**МИНОБРНАУКИ РОССИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕНЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Г.Шухова»**

**(БГТУ им. В.Г.Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа № 4

По дисциплине: Алгоритмы и СД

Тема: Сравнительный анализ алгоритмов поиска (Pascal/C)

Выполнил: ст.группы ПВ - 21

Браткова И.О.

Белгород 2017

**Цель работы:** изучение алгоритмов поиска элемента в массиве и **з**акрепление навыков в проведении сравнительного анализа алгоритмов.

# З а д а н и е

1. Изучить алгоритмы поиска:

1) в неупорядоченном массиве:

- линейный;

- быстрый линейный;

2) в упорядоченном массиве:

- быстрый линейный;

- бинарный;

- блочный.

2. Разработать и программно реализовать средство для проведения экспериментов по определению временных характеристик алгоритмов поиска.

3. Провести эксперименты по определению временных характеристик алгоритмов поиска. Результаты экспериментов представить в виде таблиц 12 и 13. Клетки таблицы 12 содержат максимальное количество операций сравнения при выполнении алгоритма поиска, а клетки таблицы 13 —среднее число операций сравнения.

4. Построить графики зависимости количества операций сравнения от количества элементов в массиве.

5. Определить аналитическое выражение функции зависимости количества операций сравнения от количества элементов в массиве.

6. Определить порядок функций временной сложности алгоритмов

поиска.

**Задание 1**

1. **В неупорядоченном массиве:**
   1. Линейный

Работа алгоритма заключается в том, что элементы массива, начиная с первого, последовательно сравнивается с искомым элементом. Сравнение элементов продолжается до тех пор, пока не будут просмотрены все элементы или очередной элемент массива не равен искомому.

* 1. Быстрый линейный

Любой алгоритм поиска содержит блок проверки на окончание массива. В алгоритме линейного поиска эта проверка осуществляется каждый раз перед обращением к очередному элементу. Однако проверка на окончание массива может осуществляться не при каждом сравнении. Для этого в конец массива включается (N + 1)-й элемент (N — количество элементов в массиве), равный искомому. Тогда проверка на окончание массива осуществляется лишь при совпадении очередного элемента с искомым. Если этот элемент находится внутри массива, то поиск заканчивается удачно и элемент считается найденным. Если же этот элемент оказался (N + 1)-ым, то искомого элемента в массиве нет.

1. **В упорядоченном массиве:**
   1. Быстрый линейный

Применяя алгоритм быстрого линейного поиска для поиска элемента в упорядоченном массиве, поиск можно прекратить, если очередной элемент массива будет больше искомого. Это будет означать, что искомого элемента в массиве нет. В случае поиска элемента, который есть в массиве, этот алгоритм аналогичен алгоритму быстрого линейного поиска в неупорядоченном массиве.

* 1. Бинарный

Принцип, лежащий в основе алгоритма бинарного поиска (и некоторых других алгоритмов), состоит в том, что иногда удается последовательно уменьшать объем задачи до такой степени, что ее решение в конце концов становится тривиальным. Главный шаг при бинарном поиске — взять элемент из середины массива и, если он не равен искомому, то в зависимости от его значения ту или другую половину массива убрать из рассмотрения. Повторное выполнение этого шага быстро сокращает размер области поиска.

* 1. Блочный

Блочный поиск состоит в том, что массив, упорядоченный по возрастанию, разбивается на определенное число блоков. В процессе поиска искомый элемент последовательно сравнивается с последним элементом блоков. Если искомый элемент меньше последнего элемента очередного блока, то искомый элемент может находиться только внутри этого блока. Для поиска элемента в блоке можно применить линейный поиск.

**Задание 2**

**int line\_neup (int \*a, int x, size\_t n, int \*k)**

{ int i=0;

while ((i<n) && (a[i]!=x))

{ i++; (\*k)+=2; } (\*k)+=3;

if (i<n) return 1;

else return 0;

}

**int spe\_line\_neup (int \*a, int x, size\_t n, int \*k)**

{

int i=0;

while (a[i]!=x)

{ i++; (\*k)++; } (\*k)+=2;

if (i==n) return 0;

else return 1;

}

**int line\_upor (int \*a, int x, size\_t n, int \*k)**

{ int i=0;

while (a[i]<x)

{ i++; (\*k)++; } (\*k)+=3;

if (i==n || a[i]!=x) return 0;

else return 1;

}

**int binary\_upor (int \*a, int x, size\_t n, int \*k)**

{

int left = 0, right = n, midl;

while (left<=right)

{

midl = (left+right)/2;

if (a[midl]==x)

{ (\*k)++; return 1; }

else

{

(\*k)++; if (x < a[midl]) right = midl-1;

else left = midl+1;

} (\*k)++; } (\*k)++;

return 0;

}

**int blocs\_upor (int \*a, int x, size\_t n, int \*k)**

{

int i, j=(int)sqrt(n);

i=j-1;

while ((i<=(j\*j-1)) && (a[i]<x))

{

i+=j;

(\*k)+=2;

}

(\*k)+=4;

if ((i==(j\*j-1)) && (a[i]==x)) return i;

else if ((i<(j\*j-1)) && (a[i]>x))

{

i-=j;

j=i+j;

(\*k)+=2;

while ((i<j) && (a[i]!=x))

{

i++;

(\*k)+=2;

}

(\*k)+=3;

if (i<n) return i;

else return -1;

}

else

{

i=i-j+1;

while ((i<n) && (a[i]!=x))

{

i++;

(\*k)+=2;

}

(\*k)+=3;

if (i<n) return i;

else return -1;

}

}

**Для таблицы:**

#define N 1000

#define TableOk 0 #define TableEmpty 1 #define TableFull 2 #define TableNotContent 3

#define TableAlreadyContent 4 short TableError;

typedef struct { int key; int data; } t\_element;

typedef struct { t\_element buf[N]; unsigned uk; }t\_table;

t\_table up\_table(size\_t n); t\_table noup\_table(size\_t n);void output\_table (t\_table t, unsigned n);

void InitTable(t\_table \*t);void IsEmptyTable(t\_table \*t);void IsFullTable(t\_table \*t);

void ReadTable(t\_table \*t, int \*e, int k);void PutInTable(t\_table \*t, int d, int k);

void GetInTable (t\_table \*t, int \*d, int k);

#include <stdio.h>#include <stdlib.h>#include "TABLE.h"

**t\_table up\_table(size\_t n) //Упорядочнная таблица**

{

IsFullTable(t); if (TableError==TableFull) return;

size\_t i; t\_table tab;

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

{

tab.uk=i; tab.buf[i].key=i; tab.buf[i].data=i+2;

}

return tab;

}

**t\_table noup\_table(size\_t n) //Неупорядоченная таблица**

{

IsFullTable(t); if (TableError==TableFull) return;

size\_t i; t\_table tab;

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

{

tab.uk=i; tab.buf[i].key=i; tab.buf[i].data=1+rand()%n;

}

return tab;

}

**void output\_table (t\_table t, unsigned n) //Вывод**

{ int i; for(i=0; i<n; i++) printf("|%d \t\t %d|\n", t.buf[i].data, t.buf[i].key);}

**void InitTable(t\_table \*t) //Инициализация таблицы**

{ t->uk=0; TableError=TableOk;}

**void IsEmptyTable(t\_table \*t) // Пуста-Не пуста**

{

if (t->uk == 0)

{ TableError = TableEmpty; return 0; }

return 1;

}

**void IsFullTable(t\_table \*t) // Полна-Не полна**

{

if (t->uk == N)

{ TableError = TableFull; return 0; }

return 1;

}

**void ReadTable(t\_table \*t, int \*e, int k) //Чтение из таблицы**

{

int i=0;

IsEmptyTable(t); if (TableError==TableEmpty) return;

while (i<t->uk)

{

if (t->buf[i].key==k)

{ \*e=t->buf[i].data; return; }

i++;

}

TableError=TableNotContent;

}

**void PutInTable(t\_table \*t, int d, int k) // вставить в таблицу**

{

t\_element tmp;

IsFullTable(t); if (TableError==TableFull) return;

IsEmptyTable(t);

ReadTable(t, &tmp.data, k);

int i = t->uk;

if ((TableError==TableNotContent) || (TableError == TableEmpty))

{

tmp.data=d; tmp.key=k; t->buf[i]=tmp; i++; TableError=TableOk;

}

else TableError=TableAlreadyContent;

}

**void GetInTable(t\_table \*t, int \*E, int K) //Взять из таблицы**

{

int i=0;

IsEmptyTable(t); if (TableError==TableEmpty) return;

while (i<t->uk)

{

if (t->buf[i].key==K)

{

int j; \*E=t->buf[i].data; for (j=i+1; j<t->uk; j++)

{

t->buf[j-1]=t->buf[j];

}

t->uk--; return;

}

i++;

}

TableError=TableNotContent;

}

**Сортировки для таблицы:**

**int line\_neup (t\_table a, int x, size\_t n, int \*k) //линейная в неупорядоченном**

{ int i=0;

while ((i<n) && (a.buf[i].data!=x))

{ i++; (\*k)+=2; } (\*k)+=3;

if (i<n) return 1;

else return 0;

}

**int spe\_line\_neup (t\_table a, int x, size\_t n, int \*k) //быстрая линейная в нейпорядоченном**

{

int i=0;

while (a.buf[i].data !=x)

{ i++; (\*k)++; } (\*k)+=2;

if (i==n) return 0;

else return 1;

}

**int line\_upor (t\_table \*a, int x, size\_t n, int \*k) //линейная в упорядоченном**

{ int i=0;

while (a.buf[i].data <x)

{ i++; (\*k)++; } (\*k)+=3;

if (i==n || a.buf[i].data!=x) return 0;

else return 1;

}

**int binary\_upor (int \*a, int x, size\_t n, int \*k) //бинарная**

{

int left = 0, right = n, midl;

while (left<=right)

{

midl = (left+right)/2;

if (a.buf[midl].data ==x)

{ (\*k)++; return 1; }

else

{ (\*k)++; if (x < a.buf[midl].data) right = midl-1;

else left = midl+1; } (\*k)++; } (\*k)++;

return 0;

}

**int block\_poisk(t\_table tab, unsigned n, int x, unsigned \*ks)**

{

int block=sqrt(n);

int i,j=block;

while((j<n) && (x>tab.buf[j].data))

{ j+=block; (\*ks)+=2; }

if(j<n) { i=j-block;

while((i<j)&&(x!=tab.buf[i].data))

{ (\*ks)+=2; i++; } return i<=j;

}

return 0;}

**Здание 3**

Максимальное число операций сравнения

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Алгоритмы | Количество элементов в массиве | | | | | | | | |
| 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 |
| Линейный (неупорядоченный массив) | 101 | 201 | 301 | 401 | 501 | 601 | 701 | 801 | 901 |
| Быстрый линейный (неупорядоченный массив) | 51 | 101 | 151 | 201 | 251 | 301 | 351 | 401 | 451 |
| Быстрый линейный (упорядоченный массив) | 52 | 102 | 152 | 202 | 252 | 302 | 352 | 402 | 452 |
| Бинарный (упорядоченный массив) | 9 | 11 | 13 | 15 | 17 | 19 | 21 | 23 | 25 |
| Блочный (упорядоченный массив) | 21 | 22 | 41 | 41 | 85 | 61 | 93 | 42 | 65 |

Среднее число операций сравнения

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Алгоритмы  поиска | Количество элементов в массиве | | | | | | | | |
| 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 |
| Линейный (неупорядоченный массив) | 52 | 105 | 147 | 198 | 252 | 304 | 359 | 396 | 422 |
| Быстрый линейный (неупорядоченный массив) | 25 | 52 | 78 | 102 | 131 | 155 | 173 | 196 | 225 |
| Быстрый линейный (упорядоченный массив) | 26 | 51 | 77 | 100 | 129 | 152 | 175 | 217 | 235 |
| Бинарный (упорядоченный массив) | 12 | 15 | 17 | 17 | 19 | 20 | 20 | 21 | 22 |
| Блочный (упорядоченный массива) | 21 | 27 | 32 | 36 | 42 | 43 | 45 | 48 | 50 |

**Задание 4-5**

Максимальное

Среднее

**Задание 6**

Порядок временной сложности алгоритмов:

* **Неупорядоченные массивы:**
  + Линейный поиск:

Другие случаи: O(N);

Лучший случай: O(1);

* + Быстрый линейный поиск:

Другие случаи: O(N);

Лучший случай: O(1);

* **Упорядоченные массивы:**
  + Линейный поиск:

Другие случаи: O(N);

Лучший случай: O(1);

* + Бинарный поиск:

Лучший случай: O(1) (искомый элемент в середине массива)

Другие случаи: O(log 2 N).

* + Блочный поиск:

Худший случай: 

Другие случаи: 