

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МО ЭВМ**

**ОТЧЕТ**  
**по лабораторной работе №4**  
**по дисциплине «Алгоритмы и структуры данных»**  
**Тема: Алгоритмы сортировки.**

Студент гр. 9381

Преподаватель

\_\_\_\_\_  
\_\_\_\_\_

Птичкин С. А.

Фирсов М. А.

Санкт-Петербург

2020

### **Цель работы.**

Изучение и реализация алгоритмов сортировки. Изучение их особенностей, а также быстродействия и расхода памяти.

### **Задание.**

#### **Вариант 13.**

Пирамидальная сортировка.

### **Выполнение работы.**

Для применения некоторых функций были подключены заголовочные файлы `iostream`, для консольного ввода/вывода, `string`, для работы со строками, `ctype.h`, для определения принадлежности символов в входной строке, `fstream`, для файлового ввода/вывода, `sstream`, для использования строк-поточков, с возможностью заведомо проверить корректность ввода, и `algorithm`, для использования библиотечной функции сортировки `sort()`. Разработанный программный код можно посмотреть в приложении А.

### **Описание алгоритма и структуры хранения данных.**

Пирамида - бинарное дерево высоты  $k$ , в котором:

- 1) все узлы имеют глубину  $k$  или  $k-1$  - дерево сбалансированное.
- 2) при этом уровень  $k-1$  полностью заполнен, а уровень  $k$  заполнен слева направо
- 3) выполняется "свойство пирамиды": каждый элемент меньше, либо равен родителю.

Соответствие между геометрической структурой пирамиды как дерева и массивом устанавливается по следующей схеме:

- 1) в  $a[0]$  хранится корень дерева
- 2) левый и правый сыновья элемента  $a[i]$  хранятся, соответственно, в  $a[2i+1]$  и  $a[2i+2]$

Алгоритм построения пирамиды:

Начать построение пирамиды можно с  $a[k]...a[n]$ ,  $k = \lfloor \text{size}/2 \rfloor$ . Эта часть массива удовлетворяет свойству пирамиды, так как не существует индексов  $i, j$ :  $i = 2i+1$  ( или  $j = 2i+2$  ). Далее будем расширять часть массива, приведённую к пирамидальному виду, добавляя по одному элементу за шаг. Следующий элемент на каждом шаге добавления - тот, который стоит перед уже готовой частью.

Чтобы при добавлении элемента сохранялась пирамидальность, используем следующий алгоритм расширения пирамиды  $a[i+1]..a[n]$  на элемент  $a[i]$  влево:

- 1) Рассматриваем поддеревья слева и справа - в массиве это  $a[2i+1]$  и  $a[2i+2]$  и выбираем наибольшего из них.
- 2) Если этот элемент больше  $a[i]$  - меняем его с  $a[i]$  местами и идем к шагу 2, имея в виду новое положение  $a[i]$  в массиве. Иначе конец работы алгоритма.

Учитывая, что высота пирамиды  $h \leq \log n$ , алгоритм требует  $O(\log n)$  времени.

Алгоритм сортировки:

- 1) Берем верхний элемент пирамиды  $a[0]...a[n]$  (первый в массиве) и меняем с последним местами. Затем уменьшаем количество рассматриваемых элементов на 1 и далее рассматриваем массив  $a[0]...a[n-1]$ . Для превращения его в пирамиду снова применяем алгоритм построения пирамиды.
- 2) Повторяем шаг 1, пока обрабатываемая часть массива не уменьшится до одного элемента.

Построение пирамиды занимает  $O(n \log n)$  операций. Фаза сортировки занимает  $O(n \log n)$  времени:  $O(n)$  раз берется максимум и происходит просеивание бывшего последнего элемента. Плюсом является стабильность метода: среднее число пересылок  $(n \log n)/2$ , и отклонения от этого значения сравнительно малы. Пирамидальная сортировка не использует дополнительной памяти. Метод не является устойчивым: по ходу работы порядок элементов может измениться случайным образом. Поведение неестественно: частичная упорядоченность массива никак не учитывается.

### **Основные функции.**

#### **1) Функция main. int main()**

Функция не принимает никаких параметров. Данная функция предназначена для стартового диалога с пользователем. Здесь идёт выбор ввода данных, либо из консоли, либо из файла. За корректность введенных данных отвечает функция `input_num`. Ввод команды 1 вызывает функцию консольного ввода, команда 2 - файлового. Возвращаемые значения этих функций определяют, будет ли цикл продолжаться с возможностью ввести новые данные, либо программа завершится.

## **2) Функция `file_input`. `int file_input()`**

Функция не принимает никаких параметров. В начале объявляются все необходимые для работы переменные, выделяется память под имя файла. Затем считывается имя файла, и файл открывается при корректном имени. Далее из файла считывается строка. После этого вызывается функция считывания массива из строки `mass_from_string`. Затем файл закрывается и вызывается функция `data_analis`, куда передаётся считанный массив и число его элементов. Функция возвращает то же значение, что и `data_analis`.

## **3) Функция `console_input`. `int console_input()`**

Функция не принимает никаких параметров. В начале объявляются все необходимые для работы переменные. Затем из консоли считывается строка, куда пользователь должен ввести элементы массива через пробел. После этого вызывается функция считывания массива из строки `mass_from_string`. Затем вызывается `data_analis`, куда передаётся считанный массив и число его элементов. Функция возвращает то же значение, что и `data_analis`.

## **4) `int *mass_from_string(string &input_string, int* size_mass)`**

Функция принимает на вход строку и указатель на переменную, куда будет записан размер массива. Сначала проверяется что строка не пустая и что в ней только числа и пробелы. Если всё корректно, то данная строка записывается в поток `myStream` типа `stringstream`. Объявляется массив с буферным количеством элементов, при переполнении буфер увеличивается на 10, выделяется память под массив увеличенного размера, куда копируются все предыдущие элементы. Предыдущий массив очищается. Таким образом в цикле `while` из потоковой строки `myStream` считываются элементы массива, пока строка не закончится. Функция возвращает указатель на считанный массив, а также записывает количество элементов в `size_mass`.

### **5) int data\_analis(int\* data\_mass, int count\_of\_elem)**

Функция принимает на вход массив целых чисел и количество элементов в нём. Сначала выводится исходный порядок элементов. Затем создаются две копии массива, которые будут отсортированы пирамидальной сортировкой и библиотечной функцией `sort`. После этого вызывается функция `heapSort`, которая сортирует первую копию массива, по ходу сортировки выводятся промежуточные данные, а затем и результат. Дополнительно вызывается сортировка библиотечной функцией `sort` второй копии массива, результат также выводится. Затем поэлементно сравниваются массивы, для проверки их идентичности. В конце пользователь попадает в меню, где выбирает сохранить ли полученные результаты в файл, продолжить или завершить программу. При записи вызывается функция `data_save`, куда передаётся исходный и отсортированный массив с количеством элементов. Затем память, выделенная под массивы очищается. Функция возвращает 1 для продолжения работы, 0 - для завершения.

### **6) void data\_save(int\* data\_mass, int\* sort\_data\_mass, int count\_of\_elem)**

Функция принимает на вход массивы целых чисел, первый это исходный, второй уже отсортированный. Сначала пользователю предлагается назвать имя файла для записи. Затем файл открывается и в него последовательно записываются элементы первого массива, а затем и второго. Файл закрывается и функция ничего не возвращает.

### **7) void heapSort(int\* data\_mass, int n)**

Основная функция, выполняющая пирамидальную сортировку. Принимает на вход массив и количество элементов в нём. Сначала вызывается вспомогательная функция `heapify()`, которая приводит массив к пирамидальному виду. После данного преобразования на первой позиции

оказывается наибольший элемент массива, который меняется местами с минимальным, находящимся в конце массива. После каждой итерации мы учитываем на 1 элемент меньше, а после снова вызываем функцию выравнивания `heapify`, для уменьшенного диапазона элементов. Процедура продолжается, пока количество элементов не станет 0. Также в процессе выводятся промежуточные результаты. Функция ничего не возвращает.

#### **8) void heapify(int\* data\_mass, int n, int i)**

Функция предназначена для преобразования массива к пирамидальному виду. На вход подаётся массив, количество элементов и индекс узла пирамиды, относительно которого производится выравнивание. Для выравнивания в данном узле, необходимо, чтобы значение в корне было больше значений в поддеревьях. Чтобы получить индексы правого и левого поддерева в массиве необходимо вычислить значения  $2*i+1$  и  $2*i+2$ . При этом идёт проверка выхода за границу массива. Если один из элементов поддерева больше корня, они меняются местами. При этом выводятся промежуточные данные, а именно в каком узле происходит выравнивание и какие элементы меняются местами. Функция ничего не возвращает.

#### **9) void print\_mass(int\* mass, int count\_of\_elem)**

Функция печати массива на экран. Принимает на вход сам массив и количество элементов. В цикле печатает элементы массива через пробел и ничего не возвращает.

#### **10) Функция input\_num. int input\_num(string message)**

Функция принимает на вход сообщение, выводимое пользователю. Функция предназначена для корректного считывания числа из потока ввода. На вход принимается адрес строки с сообщением пользователю, что ему делать. Объявляется переменная для записи числа и выделяется буфер на 10 символов.

Затем из `cin` считывается 10 символов в буфер. Далее в цикле из данного буфера считывается число функцией `sscanf`. Пока функция не вернёт 1 - количество верно считанных аргументов, ввод не прекратится. Когда наконец число считается, оно возвращается функцией. Память, выделенная под буфер очищается.

### Тестирование.

Результаты теста входных данных представлены в таблице 1.

Таблица 1- Результаты тестирования

№ Теста	Входные данные	Выходные данные
1	1 0 3 4 7 2 9 5	Исходный массив: 1 0 3 4 7 2 9 5 Отсортированный массив: 0 1 2 3 4 5 7 9
2	1 9 -5 2 0 4 -6 3	Исходный массив: 1 9 -5 2 0 4 -6 3 Отсортированный массив: -6 -5 0 1 2 3 4 9
3	1	Исходный массив: 1 Отсортированный массив: 1
4	-1 -2	Исходный массив: -1 -2 Отсортированный массив: -2 -1
5	4 4 4 4 4 4 4 4 4 4	Исходный массив: 4 4 4 4 4 4 4 4 4 4 Отсортированный массив: 4 4 4 4 4 4 4 4 4 4
6	1 2 4 g r w	Входная строка некорректна!

Сравнение результатов пирамидальной сортировки и сортировки функцией `sort` можно увидеть на рис.1

```
Сортировка завершена!
Итоговый отсортированный массив: -10 -1 0 2 3 4 6 9 9
Тот же массив, отсортированный функцией sort: -10 -1 0 2 3 4 6 9 9
```

Рис.1 - Сравнение результатов сортировок



### **Вывод.**

Был изучен алгоритм пирамидальной сортировки, его особенности реализации. Также выявлены быстродействие и затраты в памяти данного алгоритма. Была реализована программа, сортирующая массив целых чисел с использованием данного алгоритма.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл AiSD\_lab4.cpp:

```
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <fstream>
#include <string>
#include <ctype.h>
#include <sstream>

using namespace std;

int input num(string message) {
    int num = 0;
    cout << message << '\n';
    char* input = new char[10];
    fgets(input, 10, stdin);
    while (sscanf s(input, "%d", &num) != 1) {
        cout << "Ввод некорректный!\n" << message << '\n';
        fgets(input, 10, stdin);
    }
    delete[] input;
    return num;
}

void print mass(int* mass, int count of elem) { //выводит текущее состояние массива
    for (int i = 0; i < count of elem; i++) {
        cout << mass[i] << ' ';
    }
    cout << '\n';
}

void heapify(int* data_mass, int n, int i)
{
    int largest = i;
    // Инициализируем наибольший элемент как корень
    int l = 2 * i + 1; // левый = 2*i + 1
    int r = 2 * i + 2; // правый = 2*i + 2
    // Если левый дочерний элемент больше корня
    if ((l < n) && (data_mass[l] > data_mass[largest]))
        largest = l;
    // Если правый дочерний элемент больше, чем самый большой элемент на данный
    момент
    if ((r < n) && (data_mass[r] > data_mass[largest]))
        largest = r;
    // Если самый большой элемент не корень
    if (largest != i)
    {
        if (i == 0) {
            cout << "Выравниваем элементы на позициях 1, 2, 3 (" <<
data_mass[i] << " <-> " << data_mass[largest] << ") \n";
        }
    }
}
```

```

        else {
            cout << "Выравниваем элементы на позициях " << i + 1 << ", " <<
2 * i + 2 << ", " << 2 * i + 3 << " (" << data_mass[i] << " <-> " <<
data_mass[largest] << ") \n";
        }
        swap(data mass[i], data mass[largest]);
        print mass(data mass, n);
        // Рекурсивно преобразуем в пирамиду затронутое поддереву
        heapify(data_mass, n, largest);
    }
}

// Основная функция, выполняющая пирамидальную сортировку
void heapSort(int* data_mass, int n)
{
    // Построение пирамиды
    cout << "\nВыравнивание пирамиды:\n\n";
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(data mass, n, i);
    // Один за другим извлекаем элементы из пирамиды
    for (int i = n - 1; i >= 0; i--)
    {
        // Перемещаем текущий корень в конец
        swap(data mass[0], data mass[i]);
        cout << "Перемещаем наибольший элемент из вершины в конец
массива:\n";
        print mass(data mass, i+1);
        // вызываем процедуру heapify на уменьшенной пирамиде
        cout << "Уменьшаем интервал сортировки на 1:\n";
        print mass(data_mass, i);
        if (i > 1) {
            cout << "\nВыравнивание пирамиды:\n\n";
        }
        heapify(data_mass, i, 0);
    }
    cout << "Сортировка завершена!\n";
}

void data save(int* data mass, int* sort_data_mass, int count_of_elem) {
    char* file name = new char[256];
    cout << "Введите имя файла сохранения\n";
    cin >> file name;
    getchar(); //вытаскиваем символ переноса строки из потока
    fstream output file;
    output file.open(file_name, fstream::out | fstream::app); //открытие или
создание файла на запись
    output file << "Исходный массив: ";
    for (int i = 0; i < count of elem; i++) {
        output_file << data_mass[i] << ' '; //записываем исходный массив
    }
    output file << "\nОтсортированный массив: ";
    for (int i = 0; i < count of elem; i++) {
        output file << sort_data_mass[i] << ' '; //записываем
отсортированный массив
    }
    output file << "\n\n";
    output_file.close();
}

```

```

}

int data analis(int* data_mass, int count_of_elem) {
    cout <<
    "-----\nИсходный массив:\n";
    string dialog text = "\nВыберите дальнейшее действие:\n1 - сохранить результаты в файл и продолжить\n2 - "
    "сохранить результаты в файл и выйти\n3 - продолжить без сохранения\n4 - выйти из программы";
    int* sort mass = new int[count_of_elem]; //создаём копии массива, чтобы не изменять первоначальный
    int* heap sort mass = new int[count of elem];
    for (int i = 0; i < count of elem; i++) {
        heap sort mass[i] = data mass[i];
        sort mass[i] = data mass[i];
        cout << data_mass[i] << ' ';
    }
    cout << "\n\nПромежуточные данные: \n";
    heapSort(heap_sort_mass, count_of_elem); //вызываем функцию пирамидальной сортировки
    cout << "\nИтоговый отсортированный массив: ";
    print mass(heap sort_mass, count_of_elem);
    //вызов метода sort
    cout << "\nТот же массив, отсортированный функцией sort: ";
    sort(sort mass, &sort mass[count of elem]);
    print mass(sort_mass, count_of_elem);
    int equal = 1;
    for (int i = 0; i < count of elem; i++) {
        if (sort mass[i] != heap sort_mass[i]) { // проверка на одинаковый результат сортировки разными способами
            equal = 0;
            break;
        }
    }
    if (equal) {
        cout << "\nРезультаты совпадают\n";
    }
    else {
        cout << "Результаты не совпадают!\n";
    }
    cout << "\nНажмите ENTER, чтобы продолжить";
    getchar();
    while (1) {
        switch (input_num(dialog_text)) { //выбор дальнейших действий пользователем
            case 1: data save(data mass, heap sort mass, count of elem); delete data mass; delete sort mass; delete heap_sort_mass; return 1; break; // вывод элементов в порядке возрастания в файл
            case 2: data save(data mass, heap sort mass, count of elem); delete data mass; delete sort mass; delete heap sort mass; return 0; break; // вывод элементов в порядке возрастания в файл и выход
            case 3: delete data mass; delete sort_mass; delete heap_sort_mass; return 1; break; //продолжение работы
            case 4: delete data_mass; delete sort_mass; delete heap_sort_mass; return 0; break; //выход
            default: cout << "Команда не распознана!\n"; break;
        }
    }
}

```

```

    }
}
return 0;
}

int *mass from string(string &input_string, int* size_mass) { //функция считывания
массива из строки
    if (input_string.empty())
        return nullptr;
    for (int i = 0; i < input_string.size(); i++) {
        if ((!isdigit(input_string[i])) && (input_string[i] != ' ') &&
(input_string[i] != '-')) {
            return nullptr;
        }
    }
    stringstream myStream; // открывается строковый поток
    myStream << input_string; // в него записывается введённая строка
    int count of elem = 0;
    int buff = 10;
    int* rezerv data mass;
    int* data mass = new int[buff];
    while (myStream >> data_mass[count_of_elem]) { //считываем из строки
элементы массива
        count of elem++;
        if (count of elem == buff) { //проверка на заполнение буфера
            buff += 10;
            rezerv data mass = new int[buff];
            for (int i = 0; i < buff - 10; i++) {
                rezerv_data_mass[i] = data_mass[i];
            }
            delete[] data mass;
            data mass = rezerv data mass;
            rezerv_data_mass = nullptr;
        }
    }
    if (count of elem == 0) {
        delete data mass;
        return nullptr;
    }
    *size mass = count_of_elem; //записываем количество элементов
    return data_mass;
}

int console input() {
    cout << "Введите элементы массива\n";
    string input_str;
    getline(cin, input_str, '\n'); //считываем строку из консоли
    int count of elem = 0;
    int* data mass = mass from string(input_str, &count_of_elem); //вызываем
функцию получения массива из строки
    if (data mass == nullptr) {
        cout << "Входная строка некорректна!";
        return 1;
    }
    if (data analis(data_mass, count_of_elem)) { //вызов функции анализа данных
        return 1;
    }
}

```

```

        return 0;
    }

int file input() {
    int correct file name_flag = 0; //флаг корректного имени файла ввода
    fstream file input;
    char* file name = new char[256];
    while (!correct file name flag) { //цикл до ввода корректного имени файла
        cout << "Введите имя файла\n\n";
        cin >> file name;
        file input.open(file name, fstream::in); //открывается файл ввода
        if (file input.is open()) {
            correct_file_name_flag = 1;
        }
        else {
            cout << "\nФайла с таким именем не найдено!\n";
            memset(file_name, '\0', 256);
        }
    }
    getchar(); //убираем символ переноса строки из потока ввода
    delete[] file name;
    string input str;
    getline(file input, input_str, '\n');
    int count of elem = 0;
    int* data mass = mass from string(input_str, &count_of_elem);
    if (data mass == nullptr) {
        cout << "Входная строка некорректна!";
        return 1;
    }
    if (data analis(data_mass, count_of_elem)) { //вызов функции анализа данных
        return 1;
    }
    return 0;
}

int main()
{
    setlocale(LC_ALL, "rus");
    string start dialog = "\nВыберите способ ввода данных:\n1 - Ввод с
консоли\n2 - Ввод из файла\n3 - Выйти из программы";
    while (1) {
        switch (input_num(start_dialog)) {
            case 1:
                cout << "Выбран ввод с консоли\n\n";
                if (!console input()) {
                    system("pause");
                    return 0;
                }
                break;
            case 2:
                cout << "Выбран ввод из файла\n\n";
                if (!file input()) {
                    system("pause");
                    return 0;
                }
                break;
            case 3:

```

```
        cout << "Выход из программы\n";
        system("pause");
        return 0;
        break;
default:
    cout << "Ответ некорректный!\n\n";
}
}
```