**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Г.ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и

автоматизированныхсистем

Лабораторная работа №4

дисциплина: Алгоритмы и структуры данных

тема: «Сравнительный анализ алгоритмов поиска (Pascal/C)»

Выполнил ст. группы ВТ-

Проверил: проф. Синюк В.Г.

Белгород 20

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

**Задание**

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.

**Заголовочный файл**

**“Search.h”**

#ifndef **ASD4\_SEARCH\_H**#define **ASD4\_SEARCH\_H  
int** lineSearch(**int** \*a, **int** n, **int** key);  
**int** fastLineSearch(**int** \*a, **int** n, **int** key);  
**int** binSearch(**int** \*a, **int** n, **int** key);  
**int** blockSearch(**int** \*a, **int** n, **int** key);  
#endif *//ASD4\_SEARCH\_H*

**Модуль**

**“Search.с”**

#include **<stdlib.h>**#include **<math.h>**

**int** countComparison;  
  
*//линейный поиск***int** lineSearch(**int** \*a, **int** n, **int** key) {  
 countComparison = 0;  
 **int** i = 0;  
 **while** ((countComparison += 2) && (i < n) && (a[i] != key))  
 i++;  
 **return** (i != n) ? i : -1;  
}  
  
*//быстрый линейный поиск***int** fastLineSearch(**int** \*a, **int** n, **int** key) {  
 countComparison = 0;  
 **int** i = 0;  
 a[n] = key;  
 **while** ((countComparison += 1) && (a[i] != key))  
 i++;  
 **return** (i != n) ? i : -1;  
}  
  
*//Быстрый линейный поиск для массива упорядоченного по возрастанию***int** sortfastLineSearch(**int** \*a, **int** n, **int** x)  
{  
 countComparison = 1;  
  
 a[n] = x;  
 **int** i = 0;  
 **while** ((a[i] < x)) {  
 i++;  
 ++countComparison;  
 }  
 **return** (x < a[i]) || (i == n) ? -1 : i; *//-1 в случае ненаходы*}  
  
*//бинарный поиск***int** binSearch(**int** \*a, **int** n, **int** key) {  
 countComparison = 0;  
 **int** left = 0;  
 **int** right = n;  
 **int** search = -1;  
 **while** (left <= right) {  
 **int** mid = (left + right) / 2;  
 countComparison++;  
 **if** (key == a[mid]) {  
 search = mid;  
 **break**;  
 }  
 **if** (key < a[mid])  
 right = mid - 1;  
 **else** left = mid + 1;  
 }  
 **return** search;  
}  
  
*//блочный поиск***int** blockSearch(**int** \*a, **int** n, **int** key) {  
 **int** step = sqrt(n);  
 **int** stblock = 0;  
 countComparison = 0;  
  
 **while** (stblock < n + step) {  
 countComparison += 1;  
 **if** (a[stblock] > key) {  
 **for** (**int** i = step; i > 0; i--) {  
 countComparison += 1;  
 **if** ((stblock - i) < n && a[stblock - i] == key)  
 **return** stblock - i;  
 }  
 **return** -1;  
 }  
 stblock += step;  
 }  
 **return** -1;  
}

**“Main.с”**

#include **"Search.h"**

#include **<stdio.h>**#include **<stdlib.h>**

**const int** N = 10;

*// заполнение массива***void** initArr(**int** \*arr, **int** n, **int** t) {  
  
 **for** (**int** i = 0; i < n; i++) {  
 **switch** (t) {  
 **case** 1:  
 arr[i] = i;  
 **break**;  
 **case** 2:  
 arr[i] = rand() % 100 + 1;  
 **break**;  
 }  
 }  
}  
*//возвращает наихудшее количество шагов данного поиска***int** getBadStep(**int** \*a, **int** n, **int**(\*find)(**int** \*, **int**, **int**)) {  
 **int** max = 0;  
 **for** (**int** i = 0; i < n; i++) {  
 find(a, n, a[i]);  
 **if** (max < countComparison)  
 max = countComparison;  
 }  
 **return** max;  
}  
  
*//возвращает среднее количество шагов данного поиска***int** getAverageStep(**int** \*a, **int** n, **int**(\*find)(**int** \*, **int**, **int**)) {  
 **int** sum = 0;  
 **for** (**int** i = 0; i < n; i++) {  
 find(a, n, a[i]);  
 sum += countComparison;  
 }  
 **return** sum / n;  
}  
  
  
**int** main() {  
 **int** a[N];  
 initArr(a,N,1);  
 **for** (**int** i = 0; i < N; ++i) {  
 printf(**"%d "**,a[i]);  
 }  
 printf(**"\n Линейный поиск\n икомый элемент 4 index: %d\n"**,lineSearch(a,N,4));  
 printf(**"Среднее число сранений: %d Наихудшее число сравнений: %d\n"**,getAverageStep(a,N,&lineSearch),getBadStep(a,N,&lineSearch));  
 printf(**"Быстрый линейный поиск\n икомый элемент 4 index: %d\n"**,fastLineSearch(a,N,4));  
 printf(**"Среднее число сранений: %d Наихудшее число сравнений: %d\n"**,getAverageStep(a,N,&fastLineSearch),getBadStep(a,N,&fastLineSearch));  
 printf(**"Быстрый линейный поиск в упорядоченном\n икомый элемент 4 index: %d\n"**,sortfastLineSearch(a,N,4));  
 printf(**"Среднее число сранений: %d Наихудшее число сравнений: %d\n"**,getAverageStep(a,N,&sortfastLineSearch),getBadStep(a,N,&sortfastLineSearch));  
 printf(**"Бинарный поиск\n икомый элемент 4 index: %d\n"**,binSearch(a,N,4));  
 printf(**"Среднее число сранений: %d Наихудшее число сравнений: %d\n"**,getAverageStep(a,N,&binSearch),getBadStep(a,N,&binSearch));  
 printf(**"Блочный поиск\n икомый элемент 4 index: %d\n"**,blockSearch(a,N,4));  
 printf(**"Среднее число сранений: %d Наихудшее число сравнений: %d\n"**,getAverageStep(a,N,&blockSearch),getBadStep(a,N,&blockSearch));  
 **return** 0;

}

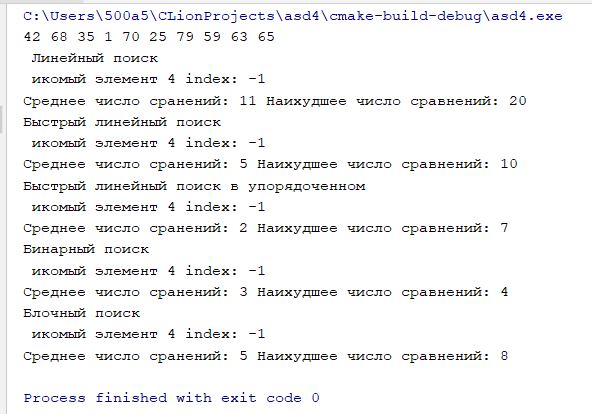
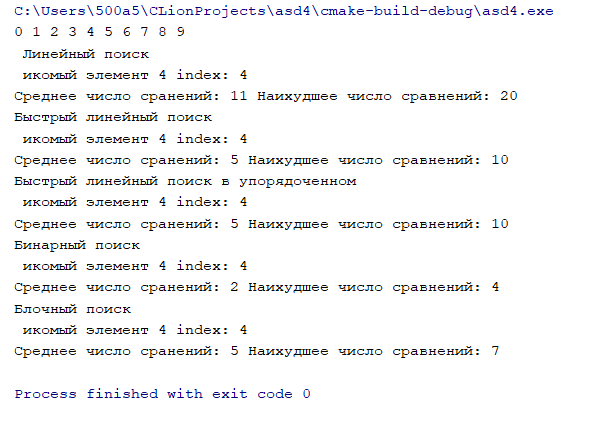


Таблица 12

**Максимальное количество операций сравнения**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Алгоритмы  поиска | Количество элементов в массиве | | | | | | | | |
| 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 |
| Линейный (неупорядоченный массив) | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 |
| Быстрый линейный (неупорядоченный массив) | 51 | 101 | 151 | 201 | 251 | 301 | 351 | 401 | 451 |
| Быстрый линейный (упорядоченный массив) | 51 | 101 | 151 | 201 | 251 | 301 | 351 | 401 | 451 |
| Бинарный (упорядоченный массив) | 6 | 7 | 8 | 8 | 8 | 9 | 9 | 9 | 9 |
| Блочный (упорядоченный массив) | 15 | 20 | 25 | 29 | 32 | 35 | 38 | 40 | 43 |

Таблица 13

**Среднее количество операций сравнения**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Алгоритмы  поиска | Количество элементов в массиве | | | | | | | | |
| 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 |
| Линейный (неупорядоченный массив) | 51 | 101 | 151 | 201 | 251 | 301 | 351 | 401 | 451 |
| Быстрый линейный (неупорядоченный массив) | 26 | 51 | 76 | 101 | 126 | 151 | 176 | 201 | 226 |
| Бстрый линейный (упорядоченный массив) | 26 | 51 | 76 | 101 | 126 | 151 | 176 | 201 | 226 |
| Бинарный (упорядоченный массив) | 6 | 7 | 8 | 8 | 8 | 9 | 9 | 9 | 9 |
| Блочный (упорядоченный массива) | 8 | 10 | 13 | 15 | 16 | 18 | 19 | 20 | 22 |

**Таблица**

**Заголовочный файл**

**“** **table\_array.h”**

#ifndef **ASD4\_TABLE\_ARRAY\_H**#define **ASD4\_TABLE\_ARRAY\_H**#define **TABL\_SIZE** 3  
  
*//Все в порядке***static const short** TABL\_OK = 0;  
  
*//Таблица переполнена***static const short** TABL\_FULL = 1;  
  
*//Таблица пуста***static const short** TABL\_EMPTY = 2;  
  
*//Ключ не найден***static const short** TABL\_NOT\_KEY = 3;  
  
*//Переменная для ошибок***extern short** TABL\_ERROR;  
  
**typedef struct**{  
  
 **short** key;  
  
 **int** data;  
  
}elementTabl;  
  
**typedef struct**{  
  
 elementTabl buf[**TABL\_SIZE**];  
  
 **short** Uk;  
  
}tabl\_array;  
  
*//Инициализация таблицы***void** initTable(tabl\_array \*tabl);  
  
*//Поместить элемент в таблицу***void** putTable(tabl\_array \*tabl, elementTabl el);  
  
*//Взять элемент из таблицы***void** getTable(tabl\_array \*tabl, **short** key, elementTabl \*el);  
  
*//Таблица полная?***int** fullTable(tabl\_array t);  
  
*//Таблица пустая?***int** emptyTable(tabl\_array t);  
  
#endif *//ASD4\_TABLE\_ARRAY\_H*

**Модуль**

**“** **table\_array.с”**

#include **"table\_array.h"  
short** TABL\_ERROR;  
  
*//Инициализация таблицы***void** initTable(tabl\_array \*tabl)  
{  
  
 tabl->Uk = 0;  
 TABL\_ERROR = TABL\_OK;  
  
}  
  
*//Поместить элемент в таблицу***void** putTable(tabl\_array \*tabl, elementTabl el)  
{  
  
 **if**(fullTable(\*tabl))  
 TABL\_ERROR = TABL\_FULL;  
 **else**{  
  
 **for** (**int** i = 0; i < tabl->Uk; ++i)  
 **if**(tabl->buf[i].key == el.key){  
  
 tabl->buf[i].data = el.data;  
 **return**;  
  
 }  
  
 tabl->buf[tabl->Uk] = el;  
 ++tabl->Uk;  
 TABL\_ERROR = TABL\_OK;  
  
 }  
  
}  
  
  
*//Взять элемент из таблицы***void** getTable(tabl\_array \*tabl, **short** key, elementTabl \*el)  
{  
  
 **if**(emptyTable(\*tabl))  
 TABL\_ERROR = TABL\_EMPTY;  
 **else**{  
  
 **for** (**int** i = 0; i < tabl->Uk; ++i) {  
  
 **if**(tabl->buf[i].key == key){  
  
 \*el = tabl->buf[i];  
 --tabl->Uk;  
 tabl->buf[i] = tabl->buf[tabl->Uk];  
 TABL\_ERROR = TABL\_OK;  
 **return**;  
  
 }  
  
 }  
  
 TABL\_ERROR =TABL\_NOT\_KEY;  
  
 }  
  
  
}  
  
*//Таблица полная?***int** fullTable(tabl\_array t)  
{  
  
 **return** t.Uk == **TABL\_SIZE**;  
  
}  
  
*//Таблица пустая?***int** emptyTable(tabl\_array t)  
{  
  
 **return** t.Uk == 0;  
  
}

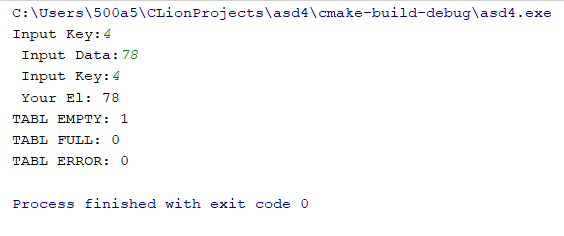
**“Main.с”**

#include **<stdio.h>**#include **<stdlib.h>**

#include **"table\_array.h"**  
  
**int** main() {  
 tabl\_array tabl;  
  
elementTabl el;  
**short** key;  
initTable(&tabl);  
printf(**"Input Key: "**);  
scanf(**"%hd"**, &(el.key));  
printf(**"Input Data: "**);  
scanf(**"%d"**, &(el.data));  
putTable(&tabl, el);  
  
printf(**"Input Key: "**);  
scanf(**"%hd"**, &(key));  
getTable(&tabl, key, &el);  
printf(**"Your El: %d\n"**, el.data);  
printf(**"TABL EMPTY: %d\n"**, emptyTable(tabl));  
  
printf(**"TABL FULL: %d\n"**, fullTable(tabl));  
printf(**"TABL ERROR: %d\n"**, TABL\_ERROR);

**return** 0;

}



**Приоритетная очередь**

**Заголовочный файл**

**“** **FIFO\_prior\_array.h”**

#ifndef **ASD4\_FIFO\_PRIOR\_ARRAY\_H**#define **ASD4\_FIFO\_PRIOR\_ARRAY\_H***//Размер очереди*#define **FIFO\_P\_SIZE** 3  
  
*//Очередь в порядке***static const short** FIFO\_P\_OK = 0;  
  
*//Очередь переполнена***static const short** FIFO\_P\_FULL = 1;  
  
*//Очередь пуста***static const short** FIFO\_P\_EMPTY = 2;  
  
*//Переменная ошибки***extern short** FIFO\_P\_ERROR;  
  
*//Базовый тип***typedef int** t\_baseTypeFIFOP;  
  
*//Тип для приоритета***typedef int** priorityType;  
  
*//Тип эелемнта очереди***typedef struct** {  
  
 t\_baseTypeFIFOP data;  
  
 priorityType priority;  
  
}elementFP;  
  
**typedef struct**{  
  
 elementFP Buf[**FIFO\_P\_SIZE**];  
  
 **short** Uk1;  
 **short** Uk2;  
  
 **short** n;  
  
} FIFOP;  
  
*//Операция инициализации***void** InitFIFOP(FIFOP \*q);  
  
*//Если очередь не переполнена, то включает элемент E в очередь q  
//с приоритетом priority***void** PutFIFOP(FIFOP \*q, t\_baseTypeFIFOP E, priorityType priority);  
  
*//Если очередь не пуста, то перемещает элемент очереди  
//с наибольшим приоритетом q по адресу E***void** GetFIFOP(FIFOP \*q, t\_baseTypeFIFOP \*E);  
  
*//Возвращает 1 если очередь q пуста  
//Иначе 0***short** EmptyFIFOP(FIFOP \*q);  
  
*//Возвращает 1 если очередь q полная  
//Иначе 0***short** FullFIFOP(FIFOP \*q);  
  
#endif *//ASD4\_FIFO\_PRIOR\_ARRAY\_H*

**Модуль**

**“** **FIFO\_prior\_array.с”**

#include **"FIFO\_prior\_array.h"  
  
  
short** FIFO\_P\_ERROR;  
  
*//Операция инициализации***void** InitFIFOP(FIFOP \*q)  
{  
  
 q->Uk1 = 0;  
 q->Uk2 = 0;  
 q->n = 0;  
 FIFO\_P\_ERROR = FIFO\_P\_OK;  
  
}  
  
*//Если очередь не переполнена, то включает элемент E в очередь q  
//с приоритетом priority***void** PutFIFOP(FIFOP \*q, t\_baseTypeFIFOP E, priorityType priority)  
{  
  
 **if**(FullFIFOP(q))  
 FIFO\_P\_ERROR = FIFO\_P\_FULL;  
 **else**{  
  
 q->Buf[q->Uk1].data = E;  
 q->Buf[q->Uk1].priority = priority;  
  
 q->Uk1 = (q->Uk1 + 1) % **FIFO\_P\_SIZE**;  
  
 ++q->n;  
 FIFO\_P\_ERROR = FIFO\_P\_OK;  
  
 }  
  
}  
  
  
*//Если очередь не пуста, то перемещает элемент очереди  
//с наибольшим приоритетом q по адресу E***void** GetFIFOP(FIFOP \*q, t\_baseTypeFIFOP \*E)  
{  
  
 **if**(EmptyFIFOP(q))  
 FIFO\_P\_ERROR = FIFO\_P\_EMPTY;  
 **else**{  
  
 **int** position = q->Uk2;  
 **int** positionMaxP = q->Uk2;  
  
 **while** (position != q->Uk1){  
  
 **if**(q->Buf[position].priority > q->Buf[positionMaxP].priority)  
 positionMaxP = position;  
  
 position = (position + 1) % **FIFO\_P\_SIZE**;  
  
 }  
  
 \*E = q->Buf[positionMaxP].data;  
  
 q->Buf[positionMaxP] = q->Buf[q->Uk2];  
  
 q->Uk2 = (q->Uk2 + 1) % **FIFO\_P\_SIZE**;  
  
 --q->n;  
 FIFO\_P\_ERROR = FIFO\_P\_OK;  
  
 }  
  
}  
  
*//Возвращает 1 если очередь q пуста  
//Иначе 0***short** EmptyFIFOP(FIFOP \*q)  
{  
  
 **return** q->n == 0;  
  
}  
  
*//Возвращает 1 если очередь q полная  
//Иначе 0***short** FullFIFOP(FIFOP \*q)  
{  
  
 **return** q->n == **FIFO\_P\_SIZE**;  
  
}

**“Main.с”**

#include **<stdio.h>**#include **<stdlib.h>**

#include **"FIFO\_prior\_array.h"**

**int** main() {  
FIFOP fifo;  
**int** el;  
InitFIFOP(&fifo);  
printf(**"Input El: "**);  
scanf(**"%d"**, &el);  
PutFIFOP(&fifo, el, el);  
printf(**"Input El: "**);  
scanf(**"%d"**, &el);  
PutFIFOP(&fifo, el, el);  
GetFIFOP(&fifo, &el);  
printf(**"Your El: %d\n"**, el);  
printf(**"FIFO EMPTY: %d\n"**, EmptyFIFOP(&fifo));  
printf(**"FIFO FULL: %d\n"**, FullFIFOP(&fifo));  
printf(**"FIFO ERROR: %d\n"**, FIFO\_P\_ERROR);

**return** 0;

}

