**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 автоматизированных систем

Лабораторная работа №4

дисциплина: Алгоритмы и Структуры данных

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

Выполнил: ст. группы ПВ-21

Донцов Александр Алексеевич

Проверил: Синюк Василий Григорьевич

Белгород 2018

**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.

**Реализация**

#include "Search.h"

#include "windows.h"

#include <math.h>

#include <stdio.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;

}

//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;

}

//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;

}

// заполнение массива

void initArr(int \*arr, int n, short t) {

srand(50);

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

switch (t) {

case ORDERLY: arr[i] = i; break;

case DISORDERED:

arr[i] = rand() % 100 + 1;

break;

}

}

}

// функция - Эксперимент - эффективности сортировок

void experiment(void) {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

// тестовые массивы

int a[MAX\_SIZE\_ARR];

//цикл отвечает за тип массива (упорядоченный/ неупорядоченный)

for (short t = 0; t < 2; t++) {

switch (t) {

case ORDERLY:

printf("--------------Упорядоченный массив--------------\n");

break;

case DISORDERED:

printf("--------------Непорядоченный массив-------------\n");

break;

}

//цикл отвечает за изменение длинный массива

for (int size = 50; size <= 450; size += 50) {

printf("--------------Размер массива - %d---------------\n", size);

switch (t) {

case DISORDERED:

initArr(a, size, DISORDERED);

printf("Линейный поиск\n");

printf(" Наихудшее время: %d\n", getBadStep(a, size, &lineSearch));

printf(" Среднее время время: %d\n", getAverageStep(a, size, &lineSearch));

printf("Быстрый линейный поиск\n");

printf(" Наихудшее время: %d\n", getBadStep(a, size, &fastLineSearch));

printf(" Среднее время время: %d\n", getAverageStep(a, size, &fastLineSearch));

break;

case ORDERLY:

initArr(a, size, ORDERLY);

printf("Быстрый линейный поиск\n");

printf(" Наихудшее время: %d\n", getBadStep(a, size, &fastLineSearch));

printf(" Среднее время время: %d\n", getAverageStep(a, size, &fastLineSearch));

printf("Бинарный поиск\n");

printf(" Наихудшее время: %d\n", getBadStep(a, size, &binSearch));

printf(" Среднее время время: %d\n", getAverageStep(a, size, &binSearch));

printf("Блочный поиск\n");

printf(" Наихудшее время: %d\n", getBadStep(a, size, &blockSearch));

printf(" Среднее время время: %d\n", getAverageStep(a, size, &blockSearch));

break;

}

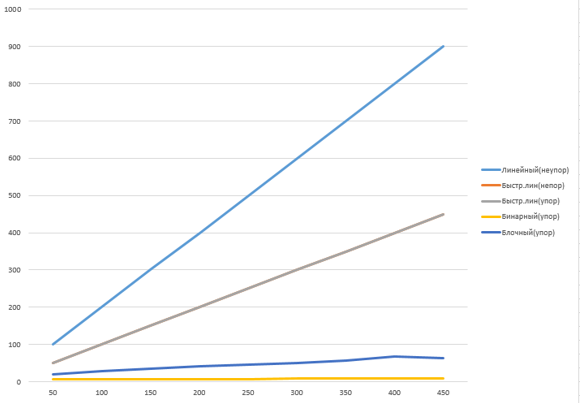
}

}

}

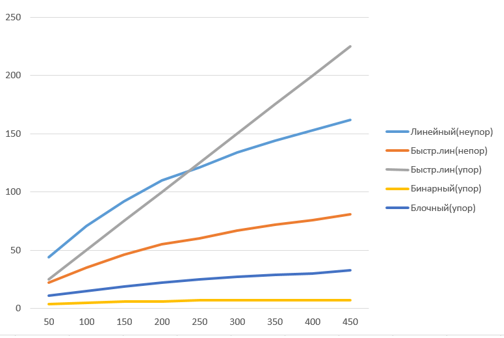
**Максимальное количество операций сравнения**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Алгоритмы  поиска | Количество элементов в массиве | | | | | | | | |
| 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 |
| Линейный (неупорядоченный массив) | 96 | 200 | 300 | 398 | 448 | 574 | 656 | 794 | 822 |
| Быстрый линейный (неупорядоченный массив) | 48 | 100 | 150 | 199 | 224 | 287 | 328 | 397 | 411 |
| Быстрый линейный (упорядоченный массив) | 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 |
| Бинарный (упорядоченный массив) | 6 | 7 | 8 | 8 | 8 | 9 | 9 | 9 | 9 |
| Блочный (упорядоченный массив) | 15 | 20 | 25 | 29 | 32 | 35 | 38 | 40 | 43 |



**Среднее количество операций сравнения**

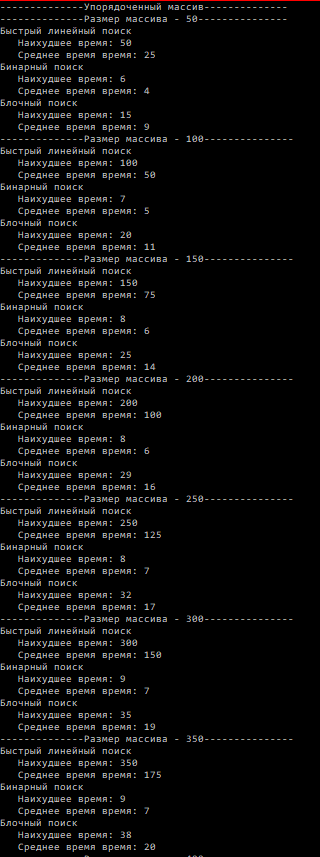
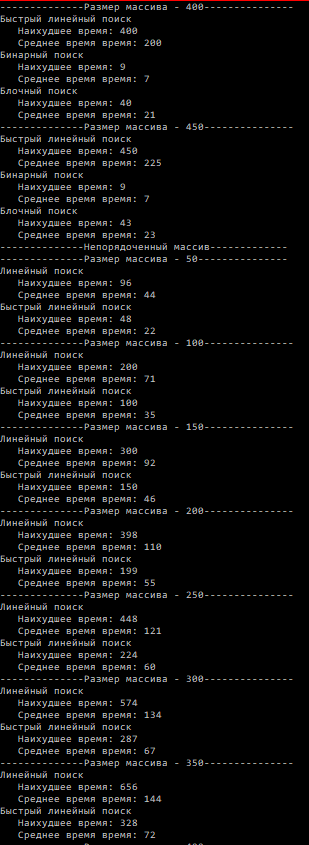
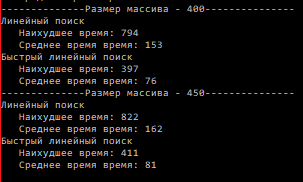
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Алгоритмы  поиска | Количество элементов в массиве | | | | | | | | |
| 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 |
| Линейный (неупорядоченный массив) | 44 | 71 | 92 | 110 | 121 | 134 | 144 | 153 | 162 |
| Быстрый линейный (неупорядоченный массив) | 22 | 35 | 46 | 55 | 60 | 67 | 72 | 76 | 81 |
| Быстрый линейный (упорядоченный массив) | 25 | 50 | 75 | 100 | 125 | 150 | 175 | 200 | 225 |
| Бинарный (упорядоченный массив) | 4 | 5 | 6 | 6 | 7 | 7 | 7 | 7 | 7 |
| Блочный (упорядоченный массива) | 9 | 11 | 14 | 16 | 17 | 19 | 20 | 21 | 23 |



**Линейный поиск:** O(N)

**Бинарный поиск:** O(log 2 N)

**Блочный поиск:** O(2\*)

**Таблица**

**Файл Table\_Array.cpp**

#include "Table\_Array.h"

#include <stdio.h>

//Инициализация таблицы

void initTableArray(TableArray \*T) {

T->uk = 0;

errorTableArray = tableOk;

}

//Включение элемента в таблицу

void putTAbleArray(TableArray \*T, elementTableArray E) {

if (isFullTAbleArray(T))//Если таблица переполнена

return;

int posElement;

if ((posElement = searchKey(T, E.key)) != -1)//Обновление данных

T->buf[posElement].date = E.date;

else {//Иначе включение элемента по указателю

T->buf[T->uk] = E;// Добавление данных

T->uk++;// Сдвиг указателя

}

}

//Исключение элемента из таблицы

void getTableArray(TableArray \*T, elementTableArray \*E, KeyBaseType key) {

if (isEmptyTableArray(T))//Если таблина пуста

return;

int posElement;

if ((posElement = searchKey(T, key)) != -1) {// Ecли такой ключ найден

E->date = T->buf[posElement].date;// Запись данных в переменную

E->key = T->buf[posElement].key;// Запись ключа в переменную

if (posElement != T->uk - 1) // Если найденный элемент - не последний

T->buf[posElement] = T->buf[T->uk - 1];// Запись на место найденного элемента последнего

T->uk--;// Сдвиг указателя

}

}

//Поиск элемента с данным ключом

int searchKey(TableArray \*T, KeyBaseType key) {

unsigned i;

for (i = 0; i < T->uk && T->buf[i].key != key;) //Для всех элементов таблицы выполнить поиск по ключу

i++;

if (i == T->uk) {// Если ключ не был найден

errorTableArray = tableNoKey;// Установка исключительной ситуации

return -1;// Возврат -1

}

return i;// Иначе возврат позиции элемента с таким ключом

}

//Предикат: Пуста ли таблица

int isEmptyTableArray(TableArray \*T) {

if (T->uk == 0) {

errorTableArray = tableEmpty;

return 1;

}

}

//Предикат: Переполнена ли таблица

int isFullTAbleArray(TableArray \*T) {

if (T->uk == SIZE\_TABLE\_ARRAY) {

errorTableArray = tableFull;

return 1;

}

return 0;

}

**Заголовочный файл Table\_Array.h**

#pragma once

#ifndef \_TABLE\_ARRAY\_

#define \_TABLE\_ARRAY\_

#define SIZE\_TABLE\_ARRAY 10

//исключительные ситуации

const int tableOk = 0;

const int tableEmpty = 1;

const int tableFull = 2;

const int tableNoKey = 3;

extern int errorTableArray;//Переменная ошибок

/\*Описание типов\*/

typedef int BaseType; // Базовый тип данных таблицы

typedef int KeyBaseType; // Базовый тип ключа таблицы

typedef struct {

KeyBaseType key;//Поле ключа

BaseType date;//Поле данных

} elementTableArray;

typedef struct {

elementTableArray buf[SIZE\_TABLE\_ARRAY];// Буфер таблицы

unsigned uk;//Указатель

} TableArray;

void initTableArray(TableArray \*T);// Инициализация таблиц

void putTAbleArray(TableArray \*T, elementTableArray E);// Включение элемента в таблицу

void getTableArray(TableArray \*T, elementTableArray \*E, KeyBaseType key);// Исключение элемента из таблицы

int searchKey(TableArray \*T, KeyBaseType key);// Поиск элемента с данным ключом

int isEmptyTableArray(TableArray \*T);// Предикат: Пуста ли таблица

int isFullTAbleArray(TableArray \*T);// Предикат: Переполнена ли таблица

#endif