

Лабораторная работа №2  
по дисциплине «Структура и алгоритмы и обработки данных»  
на тему:  
«Методы поиска»

Выполнили: студ. гр. БСТ1902

Козлов М. С.

Вариант №7

Москва 2021

## 1. Цель работы

Реализовать методы поиска в соответствии с заданием. Организовать генерацию начального набора случайных данных. Для всех вариантов добавить реализацию добавления, поиска и удаления элементов. Оценить время работы каждого алгоритма поиска и сравнить его со временем работы стандартной функции поиска, используемой в выбранном языке программирования.

## 2. Ход выполнения лабораторной работы

### 2.1 Задание 1

Бинарный поиск	Бинарное дерево	Фибоначчиев	Интерполяционный
----------------	-----------------	-------------	------------------

Код программы:

Бинарный поиск

```
class BinarySearch<TValue> where TValue : IComparable
{
    public static int Search(TValue[] arr, TValue value)
        => Search(arr, value, 0, arr.Length - 1);

    private static int Search(TValue[] array, TValue value, int left, int
right)
    {
        if (left >= right)
            return (array[right].CompareTo(value) == 0) ? right : -1;
        var middle = (right + left) / 2;
        if (array[middle].CompareTo(value) < 0)
            return Search(array, value, middle + 1, right);
        return Search(array, value, left, middle);
    }
}
```

Фибоначчиев поиск

```
class FibonacciSearch
{
    private int item2, item1, index;
    private bool stop = false;

    private void Init(int[] array)
```

```

    {
        stop = false;
        var k = 0;
        var n = array.Length;
        while (GetNumber(k + 1) < n + 1) k++;
        var m = (int) GetNumber((k + 1) - (n + 1));
        index = (int) (GetNumber(k) - m);
        item1 = (int) GetNumber(k - 1);
        item2 = (int) GetNumber(k - 2);
    }

    private long GetNumber(int v)
    {
        long firstNumber = 0;
        long secondNumber = 1;

        for (int i = 0; i < v; i++)
        {
            long temp = secondNumber;
            secondNumber += firstNumber;
            firstNumber = temp;
        }
        return firstNumber;
    }

    private void UpIndex()
    {
        if (item1 == 1) stop = true;
        index += item2;
        item1 -= item2;
        item2 -= item1;
    }

    private void DownIndex()
    {
        if (item2 == 0) stop = true;
        index -= item2;
        int temp = item2;
        item2 = item1 - item2;
        item1 = temp;
    }

    public int Search(int[] array, int value)
    {
        Init(array);
        var n = array.Length;
        var resultIndex = -1;
        while (!stop)
        {
            if (index < 0) UpIndex();
            else if (index >= n) DownIndex();
            else if (array[index] == value)
            {
                resultIndex = index;
                break;
            }
            else if (value < array[index]) DownIndex();
            else if (value > array[index]) UpIndex();
        }
        return resultIndex;
    }
}

```

## Интерполяционный поиск

```
class InterpolationSearch
{
    public int Search(int[] array, int value)
        => Search(array, value, 0, array.Length - 1);

    private int Search(int[] array, int value, int left, int right)
    {
        int index = -1;

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

            return Search(array, value, left, right);
        }

        return index;
    }
}
```

## Бинарное дерево

```
public class BinaryTree<TValue> where TValue : IComparable<TValue>
{
    class Node
    {
        public TValue Value;
        public Node Left, Right;
    }

    private Node root;

    public void Add(TValue value)
    {
        if (root == null)
        {
            root = new Node { Value = value };
            return;
        }
        Add(root, value);
    }

    private void Add(Node currentNode, TValue value)
    {
        if (value.CompareTo(currentNode.Value) < 0)
        {
            if (currentNode.Left == null)
```

```

        {
            currentNode.Left = new Node() { Value = value };
        }
        else if (currentNode.Left != null) Add(currentNode.Left,
value);
    }
    else
    {
        if (currentNode.Right == null)
        {
            currentNode.Right = new Node { Value = value };
        }
        else if (currentNode.Right != null) Add(currentNode.Right,
value);
    }
}

public TValue Find(TValue data)
{
    Node current = root;

    while (current != null)
    {
        var result = current.Value.CompareTo(data);
        if (result == 0) return current.Value;
        current = (result > 0)
            ? current.Left
            : current.Right;
    }

    return default;
}

public bool Contains(TValue data)
{
    Node current = root;

    while (current != null)
    {
        var result = current.Value.CompareTo(data);
        if (result == 0) return true;
        current = (result > 0)
            ? current.Left
            : current.Right;
    }

    return false;
}

public void Write() => Write(root);

private void Write(Node current)
{
    if (current == null) return;
    Write(current.Left);
    Console.WriteLine(current.Value.ToString());
    Write(current.Right);
}

public bool Remove(TValue data)
{
    if (root == null) return false;

    Node current = root, parent = null;

```

```

int result = current.Value.CompareTo(data);
while (result != 0)
{
    if (result > 0)
    {
        parent = current;
        current = current.Left;
    }
    else if (result < 0)
    {
        parent = current;
        current = current.Right;
    }
    if (current == null)
        return false;
    else
        result = current.Value.CompareTo(data);
}
if (current.Right == null)
{
    if (parent == null)
        root = current.Left;
    else
    {
        result = parent.Value.CompareTo(current.Value);
        if (result > 0)

            parent.Left = current.Left;
        else if (result < 0)

            parent.Right = current.Left;
    }
}
else if (current.Right.Left == null)
{
    current.Right.Left = current.Left;

    if (parent == null)
        root = current.Right;
    else
    {
        result = parent.Value.CompareTo(current.Value);
        if (result > 0)
            parent.Left = current.Right;
        else if (result < 0)

            parent.Right = current.Right;
    }
}
else
{
    Node leftmost = current.Right.Left, lmParent = current.Right;
    while (leftmost.Left != null)
    {
        lmParent = leftmost;
        leftmost = leftmost.Left;
    }
    lmParent.Left = leftmost.Right;
    leftmost.Left = current.Left;
    leftmost.Right = current.Right;

    if (parent == null)

```

```

        root = leftmost;
    else
    {
        result = parent.Value.CompareTo(current.Value);
        if (result > 0)
            parent.Left = leftmost;
        else if (result < 0)
            parent.Right = leftmost;
    }
}

return true;
}
}

```

## 2.2 Задание 2

Простое рехэширование	Рехэширование с помощью псевдослучайных чисел	Метод цепочек
--------------------------	--	---------------

Код программы:

Абстрактный класс хеш таблицы

```

abstract public class Hashtable<TKey, TValue>
{
    public struct KeyValue<TKey, TValue>
    {
        public TKey Key { get; set; }
        public TValue Value { get; set; }
    }

    public readonly int Size;

    public Hashtable(int size)
    {
        Size = size;
    }

    abstract public TValue Find(TKey key);

    abstract public void Add(TKey key, TValue value);

    abstract public void Remove(TKey key);
}

```

## Простое рехэширование

```
class SimpleHashtable<TKey, TValue> : Hashtable<TKey, TValue>
{
    /* Простое рехэширование */

    protected KeyValue<TKey, TValue>[] items;

    public SimpleHashtable(int size) : base(size)
    {
        items = new KeyValue<TKey, TValue>[size];
    }

    protected virtual int GetArrayPosition(TKey key)
    {
        int position = key.GetHashCode() % Size;
        return Math.Abs(position);
    }

    public override TValue Find(TKey key)
    {
        var position = GetArrayPosition(key);
        return items[position].Value;
    }

    public override void Add(TKey key, TValue value)
    {
        var position = GetArrayPosition(key);
        if (!items[position].Key.Equals(default(TKey)))
        {
            //Console.WriteLine(items[position].Key);
            //Console.WriteLine($"Ключ: {key} хеш: {position} значение:
{value}");
            return;
            //throw new Exception("Ключ должен быть уникальным");
        }
        items[position] = new KeyValue<TKey, TValue>() { Key = key, Value
= value };
    }

    public override void Remove(TKey key)
    {
        var position = GetArrayPosition(key);
        items[position] = default(KeyValue<TKey, TValue>);
    }
}
```

## Рехэширование с помощью псевдослучайных чисел

```
class RandomHashtable<TKey, TValue> : SimpleHashtable<TKey, TValue>
{
    /* Рехэширование с помощью псевдослучайных чисел */

    public RandomHashtable(int size) : base(size) { }

    protected override int GetArrayPosition(TKey key)
    {
        int x = key.GetHashCode();
        int position = GetRandomHash(x);
        int startHash = position;
    }
}
```



```

        int counter = 0;

        while (!items[position].Key.Equals(default(TKey)))
        {
            if (startHash == position) break; // Таблица заполнена
            position = GetRandomHash(position);
            counter++;
        }

        return Math.Abs(position);
    }

    private int GetRandomHash(int x)
        => x + ((625 * x + 6571) % 31104) % Size;
}

```

## Метод цепочек

```

public class ChainHashtable<TKey, TValue> : Hashtable<TKey, TValue>
{
    private readonly LinkedList<KeyValuePair<TKey, TValue>>[] items;

    public ChainHashtable(int size) : base(size)
    {
        items = new LinkedList<KeyValuePair<TKey, TValue>>[size];
    }

    protected int GetArrayPosition(TKey key)
    {
        int position = key.GetHashCode() % Size;
        return Math.Abs(position);
    }

    public override TValue Find(TKey key)
    {
        var position = GetArrayPosition(key);
        var linkedList = GetLinkedList(position);

        return linkedList.FirstOrDefault(x => x.Key.Equals(key)).Value;
    }

    public override void Add(TKey key, TValue value)
    {
        var position = GetArrayPosition(key);
        var linkedList = GetLinkedList(position);
        var item = new KeyValuePair<TKey, TValue>() { Key = key, Value =
value };
        linkedList.AddLast(item);
    }

    public override void Remove(TKey key)
    {
        var position = GetArrayPosition(key);
        var linkedList = GetLinkedList(position);

        var foundItem = default(KeyValuePair<TKey, TValue>);
        var itemFound = false;

        foreach (var item in linkedList)

```

```

        {
            if (item.Key.Equals(key))
            {
                itemFound = true;
                foundItem = item;
            }
        }

        if (itemFound)
        {
            linkedList.Remove(foundItem);
        }
    }

    protected LinkedList<KeyValue<TKey, TValue>> GetLinkedList(int
position)
    {
        var linkedList = items[position];
        if (linkedList == null)
        {
            linkedList = new LinkedList<KeyValue<TKey, TValue>>();
            items[position] = linkedList;
        }

        return linkedList;
    }
}

```

## Время работы поиска для 1000000 элементов

```

Ищем элемент: 5377

Найден элемент: 5377
Под индексом: 538301
Встроенный поиск - время поиска: 1,8633 мс.

Найден элемент: 5377
Под индексом: 538301
Бинарный поиск - время поиска: 0,273 мс.

Найден элемент: 5377
Под индексом: 538328
Фибоначчиев поиск - время поиска: 0,6212 мс.

Найден элемент: 5377
Под индексом: 538394
Интерполяционный поиск - время поиска: 0,2948 мс.

Найден элемент: 5377
Бинарное дерево - время поиска: 0,1439 мс.

Найден элемент: 5377
Рехэширование с помощью псевдослучайных чисел - время поиска: 0,163 мс.

Найден элемент: 5377
Простое рехэширование - время поиска: 0,0017 мс.

Найден элемент: 5377
Метод цепочек - время поиска: 2,076 мс.

Конец

```

## 2.3 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.

## Код программы

```

/*
 * 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.
 */
class Task8F
{
    public static void Start() => Start(new byte[9]);

    //Перебор 8^8 вариантов
    private static void Start(byte[] p, int pos = 1)
    {
        for (byte i = 1; i <= 8; i++)
        {
            p[pos] = i;
            if (pos != 8) Start(p, pos + 1);
            else
            {
                p[0] = 0; // обнуление счетчика
                GetScore(p);
                if (p[0] == 28) WriteResult(p);
            }
        }
    }

    /// <summary>
    /// Проверка на пересечения ферзей
    /// </summary>
    /// <param name="p"></param>
    private static void GetScore(byte[] p)
    {
        bool hPoints, dPoints45, dPoints135;
        for (byte i = 1; i < 8; i++)
        {
            for (byte j = (byte)(i + 1); j <= 8; j++)
            {
                hPoints = (p[i] != p[j]); // не на одной горизонтали
                dPoints45 = (p[j] - p[i]) != (j - i); // не на одной
                dPoints135 = (p[i] + i) != (p[j] + j); // не на одной
                if (hPoints && dPoints45 && dPoints135) p[0]++; // если
            }
        }
    }
}

```

```

    /// <summary>
    /// Вывод комбинации на консоль
    /// </summary>
    /// <param name="p"></param>
    private static void WriteResult(byte[] p)
    {
        for (int j = 1; j <= 8; j++)
            Console.Write($" {p[j]} ");
        Console.WriteLine();
    }
}

```

## Вывод программы

КОНСОЛЬ ОЖИДАЕТ ПРОБЕЛЫ ИЛИ ПУНКТОВ

```

1 5 8 6 3 7 2 4
1 6 8 3 7 4 2 5
1 7 4 6 8 2 5 3
1 7 5 8 2 4 6 3
2 4 6 8 3 1 7 5
2 5 7 1 3 8 6 4
2 5 7 4 1 8 6 3

```