Министерство цифрового развития, связи и массовых коммуникаций

Ордена Трудового Красного Знамени федеральное государственное бюджетное

образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра Математической кибернетики и информационных технологий

Отчёт по лабораторной работе № 2

«Методы поиска»

по дисциплине «Структуры и алгоритмы обработки данных»

Выполнил: студент группы БВТ1902

Мартынов Николай Владимирович

Москва

2021

Оглавление

[Введение 3](#_Toc72437756)

[Листинг программы 4](#_Toc72437757)

[Вывод 20](#_Toc72437758)

Введение

Цель данной лабораторной – получить знания и навыки реализовав 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.

Задание №1:

1. Бинарный поиск
2. Бинарное дерево
3. Фибоначчиев
4. Интерполяционный

Задание №2:

1. Простое рехэширование
2. Рехэширование с помощью псевдослучайных чисел
3. Метод цепочек

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.

Листинг программы

Класс Lab2

package com.company;

import java.util.\*;

import java.lang.\*;

import java.util.stream.Collectors;

public class Lab2 {

//БВТ1902 Мартынов Николай 16 вариант

public static int binSearch(int[] sortedArray, int key, int low, int high) {

int index = -1;

while (low <= high) {

int mid = (low + high) / 2;

if (sortedArray[mid] < key) {

low = mid + 1;

} else if (sortedArray[mid] > key) {

high = mid - 1;

} else if (sortedArray[mid] == key) {

index = mid;

break;

}

}

return index;

}

public static int interpolationSearch(int[] sortedArray, int toFind) {

// Возвращает индекс элемента со значением toFind или -1, если такого элемента не существует

int mid;

int low = 0;

int high = sortedArray.length - 1;

while (sortedArray[low] < toFind && sortedArray[high] > toFind) {

if (sortedArray[high] == sortedArray[low]) // Защита от деления на 0

break;

mid = low + ((toFind - sortedArray[low]) \* (high - low)) / (sortedArray[high] - sortedArray[low]);

if (sortedArray[mid] < toFind)

low = mid + 1;

else if (sortedArray[mid] > toFind)

high = mid - 1;

else

return mid;

}

if (sortedArray[low] == toFind)

return low;

if (sortedArray[high] == toFind)

return high;

return -1; // Not found

}

static int[] addElement(int[] a, int e) {

a = Arrays.copyOf(a, a.length + 1);

a[a.length - 1] = e;

return a;

}

public static int[] removeElement(int index, int[] n) {

int end = n.length;

for(int j = index; j < end - 1; j++) {

n[j] = n[j + 1];

}

end--;

int[] newArr = new int[end];

for(int k = 0; k < newArr.length; k++) {

newArr[k] = n[k];

}

return newArr;

}

public static class Map<K, V> {

class MapNode<K, V> {

K key;

V value;

MapNode<K, V> next;

public MapNode(K key, V value)

{

this.key = key;

this.value = value;

next = null;

}

}

// Массив ведра, где

// узлы, содержащие пары K-V, хранятся

ArrayList<MapNode<K, V> > buckets;

//Количество хранимых пар - n

int size;

// Размер bucketArray - b

int numBuckets;

//LoadFactor по умолчанию

final double DEFAULT\_LOAD\_FACTOR = 0.75;

public Map()

{

numBuckets = 5;

buckets = new ArrayList<>(numBuckets);

for (int i = 0; i < numBuckets; i++) {

// Инициализация до нуля

buckets.add(null);

}

System.out.println("HashMap созданный");

System.out.println("\n" + "Количество пар на Map: " + size);

System.out.println("Размер Map: " + numBuckets);

System.out.println("Коэффициент нагрузки по умолчанию : " + DEFAULT\_LOAD\_FACTOR + "\n");

}

private int getBucketInd(K key)

{

// Использование встроенной функции из объектного класса

int hashCode = key.hashCode();

// array index = hashCode%numBuckets

return (hashCode % numBuckets);

}

public void insert(K key, V value)

{

//Получение индекса, по которому его нужно вставить

int bucketInd = getBucketInd(key);

// Первый узел в этом индексе

MapNode<K, V> head = buckets.get(bucketInd);

// Сначала прокрутите все узлы, присутствующие в этом индексе

// чтобы проверить, существует ли уже ключ

/\* while (head != null) {

// Если уже присутствует, значение обновляется

if (head.key.equals(key)) {

head.value = value;

return;

}

head = head.next;

}\*/

//новый узел с K и V

MapNode<K, V> newElementNode = new MapNode<K, V>(key, value);

// главный узел в индексе

head = buckets.get(bucketInd);

// новый узел вставлен

// сделав это head

// и это следующая это предыдущая head

newElementNode.next = head;

buckets.set(bucketInd, newElementNode);

System.out.println("Пара (" + key + ", " + value + ") вставлено успешно.\n");

// Увеличение размера

// по мере добавления новой пары K-V на map

size++;

// Расчетный коэффициент нагрузки

double loadFactor = (1.0 \* size) / numBuckets;

System.out.println("Текущий коэффициент нагрузки = " + loadFactor);

// Если коэффициент нагрузки> 0,75, выполняется повторное хеширование.

if (loadFactor > DEFAULT\_LOAD\_FACTOR) {

System.out.println(loadFactor + " больше, чем " + DEFAULT\_LOAD\_FACTOR);

System.out.println("Поэтому повторное хеширование будет выполнено.\n");

// Rehash

rehash();

System.out.println("Новый размер для Map: " + numBuckets + "\n");

}

System.out.println("Количество пар в Map: " + size);

System.out.println("Размер Map: " + numBuckets + "\n");

}

private void rehash()

{

System.out.println("\nНачало рехеширования\n");

// Настоящий список ведра составлен на временной основе.

ArrayList<MapNode<K, V> > temp = buckets;

// Создается новый bucketList вдвое больше старого

buckets = new ArrayList<MapNode<K, V> >(2 \* numBuckets);

for (int i = 0; i < 2 \* numBuckets; i++) {

// Initialised to null

buckets.add(null);

}

// Теперь размер обнулен

// мы перебираем все узлы в исходном списке ведра (temp)

// и вставляем в новый список

size = 0;

numBuckets \*= 2;

for (int i = 0; i < temp.size(); i++) {

// глава цепочки по этому индексу

MapNode<K, V> head = temp.get(i);

while (head != null) {

K key = head.key;

V val = head.value;

// вызов функции вставки для каждого узла в temp

// поскольку новый список теперь bucketArray

insert(key, val);

head = head.next;

}

}

System.out.println("\nРехеширование закончилось\n");

}

public void printMap()

{

// Настоящий список ведра составлен на временной основе.

ArrayList<MapNode<K, V> > temp = buckets;

System.out.println("Построенный HashMap:");

// loop through all the nodes and print them

for (int i = 0; i < temp.size(); i++) {

// пройтись по всем узлам и распечатать их

MapNode<K, V> head = temp.get(i);

while (head != null) {

System.out.println("ключ = " + head.key + ", значение = " + head.value);

head = head.next;

}

}

System.out.println();

}

}

public static class HashNode<K, V> {

K key;

V value;

// Ссылка на следующий узел

HashNode<K, V> next;

// Конструктор

public HashNode(K key, V value)

{

this.key = key;

this.value = value;

}

}

// Класс для представления всей хеш-таблицы

public static class Map2<K, V> {

// bucketArray используется для хранения массива цепочек

private ArrayList<HashNode<K, V>> bucketArray;

// Текущая емкость списка массивов

private int numBuckets;

// Текущий размер списка массивов

private int size;

// Конструктор (инициализирует емкость, размер и

// пустые цепи.

public Map2() {

bucketArray = new ArrayList<>();

numBuckets = 10;

size = 0;

// Создание пустых цепей

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

bucketArray.add(null);

}

public int size() {

return size;

}

public boolean isEmpty() {

return size() == 0;

}

// Это реализует хеш-функцию для поиска индекса

// для ключа

private int getBucketIndex(K key) {

int hashCode = key.hashCode();

int index = hashCode % numBuckets;

// key.hashCode() может быть отрицательным.

index = index < 0 ? index \* -1 : index;

return index;

}

// Метод удаления данного ключа

public V remove(K key) {

// Применить хеш-функцию, чтобы найти индекс для данного ключа

int bucketIndex = getBucketIndex(key);

// Получить head цепи

HashNode<K, V> head = bucketArray.get(bucketIndex);

// Поиск ключа в цепочке

HashNode<K, V> prev = null;

while (head != null) {

// Если ключ найден

if (head.key.equals(key))

break;

// Иначе продолжайте двигаться по цепочке

prev = head;

head = head.next;

}

// Если бы ключа не было

if (head == null)

return null;

// Уменьшить размер

size--;

// Удалить ключ

if (prev != null)

prev.next = head.next;

else

bucketArray.set(bucketIndex, head.next);

return head.value;

}

// Возвращает значение ключа

public V get(K key) {

// Найти головку цепочки для данного ключа

int bucketIndex = getBucketIndex(key);

HashNode<K, V> head = bucketArray.get(bucketIndex);

// Поиск ключей в цепочке

while (head != null) {

if (head.key.equals(key))

return head.value;

head = head.next;

}

// Если ключ не найден

return null;

}

// Добавляет пару "ключ-значение" в хэш

public void add(K key, V value) {

// Найти head цепочки для данного ключа

int bucketIndex = getBucketIndex(key);

HashNode<K, V> head = bucketArray.get(bucketIndex);

// Проверить, присутствует ли уже ключ

while (head != null) {

if (head.key.equals(key)) {

head.value = value;

return;

}

head = head.next;

}

// Вставить ключ в цепочку

size++;

head = bucketArray.get(bucketIndex);

HashNode<K, V> newNode

= new HashNode<K, V>(key, value);

newNode.next = head;

bucketArray.set(bucketIndex, newNode);

// Если коэффициент загрузки превышает пороговое значение, то

// размер двойной хеш-таблицы

if ((1.0 \* size) / numBuckets >= 0.7) {

ArrayList<HashNode<K, V>> temp = bucketArray;

bucketArray = new ArrayList<>();

numBuckets = 2 \* numBuckets;

size = 0;

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

bucketArray.add(null);

for (HashNode<K, V> headNode : temp) {

while (headNode != null) {

add(headNode.key, headNode.value);

headNode = headNode.next;

}

}

}

}

}

public static void main (String[]args) {

Scanner scan = new Scanner(System.in);

int n, min, max;

System.out.println("Введите размерность набора данных:");

n = scan.nextInt();

System.out.println("Минимальный элемент генерации набора данных:");

min = scan.nextInt();

System.out.println("Максимальный элемент генерации набора данных:");

max = scan.nextInt();

int a[] = new int[n];

for (int i = 0; i < n; i++) {

a[i] = (int) (Math.random() \* ((max - min) + 1)) + min;

}

System.out.println("Набор данных:");

Lab\_1.heapSort(a);

Tree tree = new Tree();

for (int i = 0; i < n; i++) {

System.out.print(a[i] + " ");

tree.insertNode(a[i]);

}

int keySearch = 0;

System.out.println("\n"+"Элемент для поиска:");

keySearch = scan.nextInt();

int first = 0; //первый элемент массива

int last = a.length - 1; //последний элемент массива

long time1 = System.nanoTime();

System.out.println("Бинарный поиск нашел индекс = "+ binSearch(a,keySearch,first,last)+" time: "+(System.nanoTime()-time1) +"ns");

long time2 = System.nanoTime();

Node foundNode = tree.findNodeByValue(keySearch);

foundNode.printNode();

System.out.println(" time: "+(System.nanoTime()-time2) +"ns");

long time3 = System.nanoTime();

System.out.println("Фибоначчиев поиск нашел индекс = " + Fibonacci.fibMonaccianSearch(a,keySearch,n)+" time: "+(System.nanoTime()-time3) +"ns");

long time4 = System.nanoTime();

System.out.println("Интерполяционный поиск нашел индекс = " + interpolationSearch(a,keySearch)+" time: "+(System.nanoTime()-time4) +"ns");

long time5 = System.nanoTime();

System.out.println("Стандартный поиск = " + Arrays.stream(a).boxed().collect(Collectors.toList()).indexOf(keySearch) +" time: "+(System.nanoTime()-time5) +"ns");

System.out.println("Добавть элемент: ");

int addInt;

addInt = scan.nextInt();

a = addElement(a,addInt);

Lab\_1.heapSort(a);

for (int i = 0; i < n+1; i++) {

System.out.print(a[i] + " ");

}

System.out.println("\n"+"Элемент для поиска:");

keySearch = scan.nextInt();

System.out.println("Интерполяционный поиск нашел индекс = " + interpolationSearch(a,keySearch));

System.out.println("\n"+"Элемент для удаления:");

keySearch = scan.nextInt();

a = removeElement(interpolationSearch(a,keySearch),a);

Lab\_1.heapSort(a);

for (int i = 0; i < n; i++) {

System.out.print(a[i] + " ");

}

Hashtable<Integer, Integer> table = new Hashtable<Integer, Integer>(a.length);

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

{

table.put(i,a[i]);

}

System.out.println(table);

table.get(5);

System.out.println("\nРехеширование");

Map<Integer, Integer> map = new Map<Integer, Integer>();

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

{

map.insert(i,a[i]);

}

map.printMap();

map.rehash();

map.printMap();

System.out.println("\nМетод цепочек");

Map2<Integer, Integer> map2 = new Map2<Integer, Integer>();

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

{

map2.add(i,a[i]);

System.out.println("Ключ: " + map2.getBucketIndex(i) + " Значение: "+ map2.get(i));

}

System.out.println("Размер: "+map2.size);

System.out.println("Удалить элемент по ключу: ");

int d;

d = scan.nextInt();

map2.remove(d);

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

{

System.out.println("Ключ: " + map2.getBucketIndex(i) + " Значение: "+ map2.get(i));

}

System.out.println("Размер: "+map2.size);

System.out.println("Найти элемент по ключу: ");

int p;

p = scan.nextInt();

System.out.println("Найден элемент: " +map2.get(p));

/\*

Table tab = new Table(a.length);

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

{

tab.put(i,a[i]);

}

tab.hashCode();

tab.remove(5);

tab.get(5);

tab.showTable(); \*/

/\*for (int i=0; i<a.length; i++)

{

table.put(i,a[i]);

}

System.out.println("\n");

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

{

System.out.println(table);

}

table.remove(9);

table.put(2,10);

System.out.println("\n");

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

{

System.out.println(table);

}

System.out.println(table.get(3));\*/

}

}

Класс Fibonacci

package com.company;

public class Fibonacci {

// Сервисная функция для поиска минимума

// из двух элементов

public static int min(int x, int y) {

return (x <= y) ? x : y;

}

// Возвращает индекс x, если присутствует, иначе возвращает -1

public static int fibMonaccianSearch(int arr[], int x, int n) {

// Инициализировать числа Фибоначчи

int fibMMm2 = 0; // (м-2) -ый номер Фибоначчи

int fibMMm1 = 1; // (m-1) '-ый номер Фибоначчи

int fibM = fibMMm2 + fibMMm1; // м Фибоначчи

// fibM собирается хранить самые маленькие

// Число Фибоначчи, большее или равное n

while (fibM < n) {

fibMMm2 = fibMMm1;

fibMMm1 = fibM;

fibM = fibMMm2 + fibMMm1;

}

// Отмечает удаленный диапазон спереди

int offset = -1;

// пока есть элементы для проверки.

//Обратите внимание, что мы сравниваем arr [fibMm2] с x.

// Когда fibM становится 1, fibMm2 становится 0

while (fibM > 1) {

// Проверяем, является ли fibMm2 действительным местоположением

int i = min(offset + fibMMm2, n - 1);

//Если х больше значения в

//индекс fibMm2, вырезать массив подмассива

//от смещения до i

if (arr[i] < x) {

fibM = fibMMm1;

fibMMm1 = fibMMm2;

fibMMm2 = fibM - fibMMm1;

offset = i;

}

// Если х больше, чем значение в индексе

// fibMm2, вырезать подрешетку после i + 1

else if (arr[i] > x) {

fibM = fibMMm2;

fibMMm1 = fibMMm1 - fibMMm2;

fibMMm2 = fibM - fibMMm1;

}

//элемент найден. индекс возврата

else return i;

}

//сравнение последнего элемента с x

if (fibMMm1 == 1 && arr[offset + 1] == x)

{ return offset + 1;}

//элемент не найден. возврат -1

return -1;

}

}

Класс Node

package com.company;

public class Node {

private int value; // ключ узла

private Node leftChild; // Левый узел потомок

private Node rightChild; // Правый узел потомок

public void printNode() { // Вывод значения узла в консоль

System.out.println("Бинарное дерево имеет значение :" + value);

}

public int getValue() {

return this.value;

}

public void setValue(final int value) {

this.value = value;

}

public Node getLeftChild() {

return this.leftChild;

}

public void setLeftChild(final Node leftChild) {

this.leftChild = leftChild;

}

public Node getRightChild() {

return this.rightChild;

}

public void setRightChild(final Node rightChild) {

this.rightChild = rightChild;

}

@Override

public String toString() {

return "Node{" +

"value=" + value +

", leftChild=" + leftChild +

", rightChild=" + rightChild +

'}';

}

}

Класс Tree

package com.company;

import java.util.\*;

public class Tree {

private Node rootNode; // корневой узел

public Tree() { // Пустое дерево

rootNode = null;

}

public Node findNodeByValue(int value) { // поиск узла по значению

Node currentNode = rootNode; // начинаем поиск с корневого узла

while (currentNode.getValue() != value) { // поиск покуда не будет найден элемент или не будут перебраны все

if (value < currentNode.getValue()) { // движение влево?

currentNode = currentNode.getLeftChild();

} else { //движение вправо

currentNode = currentNode.getRightChild();

}

if (currentNode == null) { // если потомка нет,

return null; // возвращаем null

}

}

return currentNode; // возвращаем найденный элемент

}

public void insertNode(int value) { // метод вставки нового элемента

Node newNode = new Node(); // создание нового узла

newNode.setValue(value); // вставка данных

if (rootNode == null) { // если корневой узел не существует

rootNode = newNode;// то новый элемент и есть корневой узел

}

else { // корневой узел занят

Node currentNode = rootNode; // начинаем с корневого узла

Node parentNode;

while (true) // мы имеем внутренний выход из цикла

{

parentNode = currentNode;

if(value == currentNode.getValue()) { // если такой элемент в дереве уже есть, не сохраняем его

return; // просто выходим из метода

}

else if (value < currentNode.getValue()) { // движение влево?

currentNode = currentNode.getLeftChild();

if (currentNode == null){ // если был достигнут конец цепочки,

parentNode.setLeftChild(newNode); // то вставить слева и выйти из методы

return;

}

}

else { // Или направо?

currentNode = currentNode.getRightChild();

if (currentNode == null) { // если был достигнут конец цепочки,

parentNode.setRightChild(newNode); //то вставить справа

return; // и выйти

}

}

}

}

}

public boolean deleteNode(int value) // Удаление узла с заданным ключом

{

Node currentNode = rootNode;

Node parentNode = rootNode;

boolean isLeftChild = true;

while (currentNode.getValue() != value) { // начинаем поиск узла

parentNode = currentNode;

if (value < currentNode.getValue()) { // Определяем, нужно ли движение влево?

isLeftChild = true;

currentNode = currentNode.getLeftChild();

}

else { // или движение вправо?

isLeftChild = false;

currentNode = currentNode.getRightChild();

}

if (currentNode == null)

return false; // yзел не найден

}

if (currentNode.getLeftChild() == null && currentNode.getRightChild() == null) { // узел просто удаляется, если не имеет потомков

if (currentNode == rootNode) // если узел - корень, то дерево очищается

rootNode = null;

else if (isLeftChild)

parentNode.setLeftChild(null); // если нет - узел отсоединяется, от родителя

else

parentNode.setRightChild(null);

}

else if (currentNode.getRightChild() == null) { // узел заменяется левым поддеревом, если правого потомка нет

if (currentNode == rootNode)

rootNode = currentNode.getLeftChild();

else if (isLeftChild)

parentNode.setLeftChild(currentNode.getLeftChild());

else

parentNode.setRightChild(currentNode.getLeftChild());

}

else if (currentNode.getLeftChild() == null) { // узел заменяется правым поддеревом, если левого потомка нет

if (currentNode == rootNode)

rootNode = currentNode.getRightChild();

else if (isLeftChild)

parentNode.setLeftChild(currentNode.getRightChild());

else

parentNode.setRightChild(currentNode.getRightChild());

}

else { // если есть два потомка, узел заменяется преемником

Node heir = receiveHeir(currentNode);// поиск преемника для удаляемого узла

if (currentNode == rootNode)

rootNode = heir;

else if (isLeftChild)

parentNode.setLeftChild(heir);

else

parentNode.setRightChild(heir);

}

return true; // элемент успешно удалён

}

// метод возвращает узел со следующим значением после передаваемого аргументом.

// для этого он сначала переходим к правому потомку, а затем

// отслеживаем цепочку левых потомков этого узла.

private Node receiveHeir(Node node) {

Node parentNode = node;

Node heirNode = node;

Node currentNode = node.getRightChild(); // Переход к правому потомку

while (currentNode != null) // Пока остаются левые потомки

{

parentNode = heirNode;// потомка задаём как текущий узел

heirNode = currentNode;

currentNode = currentNode.getLeftChild(); // переход к левому потомку

}

// Если преемник не является

if (heirNode != node.getRightChild()) // правым потомком,

{ // создать связи между узлами

parentNode.setLeftChild(heirNode.getRightChild());

heirNode.setRightChild(node.getRightChild());

}

return heirNode;// возвращаем приемника

}

public void printTree() { // метод для вывода дерева в консоль

Stack globalStack = new Stack(); // общий стек для значений дерева

globalStack.push(rootNode);

int gaps = 32; // начальное значение расстояния между элементами

boolean isRowEmpty = false;

String separator = "-----------------------------------------------------------------";

System.out.println(separator);// черта для указания начала нового дерева

while (isRowEmpty == false) {

Stack localStack = new Stack(); // локальный стек для задания потомков элемента

isRowEmpty = true;

for (int j = 0; j < gaps; j++)

System.out.print(' ');

while (globalStack.isEmpty() == false) { // покуда в общем стеке есть элементы

Node temp = (Node) globalStack.pop(); // берем следующий, при этом удаляя его из стека

if (temp != null) {

System.out.print(temp.getValue()); // выводим его значение в консоли

localStack.push(temp.getLeftChild()); // соохраняем в локальный стек, наследники текущего элемента

localStack.push(temp.getRightChild());

if (temp.getLeftChild() != null ||

temp.getRightChild() != null)

isRowEmpty = false;

}

else {

System.out.print("\_\_");// - если элемент пустой

localStack.push(null);

localStack.push(null);

}

for (int j = 0; j < gaps \* 2 - 2; j++)

System.out.print(' ');

}

System.out.println();

gaps /= 2;// при переходе на следующий уровень расстояние между элементами каждый раз уменьшается

while (localStack.isEmpty() == false)

globalStack.push(localStack.pop()); // перемещаем все элементы из локального стека в глобальный

}

System.out.println(separator);// подводим черту

}

}

Класс chess

package com.company;

public class chess {

static int total = 0;

public static void main(String[] args) {

int n=8;

queen(n);

System.out.println("\nВсего решений: " + total);

}

static int[] recQueen(int[] p, int k) {

int n = p.length;

if (k == n) return p;

for (int j = 1; j <= n; j++) {

boolean correct = true;

for (int i = 0; i < k; i++) {

if (p[i] == j || k - i == Math.abs(j - p[i])) {

correct = false;

break;

}

}

if (correct) {

p[k] = j ;

int[] pos = recQueen(p, k+1);

if (pos != null) {

total++;

printBoard(pos);

}

}

}

return null;

}

static void queen(int n) {

recQueen(new int[n], 0);

}

static void printBoard(int[] pos) {

System.out.println("\nРешение №" + total );

for (int i = 0; i < pos.length; i++) {

int queenPos = pos[i];

for (int k = 1; k < queenPos; k++) {

System.out.print("\_ ");

}

System.out.print("♛ ");

for (int k = queenPos + 1; k <= pos.length; k++) {

System.out.print("\_ ");

}

System.out.print("\n");

}

}

}

Вывод

В результате выполненной лабораторной работы я реализовал меды поиска, добавления и удаления элементов (бинарный поиск, бинарное дерево, Фибоначчиев, интерполяционный, простое рехэширование, рехэширование с помощью псевдослучайных чисел, метод цепочек) для сгенерированного набора данных на java.