**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»**

Кафедра математической кибернетики и информационных технологий

Отчет по лабораторной работе №2

на тему «Методы поиска»

по дисциплине «Структуры и алгоритмы обработки данных»

Выполнила: студентка группы БВТ1904

Кремер К.С.

# 

Москва 2021

**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.

Задание №1:

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

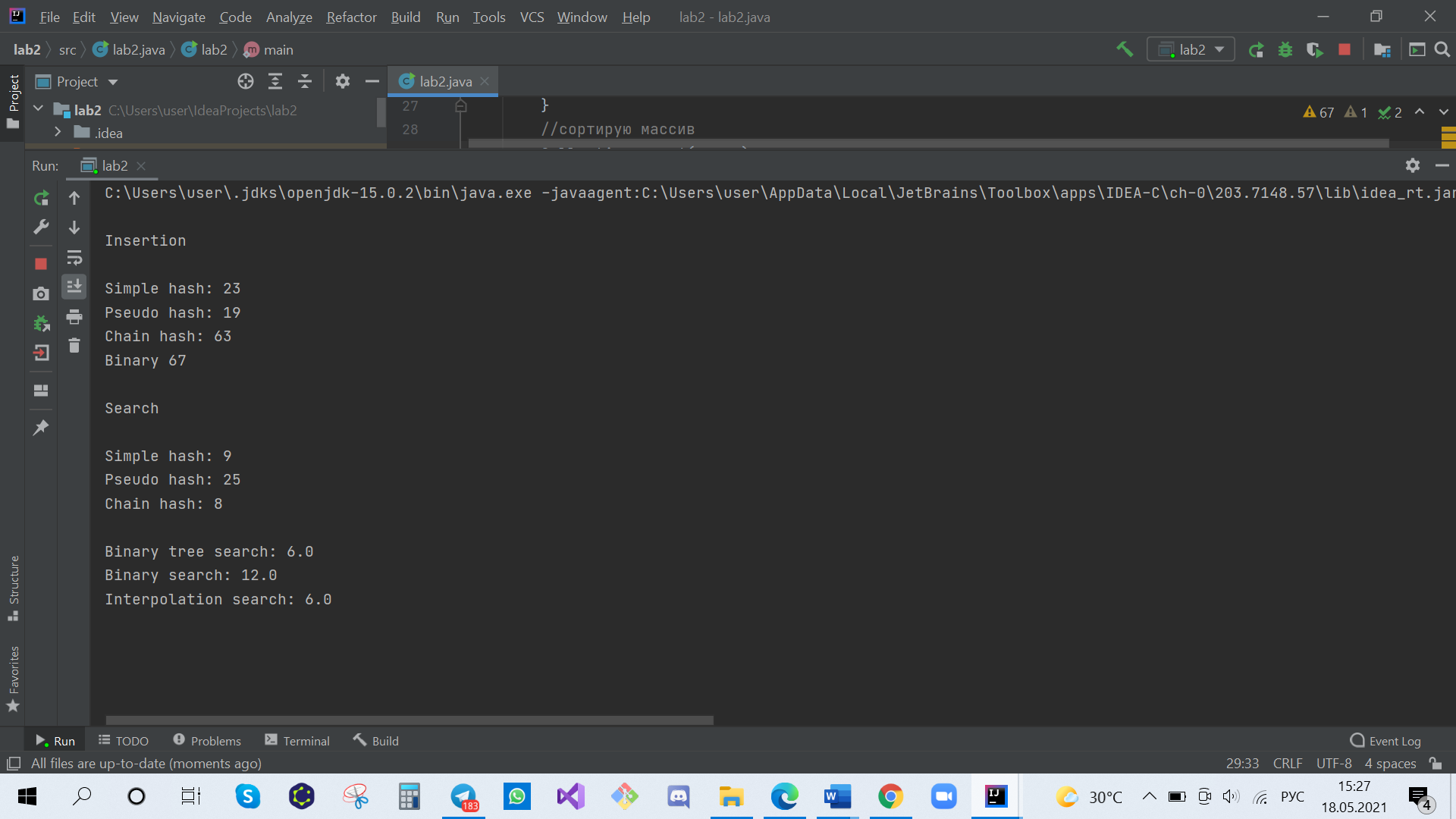
Задание №2:

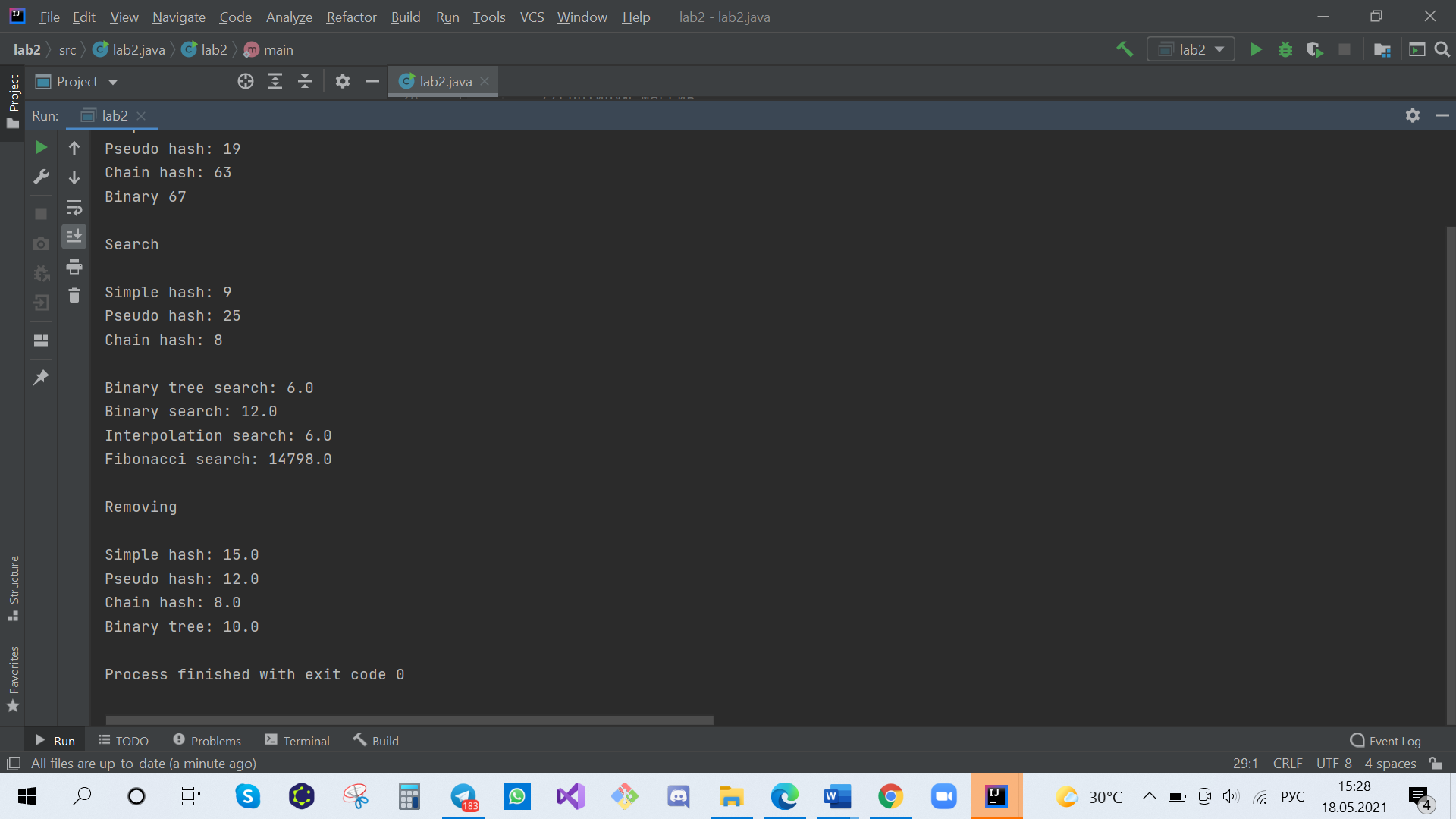
|  |  |  |
| --- | --- | --- |
| Простое рехэширование | Рехэширование с помощью псевдослучайных чисел | Метод цепочек |

**Код программы**

import java.io. \*;  
import java.util.\*;  
import java.lang. \*;  
  
public class lab2{  
 public static void main(String[] args){  
 hashTable table1 = new hashTable(300000);  
 hashTable table2 = new hashTable(300000);  
 Random rand = new Random(System.*currentTimeMillis*());  
  
 final int ARRSIZE = 100000;  
  
 ArrayList<Integer> array = new ArrayList<Integer>();  
  
 Tree tree = null;  
 //что за l ? непонел  
 ArrayList<Integer> l = new ArrayList<>(9);  
 //заполняю массив элементами  
 for (int i = 0; i < ARRSIZE; i++){  
 int number = rand.nextInt(10 \* ARRSIZE);  
  
 array.add(number);  
 if(tree == null){  
 tree = new Tree(number);  
 }  
 tree.add(number);  
 }  
 //сортирую массив  
 Collections.*sort*(array);  
  
 System.*out*.println("\nInsertion\n");  
  
 for(int i = 0; i < 4; i++){  
 long start = System.*currentTimeMillis*();  
 switch(i){  
 case 0:  
 for(int j = 0; j < ARRSIZE; j++){  
 int num = rand.nextInt(table1.size()\*10);  
 table1.insert\_simple(num);  
 }  
 System.*out*.println("Simple hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 1:  
 for(int j = 0; j < ARRSIZE; j++){  
 int num = rand.nextInt(table2.size()\*10);  
 table2.insert\_pseudo(num);  
 }  
 System.*out*.println("Pseudo hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 2:  
 for(int j = 0; j < ARRSIZE; j++){  
 int num = rand.nextInt(table1.size()\*10);  
 table1.insert\_chain(num);  
 }  
 System.*out*.println("Chain hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 3:  
 for (int j = 0; j < ARRSIZE; j++){  
 int number = rand.nextInt(10 \* ARRSIZE);  
 if(tree == null){  
 tree = new Tree(number);  
 }  
 tree.add(number);  
 }  
 System.*out*.println("Binary " + (System.*currentTimeMillis*() - start));  
 break;  
 default:  
 break;  
 }  
 }  
  
 System.*out*.println("\nSearch\n");  
  
 for(int i = 0; i < 4; i++){  
 int num = rand.nextInt(100);  
 long start = System.*currentTimeMillis*();  
 switch(i){  
 case 0:  
 for(int j = 0; j < ARRSIZE; j++){  
 table1.search\_simple(num);  
 }  
 System.*out*.println("Simple hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 1:  
 for(int j = 0; j < ARRSIZE; j++){  
 table2.search\_pseudo(num);  
 }  
 System.*out*.println("Pseudo hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 2:  
 for(int j = 0; j < ARRSIZE; j++){  
 table1.search\_chain(num);  
 }  
 System.*out*.println("Chain hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 3:  
 break;  
 default:  
 break;  
 }  
 }  
 double start = System.*currentTimeMillis*();  
 for(int i = 0; i < 10000; i++){  
 tree.B\_search(i);  
 }  
 double end = System.*currentTimeMillis*();  
 System.*out*.println("\nBinary tree search: " + (end - start));  
  
 start = System.*currentTimeMillis*();  
 for(int i = 0; i < 10000; i++){  
 *binary\_search*(array, i, 0, array.size() - 1);  
 }  
 end = System.*currentTimeMillis*();  
 System.*out*.println("Binary search: " + (end - start));  
  
 start = System.*currentTimeMillis*();  
 for(int i = 0; i < 10000; i++){  
 *interpolation\_search*(array, i);  
 }  
 end = System.*currentTimeMillis*();  
 System.*out*.println("Interpolation search: " + (end - start));  
  
 start = System.*currentTimeMillis*();  
 for(int i = 0; i < 10000; i++){  
 *fibonacci\_search*(array, i);  
 }  
 end = System.*currentTimeMillis*();  
 System.*out*.println("Fibonacci search: " + (end - start));  
  
  
  
 System.*out*.println("\nRemoving\n");  
  
 for(int i = 0; i < 4; i++){  
 int num = rand.nextInt(100);  
 start = System.*currentTimeMillis*();  
 switch(i){  
 case 0:  
 for(int j = 0; j < ARRSIZE; j++){  
 table1.delete\_simple(num);  
 }  
 System.*out*.println("Simple hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 1:  
 for(int j = 0; j < ARRSIZE; j++){  
 table2.delete\_pseudo(num);  
 }  
 System.*out*.println("Pseudo hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 2:  
 for(int j = 0; j < ARRSIZE; j++){  
 table1.delete\_chain(num);  
 }  
 System.*out*.println("Chain hash: " + (System.*currentTimeMillis*() - start));  
 break;  
 case 3:  
 for(int j = 0; j < ARRSIZE; j++){  
 tree.remove(num);  
 }  
 System.*out*.println("Binary tree: " + (System.*currentTimeMillis*() - start));  
 break;  
 default:  
 break;  
 }  
 }  
 }  
  
 static int binary\_search(ArrayList<Integer> array, int num, int left, int right){  
  
 if(left > right){  
 return -1;  
 }  
  
 int key = (right + left) / 2;  
  
 if(num == array.get(key)){  
 return key;  
 }  
 else if(num < array.get(key)){  
 key = *binary\_search*(array, num, left, key - 1);  
 }  
 else if(num > array.get(key)){  
 key = *binary\_search*(array, num, key + 1, right);  
 }  
  
 return key;  
 }  
  
 static int fibonacci\_search(ArrayList<Integer> array, int num){  
 int k = 1, M, i, p = 1, q = 1;  
 while(array.size() + 1 > p){  
 p += q;  
 q = p - q;  
 k++;  
 }  
  
 M = *F*(k + 1) - array.size() - 1;  
 i = *F*(k) - M;  
 p = *F*(k - 1);  
 q = *F*(k - 2);  
  
 while(true){  
 if(i < 0){  
 if(q == 0) return -1;  
 i -= q;  
 q = *swap*(p - q, p = q);  
 }  
 else if(i >= array.size()){  
 if(p == 1) return -1;  
 i += q;  
 p -= q;  
 q -= p;  
 }  
 else{  
 if(num < array.get(i)){  
 if(q == 0) return -1;  
 i -= q;  
 q = *swap*(p - q, p = q);  
 }  
 else if(num > array.get(i)){  
 if(p == 1) return -1;  
 i += q;  
 p -= q;  
 q -= p;  
 }  
 else{  
 return i;  
  
 }  
 }  
 }  
  
 }  
 static int F(int a){  
 if(a <= 1)  
 return a;  
  
 return *F*(a - 1) + *F*(a - 2);  
 }  
  
 static int interpolation\_search(ArrayList<Integer> array, int num){  
 int mid, left = 0, right = array.size() - 1;  
  
 while(array.get(left) <= num && array.get(right) >= num){  
 mid = left + ((num - array.get(left)) \* (right - left)) / (array.get(right) - array.get(left));  
 if (array.get(mid) < num) left = mid + 1;  
 else if (array.get(mid) > num) right = mid - 1;  
 else return mid;  
 }  
  
 if((array.get(left) == num)) return left;  
 return -1;  
 }  
  
 static int swap(int a, int b){  
 return a;  
 }  
}  
  
class Tree {  
 int value;  
 Tree left = null;  
 Tree right = null;  
 Tree parent = null;  
  
 public Tree(int value, Tree left, Tree right) {  
 this.value = value;  
 this.left = left;  
 this.right = right;  
 }  
  
 public Tree(int value) {  
 this.value = value;  
 }  
 public Tree(int value, Tree parent) {  
 this.value = value;  
 this.parent = parent;  
 }  
  
 public int sum() {  
 int summ = value;  
  
 if (left != null) {  
 summ += left.sum();  
 }  
 if (right != null) {  
 summ += right.sum();  
 }  
 return summ;  
 }  
  
 public int B\_search(int num){  
 if(num == value)  
 return 1;  
  
 if(num < value){  
 if(left == null)  
 return -1;  
 return left.B\_search(num);  
 }  
 else{  
 if(right == null)  
 return -1;  
 return right.B\_search(num);  
 }  
 }  
  
 public int add(int num){  
 if(num == value)  
 return -1;  
  
 if(num < value){  
 if(left == null){  
 left = new Tree(num, this);  
 return 1;  
 }  
 return left.add(num);  
 }  
 else{  
 if(right == null){  
 right = new Tree(num, this);  
 return 1;  
 }  
 return right.add(num);  
 }  
 }  
  
 public int remove\_min(){  
 if(left == null){  
 int a = value;  
 remove(value);  
 return a;  
 }  
 return left.remove\_min();  
 }  
  
 public int remove\_max(){  
 if(right == null){  
 int a = value;  
 remove(value);  
 return a;  
 }  
 return right.remove\_max();  
 }  
  
 public void remove(int num){  
 if(num < value){  
 if(left != null)  
 left.remove(num);  
 }  
 else if(num > value){  
 if(right != null)  
 right.remove(num);  
 }  
 else{  
 if(parent != null){  
 if(parent.left != null){  
 if(parent.left.value == num){  
 //если элемент является левым листом родителя  
 if(left == null && right == null){  
 parent.left = null;  
 }  
 //если есть левое поддерево  
 else if(left != null && right == null){  
 parent.left = left;  
 }  
 else if(left == null && right != null){  
 parent.left = right;  
 }  
 else if(parent.left.left != null && parent.left.right != null){  
 value = right.remove\_min();  
 }  
 }  
 }  
 if(parent.right != null){  
 if(parent.right.value == num){  
 //если элемент является правым листом родителя  
 if(left == null && right == null){  
 System.*out*.println("I'm here ");  
 parent.right = null;  
 }  
 else if(left != null && right == null){  
 parent.right = left;  
 }  
 else if(left == null && right != null){  
 parent.right = right;  
 }  
 else if(left != null && right != null){  
 value = right.remove\_min();  
 }  
 }  
 }  
 }  
 else{  
 if(right != null && left != null || left == null && right != null){  
 value = right.remove\_min();  
 }  
 else if(left != null && right == null){  
 value = left.remove\_max();  
 }  
 }  
 }  
 }  
  
}  
  
class hashTable{  
 private Integer[] table;  
 private LinkedList<Integer>[] tbl;  
 private int size;  
  
 public hashTable(int size){  
 this.size = size;  
 table = new Integer[size];  
 tbl = new LinkedList[size];  
 }  
  
 private int hash(int key){  
 return key % size;  
 }  
  
 private int simple\_rehash(int hash){  
 return (hash + 1) % size;  
 }  
  
 private int pseudo\_rehash(int hash, Random rand){  
 return (hash + Math.*abs*(rand.nextInt(size))) % size;  
 }  
  
 public int size(){  
 return size;  
 }  
  
 public Integer get(int i){  
 return table[i];  
 }  
  
 public LinkedList<Integer> take(int i){  
 return tbl[i];  
 }  
  
 public void insert\_chain(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 if(tbl[hash] == null){  
 tbl[hash] = new LinkedList<Integer>();  
 }  
 tbl[hash].add(item);  
 }  
  
 public void insert\_simple(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 while(table[hash] != null){  
 hash = simple\_rehash(hash);  
 }  
 table[hash] = item;  
 }  
  
 public void insert\_pseudo(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 while(table[hash] != null){  
 hash = pseudo\_rehash(hash, rand);  
 }  
 table[hash] = item;  
 }  
  
 public boolean search\_pseudo(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 while(table[hash] != null && table[hash] != key){  
 hash = pseudo\_rehash(hash, rand);  
 }  
 if(table[hash] == null){  
 return false;  
 }  
 return true;  
 }  
  
 public boolean search\_simple(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 while(table[hash] != null && table[hash] != key){  
 hash = simple\_rehash(hash);  
 }  
 if(table[hash] == null){  
 return false;  
 }  
 return true;  
 }  
  
 public boolean search\_chain(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 if(tbl[hash] == null){  
 return false;  
 }  
 else{  
 if(tbl[hash].contains(key)){  
 return true;  
 }else{  
 return false;  
 }  
 }  
 }  
  
 public void delete\_chain(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 if(tbl[hash] != null){  
 if(tbl[hash].contains(key)){  
 tbl[hash].remove(key);  
 }  
 }  
 }  
  
 public void delete\_simple(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 while(table[hash] != null && table[hash] != key){  
 hash = simple\_rehash(hash);  
 }  
 table[hash] = null;  
 }  
  
 public void delete\_pseudo(int key){  
 Integer item = new Integer(key);  
 int hash = hash(key);  
 Random rand = new Random(hash);  
  
 while(table[hash] != null && table[hash] != key){  
 hash = pseudo\_rehash(hash, rand);  
 }  
 table[hash] = null;  
 }  
}

**Результат работы программы**





**Вывод**

В ходе данной лабораторной работы я реализовала методы поиска в соответствии с заданием. Среди всех методов поисков наименьшее время занимает интерполяционный, а среди рехеширования – метод simple\_hash.