|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«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)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.03 Прикладная информатика**

**Отчет**

|  |  |
| --- | --- |
| **по лабораторной работе №** | **1** |

**Название:**

Выбор структур и методов обработки данных

**Дисциплина:** Технологии разработки программных систем

**Вариант 14**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ИУ6-45Б |  | 01.09.2024 | И. А. Буханцев |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  | Е. К. Пугачев |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

Москва, 2024

**Содержание**

**Введение** 4

**1.** **Цель лабораторной работы** 4

**2.** **Описание задания** 4

**2.1.** **Задание** 4

**2.2.** **Основные требования** 6

**2.3.** **Задача** 7

**3.** **Описание основного варианта задания** 7

**3.1.** **Структура данных** 7

**3.1.1.** **Реализация структуры на языке C++** 7

**3.1.2.** **Определение объёма данных** 8

**3.1.3.** **Оценка времени доступа к i-му элементу** 8

**3.2.** **Анализ метода поиска** 8

**3.2.1.** **Реализация метода поиска** 9

**3.2.2.** **Среднее количество сравнений** 9

**3.2.3.** **Оценка времени поиска** 9

**3.3.** **Анализ метода упорядочивания** 10

**3.3.1.** **Реализация метода упорядочивания** 11

**3.3.2.** **Оценка количества сравнений** 13

**3.3.3.** **Оценка времени упорядочивания** 13

**3.4.** **Метод корректировки** 14

**3.4.1.** **Реализация метода** 14

**3.4.2.** **Расчет времени удаления элемента** 15

**4.** **Альтернативный вариант** 15

**4.1.** **Выбор альтернативной структуры данных** 15

**4.1.1.** **Реализация структуры на языке C++** 16

**4.1.2.** **Расчет занимаемой памяти** 17

**4.2.** **Анализ метода поиска** 17

**4.2.1.** **Реализация метода поиска** 17

**4.2.2.** **Среднее количество сравнений** 18

**4.2.3.** **Оценка времени поиска** 18

**4.3.** **Метод упорядочивания** 19

**4.3.1.** **Реализация метода поиска** 19

**4.3.2.** **Среднее количество сравнений** 20

**4.3.3.** **Оценка времени поиска** 21

**4.4.** **Метод корректировки** 21

**4.4.1.** **Реализация метода** 22

**4.4.2.** **Расчет времени удаления элемента** 23

**5.** **Таблица результатов** 23

**6.** **Заключение** 24

# **Введение**

При разработке алгоритмов программ часто возникает задача выбора структур данных и методов их обработки. Исходными составляющими для решения этой задачи являются описание набора и типов хранимых данных, а также перечень операций, выполняемых с ними. Можно выделить следующие основные вопросы, на которые необходимо ответить при решении поставленной задачи:

* 1. Как логически организовать структуру данных и реализовать ее?
  2. Как осуществлять поиск информации и как упорядочить данные?
  3. Как выполнить функции корректировки данных?

1. **Цель лабораторной работы**

Определить основные критерии оценки структуры данных и методов ее обработки применительно к конкретной задаче.

1. **Описание задания**
   1. **Задание**

1. На основе теоретических сведений выделить критерии оценки структур данных, принципы работы и критерии оценки операций поиска, сортировки и корректировки.

2. В соответствии с вариантом задания (Вариант 14) предложить конкретную схему структуры данных (в задании указана абстрактная структура данных) и способ ее реализации на выбранном языке программирования.

3. Определить качественные критерии оценки (универсальность, тип доступа и др.) полученной на шаге 2 структуры данных с учетом специфики задачи по выданному варианту.

4. Определить качественные критерии оценки, полученной на шаге 2 структуры данных: требуемый объем памяти на единицу информации, на структуру данных в целом и др.

5. Провести сравнительный анализ структуры данных, предложенной на шаге 2, на основе оценок, полученных на шаге 3 и шаге 4, с другими возможными вариантами реализации с целью поиска лучшей структуры данных к заданию по варианту.

6. Если цель шага 5 достигнута, то необходимо выполнить шаг 2, но для новой абстрактной структуры данных с указанием качественных и количественных критериев.

7. Оценить применимость метода поиска, который указан в варианте задания, с учетом структуры данных.

8. Если метод поиска применим, то необходимо сформулировать его достоинства и недостатки, используя качественные и количественные критерии: универсальность, требуемые ресурсы для реализации, среднее количество сравнений, время выполнения (такты) и др.

9. Предложить альтернативный, более эффективный метод поиска (от­ личный от задания), если такой существует, с учетом специфики за­ дачи по варианту, а также с учетом структур данных, полученных на предыдущих шагах. Для обоснования выбора альтернативного метода поиска использовать качественные и количественные критерии.

10. Оценить применимость метода упорядочивания, который указан в варианте задания, с учетом структуры данных.

11. Если метод упорядочивания применим, то необходимо сформулировать его достоинства и недостатки, используя качественные и количественные критерии: универсальность, требуемые ресурсы для реализации, среднее количество сравнений, время выполнения (такты) и др.

12. Предложить альтернативный метод упорядочивания, более эффективный и отличный от задания, если такой существует. При этом должны учитываться задача по варианту и структура данных. Для обоснования выбора альтернативного метода упорядочивания использовать качественные и количественные критерии.

13. Оценить применимость метода корректировки, который указан в за­ дании, к структуре данных.

14. Если метод корректировки применим, то необходимо сформулировать его достоинства и недостатки, используя качественные и количественные критерии: универсальность, требуемые ресурсы для реализации, время выполнения (такты) и др.

15. Предложить альтернативный способ корректировки, более эффективный и отличный от задания, если такой существует. При этом должны учитываться задача по варианту, структура данных. Для обоснования выбора альтернативного способа корректировки использовать качественные и количественные критерии.

16. Определить влияние метода корректировки на выполнение операций поиска и упорядочивания.

17. Определить основной режим работы программы и с учетом этого сделать выводы, а итоговые полученные результаты внести в таблицу. Из данной таблицы должно следовать, что предложенный альтернативный вариант решения задачи лучше. Как минимум должно быть одно улучшение, но могут быть заменены все методы обработки и сама структура данных.

* 1. **Основные требования**

Основной вариант задания включает в себя следующие требования:

* Структура: Таблица;
* Поиск: Последовательный;
* Упорядочение: Квадратичной выборки;
* Корректировка: Удаление сдвигом;
  1. **Задача**

Даны M записей вида: код группы, ФИО, дата рождения.

1. **Описание основного варианта задания**
   1. **Структура данных**

Структуру данных по заданию (таблицу) предлагается реализовать с помощью одномерного массива записей, структура которого представлена на рисунке 1.



Рисунок 1 – Структура массива записей

Каждый элемент массива (запись) содержит поля:

* код группы (целое число – длина: 6 символов);
* ФИО (макс. длина: 255 символов);
* дата рождения (10 символов);
  + 1. **Реализация структуры на языке C++**

Предлагается структуру данных реализовать следующим образом:

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

// Структура записи

struct Record {

int group\_id; // Код группы (целое число, 6 символов)

char full\_name[256]; // ФИО (макс. длина 255 символов)

char date\_of\_birth[11]; // Дата рождения (10 символов)

};

// Макс. количество записей

const int MAX\_RECORDS = 5;

// Таблица (одномерный массив записей)

Record records[MAX\_RECORDS];

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* + 1. **Определение объёма данных**

Объем занимаемой памяти массивом *V = k\*Vэ*, где *k* — количество элементов, а *Vэ* — размер одного элемента. Множитель *k* определяется пользователем, но не может быть динамически изменен. Размер элемента является суммой размера полей элемента. Рассчитаем эти размеры, учитывая, что 1 символ занимает 1 байт.

Получаем:

Если массив статический, а *k* неизвестно, то это приведет к неэффективному использованию оперативной памяти.

* + 1. **Оценка времени доступа к i-му элементу**

В массиве доступ выполняется по индексу. Соответственно, количество тактов, необходимых для доступа к *i*-му элементу, складывается из количе­ства тактов, необходимых для умножения и для сложения с постоянной:

* 1. **Анализ метода поиска**

Используется последовательный поиск, так как массив обеспечивает прямой доступ к элементам данных. При последовательном поиске элементы в массиве проверяются по очереди, начиная с первого элемента и заканчивая последним, пока не будет найден элемент с требуемым значением или пока не будет достигнут конец массива. Последовательный поиск не требует предварительной сортировки массива и прост в реализации, однако его эффективность может снижаться на больших массивах или в случае, если искомый элемент находится в конце массива.

* + 1. **Реализация метода поиска**

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

// Последовательный поиск

Record\* FindRecord(const char\* search\_term) {

// Перебор всех записей

for (int i = 0; i < record\_count; i++) {

// Проверка ФИО на наличие поискового термина

if (strstr(records[i].full\_name, search\_term) != nullptr) {

// Возвращаем указатель на найденную запись

return &records[i];

}

}

// Возвращаем nullptr, если запись не найдена

return nullptr;

}

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

* + 1. **Среднее количество сравнений**
    2. **Оценка времени поиска**

**tfor:**

*tуст* — время начальной установки цикла: *int i = 0 – 1 такта*

*tпров* — время проверки условия: *i < record\_count – 2 такта*

*kмод* — время модификации переменной цикла: i++ – 1 такта

**tif:**

*records[i] – 2 такта*

*records[i].full\_name – 2 такта*

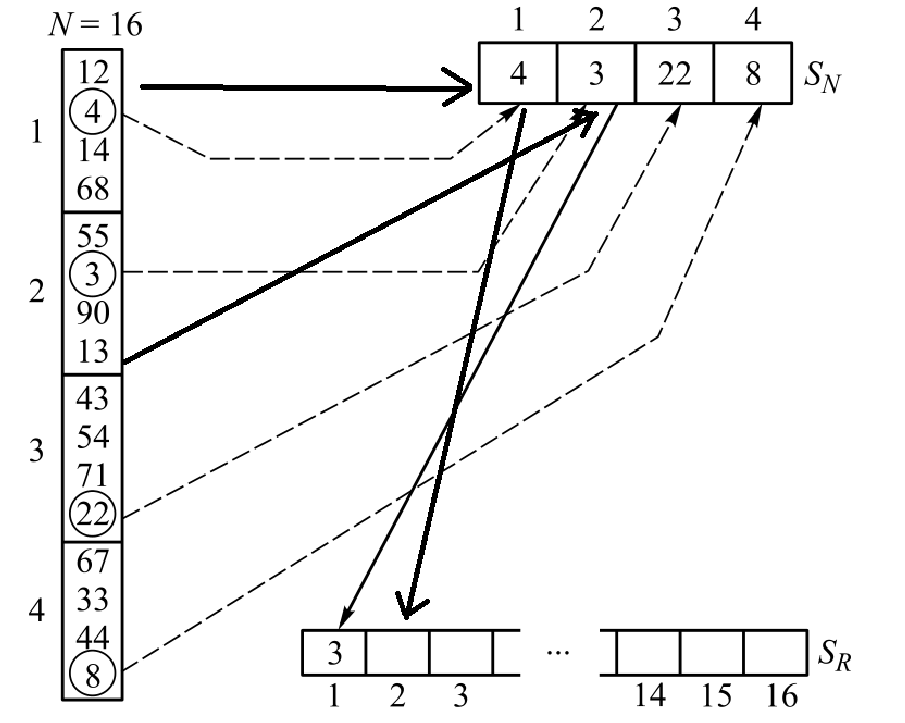
*strstr(records[i].full\_name, search\_term) – C(256/2) \* (доступ (2), доступ (2), сравнение (1))*

*!= – 1 такт*

* 1. **Анализ метода упорядочивания**

Упорядочиваемый массив, состоящий из N элементов, делится на групп по элементов в каждой. Для каждой группы необходимо выделить поле памяти, достаточное для размещения значения ключевого признака. Совокупность таких полей образует зону линейного накопления SN. В результате просмотра элементов каждой группы определяют наименьший элемент, который заносят в соответствующее поле зоны линейного накопления SN. Затем просматривают элементы в SN и наименьший из элементов заносят в зону формирования SR. Далее осуществляют просмотр в группе, элемент которой записан в зону SR. Наименьший элемент из оставшихся элементов в группе заносят в зону накопления SN на место элемента, перенесенного в зону формирования SR. После этого снова просматривают элементы в SN и наименьший из элементов заносят в зону формирования SR. Процесс повторяют, пока в зоне SN не останется элементов. На рисунке 1.2 приведен пример схемы сортировки по методу квадратичной выборки.

Квадратичная выборка по сравнению с сортировкой выбором имеет меньшее число сравнений элементов, однако требует дополнительного объема памяти.



Рисунке 1.2 – Схема работы алгоритма

* + 1. **Реализация метода упорядочивания**

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

int min\_index(int start, int end) {

int ind\_min = -1;

int min\_num = LONG\_MAX;

for (int i = start; i < end; i++) {

if (i < record\_count) {

if (records[i].group\_id < min\_num && records[i].group\_id != -1) {

min\_num = records[i].group\_id;

ind\_min = i;

}

}

}

return ind\_min;

}

int min\_index\_sn(int \*arr, int length) {

int ind\_min = -1;

int min\_num = LONG\_MAX;

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

if (arr[i] != -1) {

if (records[arr[i]].group\_id < min\_num) {

min\_num = records[arr[i]].group\_id;

ind\_min = i;

}

}

}

return ind\_min;

}

// Упорядочивание квадратичной выборкой

void SortByGroupId() {

// Количество групп

const int num\_groups = ceil(sqrt(record\_count));

// Массив для хранения индексов начала каждой группы

int\* group\_starts = new int[num\_groups];

// Разделение на группы SN

for (int i = 0; i < num\_groups; i++) {

group\_starts[i] = min\_index(i \* num\_groups, (i + 1) \* num\_groups);

}

int index\_groups = min\_index\_sn(group\_starts, num\_groups);

Record\* result\_arr = new Record[MAX\_RECORDS];

int i = 0;

while (index\_groups != -1) {

result\_arr[i] = records[group\_starts[index\_groups]];

records[group\_starts[index\_groups]].group\_id = -1;

group\_starts[index\_groups] = min\_index(index\_groups \* num\_groups, (index\_groups + 1) \* num\_groups);

index\_groups = min\_index\_sn(group\_starts, num\_groups);

i++;

}

delete[] records;

records = result\_arr;

}

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* + 1. **Оценка количества сравнений**
    2. **Оценка времени упорядочивания**

Так как сама функция достаточно сложна подсчитаем только такты, времени выполнения функций *min\_index* и *min\_index\_sn*.

– корень из k раз вызовем тот же цикл for (т.к. в функцию передается группа из элементов.

– аналогично, как в tmin\_index

*while* – отработает за *k* тактов

Время выполнения тела цикла while, где *ti*– обозначение команды целиком:

* 1. **Метод корректировки**

Алгоритм удаления сдвигом (shift deletion) заключается в смещении всех последующих элементов на одну позицию влево после удаления элемента из массива, за счет чего удаляемый элемент заменяется. Этот метод прост в реализации и эффективен на небольших массивах, но становится неэффективным на больших объемах данных из-за *O(n)* временной сложности, где *n* ­– количество элементов в массиве. Он также приводит к потере порядка элементов и может быть неэффективным при частых операциях удаления.

* + 1. **Реализация метода**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

void DeleteRecord(int index) {

// Сдвиг элементов

for (int i = index; i < record\_count - 1; i++) {

records[i] = records[i + 1];

}

// Уменьшение количества записей

record\_count--;

}

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

* + 1. **Расчет времени удаления элемента**

1. **Альтернативный вариант**

Альтернативный вариант задания включает в себя следующие изменения:

* Структура: Таблица с записями в виде динамического массива
* Поиск: Двоичный (бинарный);
* Упорядочение: Quick sort;
* Корректировка: Удаление маркировкой;
  1. **Выбор альтернативной структуры данных**

Структура данных была выбрана таже – таблица, только теперь она представляет собой динамический массив, а не статический. Из плюсов – эффективное использование памяти (выделяется под конкретный элемент), изменяемый размер. Главный минус, это медленный доступ к элементам: при каждом доступе к элементу необходимо вычислить его адрес.



Рисунок 2 – Структура массива записей

* + 1. **Реализация структуры на языке C++**

Ниже представлен код реализации структуры элемента и код инициализации массива:

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

// Структура записи

struct Record {

int group\_id; // Код группы (целое число, 6 символов)

char full\_name[256]; // ФИО (макс. длина 255 символов)

char date\_of\_birth[11]; // Дата рождения (10 символов)

bool delete\_flag; // Маркировка на удаление

};

// Макс. количество записей

const int MAX\_RECORDS = 5;

// Таблица (одномерный массив записей)

vector<Record> records;

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Каждый элемент массива (запись) содержит поля:

* код группы (целое число – длина: 6 символов);
* ФИО (макс. длина: 255 символов);
* дата рождения (10 символов);
  + 1. **Расчет занимаемой памяти**

Для элементов в группах *V = k\*Vэ*, где *k* — количество элементов; *Vэ* — размер одного элемента.

Получаем:

Размер памяти, занимаемой одной записью, определяется по тем же правилам, что и в исходном варианте структуры.

* 1. **Анализ метода поиска**

Используется бинарный поиск. Представляет собой эффективный метод поиска элемента в отсортированном массиве, который имеет ряд преимуществ. Среди них высокая скорость выполнения благодаря уменьшению количества сравнений и итераций, что особенно заметно при работе с большими объемами данных, а также минимизация времени выполнения поиска в сравнении с линейным поиском. Однако бинарный поиск требует, чтобы данные были предварительно отсортированы, что может стать затратным в случае часто изменяющихся данных. Кроме того, бинарный поиск сложнее в реализации и понимании, чем линейный поиск, и не подходит для неупорядоченных структур данных.

* + 1. **Реализация метода поиска**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Record\* binary\_search(const char\* name) {

int left = 0;

int right = records.size() - 1;

while (left <= right) {

int middle = (left + right) / 2;

// Сравниваем имя

int cmp = strcmp(records[middle].full\_name, name);

if (cmp == 0) {

return &records[middle];

}

else if (cmp < 0) {

left = middle + 1;

}

else {

right = middle - 1;

}

}

return nullptr;

}

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

* + 1. **Среднее количество сравнений**
    2. **Оценка времени поиска**
  1. **Метод упорядочивания**

Быстрая сортировка организована так: берется центральный элемент массива, а также берутся крайние элементы. Пока крайние элементы меньше (для левого) или больше (для правого) центрального мы сдвигаем указатель к центральному, не трогая элементы. После этой операции есть 2 варианта: указатели перешли через друг друга или нашлись числа, подходящие под все условия. Если 2 – верно, то меняем элементы местами и далее идем по той же схеме, пока не будет верно первое, попутно меняя элементы местами. После всех операций мы получим слева от центра только элементы большие центрального (в не отсортированном виде), а справа только элементы меньшие. Следующим шагом разделяем массив уже на 4 части (пополам, а потом каждую часть еще раз пополам) и проделываем те же действия с половинками, так идем, пока не закончатся половинки. После этого получим отсортированный массив.

* + 1. **Реализация метода поиска**

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

int partition(int low, int high) {

int pivot = records[high].group\_id;

int i = low - 1;

for (int j = low; j <= high - 1; j++) {

if (records[j].group\_id <= pivot) {

i++;

swap(records[i], records[j]);

}

}

swap(records[i + 1], records[high]);

return i + 1;

}

void quickSort(int low, int high) {

if (low < high) {

int pi = partition(low, high);

quickSort(low, pi - 1);

quickSort(pi + 1, high);

}

}

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

* + 1. **Среднее количество сравнений**

В функции partition используется 2 сравнения *j <= high – 1* и *records[j].group\_id <= pivot.* Тут 𝑛 – кол-во обрабатываемых элементов.

В итоге, среднее кол-во сравнений для одной функции partition:

Кол-во сравнение *low < high:*

Среднее кол-во сравнений для выполнения *quicksort* для 1-ой половины partiotin (*quickSort(low, pi - 1)*) в 1-ый раз:

Среднее кол-во сравнений для выполнения *quicksort* для 2-ой половины partiotin (*quickSort(*pi + 1, high*)*) в 1-ый раз:

Для 2-ого раза:

Для 3-го:

где:

* + 1. **Оценка времени поиска**

Так как сама функция достаточно сложна подсчитаем только такты,

участвующие непосредственно в сравнениях.

* 1. **Метод корректировки**

Для корректировки применялся метод маркировки удаляемого элемента. Серди плюсов данного метода: быстрое удаление ­– помеченные элементы можно легко удалить в пакетном режиме позже; простота реализации; потенциально уменьшает фрагментацию памяти. Минусы: дополнительные затраты памяти – требуется дополнительное пространство для хранения метки удаления.

* + 1. **Реализация метода**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

// Помечаем запись на удаление

void delete\_by\_id(int index) {

// Если запись найдена, помечаем ее как удаленную

records[index].delete\_flag = true;

}

// Сдвигаем

void compact\_records() {

int j = 0;

const int length = records.size();

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

// Если запись не удалена, копируем ее в начало массива

if (!records[i].delete\_flag) {

records[j++] = records[i];

}

else {

record\_count--;

}

}

// Урезаем массив до нового размера

records.resize(j);

}

// Восстанавливаем, если помечено на удаление

void recovery() {

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

if (records[i].delete\_flag == true) {

records[i].delete\_flag = false;

}

}

}

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

* + 1. **Расчет времени удаления элемента**

Время маркировки:

Время удаления i-го элемента:

*tуст*— время установки начального значения индекса

*tпр*— время проверки условия.

1. **Таблица результатов**

Таблица 1 – таблица результатов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Варианты** | **Структура данных** | **Метод поиска** | **Метод упорядочения** | **Метод корректировки** |
| Основной | Статический массив  *K \* 288 Байт*  *K – константа*  *(жесткое ограничение)* | Последовательный | Квадратичная выборка | Удаление сдвигом |
| Альтернативный | Динамический массив  *k \* 288 Байт*  *k – изменяемое число (гибкость работы с массивом)* | Бинарный | Быстрая сортировка | Метод маркировки |

Как видно из таблицы, альтернативный вариант выигрывает основной по временя (во всех методах), но занимает больше места в оперативной памяти. Также, бинарный поиск накладывает ограничение на то, что массив должен быть отсортирован.

1. **Заключение**

В результате выполнения лабораторной работы были проведены качественные и количественные оценки структур данных и методов их обработки в соответствии с вариантом задания. В альтернативном варианте предложены решения, которые обеспечат более эффективные поиск, сортировку и удаление данных.