Министерство Образования, Культуры и Исследований  
Молдавский Государственный Университет  
Факультет Математики и Информатики  
Департамент Информатики

Лабораторная работа №1

По предмету: «Алгоритмы, структуры данных и сложность».  
ТЕМА: Методы и процедуры поиска.

Преподаватель: Кроитор М.  
Студент: Журба Дарья.  
Группа: IА2102.

Кишинёв 2023

Задание: Требуется создать текстовый файл который содержит 50 записей. Каждая запись имеет минимум 5 полей с минимум 2-мя типами данных. Ключевое поле должно быть уникальным. Записи не упорядочены. Тематика структуры данных определяется каждым студентом совместно с преподавателем на лабораторных занятиях.

Требуется реализовать в C++ или Java программу в которой внедрены 4 метода поиска в отсортированных и неотсортированных массивах по ключевому полю в текстовом файле созданном ранее. Для каждого метода поиска требуется провести анализ среднего теоретического времени работы и вывод на экран практического времени поиска. Описать по шагам алгоритмы методов поиска.

Метод последовательного (линейного) поиска

Метод бинарного поиска

Метод поиска интерполированием

Метод поиска Фибоначчи

Метод поиска в древовидных неупорядоченных таблицах

Ход работы:

**Метод последовательного(линейного) поиска**

Алгоритм линейного поиска – простейший алгоритм поиска. Начиная с начала набора данных, проверяется каждый элемент на соответствие некоторому значению. Поиск прекращается после нахождения соответствия. Применим как для отсортированных, так и для неотсортированных данных.

Сложность: O(n)

Реализация:public class LinearSearch {  
public static void linearSearch(int searchValue, ArrayList<String> list)  
{  
System.out.println("Linear Search");  
int iterationsNum=0,keyValue;  
String substring i;  
for(String i:list)  
{  
iterationsNum++;  
String[] line = i.split("\\t");  
keyValue = Integer.valueOf(line[0]);  
if(keyValue==searchValue) {  
System.out.println(i);  
break;  
}  
}  
System.out.println("Number of iterations = " + iterationsNum);  
}  
}

Результат:

Linear Search

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кириленко | Юрий | 25 | Водитель | 304 |

Number of iterations = 46

**Метод бинарного поиска**

Бинарный поиск работает с отсортированными массивами данных. Суть метода заключается в сравнении искомого ключа с ключом среднего элемента массива.

**mid = (first + last)/2;**

В случае равенства поиск завершается. Если искомый ключ меньше ключа среднего элемента массива, продолжается поиск в правой половине, иначе – в левой. Поиск продолжается до тех пор, пока не будет найден элемент, равный значению исходного ключа, либо не станет пустым интервал поиска.

Сложность: O(log2n)

Реализация:public class BinarySearch {  
public static void binarySearch(int searchValue, ArrayList<String> list)  
{  
System.out.println("Binary Search");  
Collections.sort(list);  
int iterationsNum=0;  
int first=0;  
int last = list.size()-1;  
int key;  
int mid = (first + last)/2;  
while( first<=last)  
{  
key =  
Integer.parseInt(list.get(mid).substring(0,list.get(mid).indexOf("\t")));  
if(key==searchValue)  
{  
System.out.println(list.get(mid));  
break;  
}  
else if(key>searchValue)  
{  
last = mid-1;  
}  
else first = mid+1;  
iterationsNum++;  
mid = (first + last)/2;  
}  
if(first>last) System.out.println("Record not found");  
System.out.println("Number of iterations = " + iterationsNum);  
}  
}

Результат:

Binary Search

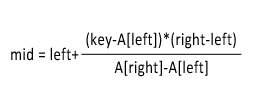
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кириленко | Юрий | 25 | Водитель | 304 |

Number of iterations = 3

**Метод поиска интерполированием**

Основан на интерполировании. Интерполирование – нахождение промежуточных значений величины по имеющемуся дискретному набору известных значений. Интерполяционный поиск вычисляет, в каком оставшемся интервале для поиска может находится искомый элемент. Это вычисление осуществляется на основе нижнего и верхнего значения интервала и значения искомого элемента.

Формула:



Где:

mid – номер элемента, с которым сравнивается значение ключа,

key – ключ (искомый элемент),

A – массив упорядоченных элементов,

left и right – номера крайних элементов области поиска.

Операция деления в формуле строго целочисленная.

Применим только к упорядоченным массивам.

Сложность: O(log logn)

Реализация:public class Interpolation {  
public static void interpolationSearch(int searchValue,  
ArrayList<String> list)  
{  
System.out.println("Interpolation Search");  
int iterationsNum = 0;  
int low = 0;  
int hight = list.size()-1;  
int keyLeft=  
Integer.parseInt(list.get(low).substring(0,list.get(low).indexOf("\t")));  
int keyRight=  
Integer.parseInt(list.get(hight).substring(0,list.get(hight).indexOf("\t")));  
while(!Objects.equals(list.get(low), list.get(hight)) && searchValue  
>= keyLeft && searchValue<=keyRight)  
{  
iterationsNum++;  
int mid = low + ((searchValue-keyLeft)\*(hight-low)/(keyRightkeyLeft));  
int keyMid=  
Integer.parseInt(list.get(mid).substring(0,list.get(mid).indexOf("\t")));  
if(searchValue ==keyMid ) { System.out.println(list.get(mid));  
break;}

else if   
{

(searchValue < keyMid)

hight=mid -1;  
}  
else low = mid+1;  
keyLeft=  
Integer.parseInt(list.get(low).substring(0,list.get(low).indexOf("\t")));  
keyRight=  
Integer.parseInt(list.get(hight).substring(0,list.get(hight).indexOf("\t")));  
}  
if(searchValue == keyLeft)  
{  
System.out.println(list.get(low));  
}  
else if (!(searchValue >= keyLeft && searchValue<=keyRight))  
System.out.println("Record not found");  
System.out.println("Number of iterations = " + iterationsNum);  
}  
}

Результат:

Interpolation Search

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кириленко | Юрий | 25 | Водитель | 304 |

Number of iterations = 4

**Метод поиска Фибоначчи**

В основе метода Фибоначчи лежит идея золотого сечения, которая позволяет найти минимум и максимум функции. Идея состоит в том, чтобы сначала найти наименьшее число Фибоначчи, которое больше или равно длине данного массива. Для таблицы записей R1 , R2 , ..., RN , ключи которых расположены в порядке возрастания K1< K2< ... < KN, алгоритм ищет заданный аргумент K. N + 1 = F k + 1. Пусть p и q будут последовательными числами Фибоначчи.

1. [Инициализация] i ← Fk , p ← Fk -1 , q ← Fk-2;

2. Если K < Ki , переход к этапу 3; если K>K i переход к этапу 4; и если K = K i , алгоритм

успешно завершается;

3. [Уменьшение i] Если q = 0, алгоритм завершается неудачно. В противном случае  
(i, p, q) ← (i - q, q, p - q); переход к шагу 2;

4. [Увеличить i] Если p = 1, алгоритм завершится безуспешно. В противном случае  
(i, p, q) ← (i + q, p - q, 2q - p); и переход к шагу 2;

Числа Фибоначчи рекурсивно определяются как F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1.  
Первые несколько чисел Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Сложность: O(logn)

class Fibonacci {  
public static int min(int o, int n)  
{  
return (o <= n) ? o : n;  
}  
public static void Fibonacci(int searchValue, ArrayList<String> list)  
{  
System.out.println("\nFibonacci Search");  
int q = 0;  
int p = 1;  
int i = q + p;  
int n=50;  
int iterationsNum =0;  
while (i < n) {  
q = p;  
p = i;  
i = q + p;  
}  
int offset = -1;  
while (i > 1) {  
int j = min(offset + q, n - 1);  
String[] l = list.get(j).split("\\t");  
int key = Integer.valueOf(l[0]);  
if (key < searchValue ) {  
i = p;  
p = q;  
q = i - p;  
offset = j;  
}  
else if (key > searchValue ) {  
i = q;  
p = p - q;  
q = i - p;  
}  
else  
{  
System.out.println(list.get(j));  
System.out.println("Number of iterations = " +iterationsNum);  
return;  
}  
iterationsNum++;  
}  
String[] l = list.get(n-1).split("\\t");  
int key = Integer.valueOf(l[0]);

if   
{

(key == searchValue && p == 1)

System.out.println(list.get(n-1));  
System.out.println("Number of iterations = " + iterationsNum);  
}  
else System.out.println("Record not found");  
}  
}

Результат:

Fibonacci Search

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кириленко | Юрий | 25 | Водитель | 304 |

Number of iterations = 5

**Метод поиска в древовидных неупорядоченных таблицах**

Двоичное дерево — иерархическая структура данных, в которой каждый узел имеет не более двух потомков.

Свойства:

Левое поддерево узла содержит только узлы с ключами меньше, чем ключ корневого узла.

Правое поддерево узла содержит только узлы с ключами больше, чем ключ корневого узла.

Каждое из левого и правого поддеревьев также должно быть бинарным деревом поиска.

В двоичном дереве поиска хранятся элементы, которые можно сравнивать между собой при помощи операций "меньше" и "больше", например, это могут быть числа или строки.

Все элементы, которые хранятся в дереве - уникальные, то есть одно число не может быть записано в двух вершинах дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. Поиск осуществляется посредством сравнения искомого значения со значением корневого элемента. Если искомый элемент больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Сложность: O(logn)

class BinarySearchTree {  
int iterationsNum =0;  
class Node {  
String key;  
Node left;  
Node right;  
public Node(String data){  
key = data;  
left = right = null;  
}  
}  
Node root;  
BinarySearchTree(ArrayList<String> list){  
root = null;  
for(int i = 0;i<list.size()-1;i++)  
{  
this.insert(list.get(i));  
}  
}  
void insert(String key) {  
root = insert\_Rec(root, key);  
}  
Node insert\_Rec(Node root, String key) {

(root == null) {

if  
root = new Node(key);

return root;  
}  
int k = Integer.parseInt(key.substring(0, key.indexOf("\t")));  
int keyr = Integer.parseInt(root.key.substring(0,  
root.key.indexOf("\t")));

if   
{

(k < keyr)

root.left = insert\_Rec(root.left, key);

} else if   
{

(k > keyr)

root.right = insert\_Rec(root.right, key);  
}  
return root;  
}  
void search(int key) {  
System.out.println("\nBinary Search Tree");  
root = search\_Rec(root, key);  
}  
Node search\_Rec(Node root, int key) {  
iterationsNum++;  
int keyr = Integer.parseInt(root.key.substring(0,  
root.key.indexOf("\t")));

if   
{

( keyr==key)

System.out.println(root.key);  
System.out.println("Number of iterations = " + iterationsNum);  
return root;  
}

if

(keyr> key)

return search\_Rec(root.left, key);

return search\_Rec(root.right, key);  
}  
}

Результат:

Binary Search Tree

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кириленко | Юрий | 25 | Водитель | 304 |

Number of iterations = 15

Приложение :  
Текстовый файл:

Учет Сотрудников компании.

