МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ИНСТИТУТ НЕПРЕРЫВНОГО И ДИСТАНЦИОННОГО ОБРАЗОВАНИЯ

|  |
| --- |
| КАФЕДРА компьютерных технологий и программной инженерии |

ОЦЕНКА

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| «ХЕШИРОВАНИЕ ДАННЫХ» |
| по дисциплине: Структуры и алгоритмы обработки данных |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | Z9431 |  |  |  | Андреев Д.И. |
|  | номер группы |  | подпись, дата |  | инициалы, фамилия |
| Студенческий билет № | 2019/3781 | |  |  |  |

Санкт-Петербург 2020

Оглавление

[1. Цель работы 3](#_Toc55577180)

[2. Вариант задания 3](#_Toc55577181)

[3. Листинг программы 3](#_Toc55577182)

[4. Контрольный пример 9](#_Toc55577183)

[5. Количество выполненных операций сравнения и перестановок элементов массива 10](#_Toc55577185)

[6. Временная и пространственная сложность алгоритма 10](#_Toc55577186)

[7. Выводы 11](#_Toc55577187)

1. Цель работы

Целью работы является изучение алгоритмов внутренней сортировки и получение практических навыков их использования, и анализа их сложности.

1. Вариант задания

Вариант 26.

Использовать неупорядоченный массив A, содержащий n целочисленных элементов.

Дополнительно в программе должны быть реализованы следующие функции:

1. Поиск элемента либо по его порядковой позиции, либо по его содержимому;
2. Добавление/удаление элемента с последующей пересортировкой последовательности;
3. В программе должен быть реализован подсчет количества сравнений и перестановок, при осуществлении сортировки.

Задание на лабораторную работу: найти k-ое по порядку число среди элементов массива.

Алгоритм сортировки: четно-нечетная

1. Листинг программы

int main()

{

setlocale(LC\_ALL, "Russian");

cout << "Лабораторная №5.\nАлгоритмы сортировки.\nВариант " << (81 % 28) + 1

<< "\n\n";

short command = -1;

int\* data = nullptr;

int size = 0;

//Для тестов, чтобы можно было тестировать на случайном размере

cout << "Введите размер массива: ";

cin >> size;

while (!cin || !(size > 0)) {

cout << "Введено неверное значение. Повторите ввод: ";

std::cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

cin >> size;

}

data = clearData(data, size, false);

do {

command = getCommand();

switch (command)

{

case 0:

randGenerate(data, size);

break;

case 1:

\_\_sort(data, size);

break;

case 2:

showData(data, size);

break;

case 3:

data = clearData(data, size);

break;

case 4:

findByPos(data, size);

break;

case 5:

erase(data, size);

break;

case 6:

data = insert(data, size);

break;

case 7:

showHint();

break;

case 8:

break;

default:

cout << "Неизвестная команда\n";

break;

}

} while (command != 8);

delete[] data;

return 0;

}

void \_\_sort(int\* m, const int size, bool showInfo)

{

//Количество перестановок и сравнений

int swapNum = 0;

int compNum = 0;

//Флаги об окончании сортировки на четных и

//нечетных обходах соответственно

bool evenSwap = false;

bool oddSwap = false;

//Побитовый оператор должен работать быстрее логического,

//но из-за компилятора это скорее всего не так

while (!(evenSwap & oddSwap)) {

//В цикле тоже имеется операция сравнения

++compNum;

//На каждой итерации флаги должны быть true

//Если на данной итерации чет или нечет обход будет иметь перестановку

//Поднимается соответствующий флаг и алгоритм продолжает работу

evenSwap = true;

oddSwap = true;

//Нечетный обход

for (int i = 0; i + 1 < size; i += 2) {

if (m[i] > m[i + 1]) {

//Уже сравнение самих значений

++compNum;

int temp = m[i];

m[i] = m[i + 1];

m[i + 1] = temp;

//И один обмен

++swapNum;

oddSwap = false;

}

}

//Четный обход (все тоже самое, что и для нечетного, кроме флага)

for (int i = 1; i + 1 < size; i += 2) {

if (m[i] > m[i + 1]) {

++compNum;

int temp = m[i];

m[i] = m[i + 1];

m[i + 1] = temp;

++swapNum;

evenSwap = false;

}

}

}

if (showInfo) {

cout << "Количество сравнений в последней сортировке: " << compNum << '\n';

cout << "Количество перестановок в последней сортировке: " << swapNum << '\n';

}

}

short getCommand()

{

short com;

cout << "Введите команду для продолжения (8 - показать все команды): ";

cin >> com;

while (!cin) {

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

cout << "Введите корректное значение (9 для выхода):";

cin >> com;

}

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

return com;

}

void showHint()

{

cout << "0 - Сгенерировать массив случайных чисел\n";

cout << "1 - Сортировать массив\n";

cout << "2 - Вывести массив на экран\n";

cout << "3 - Очистить массив\n";

cout << "4 - Показать элемент в позиции\n";

cout << "5 - Удалить элемент из массива\n";

cout << "6 - Вставить элемент в массив\n";

cout << "7 - Добавить элементы в массив\n";

cout << "8 - Показать все команды\n";

cout << "9 - Выход\n";

}

//Производит поиск элемента searchElem по значению в массиве m размера size.

//Если элемент не найдет, поднимает флаг notFindFlag, возвращает 0

//Иначе возвращает позицию элемента в массиве (бинарный поиск)

int findByData(const int\* m, const int size, int searchElem, bool& notFindFlag)

{

//Левая, правая и центральная позиции в массиве соответственно

int left = 0;

int right = size;

int mid = -1;

while (right != left) {

mid = (left + right) / 2;

if (m[mid] == searchElem)

return mid;

else if (m[mid] < searchElem)

left = mid + 1;

else if (m[mid] > searchElem)

right = mid;

}

notFindFlag = true;

return 0;

}

//Выводит элемент в консоль по переданной пользователем позиции

void findByPos(const int\* m, const int size)

{

int pos = 0;

cout << "Введите позицию искомого элемента: ";

cin >> pos;

if (!cin) {

cout << "Введено неверное значение\n";

std::cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

return;

}

cout << "Элемент в позиции " << pos << ": " << m[pos] << '\n';

}

//Вставляет элементы в массив. Если есть пустые места (0 - считается пустой позицией),

//вставит в нее, иначе делает resize массива и вставляет в его конец.

//Возвращает указатель на начало массива

int\* insert(int\* m, int& size, bool newElements)

{

int\* newData = nullptr;

int elem = 0;

cout << "Введите значение элемента для вставки: ";

while (cin.peek() != '\n')

{

cin >> elem;

if (!cin) {

std::cin.clear();

cin.ignore(sizeof(char));

continue;

}

//Поиск пустой позиции

bool notFindFlag = false;

int pos = findByData(m, size, 0, notFindFlag);

//Если пустая позиция есть - вставить в нее

if (!notFindFlag && !newElements) {

m[pos] = elem;

newData = m;

//сортировать массив нужно на каждой итерации,

//так как бинарный поиск работает только с отсортированным массивом

\_\_sort(m, size, false);

continue;

}

//Иначе увеличить размер массива

//(создается новый массив, копируются все элементы из старого в новый,

//старый удаляется, новый элемент вставляется в конец нового массива)

int oldSize = size++;

newData = new int[size];

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

newData[i] = m[i];

}

newData[size - 1] = elem;

delete[] m;

//Сортировка после вставки

m = newData;

}

\_\_sort(newData, size);

return newData;

}

//Удаляет элемент по значению. Если элемента в массиве нет- ничего не делает

void erase(int\* m, const int size)

{

int elem = 0;

cout << "Введите значение элемента для удаления: ";

cin >> elem;

if (!cin) {

cout << "Введено неверное значение\n";

std::cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

return;

}

//Поиск элемента, переданного пользователем

bool notFindFlag = false;

int pos = findByData(m, size, elem, notFindFlag);

//Если такого в массиве нет - ничего не делать

if (notFindFlag) {

cout << "Такого элемента в массиве нет\n";

return;

}

//Иначе удалить элемент

m[pos] = 0;

//Сортировна после удаления

\_\_sort(m, size);

}

//Заполняет массив случайными значениями и сортирует

void randGenerate(int\* m, const int size)

{

int dev = (size > 1000) ? size : 1000;

srand(clock());

for (int i = 0; i < size; ++i)

m[i] = rand() % dev + 1;

\_\_sort(m, size);

}

void showData(const int\* m, const int size)

{

cout << " Элементы массива:\n";

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

cout << m[i] << " ";

}

cout << '\n';

}

//Функция выполняет очистку массива или изменение его размера

//Выбор предоставляется пользователю

//Можно передать другой size и изменить размер без пользователя

int\* clearData(int\* m, int& size, bool needResize)

{

if (needResize) {

cout << "Хотите задать новый размер массива? (y/n): ";

char com;

cin >> com;

if (com == 'y') {

cout << "Введите новый размер массива: ";

cin >> size;

if (!cin) {

cout << "Введено неверное значение\n";

std::cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

}

}

delete[] m;

m = new int[size] {0};

return m;

}

Основные алгоритмы:

Алгоритм сортировки:

void \_\_sort(int\* m, const int size, bool showInfo)

{

//Количество перестановок и сравнений

int swapNum = 0;

int compNum = 0;

//Флаги об окончании сортировки на четных и

//нечетных обходах соответственно

bool evenSwap = false;

bool oddSwap = false;

//Побитовый оператор должен работать быстрее логического,

//но из-за компилятора это скорее всего не так

while (!(evenSwap & oddSwap)) {

//В цикле тоже имеется операция сравнения

++compNum;

//На каждой итерации флаги должны быть true

//Если на данной итерации чет или нечет обход будет иметь перестановку

//Поднимается соответствующий флаг и алгоритм продолжает работу

evenSwap = true;

oddSwap = true;

//Нечетный обход

for (int i = 0; i + 1 < size; i += 2) {

if (m[i] > m[i + 1]) {

//Уже сравнение самих значений

++compNum;

int temp = m[i];

m[i] = m[i + 1];

m[i + 1] = temp;

//И один обмен

++swapNum;

oddSwap = false;

}

}

//Четный обход (все тоже самое, что и для нечетного, кроме флага)

for (int i = 1; i + 1 < size; i += 2) {

if (m[i] > m[i + 1]) {

++compNum;

int temp = m[i];

m[i] = m[i + 1];

m[i + 1] = temp;

++swapNum;

evenSwap = false;

}

}

}

if (showInfo) {

cout << "Количество сравнений в последней сортировке: " << compNum << '\n';

cout << "Количество перестановок в последней сортировке: " << swapNum << '\n';

}

}

Алгоритм поиска:

int findByData(const int\* m, const int size, int searchElem, bool& notFindFlag)

{

//Левая, правая и центральная позиции в массиве соответственно

int left = 0;

int right = size;

int mid = -1;

while (right != left) {

//Вычисление центрального элемента

mid = (left + right) / 2;

//Если центральный элемент равен искомому, завершить поиск

if (m[mid] == searchElem)

return mid;

//Если центральный элемент меньше искомого, сдвинуть левую границу в середину

else if (m[mid] < searchElem)

left = mid + 1;

//Если центральный элемент больше искомого, сдвинуть правую границу в середину

else if (m[mid] > searchElem)

right = mid;

}

//Если левая и правая границы сравнялись, а элемент все еще не найден –

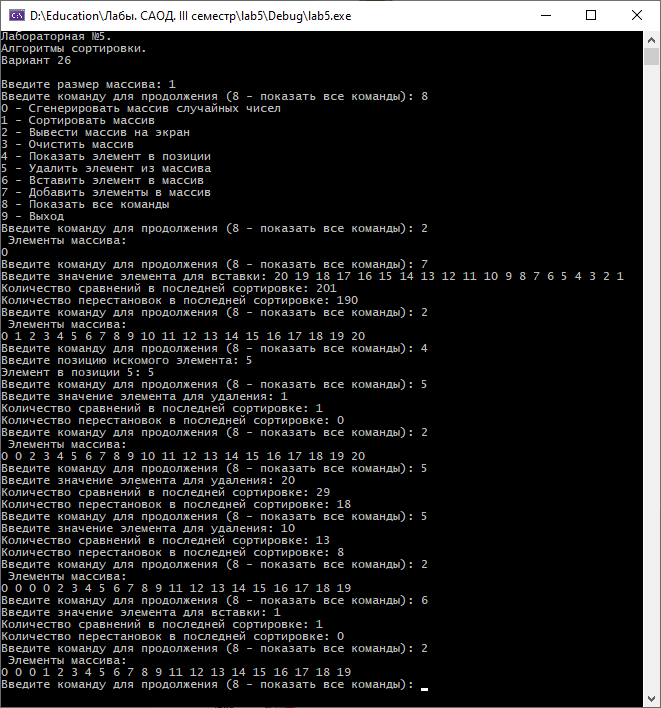
//такого элемента в массиве нет, завершить выполнение алгоритма

notFindFlag = true;

return 0;

}

1. Контрольный пример



1. Количество выполненных операций сравнения и перестановок элементов массива

Для массива из 20 элементов количество перестановок в самом худшем случае, когда элементы располагаются в порядке убывания – 190, количество сравнений в самом худшем случае – 201.

1. Временная и пространственная сложность алгоритма

Временная сложность:

На основе реализованного алгоритма можно сказать, что теоретическая временная сложность определяется следующим образом:

*Tsort = O(max(O(K1), O(3 \* n2 \* K2), O(2 \* n2 \* K3))) =*

*= O(max(O(1), O(n2), O(n2))) = O(n2),*

где *K1* – операции присвоения, имеющие временную сложность 1; *K2* – операция сравнения в цикле, имеющая в худшем случае временную сложность *n2*; *К3*– операция перестановки, имеющая в худшем случае временную сложность *n2.*

Пространственная сложность:

Алгоритм требует следующие данные:

- массив размерностью *n = sizeof(mass) / sizeof(uint\_16)*

- четыре переменные целого типа и одну логического

Пространственная сложность алгоритма определяется в этом случае следующим образом:

*V = n \* Cuint\_16 + 4 \* Cuint\_16 + 2 \* Cbool,*

где *Cuint\_16* – количество памяти, занимаемое целочисленной переменной беззнакового 16-тиразрядного целого типа, *Сbool* - количество памяти, занимаемое логической переменной.

Теоретическая пространственная сложность алгоритма сортировки:

*V(n) = O(v) = O(max(O(n \* Cuint\_16), (2 \* Cuint\_16), O(2 \* Cbool))) =*

*= O(max(O(n), O(1), O(1))) = O(n)*

1. Выводы

В рамках лабораторной работы были реализованы четно-нечетный алгоритм сортировки, а также бинарный алгоритм поиска, позволяющие сортировать массив элементов и искать в нем элементы по значению. Также была дана оценка временной и пространственной сложности алгоритма сортировки. Были реализованы вставка и удаление элементов в динамический массив.