**Федеральное агентство связи**

**Государственное бюджетное образовательное учреждение высшего образования Ордена Трудового Красного Знамени**

**«Московский технический университет связи и информатики»**

**Лабораторная работа № 2**

«Методы поиска»

Выполнила: студентка группы БВТ1904

Хорикова Софья Гарегиновна

Проверил: Павликов Артём Евгеньевич

Москва, 2020

**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.

**Листинг программы:**

using System;

using System.Collections.Generic;

using System.Text;

using System.Diagnostics;

using static Lab\_number.Lab1;

namespace Lab\_number

{

static class Lab2

{

public static void Start\_Lab2()

{

while (true)

{

Console.WriteLine("Введите число элементов массива");

int size = int.Parse(Console.ReadLine());

Stopwatch time = new Stopwatch();

Random rnd = new Random();

HashSet<int> unic = new HashSet<int>();

while (unic.Count != size)

{

unic.Add(rnd.Next(-1000, 1000));

}

int[] array = new int[unic.Count];

unic.CopyTo(array);

int[] baza = array;

Array.Sort(baza);

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

Console.Write(" " + baza[i]);

Console.WriteLine('\n' + "Введите нужное число для поиска");

int num = int.Parse(Console.ReadLine());

time.Start();

int bin = BinarySearch(baza, num);

time.Stop();

Console.WriteLine("Бинарный. Число находится на позиции " + bin);

Console.WriteLine("Время выполнения - " + time.Elapsed);

time.Reset();

time.Start();

int fib = Fibonaccian(baza, num);

time.Stop();

Console.WriteLine('\n' + "Фибоначчиев. Число находится на позиции " + fib);

Console.WriteLine("Время выполнения - " + time.Elapsed);

time.Reset();

time.Start();

int inter = Interpolation(baza, num);

time.Stop();

Console.WriteLine('\n' + "Интерполяционный. Число находится на позиции " + inter);

Console.WriteLine("Время выполнения - " + time.Elapsed);

time.Reset();

time.Start();

int vstr = Array.IndexOf(baza, num);

time.Stop();

Console.WriteLine('\n' + "Встроенный. Число находится на позиции " + vstr);

Console.WriteLine("Время выполнения - " + time.Elapsed);

time.Reset();

BinaryTree tree = new BinaryTree(new int[] { 8, 6, 10, 9, 11, 7 });

tree.Add(6);

time.Start();

Console.WriteLine('\n' + "ДЕРЕВО" + '\n' + "Число добавлено в дерево " + tree.Contains(6));

time.Stop();

Console.WriteLine("Время выполнения на поиск - " + time.Elapsed);

time.Reset();

time.Start();

tree.Delete(6);

time.Stop();

Console.WriteLine("Число содержится в дереве после удаления " + tree.Contains(6));

Console.WriteLine("Время выполнения на удаление - " + time.Elapsed);

time.Reset();

SimpleHash hash = new SimpleHash(new int[] { 8, 6, 10, 9, 11, 7, 14, 19, 61, 24, 1 });

Console.WriteLine('\n' + "ПРОСТОЕ ХЕШИРОВАНИЕ" + '\n' + "Число находится в таблице на позиции " + hash.Search(24));

Console.WriteLine("Число находится в таблице на позиции " + hash.Search(10));

time.Start();

hash.Delete(10);

Console.WriteLine("Число находится в таблице на позиции " + hash.Search(10));

time.Stop();

Console.WriteLine("Затраченное время на удаление и поиск числа = " + time.Elapsed);

time.Reset();

RandHash randhash = new RandHash(new int[] { 8, 6, 10, 9, 11, 7, 14, 19, 61, 24, 1 });

Console.WriteLine('\n' + "РАНДОМНОЕ ХЕШИРОВАНИЕ" + '\n' + "Число находится в таблице на позиции " + randhash.Search(20));

Console.WriteLine("Число находится в таблице на позиции " + randhash.Search(10));

time.Start();

randhash.Delete(10);

Console.WriteLine("Число находится в таблице на позиции " + randhash.Search(10));

time.Stop();

Console.WriteLine("Затраченное время на удаление и поиск числа = " + time.Elapsed);

time.Reset();

ChainHash chainhash = new ChainHash(new int[] { 8, 6, 10, 9, 11, 7, 14, 19, 61, 24, 1 });

Console.WriteLine('\n' + "МЕТОД ЦЕПОЧЕК" + '\n' + "Число есть в таблице - " + chainhash.HasValue(20));

Console.WriteLine("Число есть в таблице - " + chainhash.HasValue(10));

time.Start();

chainhash.Delete(10);

Console.WriteLine("Число есть в таблице - " + chainhash.HasValue(10));

time.Stop();

Console.WriteLine("Затраченное время на удаление и поиск числа = " + time.Elapsed);

time.Reset();

Console.WriteLine("Нажмите любую кнопку для повторения");

Console.ReadLine();

}

}

//Задания 1/3

static public int BinarySearch(int[] array, int search)

{

/\*Двоичный поиск — классический алгоритм поиска элемента в отсортированном массиве,

использующий дробление массива на половины. \*/

int left = 0;

int right = array.Length - 1;

while (left <= right)

{

var middle = (left + right) / 2;

if (search == array[middle])

{

return middle;

}

else if (search < array[middle])

{

//сужаем рабочую зону массива с правой стороны

right = middle - 1;

}

else

{

//сужаем рабочую зону массива с левой стороны

left = middle + 1;

}

}

return -1;

}

public static int min(int x, int y)

{

return (x <= y) ? x : y; //если да, то x, иначе y

}

public static int Fibonaccian(int[] arr, int x)

{

/\* В этом поиске анализируются элементы, находящиеся в позициях,

\* равных числам Фибоначчи. Числа Фибоначчи получаются по следующему

\* правилу: последующее число равно сумме двух предыдущих чисел \*/

int n = arr.Length;

int fib2 = 0;

int fib1 = 1;

int fibM = fib2 + fib1; //следующее число

while (fibM < n) //пока не дошло до конца массива

{

fib2 = fib1;

fib1 = fibM;

fibM = fib2 + fib1; //делаем новые следующие числа

}

int offset = -1; //для запоминания индекса

while (fibM > 1)

{

int i = min(offset + fib2, n - 1);

if (arr[i] < x)

{

fibM = fib1;

fib1 = fib2;

fib2 = fibM - fib1;

offset = i;

}

else if (arr[i] > x)

{

fibM = fib2;

fib1 = fib1 - fib2;

fib2 = fibM - fib1;

}

else

return i;

}

if (fib1 == 1 && arr[n - 1] == x)

return n - 1;

return -1;

}

public static int Interpolation(int[] arr, int key)

{

/\*Поиск происходит подобно двоичному поиску,

\* но вместо деления области поиска на две примерно равные части,

\* интерполирующий поиск производит оценку новой области поиска по

\* расстоянию между ключом и текущим значением элемента.

\*/

int mid;

int i = 0;

int j = arr.Length - 1;

while (arr[i] < key && arr[j] > key)

{

if (arr[j] == arr[i]) //чтобы не было деления на 0

break;

mid = i + ((key - arr[i]) \* (j - i)) / (arr[j] - arr[i]);

if (arr[mid] < key)

i = mid + 1;

else if (arr[mid] > key)

j = mid - 1;

else

return i;

}

if (arr[i] == key) return i;

else if (arr[j] == key) return j;

else return -1;

}

public class BinaryTree

{

int value;

BinaryTree rightchild;

BinaryTree leftchild;

BinaryTree parent;

public BinaryTree(int[] mas)

{

value = mas[0];

rightchild = null;

leftchild = null;

parent = null;

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

{

Add(mas[i]);

}

}

public BinaryTree(int num)

{

value = num;

rightchild = null;

leftchild = null;

parent = null;

}

public void Add(int num)

{

BinaryTree temp = new BinaryTree(num);

if (temp.value > this.value && rightchild == null) //если идем вправо и никого нет, то привязываем

{

rightchild = temp;

temp.parent = this;

}

else if (temp.value > this.value && rightchild != null) //если больше и справа кто-то уже есть

{

rightchild.Add(num);

}

else if (temp.value < this.value && leftchild == null) //если идем влево и никого нет, то привязываем

{

leftchild = temp;

temp.parent = this;

}

else if (temp.value < this.value && leftchild != null) //если больше и слева кто-то уже есть

{

leftchild.Add(num);

}

}

public bool Contains(int num) //когда мы обращаемся к дереву, мы обращаемся к 1 элементу

{

if (num == value) //если равно текущему элементу

{

return true;

}

if (num > value && rightchild != null)

{

return rightchild.Contains(num);

}

if (num < value && leftchild != null)

{

return leftchild.Contains(num);

}

else return false;

}

public void Delete(int num)

{

if (Contains(num))

{

List<int> childs = new List<int>();

BinaryTree deleted = GetNum(num); //временный

if (deleted.leftchild != null)

{

deleted.leftchild.FindChilds(childs);

}

if (deleted.rightchild != null)

{

deleted.rightchild.FindChilds(childs);

}

if (deleted.parent.rightchild == deleted)

{

deleted.parent.rightchild = null;

deleted.parent = null;

}

else

{

deleted.parent.leftchild = null;

deleted.parent = null;

}

foreach (var item in childs)

{

Add(item);

}

}

}

private void FindChilds(List<int> childs)

{

childs.Add(value);

if (leftchild != null)

{

FindChilds(childs);

}

if (rightchild != null)

{

FindChilds(childs);

}

}

private BinaryTree GetNum(int num) //возвращаам сам объект

{

if (num == value) //если равно текущему элементу

{

return this;

}

if (num > value && rightchild != null)

{

return rightchild.GetNum(num);

}

if (num < value && leftchild != null)

{

return leftchild.GetNum(num);

}

else return null;

}

}

//Задания 1.2

class SimpleHash

{

int?[] table;

public SimpleHash(int[] mas)

{

table = new int?[mas.Length];

foreach (var item in mas)

{

HashFunction(item);

}

}

public void HashFunction(int value)

{

int i = 0;

while (true)

{

int seat = (Math.Abs(value) + i) % table.Length;

if (table[seat] == null)

{

table[seat] = value;

return;

}

else

{

i++;

}

}

}

public int Search(int value)

{

return Array.IndexOf(table, value);

}

public void Delete(int value)

{

int i = 0;

if (Search(value) == -1) return;

while (true)

{

int seat = (Math.Abs(value) + i) % table.Length;

if (table[seat] == value)

{

table[seat] = null;

return;

}

else

{

i++;

}

}

}

}

class RandHash

{

int?[] table;

public RandHash(int[] mas)

{

table = new int?[mas.Length];

foreach (var item in mas)

{

HashFunction(item);

}

}

public void HashFunction(int value)

{

Random i = new Random();

while (true)

{

int seat = (Math.Abs(value) + i.Next(0, table.Length)) % table.Length;

if (table[seat] == null)

{

table[seat] = value;

return;

}

}

}

public int Search(int value)

{

return Array.IndexOf(table, value);

}

public void Delete(int value)

{

Random i = new Random();

if (Search(value) == -1) return;

while (true)

{

int seat = (Math.Abs(value) + i.Next(0, table.Length)) % table.Length;

if (table[seat] == value)

{

table[seat] = null;

return;

}

}

}

}

class ChainHash

{

List<int>[] links;

public ChainHash(int[] arr)

{

links = new List<int>[arr.Length];

foreach (var item in arr)

{

AddItem(item);

}

}

public void Delete(int item)

{

if (HasValue(item))

{

var hash = Hash(item);

var index = links[hash].IndexOf(item);

links[hash].RemoveAt(index);

}

}

public bool HasValue(int item)

{

var res = links[Hash(item)]?.Exists(i => i == item);

return res ?? false;

}

void AddItem(int item)

{

var index = Hash(item);

if (links[index] == null)

{

links[index] = new List<int>();

links[index].Add(item);

}

else

{

links[index].Add(item);

}

}

int Hash(int item)

{

return Math.Abs(item % links.Length);

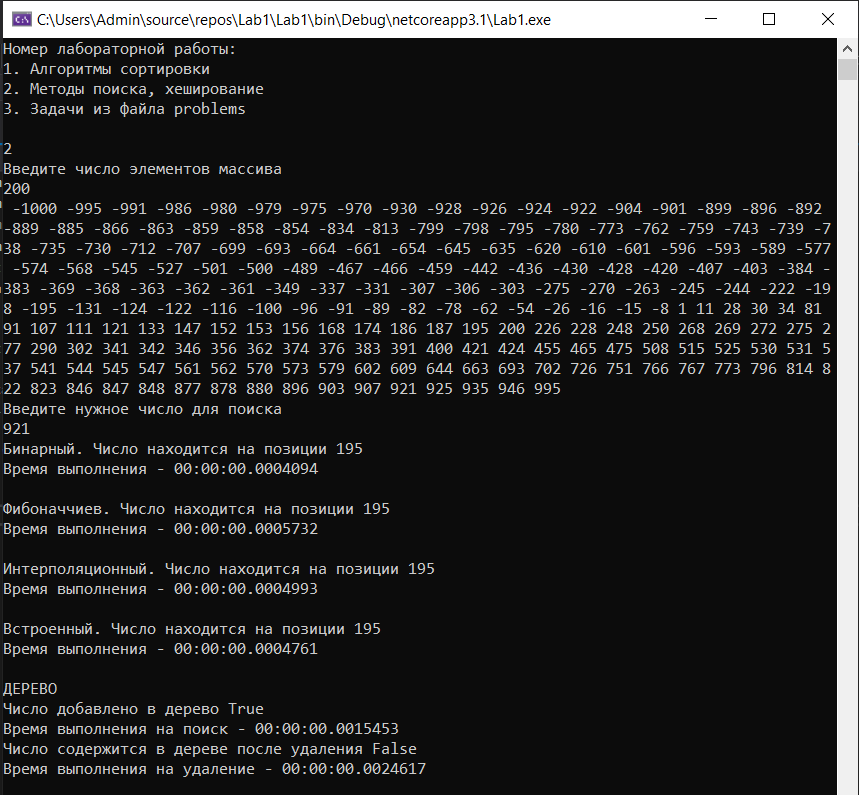
}

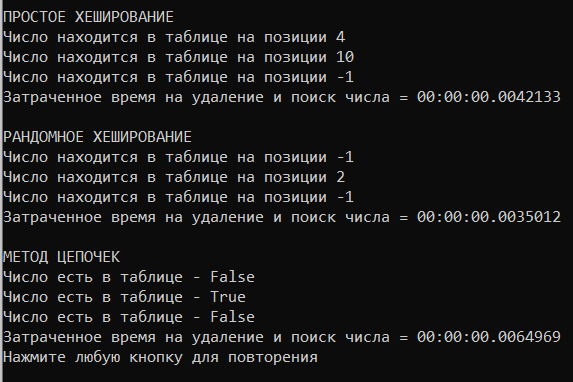
}

}

}

**Результат выполнения программы:**





**Выводы:**

Анализируя полученное время выполнения реализованных алгоритмов в сравнении со встроенными аналогами можно сделать вывод о незначительных различиях при небольших размерах входных данных и достаточно значимых различиях, когда были введены большие объемы данных.