 20
             [L]- 5
          [R]- 33
             [L]- 23
    [R]- 47
       [L]- 46
       [R]- 66
Удалим его
 [+]- 45
    [L]- 43
       [L]- 21
          [L]- 20
             [L]- 5
          [R]- 33
             [L]- 23
    [R]- 47
```

```
       [L]- 46

3.Фиббоначиев поиск

Исходный массив: 1000 1085 1919 2585 3049 3952 4033 4292 4436 4503 4965 Найдено ли число 4965 = True

Задание №2

1.Поиск по map, в которой коллизии решаются простым рехэшированием
Исходная карта:
Ключ: 1 Значение: Один
Ключ: 2 Значение: Два
Ключ: 3 Значение: Три
Ключ: 4 Значение: Четыре
Ключ: 5 Значение: Пять
Ключ: 101 Значение: Сто один

Ищем число с ключем  7 Не найдено
Ищем число с ключем 101 Сто один

Удалили 1 и 101
Ключ: 2 Значение: Два
Ключ: 3 Значение: Три
Ключ: 4 Значение: Четыре
Ключ: 5 Значение: Пять

2.Поиск по map, в которой коллизии решаются случайным рехэшированием
Исходная карта:
Ключ: 2 Значение: Два
Ключ: 3 Значение: Три
Ключ: 4 Значение: Четыре
Ключ: 5 Значение: Пять
Ключ: 102 Значение: Сто два
Ключ: 9 Значение: Девять

Ищем число с ключем  7 Не найдено
Ищем число с ключем 102 Сто два

Удалили 2 и 102
Ключ: 3 Значение: Три
```

```
Ищем число с ключем  7 Не найдено
Ищем число с ключем 102 Сто два

Удалили 2 и 102
Ключ: 3 Значение: Три
Ключ: 4 Значение: Четыре
Ключ: 5 Значение: Пять
Ключ: 9 Значение: Девять

3.Поиск по хэш таблице, в которой коллизии решаются методом цепочек
Исходная таблица:
Key : 0 Values: 65
Key : 1 Values: 51 61
Key : 2 Values: 2
Key : 3 Values: 8
Key : 4 Values: 24

Содержит ли хэш таблица число 42 False
Содержит ли хэш таблица число 8 True

Удалили 24 и 51
Key : 0 Values: 65
Key : 1 Values: 61
Key : 2 Values: 2
Key : 3 Values: 8

Задание №3
Расстановка ферзей на поле : (ферзь = 1)

0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
```