*Федеральное государственное бюджетное образовательное учреждение* *высшего образования*

|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | ***«Московский государственный технический университет  имени Н.Э. Баумана***  ***(национальный исследовательский университет)»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ (ИУ6)

**Отчет**

**по лабораторной работе № 1**

**Дисциплина:** Технологии разработки программных систем

**Название лабораторной работы:**

Исследование структур и методов обработки данных

**Вариант 5**

Студент гр. ИУ6-42  **\_\_\_\_\_\_13.04.2018 А.С. Бурлаков**

(Подпись, дата) (И.О. Фамилия)

Преподаватель  **\_\_\_\_\_\_\_\_\_\_\_ К.С. Хорунжина**

(Подпись, дата) (И.О. Фамилия)

Москва, 2018

# ВВЕДЕНИЕ

## 1. Цель работы

* Исследование структур данных, методов их обработки и оценки.

## 2. Задача

### 2.1. Порядок выполнения

1. Ознакомиться с теоретическими сведениями по абстрактным структурам данных и методами их обработки.
2. Для указанной задачи и типа данных (см. таблицу 2) предложить способ реализации и определить требуемый объем памяти.
3. Провести анализ заданных методов поиска, упорядочения и корректировки. Оценить время выполнения соответствующих операций.
4. Предложить альтернативный вариант решения задачи, в котором должно быть минимум одно улучшение. Улучшения могут касаться как структуры данных, так и основных операций. Обосновать новые решения, используя количественные и качественные критерии. Количественными критериями являются: объем памяти, среднее количество сравнений и количество тактов. Качественные критерии определяют возможность использования того или иного метода применительно к разработанной структуре. К ним можно отнести: применимость операции только к упорядоченным данным; необходимость знать количество элементов; наличие признака разбивки на гнезда; необходимость в прямом доступе к элементам; знание граничных значений; невозможность создать структуру в соответствии с арифметической прогрессией и др.

## 3. Задание

Для варианта 5 задание выглядит следующим образом:

* *Задача 2.* Дана таблица материальных нормативов, состоящая из K записей фиксированной длины вида: код детали; код материала; единица измерения; номер цеха; норма расхода.
* Структура: Таблица;
* Поиск: Дихотомический (двоичный);
* Упорядочение: слиянием;
* Корректировка: вставка записи.

# ОСНОВНАЯ ЧАСТЬ

## 1. Таблица исходных данных

Таблица 1. Исходные данные.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Вариант | Задача | Структура | Поиск | Упорядочение | Корректировка |
| 5 | 2 | Таблица | Дихотомич. | Слиянием | Вставка записи |

## 2. Постановка задачи

*Задача 2.* Дана таблица материальных нормативов, состоящая из K записей фиксированной длины вида: код детали; код материала; единица измерения; номер цеха; норма расхода.

Исходя из исходных данных и поставленной задачи следует то, что необходимо:

* Составить структуру для записи;
* Заполнить таблицу, которая будет из себя представлять одномерный массив для реализации заданных методов;
* Реализовать заданные методы;
* Провести анализ. Для этого взять несколько значений М для более точного анализа.

## 3. Анализ исходных данных

### 3.1. Определение объёма данных

Для решения поставленной задачи создадим структуру. Она представлена на рисунке 1.

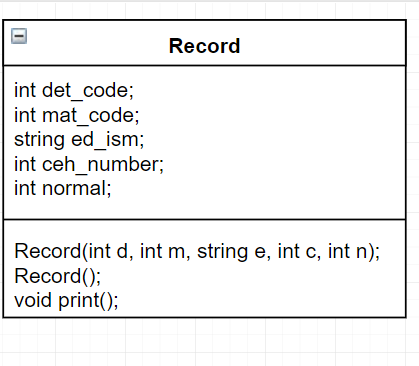


Рисунок 1 - Структура данных.

Объём данных для задачи задан как М => возьмём несколько значений М для более точной оценки методов.

Объём структуры: det\_code = 4, mat\_code = 4, ed\_ism = 10, ceh\_number = 4, normal = 4 -> Сумма = 26.

Общий объём структуры данных = М\*26.

### 3.2. Анализ алгоритма поиска

По заданию необходимо использовать дихотомический способ поиска. Для его эффективного применения необходимо, чтобы массив был заранее отсортирован.

а) Массив заранее отсортирован

В этом случае просто применяем код, представленный в Листинге 1, чтобы отсортировать наш массив.

Листинг 1. Поиск дихотомическим способом.

int Table::find(int det\_code){

int midd = 0, left = 0, right = size;

while (1)

{

midd = (left + right) / 2;

if (det\_code < mas[midd].det\_code)

right = midd - 1;

else if (det\_code > mas[midd].det\_code)

left = midd + 1;

else

return midd;

if (left > right)

return -1;

}

}

В итоге получаем следующие результаты:

1. При М = 3 количество процессорных тиков: 242
2. При М = 10 количество процессорных тиков: 184
3. При М = 25 количество процессорных тиков: 220
4. При М = 4500 количество процессорных тиков: 2006

## Исходя из полученных значений, можно говорить об сложности алгоритма O()

б) Массив нуждается в сортировке

При этом сложность поиска увеличивается на столько – сколько займёт алгоритм упорядочения.

### 3.3. Анализ алгоритма упорядочения

По заданию необходимо использовать сортировку (упорядочение) методом слияния. Метод слияния один из самых быстрых способов сортировки массива. Фрагмент кода с сортировкой методом вставки представлен в Листинге 2.

Листинг 2. Упорядочение методом вставки.

void Table::mergeSort(Record data[], int lenD)

{

if (lenD > 1){

int middle = lenD / 2;

int rem = lenD - middle;

Record\* L = new Record[middle];

Record\* R = new Record[rem];

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

if (i < middle){

L[i] = data[i];

}

else{

R[i - middle] = data[i];

}

}

mergeSort(L, middle);

mergeSort(R, rem);

merge(data, lenD, L, middle, R, rem);

}

}

void Table::merge(Record merged[], int lenD, Record L[], int lenL, Record R[], int lenR){

int i = 0;

int j = 0;

while (i < lenL || j < lenR){

if (i < lenL && j < lenR){

if (L[i].det\_code <= R[j].det\_code){

merged[i + j] = L[i];

i++;

}

else{

merged[i + j] = R[j];

j++;

}

}

else if (i < lenL){

merged[i + j] = L[i];

i++;

}

else if (j < lenR){

merged[i + j] = R[j];

j++;

}

}

}

Вычислительная сложность будет равна:

1. При М = 20 количество процессорных тиков 426592
2. При М = 200 количество процессорных тиков 6331994
3. При М = 2000 количество процессорных тиков 91826277

## Исходя из полученных значений, можно говорить об сложности алгоритма O()

### 3.4. Анализ алгоритма вставки.

По заданию необходимо использовать метод вставки записи. Метод заключается в перемещении элементов, чей индекс больше вставляемого вправо. Это не очень удобно по вычислительной сложности. Фрагмент кода с методом вставки представлен в Листинге 3.

Листинг 3. Вставка записи

void Table::increase\_buffer(){

buffer \*= 2;

Record\* m = new Record[buffer];

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

m[i] = mas[i];

}

Record\* t = mas;

mas = m;

delete[] t;

}

void Table::insert(Record new\_elem, int pos){

if (size == buffer)

increase\_buffer();

for (int i = size; i > pos; --i){

mas[i] = mas[i - 1];

}

mas[pos] = new\_elem;

size++;

}

Вычислительная сложность будет равна:

1. При М = 20 количество процессорных тиков 364188
2. При М = 200 количество процессорных тиков 17070024
3. При М = 2000 количество процессорных тиков 1404310538

## Исходя из полученных значений, можно говорить об сложности алгоритма O(n)

## 4. Вывод

Полученная структура имеет достаточно быструю сортировку и поиск элементов, однако в случае, если массив не будет отсортирован, скорость поиска сильно упадет. Вставка записи происходит достаточно быстро.

# АЛЬТЕРНАТИВНЫЙ ВАРИАНТ

## 1. Таблица измененных данных

Таблица 2. Исходные данные.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Вариант | Задача | Структура | Поиск | Упорядочение | Корректировка |
| 5 | 2 | Таблица двоичного поиска | Дихотомич. | Отсутствует | Вставка записи |

## 3. Анализ исходных данных

### 3.1. Определение объёма данных

Так как запись не меняется, количество выделяемой памяти остается неизменной, а именно 26 \* M

### 3.2. Анализ алгоритма поиска

### Алгоритм поиска не изменился, следовательно его сложность так же не поменялась

### 3.3. Анализ алгоритма вставки.

При вставке элемента сортировка происходит автоматически. Бинарным поиском находится место, в которое необходимо вставить элемент, а затем используется алгоритм вставки, описанный выше. Фрагмент кода с сортировкой методом вставки представлен в Листинге 4.

Листинг 3. Вставка записи

void Binary\_table::increase\_buffer(){

buffer \*= 2;

Record\* m = new Record[buffer];

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

m[i] = mas[i];

}

Record\* t = mas;

mas = m;

delete[] t;

}

void Binary\_table::add(Record new\_elem){

int midd = 0, left = 0, right = size;

bool b = true;

while (b){

midd = (left + right) / 2;

if (new\_elem.det\_code < mas[midd].det\_code)

right = midd - 1;

else if (new\_elem.det\_code > mas[midd].det\_code)

left = midd + 1;

else

b = false;

if (left > right){

midd = left;

b = false;

}

}

if (mas[midd].det\_code < new\_elem.det\_code)

midd--;

if (size >= buffer)

increase\_buffer();

size++;

for (int i = size - 1; i > midd; i--){

mas[i] = mas[i - 1];

}

mas[midd] = new\_elem;

}

Вычислительная сложность будет равна:

1. При М = 20 количество процессорных тиков 142
2. При М = 200 количество процессорных тиков 1866
3. При М = 2000 количество процессорных тиков 19354

## Исходя из полученных значений, можно говорить об сложности алгоритма O(n)

## 4. Вывод

Альтернативная структура не имеет функции сортировки, т.к. структура по умолчанию упорядочена. Алгоритмы вставок и бинарного поиска достаточно быстры.

# Сравнительная таблица

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Начальный вариант | | | Альтернативный вариант | | |
|  | M = 20 | M = 2000 | M = 500 | M = 20 | M = 2000 | M = 500 |
| Сортировка | 388503 | 88197594 | 18646851 | 0 | 0 | 0 |
| Поиск | 506 | 1378 | 464 | 324 | 1378 | 620 |
| Поиск\* | 551744 | 92228230 | 18481180 | - | - | - |
| Вставка | 13206 | 16193 | 16715 | 13480 | 17248 | 17168 |

Таблица 3 – Сравнительная таблица

Поиск\* - поиск в начальной структуре данных при неотсортированном массиве.

# ВЫВОД

Альтернативная структура (двоичная таблица) выигрывает по скорости в сортировке, однако немного проигрывает методу вставки основной структуры(таблица). Помимо этого, в двоичной таблице в любой момент можно начать дихромотический поиск, а в простой таблице это можно сделать только в том случае, если массив отсортировали.

Смотря на сравнительную таблицу, можно понять, что альтернативный вариант в разы лучше начального варианта из-за отстутствия потерь в скорости при сортировке.