Министерство цифрового развития, связи и массовых коммуникаций Государственное образовательного учреждение высшего образования

Ордена Трудового Красного Знамени

«Московский технический университет связи и информатики»

Лабораторная работа № 2

«Методы поиска»

Выполнил студент

группы БВТ1902

Шульпина Полина

Москва

2021

# Задание

Реализовать методы поиска в соответствии с заданием. Организовать генерацию начального набора случайных данных. Для всех вариантов добавить реализацию добавления, поиска и удаления элементов.

## Задание №1:

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

## Задание №2:

|  |  |  |
| --- | --- | --- |
| Простое рехэширование | Рехэширование с помощью  псевдослучайных чисел | 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.

# Выполнение

Листинг программы:

Класс «Lab2»:

import java.util.\*;  
import java.lang.\*;  
import java.util.stream.Collectors;  
  
  
public class Lab2 {  
  
 public static void heap\_sort(int[] arr)  
 {  
 int n = arr.length;  
  
 // Построение кучи (перегруппируем массив)  
 for (int i = n / 2 - 1; i >= 0; i--)  
 *heapify*(arr, n, i);  
  
 // Один за другим извлекаем элементы из кучи  
 for (int i = n - 1; i >= 0; i--)  
 {  
 // Перемещаем текущий корень в конец  
 int temp = arr[0];  
 arr[0] = arr[i];  
 arr[i] = temp;  
  
 // Вызываем процедуру heapify на уменьшенной куче  
 *heapify*(arr, i, 0);  
 }  
 }  
  
 // Процедура для преобразования в двоичную кучу поддерева с корневым узлом i, что является  
// индексом в arr[]. n - размер кучи  
 static void heapify(int[] arr, int n, int i)  
 {  
 int largest = i; // Инициализируем наибольший элемент как корень  
 int l = 2 \* i + 1; // левый = 2\*i + 1  
 int r = 2 \* i + 2; // правый = 2\*i + 2  
  
 // Если левый дочерний элемент больше корня  
 if (l < n && arr[l] > arr[largest])  
 largest = l;  
  
 // Если правый дочерний элемент больше, чем самый большой элемент на данный момент  
 if (r < n && arr[r] > arr[largest])  
 largest = r;  
 // Если самый большой элемент не корень  
 if (largest != i)  
 {  
 int swap = arr[i];  
 arr[i] = arr[largest];  
 arr[largest] = swap;  
  
 // Рекурсивно преобразуем в двоичную кучу затронутое поддерево  
 *heapify*(arr, n, largest);  
 }  
 }  
  
 public static int binSearch(int[] sortedArray, int key, int low, int high) { // на входе массив, число, индекс к-рого ищем в массиве, лау и хай  
 int index = -1; // если вышли из цикла, не вернув индекс, то эл-та index нет в массиве  
  
 while (low <= high) { // пока лоу не станет больше, чем хай, т.е. если в массиве sortedArray нет искомого числа или регион поиска стал пустым  
 int mid = (low + high) / 2; // ищем среднее число в массиве  
 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; // возвращаем либо индекс искомого числа, либо -1, если оно не найдено  
 }  
  
  
  
 public static int interpolationSearch(int[] sortedArray, int toFind) { // принимает массив целых чисел и целое число как искомое в этом массиве  
 // Возвращает индекс элемента со значением toFind или -1, если такого элемента не существует  
 int mid;  
 int low = 0; // создаем 2 индекса  
 int high = sortedArray.length - 1;  
  
 while (sortedArray[low] < toFind && sortedArray[high] > toFind) { // пока то, что мы ищем больше, чем то, что на индексе low и меньше high  
 if (sortedArray[high] == sortedArray[low]) // если на индексе лау и хай стоит одно и то же, т.е. область поиска сузилась до одного эл-та, то заканчиваем выполнение цикла. Защита от деления на 0  
 break;  
 mid = low + ((toFind - sortedArray[low]) \* (high - low)) / (sortedArray[high] - sortedArray[low]);  
  
 if (sortedArray[mid] < toFind) // если по вычисленному индексу стоит меньше, чем то, что мы ищем, тогда смещаем левую точку  
 low = mid + 1;  
 else if (sortedArray[mid] > toFind) // если по вычисленному индексу стоит больше, чем то, что мы ищем, тогда смещаем правую точку  
 high = mid - 1;  
 else  
 return mid;  
 }  
  
 if (sortedArray[low] == toFind) // если на индексе лау стоит то, что нам нужно, возвращаем значение лоу  
 return low;  
 if (sortedArray[high] == toFind) // если на индексе хай стоит то, что нам нужно, возвращаем значение хай  
 return high;  
  
 return -1; // Not found  
 }  
  
 static int[] addElement(int[] a, int e) { // добавление эл-та в массив  
 a = Arrays.*copyOf*(a, a.length + 1); // в метод AcO передаем наш массив, из к-рого надо скопировать знач-я и длину нового массива, в к-рый мы копируем данные  
 a[a.length - 1] = e;  
 return a;  
 }  
  
  
 public static int[] removeElement(int index, int[] n) { // удаление эл-та из массива  
  
 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 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:");  
  
 *heap\_sort*(a);  
 Tree tree = new Tree();  
 for (int i = 0; i < n; i++){  
 System.*out*.println(a[i] + " ");  
 tree.insertNode(a[i]);  
 }  
 System.*out*.println("Введите эллемент для поиска: ");  
 int key = scan.nextInt();  
 int low = 0;  
 int high = a.length - 1;  
 long time1 = System.*nanoTime*();  
 System.*out*.println("Бинарный поиск. Индекс " + *binSearch*(a, key, low, high) + " time: " + (System.*nanoTime*() - time1) + "ns");  
 long time2 = System.*nanoTime*();  
 Node foundNode = tree.findNodeByValue(key);  
 foundNode.printNode();  
 System.*out*.println(" time: " + (System.*nanoTime*()-time2) + "ns");  
 long time3 = System.*nanoTime*();  
 System.*out*.println("Фибоначчиев поиск. Индекс " + Fibonacci.*fibMonaccianSearch*(a, key, n) + " time: " + (System.*nanoTime*() - time3) + "ns");  
 long time4 = System.*nanoTime*();  
 System.*out*.println("Интерполяционный поиск. Индекс "+ *interpolationSearch*(a, key) + " time: " + (System.*nanoTime*() - time4) + "ns");  
 long time5 = System.*nanoTime*();  
 System.*out*.println("Стандартный поиск. Индекс " + Arrays.*stream*(a).boxed().collect(Collectors.*toList*()).indexOf(key) + " time: " + (System.*nanoTime*() - time5));  
 }  
  
}

Класс «Fibonacci»:

public class Fibonacci {  
  
 // Сервисная функция для поиска минимума  
  
 // из двух элементов  
  
 public static int min(int x, int y) {  
 return (x <= y) ? x : y; // тернарный оператор (одно условие - два возможных результата)  
 }  
  
  
 // Возвращает индекс x, если присутствует, иначе возвращает -1  
  
 public static int fibMonaccianSearch(int arr[], int x, int n) {  
 // Инициализировать числа Фибоначчи  
 int fibMMm2 = 0; // (m-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»:

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»:

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 «ReHashMap»:

import java.util.ArrayList;  
import java.util.Scanner;  
  
class ReHashMap<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;  
 }  
 }  
  
 ArrayList<MapNode<K, V>> buckets;  
  
 // No. of pairs stored - n  
 int size;  
  
 // Size of the bucketArray - b  
 int numBuckets;  
  
 // Default loadFactor  
 final double DEFAULT\_LOAD\_FACTOR = 0.75;  
  
 public ReHashMap()  
 {  
 numBuckets = 5;  
  
 buckets = new ArrayList<>(numBuckets);  
  
 for (int i = 0; i < numBuckets; i++)  
 {  
 // Initialising to null  
 buckets.add(null);  
 }  
 System.*out*.println("Size of Map: " + numBuckets);  
 System.*out*.println("Default Load Factor : " + DEFAULT\_LOAD\_FACTOR + "\n");  
 }  
  
 private int getBucketId(K key)  
 {  
  
 int hashCode = key.hashCode();  
  
 return (hashCode % numBuckets);  
 }  
  
 public void insert(K key, V value)  
 {  
 int bucketId = getBucketId(key);  
  
 MapNode<K, V> head;  
 MapNode<K, V> newElementNode = new MapNode<K, V>(key, value);  
  
 head = buckets.get(bucketId);  
 newElementNode.next = head;  
  
 buckets.set(bucketId, newElementNode);  
  
 System.*out*.println("Pair(" + key + ", " + value + ") inserted");  
  
 size++;  
  
 double loadFactor = size\*1.0 / numBuckets;  
 System.*out*.println("Load factor = " + loadFactor);  
  
 //check for rehash  
 if (loadFactor > DEFAULT\_LOAD\_FACTOR)  
 {  
 System.*out*.println("Load factor rather then 0.75, rehashing");  
 rehash();  
  
 System.*out*.println("New Size of Map: " + numBuckets + "\n");  
 }  
 else  
 System.*out*.println("Size of Map: " + numBuckets + "\n");  
 }  
  
 private void rehash()  
 {  
 ArrayList<MapNode<K, V>> temp = buckets;  
  
 buckets = new ArrayList<MapNode<K, V>>(2 \* numBuckets);  
  
 for (int i = 0; i < 2 \* numBuckets; i++)  
 {  
 // Initialised to null  
 buckets.add(null);  
 }  
 size = 0;  
 numBuckets \*= 2;  
  
 for (int i = 0; i < temp.size(); i++)  
 {  
  
 MapNode<K, V> head = temp.get(i);  
  
 while (head != null)  
 {  
 K key = head.key;  
 V val = head.value;  
  
 insert(key, val);  
 head = head.next;  
 }  
 }  
 System.*out*.println("Rehashing Done \n");  
 }  
  
 public void printMap()  
 {  
  
 // The present bucket list is made temp  
 ArrayList<MapNode<K, V>> temp = buckets;  
  
 System.*out*.println("Current HashMap:");  
 // loop through all the nodes and print them  
 for (int i = 0; i < temp.size(); i++)  
 {  
  
 MapNode<K, V> head = temp.get(i);  
  
 while (head != null)  
 {  
 System.*out*.println("key = " + head.key + ", val = " + head.value);  
  
 head = head.next;  
 }  
 }  
 System.*out*.println();  
 }  
  
 public static void main(String[] args)  
 {  
 Scanner scanner= new Scanner(System.*in*);  
 int count=1;  
 ReHashMap<Integer, String> reHashMap = new ReHashMap<Integer, String>();  
  
 String word;  
  
 do  
 {  
 System.*out*.println("Insert object for HashMap, '0' for exit");  
 word=scanner.nextLine();  
 if (!word.equals("0"))  
 {  
 reHashMap.insert(count, word);  
 reHashMap.printMap();  
 count++;  
 }  
 } while (!word.equals("0"));  
  
 }  
}

Класс «ChainHashMap»:

import java.util.ArrayList;  
  
class HashNode<K, V> {  
 K key;  
 V value;  
  
 HashNode<K, V> next;  
  
 public HashNode(K key, V value)  
 {  
 this.key = key;  
 this.value = value;  
 }  
}  
  
class Map<K, V>  
{  
 private ArrayList<HashNode<K, V> > bucketArray;  
  
 private int numBuckets;  
  
 private int size;  
  
 public Map()  
 {  
 bucketArray = new ArrayList<>();  
 numBuckets = 10;  
 size = 0;  
  
 for (int i = 0; i < numBuckets; i++)  
 bucketArray.add(null);  
 }  
  
 public int size() { return size; }  
  
 private int getBucketIndex(K key)  
 {  
 int hashCode = key.hashCode();  
 int index = hashCode % numBuckets;  
  
 index = index < 0 ? index \* -1 : index;  
 return index;  
 }  
  
 public V remove(K key)  
 {  
 int bucketIndex = getBucketIndex(key);  
  
 HashNode<K, V> head = bucketArray.get(bucketIndex);  
  
 HashNode<K, V> prev = null;  
  
 while (head != null)  
 {  
  
 if (head.key.equals(key))  
 break;  
  
 prev = head;  
 head = head.next;  
 }  
  
 if (head == null)  
 return null;  
  
 size--;  
  
 if (prev != null)  
 prev.next = head.next;  
 else  
 bucketArray.set(bucketIndex, head.next);  
  
 return head.value;  
 }  
  
 // Adds a key value pair to hash  
 public void add(K key, V value)  
 {  
 // Find head of chain for given key  
 int bucketIndex = getBucketIndex(key);  
 HashNode<K, V> head = bucketArray.get(bucketIndex);  
  
 // Check if key is already present  
 while (head != null) {  
 if (head.key.equals(key)) {  
 head.value = value;  
 return;  
 }  
 head = head.next;  
 }  
  
 // Insert key in chain  
 size++;  
 head = bucketArray.get(bucketIndex);  
 HashNode<K, V> newNode = new HashNode<K, V>(key, value);  
 newNode.next = head;  
 bucketArray.set(bucketIndex, newNode);  
  
 // If load factor goes beyond threshold, then  
 // double hash table size  
 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;  
 }  
 }  
 }  
 }  
  
 // Driver method to test Map class  
 public static void main(String[] args)  
 {  
 Map<String, Integer> map = new Map<>();  
 map.add("this", 1);  
 map.add("coder", 2);  
 map.add("this", 4);  
 map.add("hi", 5);  
 System.*out*.println(map.size());  
 System.*out*.println(map.remove("this"));  
 System.*out*.println(map.remove("this"));  
 System.*out*.println(map.size());  
 }  
}

Класс «chess»:

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 queenPos : pos) {  
 for (int k = 1; k < queenPos; k++) {  
 System.*out*.print("\_ ");  
 }  
 System.*out*.print("Q ");  
 for (int k = queenPos + 1; k <= pos.length; k++) {  
 System.*out*.print("\_ ");  
 }  
 System.*out*.print("\n");  
 }  
 }  
}

**Снимки экрана выполнения программы**



Рис.1 Ввод размерности набора данных



Рис.2 Ввод минимального элемента генерации набора данных



Рис.3 Ввод максимального элемента генерации набора данных

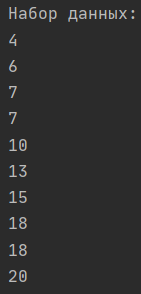


Рис.4 Генерация набора данных



Рис.5 Ввод элемента для поиска

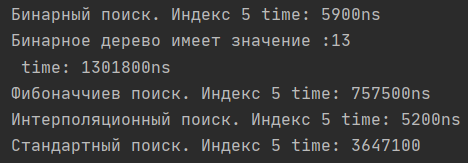


Рис.6 Реализация бинарного поиска, бинарного дерева, Фибоначчиевого поиска, Интерполяционного поиска и Стандартного поиска

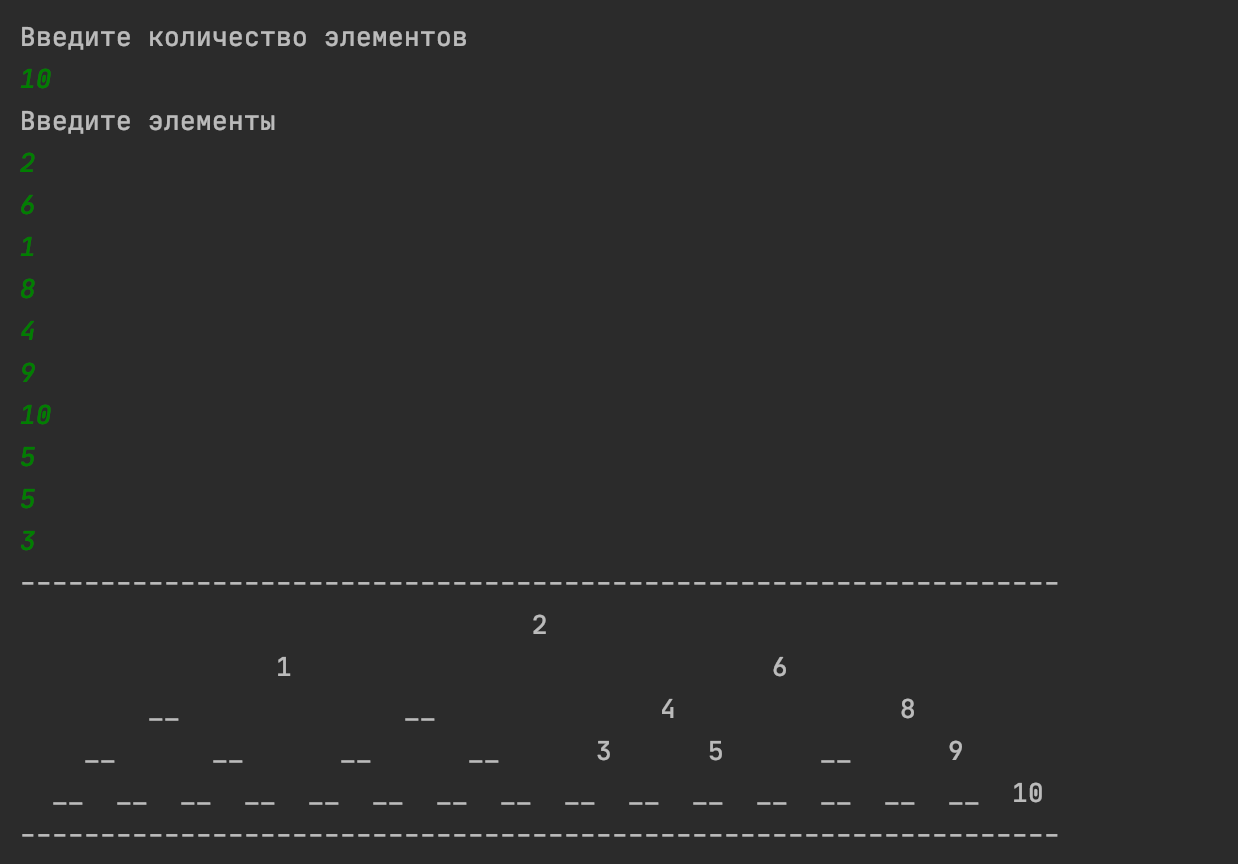


Рис. 7 Ввод и реализация бинарного дерева

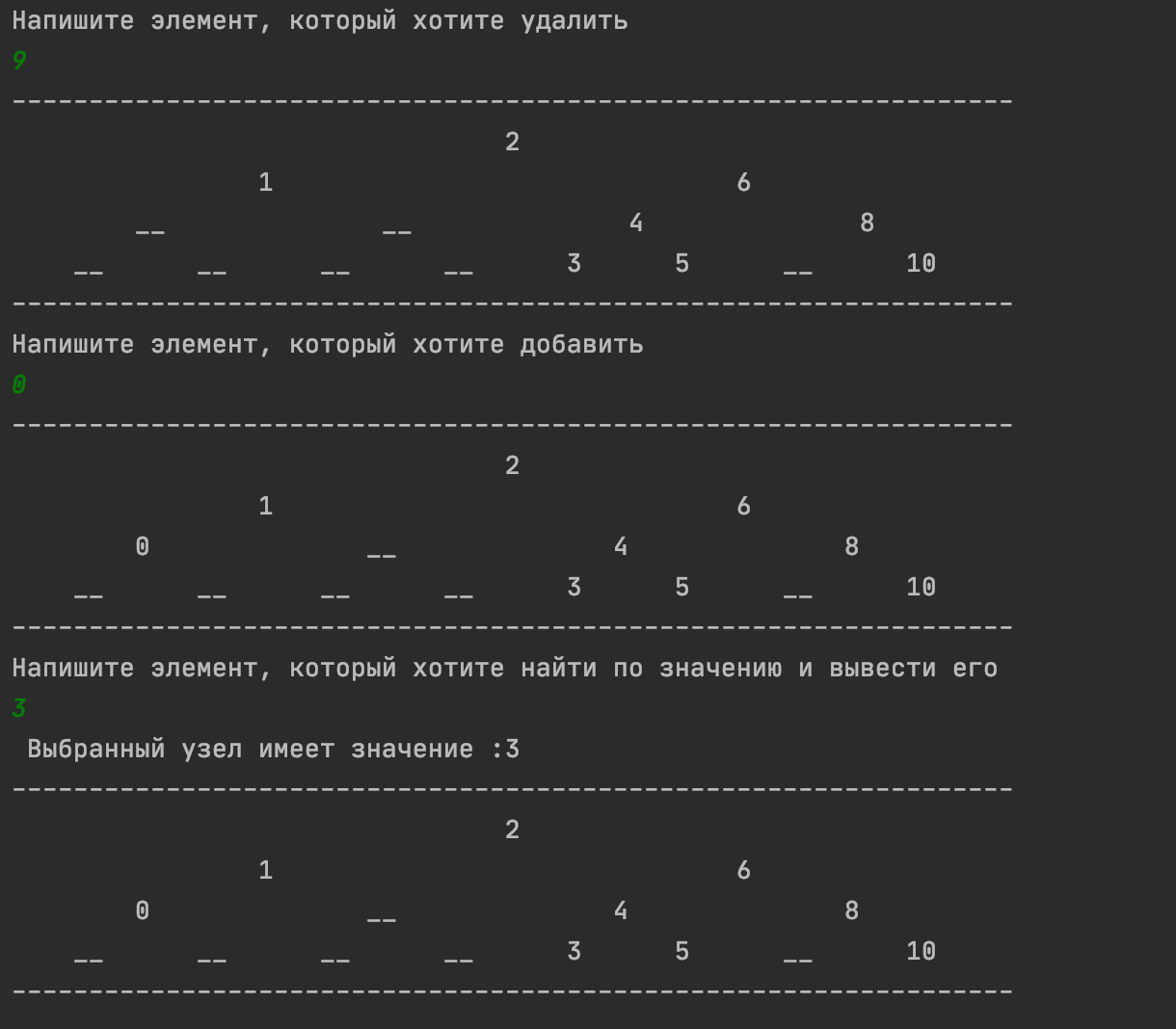


Рис. 8 Удаление, добавление и поиск элемента в бинарном дереве

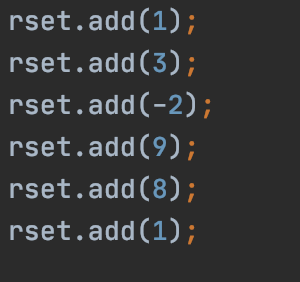


Рис. 9 Ввод данных для простого рехеширования

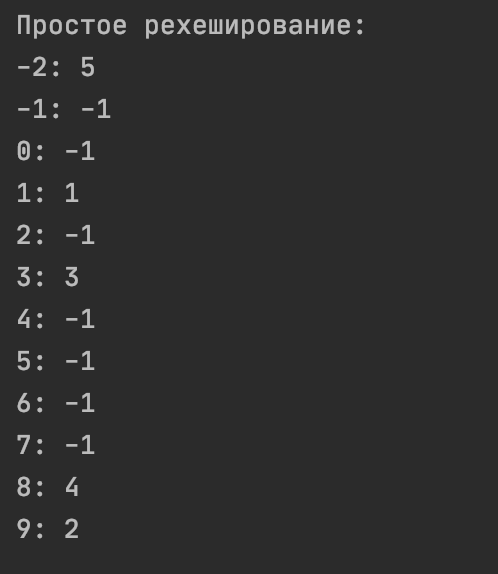


Рис. 10 Вывод данных простого рехеширования



Рис. 11 Ввод данных для рехеширования с помощью псевдослучайных чисел

# 

Рис. 12 Вывод данных рехеширования с помощью псевдослучайных чисел

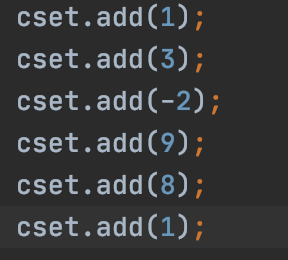


Рис. 13 Ввод данных для метода цепочек

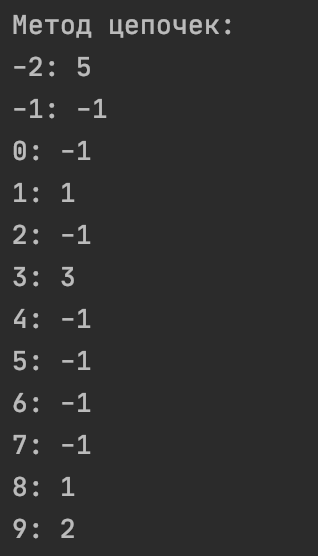


Рис. 14 Вывод данных метода цепочек

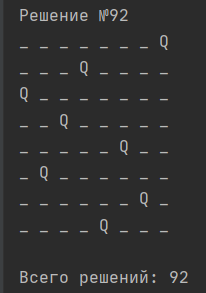


Рис. 15 Решение с шахматной доской

# Вывод

В ходе выполнения лабораторной работы, я реализовала разные методы поиска, рехэширования и решила задачу на шахматную доску.

Бинарный поиск ищет при помощи сокращения области вдвое.

Бинарное дерево представляет собой иерархическую структуру данных, в которой удобно искать, добавлять и удалять элементы.

Фибоначчиев поиск ищет с помощью золотого сечения, т.е. мы начинаем с небольшой окрестности, а затем постепенно увеличиваем шаг.

Интерполяционный поиск основывается на том, что наш массив представляет из себя что-то наподобии арифметической прогрессии.

Простое рехэширование нужно для ускоренного доступа к данным в таблицах.

Рехэширование с помощью псевдослучайных чисел позволяет быстрее заполнить таблицу числом, место которого занято.

Метод цепочек позволяет хранить все числа в удобном формате.