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 системы данных»

ТЕМА: СОРТИРОВКИ

Студент гр. 9381 _____ Любимов В.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить различные алгоритмы сортировок и реализовать один из них.

Задание.

Вариант 2

Сортировка простыми вставками; сортировка простыми вставками в список.

Описание алгоритма.

Во входном файле содержится последовательность чисел. Числа поочередно считываются из файла в массив, а их количество, то есть размер массива, запоминается. Далее начинает выполняться алгоритм сортировки простыми вставками. Алгоритм работает следующим образом:

0. Первым текущим элементом является второй элемент массива. Если его нет, то в массиве один всего элемент, значит он уже отсортирован выполнение сортировки прекращается.
1. Массив условно делиться на три части: отсортированная часть, которая может быть пустой, текущий элемент, не отсортированная часть массива, которая может быть пустой.
2. Текущий элемент копируется в отдельную переменную, тем самым «освобождая» своё место в массиве.
3. Далее текущий элемент сравнивается последовательно со всеми элементами массива, имеющими меньший индекс, то есть уже отсортированными. Если текущий элемент меньше, то сравниваемый элемент «сдвигается» в ячейку «справа», то есть с индексом на 1 больше, от него. Это возможно, так как текущий элемент был скопирован. Если текущий элемент больше или начало было пройдено на предыдущем шаге, то место для

текущего элемента в отсортированной части найдено. Он помещается «справа» в ячейку от меньшего его элемента или начало массива, если текущий элемент меньше всех отсортированных элементов массива.

4. Текущим элементом становиться следующий элемент после старого текущего элемента. Если достигнут конец массива, то сортировка завершается.

Отсортированный массив выводится на экран.

Так поиск места для текущего элемента выполняется пока он строго меньше, то сортировка является устойчивой.

Сортировка односвязного линейного списка этим алгоритмам схожа, с такой же сортировкой для массива, но имеется несколько различий:

1. Так в линейном односвязном списке можно перемещаться только от головы к хвосту, то поиск места для текущего элемента начинается с головы списка, то есть первого отсортированного элемента, а не с последнего, как в массиве, и продолжается до тех пор пока текущий элемент больше, то есть в этом случае сортировка становится неустойчивой.
2. Так в линейном односвязном списке каждый элемент хранит своё значение и место положение следующего элемента, то процесс выделения место под вставку текущего элемента в отсортированную часть списка и «удаление» текущего элемента проще, чем в массиве. Для «удаления» достаточно связать предшествующий текущему элемент с следующим после текущего элемента элементом, а для вставки – связать предшественника первого элемента, который больше текущего, с текущим, а текущий - с первым элемента больше текущего.

Данный алгоритм сортировки и для списка, и для массива использует только сам массив или список и один дополнительный элемент для временного хранения «вырезанного элемента», то есть затраты памяти характеризуются, как O(n)+O(1).

И в случае списка, и в случае массива алгоритм сортировки перебирает все полученные элементы начиная со второго, то есть делает n-1 шагов, где n – количество элементов. Для i-ого элемента происходит от 1 (массив отсортирован/список отсортирован в обратном порядке) до i-1 (массив отсортирован в обратном порядке/список полностью отсортирован) сравнений, а для массива к почти каждому сравнению прилагается операция перемещения, которой нет в сортировки списка. Операции перемещения и сравнения имеют постоянное время выполнения, а их количество примерно равно, то будем считать это одной операцией, выполняемой за постоянное время. Тогда в среднем случае для i-ого элемента будет совершенно i/2 операций сравнения-перемещения. Тогда всего таких операций будет совершенно $\sum_{i=2}^{n} \frac{i}{2} \approx \frac{1}{4}\, n^2$. Значит временные затраты можно выразить как $O(n^2)$ в среднем и худшем случае. Когда массив отсортирован или список отсортирован в обратном порядке, то время выполнения сортировки близко к линейному. Так же сортировка списка немного быстрее, так нет операции перемещения элемента при каждом сравнении, но разница будет заметна только при большом количестве сравнений для каждого элемента. Значит этот алгоритм сортировки эффективен для почти отсортированных массивов, обратно отсортированных списков и любых небольших наборов данных, так как в этом алгоритме основные используемые операции выполняются за одинаковое время для каждого элемента.

Описание реализованных классов

Программа сортирует может сортировать данные произвольных числовых типов.

Список реализовывается на базе массива.

1. Класс template <class T> class Node реализует узел
    - T m_value - приватное поле, содержащие значение этого узла.
    - int m_next - приватное поле, содержащие индекс следующего узла.
    - Node(T value = 0, int next = -1) - конструктор; по умолчанию устанавливает m_value, как нуль, а m_next – как -1, то есть отсутствие следующего узла.
    - Так же реализованы «геттеры» и «сэттеры» для получения и изменения значений полей.
2. Класс template <class T> class List реализует узел
    - Node<T>* m_head - приватное поле, содержащие массив элементов типа Node<T>, то есть узлов, хранящих данные типа T.
    - int head - приватное поле, содержащие индекс головы списка.
    - int m_size - приватное поле, содержащие размер списка.
    - int m_memory_size - приватное поле, содержащие количество памяти, выделенное под массив, хранящий список.
    - - конструктор; по умолчанию устанавливает, все поля, как нуль, а m_head – как nullptr.
    - void addNode(T value) - метод добавление узла в список. Получает значение добавляемого узла value. Если списка

ещё не существует, то создаёт голову списка, иначе добавляет элемент в конец списка. Если закончилась память под массив для хранения узлов списка, то увеличивает её объём.

- void makeList(T* &arr, int size) - метод получает на вход ссылку на массив элементов типа T и его размер. Строит из полученного массива список при помощи метода addNode().
- void printList(std::ofstream& fout) - метод выводящий список в консоль и полученный файл.
- void sort(std::ofstream& fout) - метод получает на вход файл fout для записи выходных данных. Выполняет описанный выше алгоритм сортировки простыми вставками для списка и выводит пояснительные данные в файл и консоль.
- ~Node() - деструктор очищает память, выделенную под массив хранящий узлы списка.

Описание функций.

1. template <class T> int getData(T* &arr, std::ifstream& fin) – получает на вход ссылку на массив типа T и входной файл fin. Считывает числа из файла в полученный массив arr. Если заполнен весь массив, то увеличивает количество выделенной под него памяти. Возвращает размер массива size.
2. template <class T> void sort(T* &arr, int size, std::ofstream& fout) - получает на вход ссылку на массив arr типа T, размер этого массива size и файл для записи выходных данных fout. Выполняет описанный выше алгоритм сортировки простыми

вставками для массива и выводит пояснительные данные в файл и консоль.

3. template <class T> void printArr(T* &arr, std::ofstream& fout, int size) – получает на вход ссылку на массив arr типа T, размер этого массива size и файл для записи выходных данных fout. Выводит этот массив в консоль и файл.
4. int main() – реализован простейший функционал взаимодействия с пользователем. Пользователю предлагают выбор из нескольких опций: завершить выполнение программы, обрабатывать входные данные как список, обрабатывать входные данные как список. В последних двух случаях предлагается ввести путь до файла с входными данными и путь до файла для выходных данных, предложив его перезаписать (в случае ошибочной команды файл будет перезаписан). После этого данные в обоих случаях считываются функцией getData(). В случае списка из массива создаётся список методом makeList(). После чего происходит сортировка. В случае массива сразу же начинается выполнение сортировки функцией sort(). После чего пользователю снова предлагается выбор из трёх выше указанных опций.

Тестирование.

Тестирование программы представлено в таблице 1. Здесь представлены входной набор и отсортированный набор, полученный в результате работы программы.

Таблица 1.

№	Входные данные:	Выходные данные:

1	-8 6 -0.0001 7 -1 2 3 0.7 4 5 6 7 8 -9 0	-9 -8 -1 -0.0001 0 0.7 2 3 4 5 6 6 7 7 8
2	1	1
3	1 2 3 10 7 8	1 2 3 7 8 10
4	63 28 483 595 724 949 98	28 63 98 483 595 724 949
5	36 8.8 -37163 0.0000067 -0.73 178361	-37163 -0.73 6.7e-06 8.8 36 178361
6	0 -1 -2 -3 -4 -5 -6 -7 -8 -9	-9 -8 -7 -6 -5 -4 -3 -2 -1 0
7	1 2 3 4 5 6 7 8 9 0	0 1 2 3 4 5 6 7 8 9
8	-1 1 -2 2 -3 3 -4.5 4.5 9.1 -9.1 0	-9.1 -4.5 -3 -2 -1 0 1 2 3 4.5 9.1

Файлы с этими входными данными лежат в папке test.

Выводы.

Была изучена, проанализирована и реализована сортировка простыми вставками в массиве и списке.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файлов: main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>

using namespace std;

template <class T>
int getData(T* &arr, ifstream& fin){
    int size = 0;
    int mem_size = 0;

    while(1){
        if(size == mem_size){
            mem_size += 10;
            T* new_arr = new T[mem_size];
            for(int i = 0; i < size; i++){
                new_arr[i] = arr[i];
            }
            delete[] arr;
            arr = new_arr;
        }

        if(fin.eof()){
            return size;
        }

        fin >> arr[size];
        size += 1;
    }

    return size;
}

template <class T>
void sort(T* &arr, int size, ofstream& fout){
    if(size == 1){
        cout << "Only one elememt in array! Sort doesn't need.\n";
        fout << "Only one elememt in array! Sort doesn't need.\n";
    }
    T cur = 0;
    int j = 0;
```

```
    for(int i = 1; i < size; i++){
        cur = arr[i];
        j = i-1;

        cout << "Sorted part of array: ";
        fout << "Sorted part of array: ";
        for(int k = 0; k <= j; k++){
            cout << "\033[32m" << arr[k] << ' ';
        }
        cout << "\033[0mCurent elemrnt is " << "\033[33m" << cur << ' ' <<
"\033[0mUnsorted part of array: ";
        fout << "Curent elemrnt is " << cur << ' ' << "Unsorted part of array:
";
        for(int k = i+1; k < size; k++){
            cout << "\033[36m" << arr[k] << ' ';
            fout << arr[k] << ' ';
        }
        cout << "\033[0m\nCurrent index is " << i << '\n';
        fout << "\nCurrent index is " << i << '\n';

        while(arr[j] > cur && j >= 0){
            cout << "\033[32m" << arr[j] << "\033[0m at positon " << j << " is
more than \033[33m" << cur << "\033[0m\n";
            fout << arr[j] << " at positon " << j << " is more than " << cur
<< "\n";
            arr[j+1] = arr[j];
            j -= 1;
        }
        if(j >= 0){
            cout << "\033[32m" << arr[j] << "\033[0m at positon " << j << " is
less or equal than \033[33m" << cur << "\033[0m. Element insert into position
with index " << j+1 << '\n';
            fout << arr[j] << " at positon " << j << " is less or equal than "
<< cur << ". Element insert into position with index " << j+1 << '\n';
        }
        else{
            cout << "Start of array nave been reached. \033[33m" << cur <<
"\033[0m is insert into the start of array.\n";
            fout << "Start of array nave been reached. " << cur << " is insert
into the start of array.\n";
        }

        arr[j+1] = cur;

        cout << "Current status of array: ";
        fout << "Current status of array: ";
        for(int k = 0; k < size; k++){
            cout << arr[k] << ' ';
```

```
            fout << arr[k] << ' ';
        }
        cout << "\n\n";
        fout << "\n\n";
    }
}

template <class T>
class Node{
    private:
        T m_value;
        int m_next;
    public:
        Node(T value = 0, int next = -1): m_value(value), m_next(next){
        }

        void setValue(T value){
            m_value = value;
        }

        T getValue(){
            return m_value;
        }

        void setNext(int next){
            m_next = next;
        }

        int getNext(){
            return m_next;
        }
};

template <class T>
class List{
    private:
        Node<T>* m_head = nullptr;
        int head = 0;
        int m_size = 0;
        int m_memory_size = 0;
    public:
        List(){
            m_head = nullptr;
        }

        void addNode(T value){
            if(!m_head){
                head = 0;
```

```
        m_head = new Node<T>[10];
        m_head[0].setValue(value);
        m_size += 1;
        m_memory_size = 10;
        return;
    }

    if(m_size == m_memory_size){
        m_memory_size += 10;
        Node<T>* new_arr = new Node<T>[m_memory_size];
        for(int i = 0; i < m_size; i++){
            new_arr[i] = m_head[i];
        }
        delete[] m_head;
        m_head = new_arr;
    }

    int cur = 0;
    while(m_head[cur].getNext() != -1){
        cur += 1;
    }
    m_head[cur].setNext(m_size);;
    m_head[m_size].setValue(value);
    m_size += 1;
}

void makeList(T* &arr, int size){
    for(int i = 0; i < size; i++){
        addNode(arr[i]);
    }
}

void printList(ofstream& fout){
    int cur = head;
    while(cur != -1){
        cout << m_head[cur].getValue() << ' ';
        fout << m_head[cur].getValue() << ' ';
        cur = m_head[cur].getNext();
    }
    cout << '\n';
    fout << '\n';
}

void sort(ofstream& fout){
    int precur = head;
    int cur = m_head[precur].getNext();
    int comp = head;
    int precomp = head;
```

```
            while(cur != -1){
                m_head[precur].setNext(m_head[cur].getNext());
                m_head[cur].setNext(-1);

                cout << "Current element is \033[33m" <<
m_head[cur].getValue() << "\033[0m\n";
                cout << "This element is going to be compared with elements
before its postion in list begining with the head of the list.\n";
                fout << "Current element is " << m_head[cur].getValue() <<
'\n';
                fout << "This element is going to be compared with elements
before its postion in list begining with the head of the list.\n";

                while(comp != -1 && m_head[comp].getValue() <
m_head[cur].getValue() && comp != m_head[precur].getNext()){
                    cout << "\033[32m" << m_head[comp].getValue() << "\033[0m
is less than \033[33m" <<  m_head[cur].getValue() << "\033[0m\n";
                    fout << m_head[comp].getValue() << " is less than " <<
m_head[cur].getValue() << '\n';
                    precomp = comp;
                    comp = m_head[comp].getNext();
                }

                cout << "\033[32m" << m_head[comp].getValue() << "\033[0m is
more or equal than \033[33m" <<  m_head[cur].getValue() << "\033[0m. Also its
possiblke that all elements before current have been viewed and current is
bigger then all of them. ";
                fout << m_head[comp].getValue() << " is more or equal than "
<<  m_head[cur].getValue() << ". Also its possiblke that all elements before
current have been viewed and current is bigger then all of them. ";
                if(comp == head){
                    cout << "It is the head of th list, so \033[33m" <<
m_head[cur].getValue() << "\033[0m is a new head of the list.\n";
                    fout << "It is the head of th list, so " <<
m_head[cur].getValue() << " is a new head of the list.\n";
                    m_head[cur].setNext(comp);
                    head = cur;
                }
                else{
                    cout << "So \033[33m" << m_head[cur].getValue() <<
"\033[0m has been placed between \033[32m" << m_head[precomp].getValue() <<
"\033[0m and \033[32m" << m_head[comp].getValue() << "\033[0m\n";
                    fout << "So " << m_head[cur].getValue() << " has been
placed between " << m_head[precomp].getValue() << " and " <<
m_head[comp].getValue() << '\n';
                    m_head[precomp].setNext(cur);
                    m_head[cur].setNext(comp);
                }
```

```
                if(m_head[precur].getNext() == cur){
                    precur = cur;
                    cur = m_head[cur].getNext();
                }
                else{
                    cur = m_head[precur].getNext();
                }
                comp = head;
                precomp = head;
                cout << "The current state of the list: ";
                fout << "The current state of the list: ";
                printList(fout);
                cout << '\n';
                fout << '\n';
            }
        }

        ~List(){
            delete[] m_head;
        }
};

template <class T>
void printArr(T* &arr, ofstream& fout, int size){
    for(int i = 0; i < size; i++){
        cout << arr[i] << ' ';
        fout << arr[i] << ' ';
    }
    cout << '\n';
    fout << '\n';
}

int main(){
    double* arr = nullptr;
    string fname;
    char type;
    ifstream fin;
    List<double>* list;
    int size;
    ofstream fout;

    while(1){
        cout << "Input the type of the data structure (l(for list)/a(for
array)) or input 'q' to stop the program:\n";
        cin >> type;
        switch (type){
            case 'q':
                cout << "You choose to end the programm!\n";
```

```
                return 0;

            case 'l':
                cout << "Input the path to data file:\n";
                cin >> fname;
                fin.open(fname, ifstream::in);
                if(!fin.is_open()){
                    cout << "Opening file with test data failed! Try
again!\n";
                    break;
                }

                size = getData(arr, fin);

                fin.close();
                cout << "Input the path to result file:\n";
                cin >> fname;

                cout << "Do you want to rewrite it (y/n)?:\n";
                cin >> type;
                switch (type){
                    case 'y':
                        fout.open(fname, ofstream::trunc);
                        break;
                    case 'n':
                        fout.open(fname, ofstream::app);
                        break;
                    default:
                        cout << "Error command! File will be rewritten!\n";
                        fout.open(fname, ofstream::trunc);
                        break;
                }
                if(!fout.is_open()){
                    cout << "Opening file for writing result data failed! Try
again!\n";
                    break;
                }

                list = new List<double>;
                list->makeList(arr, size);
                cout<< "List before sort: ";
                fout<< "List before sort: ";
                list->printList(fout);

                if(size == 1){
                    cout << "Only one elememt in array! Sort doesn't need.\n";
                    fout << "Only one elememt in array! Sort doesn't need.\n";
                }
```

```
                list->sort(fout);

                cout<< "List after sort: ";
                fout<< "List after sort: ";
                list->printList(fout);
                cout<< "\n";
                fout<< "\n";
                fout.close();

                delete list;
                list = nullptr;
                break;

            case 'a':
                cout << "Input the path to data file:\n";
                cin >> fname;
                fin.open(fname, ifstream::in);
                if(!fin.is_open()){
                    cout << "Opening file with test data failed! Try
again!\n";
                    break;
                }

                size = getData(arr, fin);

                fin.close();
                cout << "Input the path to result file:\n";
                cin >> fname;
                cout << "Do you want to rewrite it (y/n)?:\n";
                cin >> type;
                switch (type){
                    case 'y':
                        fout.open(fname, ofstream::trunc);
                        break;
                    case 'n':
                        fout.open(fname, ofstream::app);
                        break;
                    default:
                        cout << "Error command! File will be rewritten!\n";
                        fout.open(fname, ofstream::trunc);
                        break;
                }

                if(!fout.is_open()){
                    cout << "Opening file for writing result data failed! Try
again!\n";
                    break;
```

```
                }

                cout<< "Array before sort: ";
                fout<< "Array before sort: ";
                printArr(arr, fout, size);

                sort(arr, size, fout);

                cout<< "Array after sort: ";
                fout<< "Array after sort: ";
                printArr(arr, fout, size);
                cout<< "\n";
                fout<< "\n";
                fout.close();

                delete[] arr;
                arr = nullptr;
                break;

            default:
                cout << "Error command! Try again!\n";
                break;
        }
    }
    return 0;
}
```