Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации Ордена Трудового Красного Знамени

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Отчет по лабораторной работе №2

на тему: «Методы поиска»

Выполнил: студент группы БВТ1903

Белов Сергей Павлович

Проверил:

Павликов Артём Евгеньевич

# Задание №1

Задача: Реализовать методы бинарного поиска, бинарного дерева, Фибоначчиева поиска и интерполяционного поиска. 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.

**Бинарный поиск**

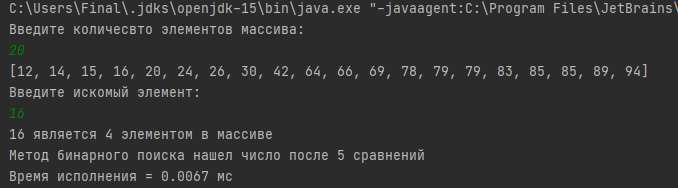
Алгоритм:

1. Определение значения элемента в середине отсортированного массива. Полученное значение сравнивается с ключом.
2. Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй.
3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
4. Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал для поиска.

Листинг Бинарного поиска

import java.util.Arrays;  
import java.util.Scanner;  
public class Binary\_Search {  
 static Scanner *scan* = new Scanner(System.*in*);  
 public static void main(String args[]) {  
 //количесвто элементов в массиве  
 System.*out*.println("Введите количесвто элементов массива:");  
 int number = *scan*.nextInt();  
 int [] array = new int [number];  
 *Random*(array);  
  
 //сортируем и выводим массив  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
  
 //вызываем поиск  
 System.*out*.println("Введите искомый элемент:");  
 int search = *scan*.nextInt();  
 long before = System.*nanoTime*();//засекаем время начала  
 int [] arr=*BinarySearch*(array,0,array.length-1,search);//int [] arr = {first,last,comparisonCount,position};  
 long after = System.*nanoTime*();//засекаем время завершения работы  
 if (arr[0] <= arr[1]) {  
 System.*out*.println(search + " является " + ++arr[3] + " элементом в массиве");  
 System.*out*.println("Метод бинарного поиска нашел число после " + arr[2] +  
 " сравнений");  
 } else {  
 System.*out*.println("Элемент не найден в массиве. Метод бинарного поиска закончил работу после "  
 + arr[2] + " сравнений");  
 }  
 System.*out*.println("Время исполнения = " + (double)(after - before)/1000000 + " мс");  
  
 System.*out*.println("Для добавления элемента в массив введите новое число:");  
 int n\_e\_w = *scan*.nextInt();  
 array=*Add*(array,n\_e\_w);  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
  
 System.*out*.println("Для удаления элемента из массива введите искомый элемент:");  
 n\_e\_w = *scan*.nextInt();  
 arr=*BinarySearch*(array,0,array.length-1,n\_e\_w);//int [] arr = {first,last,comparisonCount,position};  
 array=*Sub*(array,arr[3]);  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
 }  
 // бинарный поиск  
 public static int [] BinarySearch(int[] array, int first, int last, int item) {  
 int position;  
 int comparisonCount = 1; // для подсчета количества сравнений  
  
 // для начала найдем индекс среднего элемента массива  
 position = (first + last) / 2;  
 while ((array[position] != item) && (first <= last)) {  
 comparisonCount++;  
 if (array[position] > item) { // если число заданного для поиска  
 last = position - 1; // уменьшаем позицию на 1.  
 } else {  
 first = position + 1; // иначе увеличиваем на 1  
 }  
 position = (first + last) / 2;  
 }  
 int [] arr = {first,last,comparisonCount,position};  
 return arr;  
 }  
 public static int[] Add(int[] array, int n){  
 //получает массив и новое число  
 int[] newArray = new int[array.length+1];  
 for (int i=0;i<array.length;i++){  
 newArray[i]=array[i];  
 }  
 newArray[array.length] = n;  
 return newArray;  
 //вывод массив с числом на конце!  
 }  
 public static int[] Sub(int []array, int n){  
 //получает массив и номер числа,которое нужно удалить  
 int[] result = new int[array.length - 1]; // Array which will contain the result.  
 System.*arraycopy*(array, 0, result, 0, n);  
 System.*arraycopy*(array, n+1 , result, n, array.length - n - 1);  
 return result;  
 //выводит массив без одного числа  
 }  
 public static int[] Random (int []array){  
 //получает массив  
 for (int i=0;i< array.length;i++){  
 array[i]=(int)(Math.*random*()\*100);  
 }  
 return array;  
 //выводит массив случайных чисел  
 }  
}

Результаты работы бинарного поиска:



**Бинарное дерево**

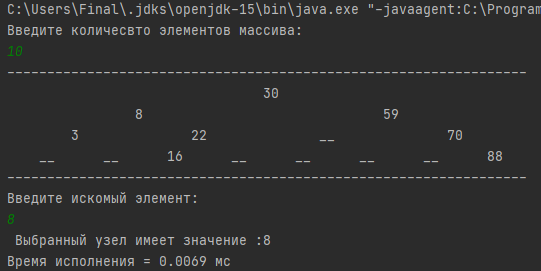
Алгоритм: Двоичное дерево состоит из узлов — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно.

Листинг бинарного дерева:

import java.util.Stack;  
import java.util.Scanner;  
  
class Binary\_tree {  
  
 public static 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(" Выбранный узел 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 узлов. Хранит и передает значения узлов  
 static class Tree {//класс со всеми методами  
 private Node rootNode; // корневой узел  
  
 public Tree() { // Пустое дерево, констурктор класса  
 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  
 }  
 }  
 }  
 }  
 }//Добавление нового элемента  
  
 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  
 }  
 //Если нет потомков  
 if (currentNode.getLeftChild() == null && currentNode.getRightChild() == null) {  
 if (currentNode == rootNode)  
 rootNode = null;//удалем узел  
 else if (isLeftChild)  
 parentNode.setLeftChild(null); //удалем узел  
 else  
 parentNode.setRightChild(null);//удалем узел  
  
 // если правого потомка нет  
 } else if (currentNode.getRightChild() == null) {  
 if (currentNode == rootNode)  
 rootNode = currentNode.getLeftChild();//узел заменяется левым поддеревом  
 else if (isLeftChild)  
 parentNode.setLeftChild(currentNode.getLeftChild());//узел заменяется левым поддеревом  
 else  
 parentNode.setRightChild(currentNode.getLeftChild());//узел заменяется левым поддеревом  
 //если левого потомка нет  
 } else if (currentNode.getLeftChild() == null) {  
 if (currentNode == rootNode)  
 rootNode = currentNode.getRightChild();// узел заменяется правым поддеревом  
 else if (isLeftChild)  
 parentNode.setLeftChild(currentNode.getRightChild());// узел заменяется правым поддеревом  
 else  
 parentNode.setRightChild(currentNode.getRightChild());// узел заменяется правым поддеревом  
 // если есть два потомка  
 } else {  
 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) {//поиск преемника  
 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 преемника  
 }  
  
 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  
 }  
 }  
 static Scanner *scan* = new Scanner(System.*in*);  
 public static void main(String args[]) {  
 Tree tree = new Tree();//создаем новое дерево  
 //количесвто элементов в массиве  
 System.*out*.println("Введите количесвто элементов массива:");  
 int number = *scan*.nextInt();  
 // вставляем узлы в дерево:  
 for (int i=0;i<number;i++){  
 tree.insertNode((int)(Math.*random*()\*100));  
 }  
 // отображение дерева:  
 tree.printTree();  
  
 //вызываем поиск  
 System.*out*.println("Введите искомый элемент:");  
 int search = *scan*.nextInt();  
 // находим узел по значению и выводим его в консоли  
 long before = System.*nanoTime*();//засекаем время начала  
 Node foundNode = tree.findNodeByValue(search);  
 long after = System.*nanoTime*();//засекаем время завершения работы  
 foundNode.printNode();  
 System.*out*.println("Время исполнения = " + (double)(after - before)/1000000 + " мс");  
  
 System.*out*.println("Для добавления элемента в массив введите новое число:");  
 int n\_e\_w = *scan*.nextInt();  
 tree.insertNode(n\_e\_w);  
  
 // отображение дерева:  
 tree.printTree();  
  
 System.*out*.println("Для удаления элемента из массива введите искомый элемент:");  
 n\_e\_w = *scan*.nextInt();  
 tree.deleteNode(n\_e\_w);  
  
 // отображение дерева:  
 tree.printTree();  
 }

}

Результаты работы бинарного дерева:



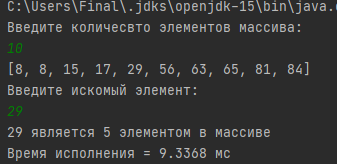
**Фибоначчиев поиск**

Алгортим: найти наименьшее число Фибоначчи, которое больше или равно длине данного массива. Пусть найденное число Фибоначчи будет F, m — число Фибоначчи. Мы используем (m-2) -ое число Фибоначчи в качестве индекса. Пусть (m-2) -ое число Фибоначчи будет i, мы сравним array [i] с X, если X одно и то же, мы вернем i. Если X больше, мы возвращаемся для подмассива после i, иначе мы возвращаемся для подмассива до i.

Листинг Фибоначчиева поиска:

import java.util.Arrays;  
import java.util.Scanner;  
  
public class Fib\_Search {  
 static Scanner *scan* = new Scanner(System.*in*);  
 /\* Возвращает индекс x, если присутствует, иначе возвращает -1 \*/  
 public static int fibSearch(int arr[], int x, int n, int counter) {  
 /\* Инициализировать числа Фибоначчи \*/  
 int fib\_n1=0;  
 int fib\_n2=1;  
 int sout = 0;  
 int fib\_i=fib\_n1+fib\_n2;  
 while (fib\_i<n){  
 if(arr[fib\_i-1]==x){  
 sout = fib\_i+counter;  
 fib\_i=n;  
 }else if(arr[fib\_i-1]<x){  
 fib\_n1=fib\_n2;  
 fib\_n2=fib\_i;  
 fib\_i=fib\_n1+fib\_n2;  
 }else if(arr[fib\_i-1]>x){  
 counter=fib\_n2;  
 int []new\_arr = new int [fib\_n1];  
 System.*arraycopy*(arr,fib\_n2,new\_arr,0,fib\_n1);  
 sout= *fibSearch*(new\_arr,x, new\_arr.length, counter);  
 fib\_i=n;  
 }else {sout=-1;}  
 }  
 return sout;  
 }  
 // код драйвера  
 public static void main(String[] args) {  
 System.*out*.println("Введите количесвто элементов массива:");  
 int number = *scan*.nextInt();  
 int [] array = new int [number];  
 *Random*(array);  
  
 //сортируем и выводим массив  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
  
 //вызываем поиск  
 System.*out*.println("Введите искомый элемент:");  
 int search = *scan*.nextInt();  
 long before = System.*nanoTime*();//засекаем время начала  
 System.*out*.println(search + " является " + (*fibSearch*(array,search,number,0)) + " элементом в массиве");  
 long after = System.*nanoTime*();//засекаем время завершения работы  
 System.*out*.println("Время исполнения = " + (double)(after - before)/1000000 + " мс");  
  
 System.*out*.println("Для добавления элемента в массив введите новое число:");  
 int n\_e\_w = *scan*.nextInt();  
 array=*Add*(array,n\_e\_w);  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
  
 System.*out*.println("Для удаления элемента из массива введите искомый элемент:");  
 n\_e\_w = *scan*.nextInt();  
 array=*Sub*(array,*fibSearch*(array,n\_e\_w,number,0));  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
 }  
 public static int[] Random (int []array){  
 //получает массив  
 for (int i=0;i< array.length;i++){  
 array[i]=(int)(Math.*random*()\*100);  
 }  
 return array;  
 //выводит массив случайных чисел  
 }  
 public static int[] Add(int[] array, int n){  
 //получает массив и новое число  
 int[] newArray = new int[array.length+1];  
 for (int i=0;i<array.length;i++){  
 newArray[i]=array[i];  
 }  
 newArray[array.length] = n;  
 return newArray;  
 //вывод массив с числом на конце!  
 }  
 public static int[] Sub(int []array, int n){  
 //получает массив и номер числа,которое нужно удалить  
 int[] result = new int[array.length - 1]; // Array which will contain the result.  
 System.*arraycopy*(array, 0, result, 0, n);  
 System.*arraycopy*(array, n+1 , result, n, array.length - n - 1);  
 return result;  
 //выводит массив без одного числа  
 }  
}

Результаты работы Фибоначчиева поиска:



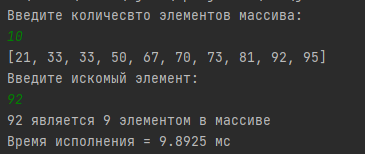
**Интерполяционный поиск**

Алгоритм: работает как бинарный поиск, но вместо того, чтобы искать середину новой области, интерполяционный «оценивает» новую область и предугадывает следующее число.

Листинг интерполяционного поиска:

import java.util.Arrays;  
import java.util.Scanner;  
  
public class Inter\_Search {  
 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  
 }  
  
 public static void main(String[] args) {  
 Scanner scan = new Scanner(System.*in*);  
 System.*out*.println("Введите количесвто элементов массива:");  
 int number = scan.nextInt();  
 int [] array = new int [number];  
 *Random*(array);  
  
 //сортируем и выводим массив  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
  
 //вызываем поиск  
 System.*out*.println("Введите искомый элемент:");  
 int search = scan.nextInt();  
 long before = System.*nanoTime*();//засекаем время начала  
 System.*out*.println(search + " является " + (*interpolationSearch*(array,search)+1) + " элементом в массиве");  
 long after = System.*nanoTime*();//засекаем время завершения работы  
 System.*out*.println("Время исполнения = " + (double)(after - before)/1000000 + " мс");  
  
 System.*out*.println("Для добавления элемента в массив введите новое число:");  
 int n\_e\_w = scan.nextInt();  
 array=*Add*(array,n\_e\_w);  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
  
 System.*out*.println("Для удаления элемента из массива введите искомый элемент:");  
 n\_e\_w = scan.nextInt();  
 array=*Sub*(array,*interpolationSearch*(array,n\_e\_w));  
 Arrays.*sort*(array);  
 System.*out*.println(Arrays.*toString*(array));  
 }  
 public static int[] Random (int []array){  
 //получает массив  
 for (int i=0;i< array.length;i++){  
 array[i]=(int)(Math.*random*()\*100);  
 }  
 return array;  
 //выводит массив случайных чисел  
 }  
 public static int[] Add(int[] array, int n){  
 //получает массив и новое число  
 int[] newArray = new int[array.length+1];  
 for (int i=0;i<array.length;i++){  
 newArray[i]=array[i];  
 }  
 newArray[array.length] = n;  
 return newArray;  
 //вывод массив с числом на конце!  
 }  
 public static int[] Sub(int []array, int n){  
 //получает массив и номер числа,которое нужно удалить  
 int[] result = new int[array.length - 1]; // Array which will contain the result.  
 System.*arraycopy*(array, 0, result, 0, n);  
 System.*arraycopy*(array, n+1 , result, n, array.length - n - 1);  
 return result;  
 //выводит массив без одного числа  
 }  
}

Результаты работы интерполяционного поиска:



# Задание №2

Задача: Реализовать простое рехэширование, рехэширование с помощью псевдослучайных чисел и метод цепочек. 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.

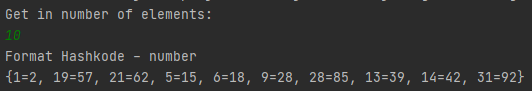
**Простое рехеширование**

Алгоритм: Если для элемента А адрес n() = h|A|, указывает на уже занятую ячейку, то есть в случае возникновения коллизии, необходимо вычислить значение функции n1 = h1|A| и проверить занятость ячейки по адресу n1. Если и она занята, то вычисляется значение h2|A|. И так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение hi|A| не совпадет с h|A|. В последнем случае считается, что таблица идентификаторов заполнена и места в ней больше нет выдается сообщение об ошибке размещения идентификатора в таблице.

Листинг простого рехэширования:

import java.util.Scanner;  
import java.util.HashMap;  
  
public class Rehash1 {  
 static int *I*=1;  
 static HashMap<Integer,Integer> *hm* = new HashMap<>();  
 public static void main (String[] args){  
 //создаем хэшкарту Ключ-значение, где ключ - это код, а значение это исходное число  
 Scanner s = new Scanner(System.*in*);  
 System.*out*.println("Get in number of elements:");  
 int n = s.nextInt();  
 int [] arr = new int [n];  
 for (int i=0;i<n;i++){  
 arr[i] = (int)(Math.*random*()\*100);  
 }  
 for (int i=0;i<n;i++){  
 *Add*(arr[i]);  
 }  
 System.*out*.println("Format Hashkode - number");  
 System.*out*.println(*hm*);  
 System.*out*.println("Get in new element:");  
 int n\_el = s.nextInt();  
 *Add*(n\_el);  
 System.*out*.println(*hm*);  
 System.*out*.println("Element for deleting?");  
 n\_el=s.nextInt();  
 *Delete*(n\_el);  
 }  
 public static int Hash(int key) {//метод хеширования  
 int ret = (key+*I*)/3;  
 return ret;  
 }  
 public static void Add(int key) {//метод добавляения нового числа  
 while (*hm*.containsKey(*Hash*(key))==true||*hm*.get(*Hash*(key))!=null){  
 *I*++;  
 }*hm*.put(*Hash*(key),key);  
 }  
 public static void Delete(int key){//метод удаления числа, но не хэша  
 if(*hm*.containsKey(*Hash*(key))==false) {  
 System.*out*.println("No Element");  
 }else{  
 *hm*.put(*Hash*(key),null);  
 System.*out*.println("Elemet is delete"+*hm*);  
 }  
 }  
}

Результаты работы простого рехэширования:



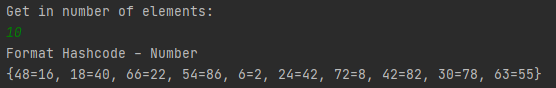
**Рехэширование с помощью псевдослучайных чисел**

Алгоритм: аналогично простому рехэшированию, но в самой формуле используется псевдослучайное число.

Листинг рехэширования:

import java.util.HashMap;  
import java.util.Scanner;  
public class Rehash2 {  
 static HashMap<Integer,Integer> *hm* = new HashMap<>();  
 static int *n*;  
 public static void main (String[] args){  
 Scanner s = new Scanner(System.*in*);  
 System.*out*.println("Get in number of elements:");  
 *n* = s.nextInt();  
 int [] arr = new int [*n*];  
 for (int i=0;i<*n*;i++){  
 arr[i] = (int)(Math.*random*()\*100);  
 }  
 for (int i=0;i<*n*;i++){  
 *Add*(arr[i]);  
 }  
 System.*out*.println("Format Hashcode - Number");  
 System.*out*.println(*hm*);  
 System.*out*.println("Get in new element:");  
 int n\_el = s.nextInt();  
 *Add*(n\_el);  
 System.*out*.println(*hm*);  
 System.*out*.println("Element for deleting?");  
 n\_el=s.nextInt();  
 *Delete*(n\_el);  
 }  
 public static int Hash(int key) {//метод хеширования  
 int ret = (key\*3)%102;  
 return ret;  
 }  
 public static void Add(int key) {//метод добавляения нового числа  
 int k= key;  
 while (*hm*.containsKey(*Hash*(key))==true||*hm*.get(*Hash*(key))!=null){  
 key=*Hash*(key);  
 }*hm*.put(*Hash*(key),k);  
 }  
 public static void Delete(int key){//метод удаления числа, но не хэша  
 if(*hm*.containsKey(*Hash*(key))==false) {  
 System.*out*.println("No Element");  
 }else{  
 *hm*.put(*Hash*(key),null);  
 System.*out*.println("Elemet is delete"+*hm*);  
 }  
 }  
}

Результаты работы хэширования:



**Метод цепочек**

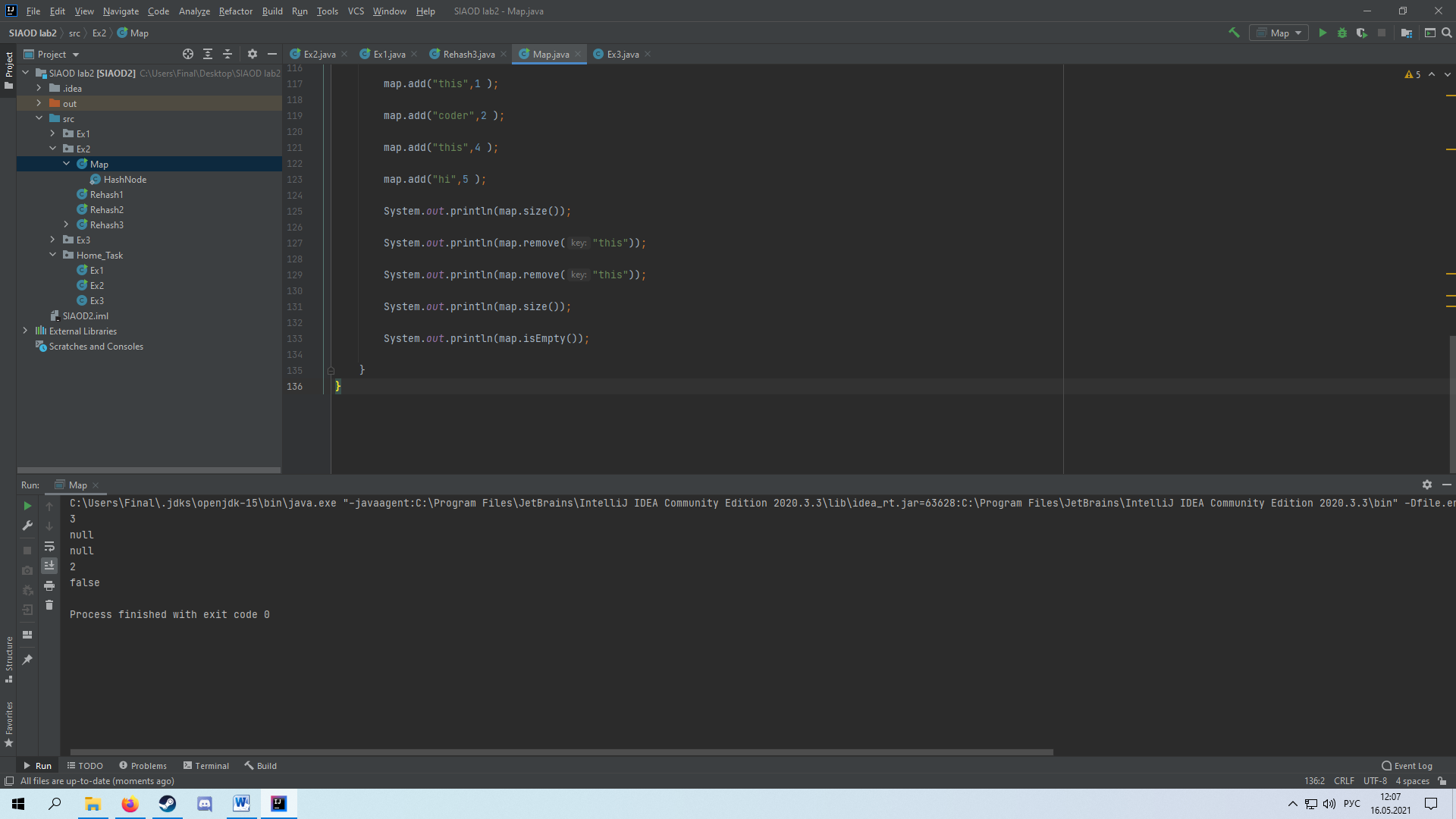
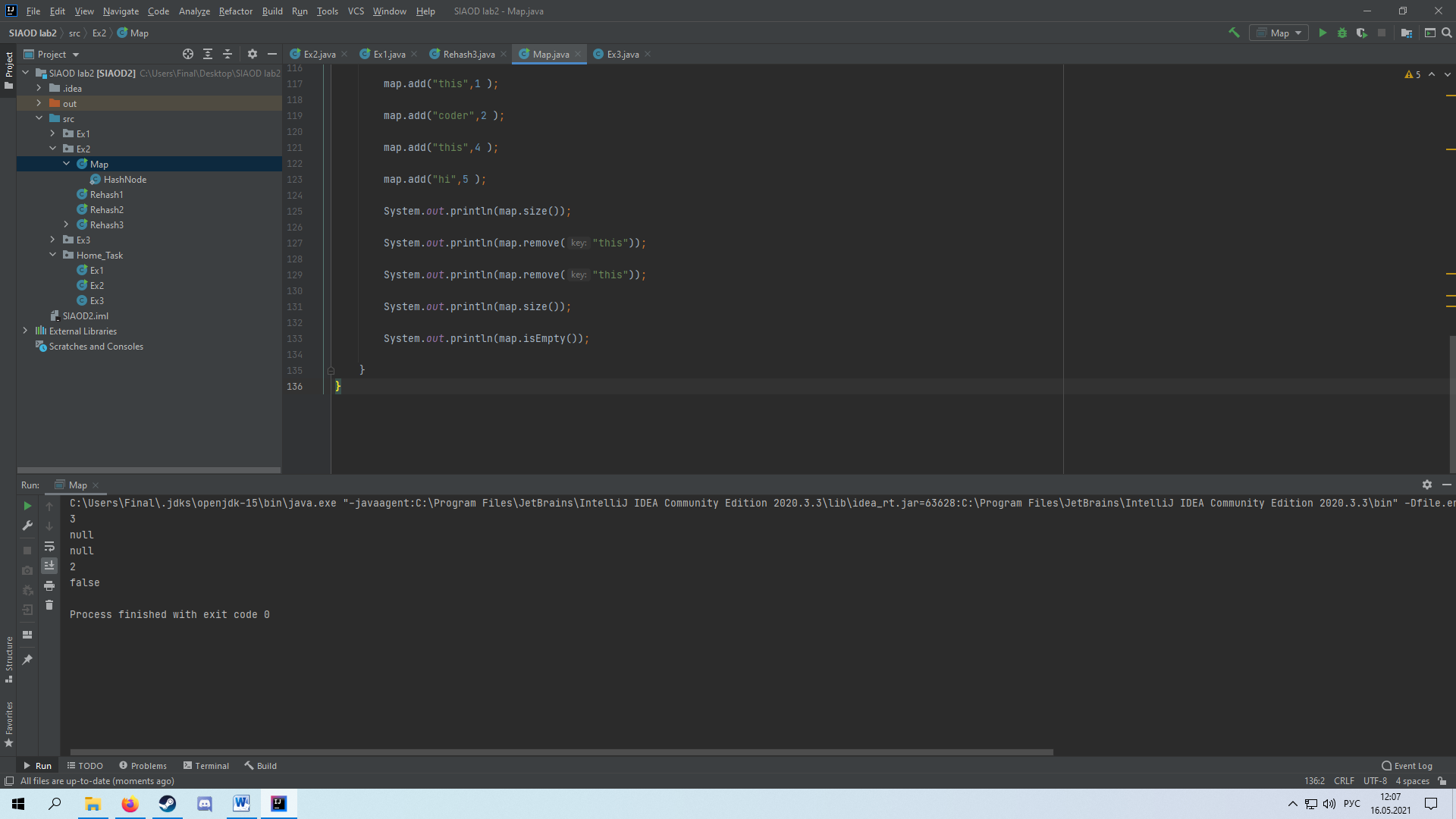
Алгоритм: каждая ячейка массива H является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.

Листинг метода цепочек:

import java.util.ArrayList;  
import java.util.Scanner;  
// Класс для представления всей хеш-таблицы  
class Map<K, V> {  
 // узел цепей  
 static class HashNode<K, V> {  
  
 K key;  
 V value;  
 HashNode<K, V> next;// Ссылка на следующий узел  
  
 public HashNode(K key, V value)// Конструктор  
 {  
 this.key = key;  
 }  
 }  
  
 private ArrayList<HashNode<K, V>> bucketArray;// bucketArray используется для хранения массива цепочек  
 private int numBuckets;// Текущая емкость списка массивов  
 private int size;// Текущий размер списка массивов  
  
 public Map(int numBuckets) {// Конструктор (Инициализирует емкость, размер и пустые цепочки.  
 bucketArray = new ArrayList<>();  
 this.numBuckets = numBuckets;  
 size = 0;  
 // Создать пустые цепочки  
 for (int i = 0; i < numBuckets; i++)  
 bucketArray.add(null);  
 }  
  
 public int size() {  
 return size;  
 }  
  
 public boolean isEmpty() {  
 return size() == 0;  
 }  
  
 private int getBucketIndex(K key) {// Реализует хеш-функцию для поиска индекса для ключа  
 int hashCode = key.hashCode();  
 int index = hashCode % numBuckets;  
 return index;  
  
 }  
  
 public V remove(K key) {// Метод для удаления данного ключа  
 int bucketIndex = getBucketIndex(key);// Применяем хеш-функцию для поиска индекса для данного ключа  
 HashNode<K, V> head = bucketArray.get(bucketIndex);// Получить голову цепи  
 HashNode<K, V> prev = null;// Поиск ключа в его цепочке  
 while (head != null) {  
 if (head.key.equals(key))// если ключ найден  
 break;  
 // Остальное продолжаем двигаться по цепочке  
 prev = head;  
 head = head.next;  
 }  
 if (head == null)// Если ключа там не было  
 return null;  
 size--;// Уменьшаем размер  
 if (prev != null) {// Удалить ключ  
 prev.next = head.next;  
 }else{  
 bucketArray.set(bucketIndex, head.next);  
 }  
 return head.value;  
 }  
  
 public V get(K key) {// Возвращает значение для ключа  
 int bucketIndex = getBucketIndex(key);// Найти начало цепочки для данного ключа  
 HashNode<K, V> head = bucketArray.get(bucketIndex);  
 while (head != null) {// Поиск ключа в цепочке  
 if (head.key.equals(key))  
 return head.value;  
 head = head.next;  
 }  
 return null;// Если ключ не найден  
 }  
  
 public void add(K key, V value) {// Добавляет пару ключ-значение в хеш  
 int bucketIndex = getBucketIndex(key);// Найти начало цепочки для данного ключа  
 HashNode<K, V> head = bucketArray.get(bucketIndex);  
 while (head != null) {// Проверяем, присутствует ли ключ  
 if (head.key.equals(key)) {  
 head.value = value;  
 return;  
 }  
 head = head.next;  
 }  
 // Вставляем 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);  
 if ((1.0 \* size) / numBuckets >= 0.7) {// Если коэффициент загрузки превышает пороговое значение,  
 ArrayList<HashNode<K, V>> temp = bucketArray;  
 bucketArray = new ArrayList<>();  
 numBuckets = 2 \* numBuckets;//то размер двойного хеш-таблицы  
 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) {  
  
 Map<String, Integer>map = new Map<>(4);  
  
 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());  
  
 System.*out*.println(map.isEmpty());  
  
 }  
}

Результаты работы метода цепочек:

Входные данные и команды: Выходные данные:



Так как при рехешировании было два одинаковых слова, они сложились в цепочку. Поэтому при запросе размера таблицы было выведено 3, вместо четырех. Две строки с надписью null говорят о том, что вместо удаленного элемента теперь пустота. И повторный запрос размера таблицы сообщает, что элементов this больше нет.

# Задание №3

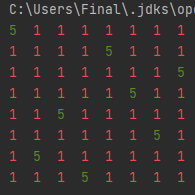
Задача: расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Подразумевается, что ферзь бьёт все клетки, расположенные по вертикалям, горизонталям и обеим диагоналям. Написать программу, которая находит хотя бы один способ решения задач.

Алгоритм: после выставления одного ферзя «закрываются» его линия, столбец и две диагонали. Этот алгоритм выполнятся несколько раз, пока есть свободные клетки. Если выставлено не 8 ферзей и все клетки закрыты, программа возвращается на шаг назад и берет другую ячейку.

Листинг шахмат:

public class Chess {  
 static int *n*=8;//quantity of cells  
 static int *q*=5;//queen number  
 static int *counter* = 0;//counter for queens (max 8)  
 public static final String *ANSI\_RED* = "\u001B[31m";//red color for text  
 public static final String *ANSI\_GREEN* = "\u001B[32m";//green color for text  
 public static final String *ANSI\_RESET* = "\u001B[0m";//reset text color  
 public static void main (String [] arg){  
 int [][] board = new int [*n*][*n*];//chess board 8\*8 cells  
 for (int i=0;i<*n*;i++){  
 for (int j=0;j<*n*;j++){  
 board [i][j]=0;//all cells are free  
 }  
 }  
 *Search*(board,0);  
 *Print\_board*(board);  
  
 }  
 public static void Closed\_cells (int [][]board, int string, int column){  
 //input: chess board, queen position  
 //step 1: delete the line  
 for (int i=0;i<*n*;i++){  
 board[string][i]--;//we can't put the queen here  
 }  
 //step 2: delete the columns  
 for (int i=0;i<*n*;i++){  
 board[i][column]--;//we can't put the queen here  
 }  
 //step 3: delete the left diagonal  
 //upper half  
 int i=string; int j=column;  
 while (i>0&&j<*n*-1){  
 i--;j++;  
 board[i][j]--;//we can't put the queen here  
 }  
 //bottom half  
 i=string; j=column;  
 while (i<*n*-1&&j>0){  
 i++;j--;  
 board[i][j]--;//we can't put the queen here  
 }  
 //step 4: delete the right diagonal  
 //upper half  
 i=string; j=column;  
 while (i>0&&j>0){  
 i--;j--;  
 board[i][j]--;//we can't put the queen here  
 }  
 //bottom half  
 i=string; j=column;  
 while (i<*n*-1&&j<*n*-1){  
 i++;j++;  
 board[i][j]--;//we can't put the queen here  
 }  
 board[string][column]=*q*;//put the queen  
 //output: board with closed cells and queen's cell  
 }  
 public static void Open\_cells (int [][]board, int string, int column){  
 //input: chess board, queen position  
 //step 1: delete the line  
 //input: chess board, queen position  
 //step 1: delete the line  
 for (int i=0;i<*n*;i++){  
 board[string][i]++;//we can't put the queen here  
 }  
 //step 2: delete the columns  
 for (int i=0;i<*n*;i++){  
 board[i][column]++;//we can't put the queen here  
 }  
 //step 3: delete the left diagonal  
 //upper half  
 int i=string; int j=column;  
 while (i>0&&j<*n*-1){  
 i--;j++;  
 board[i][j]++;//we can't put the queen here  
 }  
 //bottom half  
 i=string; j=column;  
 while (i<*n*-1&&j>0){  
 i++;j--;  
 board[i][j]++;//we can't put the queen here  
 }  
 //step 4: delete the right diagonal  
 //upper half  
 i=string; j=column;  
 while (i>0&&j>0){  
 i--;j--;  
 board[i][j]++;//we can't put the queen here  
 }  
 //bottom half  
 i=string; j=column;  
 while (i<*n*-1&&j<*n*-1){  
 i++;j++;  
 board[i][j]++;//we can't put the queen here  
 }  
 board[string][column]=0;//put the queen  
 //output: board with closed cells and queen's cell  
 }  
 public static void Print\_board(int [][] board){  
 //input: board with queens and closed cells positions  
 for (int i=0;i<*n*;i++){  
 for (int j=0;j<*n*;j++){  
 if (board [i][j]<0)//red color for closed cell  
 System.*out*.print(*ANSI\_RED*+1+" "+*ANSI\_RESET*);  
 if (board [i][j]==5)//green color for queen's sell  
 System.*out*.print(*ANSI\_GREEN*+board [i][j]+" "+*ANSI\_RESET*);//  
 }  
 System.*out*.println();  
 }  
 //output:print colorful board array to console  
 }  
 public static boolean Search (int [][] board,int s){  
 //input:chess board  
 boolean result = false;  
 for (int j=0;j<*n*;j++){  
 if(board[s][j]==0){  
 *Closed\_cells*(board, s,j);  
 *counter*++;  
 if (s == 7)  
 result = true;  
 else  
 {  
 if (!(result = *Search*(board,s + 1)))  
 *Open\_cells*(board,s, j);  
 }  
 }  
 if (result)  
 break;  
 }  
 return result;  
  
 }  
}

Результат работы шахмат:



# Вывод

В результате выполнения лабораторной работы «Методы поиска» я познакомился с различными методами поиска, а также вычислил время работы каждого из них, научился рехэшированию и разработал алгоритм для задачи «8 ферзей».