«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

О Т Ч Е Т

по лабораторной работе № 1

Дисциплина: Технология разработки программных систем

Название: Выбор структур и методов обработки данных (Вариант 11)

Студент ИУ6-42Б (Группа) 06.03.24 (Подпись, дата) А. П. Плютто (И. О. Фамилия)

Преподаватель 06.03.24 (Подпись, дата) Е. К. Пугачёв (И. О. Фамилия)

Москва 2024 г.

Содержание

1. Цель лабораторной работы

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

2. Описание задания

2.1. Задания

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 11) 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 (отличный от задания), если такой существует, с учетом специфики задачи 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 задании, 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.

2.2. Основные требования

Основные требования приведены в таблице ниже:

Номер задачи	Структура данных	Поиск	Упорядочение	Корректировка
2	Таблица	Вычисление адреса	Пузырьком	Удаление сдвигом

2.3. Задача

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

3. Основной вариант

3.1. Структура данных

В заданной таблице указана абстрактная структура данных – таблица. Это означает, что нам необходимо явно выбрать структуру данных, элементы которой связанны неявно. Из задачи следует, что столбцами таблицы будут код детали, код материала, еденица измерения, номер цеха, норма расхода. Строками мы будем заносить записи.

Каждая строка будет занимать одно и тоже количество места в памяти. Для реализации строк, так как типы данных у столбцов разные, будем использовать структуры. Они занимают столько же места, сколько бы занимали переменные, хранящиеся по отдельности. Определим тип данных для каждого столбца в зависимости от информации, которую необходимо в нем хранить.

Столбец	Тип данных
Код детали	`unsigned int`
Код материала	`unsigned int`
Единица измерения	`String (char[])`
Номер цеха	`unsigned int`
Норма расхода	`unsigned int`
Номер записи	`unsigned int`

Везде (кроме единиц измерения) будем использовать беззнаковый целочисленный тип, для того, чтобы увеличить количество чисел, которые можно сохранить в двое, по сравнению со знаковым типом. Коды, номера изначально не могут быть отрицательными в силу здравного смысла. Норма расхода – максимально допустимое плановое количество сырья, материалов, топлива, энергии на производство какой-либо единицы какой-либо продукции. Т. е. это значение тоже не может быть отрицательным, но в зависимости от единиц измерения оно может перестать быть целочисленным. В данной задаче не сказано, что это число будет вещественным, поэтому примем правило: если необходимо ввести вещественную норму расхода, необходимо ввести целую норму расхода и изменить еденицы измерения на более меньшие.

О номере записи опишем подробнее позже, когда будем рассматривать непосредственно алгоритмы.

После определения структуры необходимо задуматься о том, как хранить строки в памяти. Самым простым в реализации способом будет хранение их друг за другом – в массиве. Таким образом общей структурой данных будет массив записей. Вот его схема:

1	Запись 1				
	Код детали 1	Код материала 1	Еденица измерения 1	Номер цеха 1	Норма расхода 1
...	...				
k	Запись k				
	Код детали k	Код материала k	Еденица измерения k	Номер цеха k	Норма расхода k
...	...				
n	Запись n				
	Код детали n	Код материала n	Еденица измерения n	Номер цеха n	Норма расхода n

Рисунок 1 — *Схема структуры данных*

3.1.1. Реализация структуры на языке C++

```
struct norm {
    unsigned int detailCode;
    unsigned int materialCode;
    std::string measure;
    unsigned int workshopNum;
    unsigned int consumptionNorm;
    unsigned int no;

    unsigned int& get(int index) {
        switch (index) {
            case 0: return detailCode;
            case 1: return materialCode;
            case 3: return workshopNum;
            case 4: return consumptionNorm;
            case 5: return no;
            default: throw std::out_of_range("Invalid index");
        }
    }
}
```

Листинг 1 — *Структура данных* `norm`

3.1.2. Расчет памяти, занимаемой массивом

Объем занимаемой памяти массивом $V = kV_э$, где k — количество элементов, а $V_э$ — 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 (в среднем сокращенные еденицы измерения занимают 2 символа):

$$V_э = l_{\text{det}} + l_{\text{mat}} + l_{\text{mea}} + l_{\text{ws}} + l_{\text{cn}} + l_{\text{no}} \simeq 4 \times 5 + 3 = 23$$

Получаем $V = 23k$ байт.

Если массив статический, а k неизвестно, то это приведет к неэффективному использованию оперативной памяти, а так же к ограничению к количеству записей изменить которое можно только в коде программы (для этого объявим `#define max 300`).

Необходимо пояснить, что `std::string` является более сложной структурой, чем `char[]`, хотя бы из-за того, что в данной структуре можно легко оперировать с динамическим добалением элементов, но так как в процессе программы использования этого нет необходимости будем считать, что `std::string` занимает столько же памяти, сколько и `char[]`.

3.1.3. Оценка времени доступа к i-му элементу

В массиве доступ выполняется по индексу. Для удобства мы храним адрес лишь первого элемента массива, поэтому для доступа мы должны прибавить к этому адресу i, что займет $t_{++} = 1$ такт, доступ к оперативной памяти осуществляется за $t_{\rightarrow} = 1$ такт процессора.

$$T_д = t_{++} + t_{\rightarrow} = 1 + 1 = 2 \text{ такта.}$$

Стоит отметить, что из-за особенностей адресного поиска доступ к i-тому элементу в данной реализации будет $T_{ди} = 2i$ такта. Но операции поиска и сортировки будут выполняться по индексам ($T_д = 2$ такта), поэтому доступ к элементам (не по индексу, а по номеру записи) будет необходим только в процессе вывода или удаления элементов.

3.1.4. Оценка времени удаления i-го элемента

При удалении мы переписываем весь массив на одну ячейку влево, начиная с i-го элемента, поэтому удаление – это перезапись $k - i$ элементов. (подробнее см. 3.4.3)

3.2. Метод поиска

Поиск необходимо реализовать адресный. Адресный поиск – один из самых эффективных по времени методов поиска. Он предполагает связь искомой ячейки и индекса записи. Так как значения заренее не извесны, создать функцию поиска индекса по значению можно только проиндексировав динамически массив. Это возможно сделать в виде структур ключ-значение. Возьмем для этого структуру хеш-таблицы, куда динамически будем при записи заносить значения в виде ключей, а значениями хеша будет выступать односвязный список – одно значение может быть в нескольких записях. Список использовать будем из-за того, что результат поиска необходимо выводить весь, т.е. использовать доступ к элементам не по их порядку нет необходимости.

Для каждого значения поиска необходимо генерировать отдельный хеш, поэтому для удобства, занесем хеши одного и того же типа в массив. Отдельно создадим хеш для поля «еденицы измерения».

```
typedef unordered_map<unsigned int, list<int> > uimap;
typedef unordered_map<string, list<int> > smap;
uimap maps[4] = {{},{},{},{}};
smap  measureMap = {};
```

Листинг 2 — *Объявление новых типов и переменных для поиска*

Теперь следует поговорить о самих индексах в записи. Из-за того, что необходимо реализовать методы сортировки и удаления элементов, индексы массива не подходят для однозначного определения положения записи в массиве, ведь каждый раз при сортировке или удалении необходимо будет переписывать индексы всех (многих) записей в хеш таблицах, т.е. переиндексировать таблицу. Это будет влиять на время выполнения алгоритмов, поэтому необходимо создать дополнительное поле, содержащее свой индекс записи, который не будет изменяться при удалении других записей или при сортировке. Это увеличивает размер структуры, но позволяет наиболее эффективно по времени выполнить все три алгоритма.

Алгоритм поиска будет возвращать список с индексами найденных элементов, так как сами элементы не упорядочены по этому индексу доступ к элементам усложняется (будет равен $T_{\text{ди}}$). Для доступа к элементам будем использовать следующую функцию:

```
unsigned int idxSearch(unsigned int idx){
    for (int i = 0; i < k; i++){
        if (tbl[i].no == idx) return i;
    }
    return 0;
}
```

Листинг 3 — *Ищем запись с индексом* `idx`

Эта функция принимает «статический» индекс записи и возвращает индекс записи в массиве. При этом используется последовательный поиск по массиву.

3.2.1. Реализация метода поиска

```
list<int> adressSearch(string icol, bool wo){
    cout << "Введите текст/число для поиска: ";
    if (icol == "3"){
        string text;
        cin >> text;
        if (measureMap[text].size() == 0) {
            cout << "------------------------------\n"
                 << "Ничего не найдено\n";
        };
        printTableArr(measureMap[text], wo);
        return measureMap[text];
    }
    int i = std::stoi(icol)-1;
    if (i > 2) i--;
    int text;
    cin >> text;
    if (maps[i][text].size() == 0) {
        cout << "------------------------------\n"
             << "Ничего не найдено\n";
    };
    printTableArr(maps[i][text], wo);
    return maps[i][text];
}
```

Листинг 4 — *Метод адресного поиска*

Тут мы передаем в функцию номер колонки, по которой будем искать и `wo` – флаг, который указывает стоит ли выводить информацию о выходе из поиска после вывода результатов.

Мы запршиваем у пользователя ключ для поиска, если ищем единицы измерения, используем отдельный хеш. Поиском является само обращение к хешу.

3.2.2. Среднее количество сравнений

Таблица проиндексирована до поиска, поэтому сравнивать ничего не нужно, просто берем индексы записей, подходящих по ключу.

$$C = 0$$

3.2.3. Оценка времени поиска

В лучшем случае, если не возникает коллизий поиск выполняется за время генерации хеша из ключа (для этого используются хеш-функции). Хеш-функций много, но основная задача у них одна: используя некоторую математическую базу обеспечить одностороннее преобразование ключа в хеш (обычно обратное преобразование не требуется). При этом создается структура, похожая на динамический массив, но для доступа к элементам используются не индексы, а хеши. Если возникают коллизии программа выделяет большую облать памяти для хранения всей хеш-таблицы и полностью переносит всю хеш-таблицу туда. При вставки значений в хеш-таблицу коллизий избежать не удастся, но при получении значения по ключу они возникают редко (или вообще не возникают, зависит от хеш-функции). Возьмем для примера хеш-функцию на делении, общее количество тактов при делении – 28, поэтому на получение одного значения по ключу будет уходить около 30 тактов.

3.2.4. Оценка занимаемой памяти

Хеш-таблицы являются некоторым компромиссом между временем (удаления, доступа, вставки) и занимаемой памятью. В таблице всегда есть незаполненное пространство, которое во избежание коллизий увеличивается с увеличением самой таблицы. Отношения памяти, заполненной данными ко всей выделенной под таблицу называют коэффициентом загрузки. Оптимальным коэффициентом считают значение 0.5 и ниже, при этих значениях маленькая вероятность коллизии. Так как во всех записях обязательно должны быть значения их индексы обязательно попадут во все 5 таблиц.

$$V_{\text{таб}} = \frac{1}{\text{к-т загрузки}} \times 5 \times v_{\text{int}} \times k = 40k \text{ байт}$$

3.3. Метод упорядочивания

Метод сортировки пузырьком заключается в том, что наибольшие элементы «всплывают» (отсюда название – пузырек) в вверх массива. Для этого просто сравниваются два рядом стоящих элемента и если у элемента, индекс в массиве которого меньше, большее значение, то они меняются местами (при этом меняется их индекс в массиве, номер записи остается неизменным). Да-

лее тот же элемент сравнивается со следующим элементом, если он больше, то снова элементы меняются местами и так далее.

Так как сравниваются постоянно два элемента для реализации необходимо использовать вложенный цикл: первый элемент – тот, который будет «всплывать» будет с индексом `i` (`i = range(0, n)`), второй элемент – тот, с которым будем сравнивать первый, будет с индексом `j` (`j = range(i+1, n)`). Таким образом в худшем случае – когда массив отсортирован по убыванию будет необходимо пройти по всем элементам n^2 раз, а в лучшем, когда массив отсортирован, n раз.

3.3.1. Реализация метода упорядочивания

```
void bubbleSortUi(int nocol){
    for (int i = 0; i < k - 1; ++i) {
        for (int j = 0; j < k - i - 1; ++j) {
            if (tbl[j].get(nocol) > tbl[j + 1].get(nocol)) {
                norm temp = tbl[j];
                tbl[j] = tbl[j + 1];
                tbl[j + 1] = temp;
            }
        }
    }
}

void bubbleSortStr(){
    for (int i = 0; i < k - 1; i++) {
        for (int j = 0; j < k - 1 - i; j++) {
            if (tbl[j].measure.compare(tbl[j + 1].measure) > 0) {
                norm temp = tbl[j];
                tbl[j] = tbl[j + 1];
                tbl[j + 1] = temp;
            }
        }
    }
}
```

Листинг 5 — *Метод сортировки пузырьком*

Так как сравнения у целочисленного типа и строк сильно отличаются создадим две функции, для охвата всех полей таблицы. Отличаться тут будет только само условие: для целочисленного типа используем оператор больше, для строк используем функцию, которая последовательно сравнивает все буквы в строке, до того момента, как номер букв (в кодировке) в двух строках не будет отличаться, затем возвращет число, насколько этот номер отличается. Если это число больше 0, то вся запись с этой строкой «всплывает», таким

образом получаются записи, отсортированные в алфавитном порядке относительно единиц измерения.

3.3.2. Среднее количество сравнений

Внутри цикла мы сравниваем 2 значения (не важно какого типа), так как значение не должно сравниваться само с собой для каждого `i`-того элемента есть `k-1` значений, всего таких элементов `k`, поэтому получим:

$$C_{\text{пуз}} = k \times (k - 1)$$

3.3.3. Оценка времени упорядочивания

Посчитаем такты, которые выполняются внутри цикла, считать будем для сортировки целочисленных значений – строковые сравнения будут, в среднем выполняться в 2 раза дольше.

$$T_{\text{if}} = 2 \times \left(T_{\text{д}} + T_{\text{get}}\right) + T_{>} + t_{\text{if}} = 2 \times (2 + 2) + 2 + 1 = 11 \text{ тактов}$$

$$T_{\text{in\_if}} = 3 \times t_{=} + 4 \times T_{\text{д}} = 3 \times 2 + 4 \times 2 = 14 \text{ тактов}$$

Мы можем войти в условие по вероятности $p = 0.5$, тогда общее время упорядочивания будет равно:

$$T_{\text{пуз}} = \text{C}_{\text{пуз}} \times \left(T_{\text{if}} + p \times T_{\text{in\_if}}\right) =$$
$$= k \times (k - 1) \times (11 + 0.5 \times 14) = 18k(k - 1) \text{ тактов}$$

3.3.4. Оценка занимемой памяти

Так как все изменения происходят непосредственно в массиве, память используется только для индексов и замены записей местами.

$$V = 31 \text{ байт}$$

3.4. Метод корректировки

Удаление сдвигом. Все элементы, начиная с того, который необходимо удалить сдвигаются влево в памяти, таким образом затирая удаляемый элемент.

3.4.1. Реализация метода корректировки

```
void removeOffsetIndex(unsigned int index, bool wo){
    unsigned int tmp = index;
    index = idxSearch(index);
    int j = 0;
    for(int i = 0; i < 5 && j < 4; i++){
        if (i==2)i++;
        auto d = maps[j].find(tbl[index].get(i));
        if (d != maps[j].end()) {
            d->second.erase(whereList(d->second, tmp));
        }
        j++;
    }
    for (int i = index; i < k; ++i)
        tbl[i] = tbl[i + 1];
    cout << "Строка " << tmp << " успешно удалена\n";
    k--;
    if (wo){
        cout << "[x] В главное меню\n";
        string outi;
        cin  >> outi;
    }
}
```

Листинг 6 — *Метод удаления сдвигом*

В удалении мы используем тот же флаг, что и при выводе, чтобы при необходимости дать возможность пользователю выйти.

Удаление происходит в 3 этапа: по номеру записи находится индекс записи в массиве, номер записи удаляется из всех хешей, где он был в списках, что бы адресный поиск больше не смог найти индекс удаляемой записи и сама запись удаляется сдвигом из массива записей.

Но удалять запись по номеру не совсем удобно, поэтому я добавил так же удаление всех записей по списку, который может вернуть адресный поиск, таким образом реализовав поддрежку удаления записей по всем полям.

```
void removeOffsetArray(list<int> indexes){
    list<int> tmp(indexes);
    for (auto i: tmp){
        removeOffsetIndex(i, false);
    }
    cout << "[x] В главное меню\n";
    string outi;
    cin  >> outi;
}
```

Листинг 7 — *Метод удаления сдвигом (по полям)*

Тут видно, зачем нужен флаг в предыдущей функции, так как после каждой удаленной записи пользователю не нужно сообщение о выходе.

Необходимо так же показать реализацию метода поиска номера записи в списке. Предполагается, что списки, средней длиной $n \ll k$. При $k = 300$ длина списка в среднем будет равна 5-10 из логических соображений. Таким образом поиск по списку можно реализовать последовательный, при этом удаление все равно будет происходить быстро.

```
list<int>::const_iterator whereList(const list<int>& l, int a) {
    auto current = l.begin();
    while (*current != a && current != l.end())
        current++;
    return current;
}
```

Листинг 8 — *Последовательный поиск индекса в списке*

Поиск реализован так, что возвращает итератор, поэтому `erase` не проходится 2-ой раз по списку, а сразу освобождает элемент и переделывает ссылки предыдущего и следующего элемента (`std::list` - двусвязный список, но можно использовать собственную реализацию односвязного списка, тогда сравнивать необходимо, сохранив предыдущее значение и возвращать предыдущее значение. В целом для данной реализации это единственное отличие двусвязного списка от односвязного).

3.4.2. Среднее количество сравнений

Для поиска значений в списке используется последовательный поиск, для поиска индекса удаляемого элемента в массиве тоже используется последовательный поиск, поэтому количество сравнений будет в основном в этих поисках.

$$C_{\text{уд}} = \frac{5 \times n}{2} + \frac{k}{2} = \frac{5 \times \frac{k}{42}}{2} + \frac{k}{2} = \frac{5 \times k + 42 \times k}{84} = \frac{47k}{84}$$

3.4.3. Оценка времени удаления

Удаление, как было сказано ранее, происходит в 3 этапа, первые 2 по времени выполнения будут равны

$$T_{\text{поиск}} = (2 \times T_{\text{д}}) \times \frac{k}{2} + (2 \times T_{\rightarrow}) \times \frac{5k}{84} = 2k + \frac{5k}{84} = \frac{173}{84}k \text{ тактов}$$

со Видно, что поиск уже занимает достаточно много времени, решение этой проблемы заключается в выборе альтернативной структуры данных. (альтерантивное решение)

Для удаления тоже потребуется некоторое число тактов, а именно доступ и запись k-i (в среднем $\frac{k}{2}$) элементов.

$$T_{\text{уд}} = (T_{\text{д}} + T_{=}) \times \frac{k}{2} = 4 \times \frac{k}{2} = 2k \text{ тактов}$$

$$T_{\text{уд\_общ}} = T_{\text{поиск}} + T_{\text{уд}} = \frac{173}{84}k + 2k = \frac{341}{84}k \simeq 4k \text{ тактов}$$

Эффективное по времени удаление данных из массива в лучших случаях достигает 3-4 такта, будем стремиться к этому в альтернативном варианте.

3.4.4. Оценка освобождаемой памяти

При удалении освобождается память, занимаемая списками в хешах, но из-за статического масссива при затирании элементов длина массива не сокращается, т.е. даже после удаления элемента память, выделенная для него остается, так как в задании нет необходимости реализации добавления новых элементов эта память остается пустой в конце массива до завершения программы. За каждое удаление мы освобождаем от данных 23 байта. Полностью освобождается только память, занимаемая номером в списках, т.е. 20 байт.

4. Альтернативный вариант

4.1. Структура данных

В основном случае был расмотрен неплохой вариант организации таблицы. Поиск выполняется за константу, что при большом количестве элементов все равно лучше, чем зависимость от этого количества. Хорошим показателем памяти обладает алгоритм упорядочивания. Алгоритм удаления вышел самым неэффективным – во-первых память после удаления элемента не освобождается, во-вторых для удаления необходимо пройти по всем элементам несколько раз.

Все проблемы заключаются в двух доступах: по индексу массива и по номеру записи, но если организовывать один доступ в массиве прийдется при удалении и сортировке переиндексировать полностью таблицу. Это увеличит время алгоритмов и они станут неэффективными. Так как скорость выполнения в наше время имеет больший приоретет, можно увеличить количество памяти, занимаемой таблицей, для того, что бы ускорить сортировку и удаление.

Одним из самых эффективных алгоритмов удаления является алгоритм маркировки, так как он только помечает в памяти удаляемую запись, и только раз в несколько (десятков, сотен или даже тысяч) удалений освобождает всю память, занимаемую уже удаленными элементами.

Итак, для альтернативного решения необходимо подобрать такую структуру данных, чтобы в ней были индексы, которые не меняются при сортировке или удалении других элементов, доступ по индексу должен производиться за константу, и должен быть порядок вывода этих записей – для отображения результатов работы сортировки.

Будем использовать динамический массив, который будет хранить в себе номера записей, при сортировке эти номера будут меняться местами, а если i-тый (i - номер элемента, а не индекс в массиве) элемент удален, то поставим в его еденицы измерения «~», как флаг удаленного элемента (сборщик мусора потом пройдет по этому массиву и удалит все записи с «~» как из массива, так и из памяти).

Из самих записей уберем номер – теперь этот номер будет использоваться непосредственно как ключ в отдельной хеш таблице, значения которой будут сами записи. Таким образом мы получаем быстрый доступ к элементам по индексу массива или по номеру записи.

Предыдущая сортировка эффективна по памяти, но по времени она проигрывает быстрой сортировке, поэтому в данном решении будем использовать сортировку Хоара.

Адресный поиск оставим без изменений, он эффективен по времени.

Вот схема структуры данных для альтернаятивного решения:

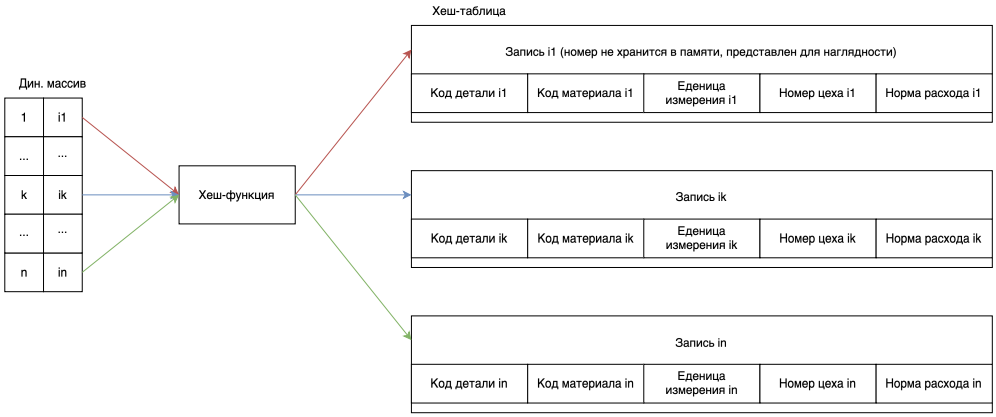

Рисунок 2 — *Схема структуры данных для альтернативного решения*

4.1.1. Реализация структуры на языке C++

```
struct norm {
    unsigned int detailCode;
    unsigned int materialCode;
    std::string measure;
    unsigned int workshopNum;
    unsigned int consumptionNorm;

    unsigned int& get(int index) {
        switch (index) {
            case 0: return detailCode;
            case 1: return materialCode;
            case 3: return workshopNum;
            case 4: return consumptionNorm;
            default: throw std::out_of_range("Invalid index");
        }
    }
}
```

Листинг 9 — *Структура записей* `norm`

```
typedef unordered_map<unsigned int, norm > tytbl;
tytbl tbl;
vector<int> stbl;
```

Листинг 10 — *Объявление хеш-таблицы записей и дин. массива индексов*

4.1.2. Рассчет памяти, занимаемой структурой

Память занимаемая структурой состоит из 2 частей: память массива и память хеш-таблицы.

$$V_{\text{хеш}} = \frac{1}{\text{к-т загрузки}} \times V_{\text{зап}} \times k = \frac{k}{\text{к-т загрузки}} \times$$

$$\times \left(l_{\text{det}} + l_{\text{mat}} + l_{\text{mea}} + l_{\text{ws}} + l_{\text{cn}}\right) \simeq \frac{k}{0.5} \times (4 \times 4 + 3) = 38k \text{ байт}$$

$$V_{\text{мас}} = V_{\text{int}} \times k = 4k \text{ байт}$$

$$V_{\text{ст}} = V_{\text{хеш}} + V_{\text{мас}} = 46k + 4k = 42k \text{ байт}$$

Почусили на $14k$ байт больше, чем в прошлой реализации, но тут данные хранятся динамически, поэтому k вводится в процессе выполнения программы и может быть изменено. При большом разбросе количества записей этот факт может компенсировать увеличение объема памяти.

4.1.3. Оценка времени доступа к i-му элементу

Увеличение памяти, занимвемой структурой приводит к уменьшению времени доступа к элементам.

Доступ к элементам может быть через индекс массива $T_{\text{д}}$ или через номер элемента $T_{\text{ди}}$. Так как теперь эти два индекса связанны в одном массиве, то справедливо равенство

$$T_{\text{д}} = t_{++} + t_{\rightarrow} + T_{\text{ди}} = 2 + T_{\text{ди}} \text{ тактов}$$

В части 3.2.3 оценено примерное время доступа к одному значению по ключу:

$$T_{\text{ди}} = 30 \text{ тактов}$$

Как мы видим теперь оба доступа к значениям константы, что хорошо для большого количества значений.

4.1.4. Оценка времени удаления i-го элемента

list adressSearch(string icol, bool wo); Так как удаляется не каждый элемент, введем `#define ev 10`, где 10 – количество вызовов, через которое элементы будут удаляться. Общее удаление будет происходить сдвигом, размер сдвига будет увеличиваться, каждый раз, когда будет необходимо пропустить-(затереть) текущий элемент и таким образом к концу прохода массива сдвиг будет равен `ev`. Очищать будем и хеш и массив, всего необходимо очистить за 1 раз 46 ev байт. (подробнее см. 4.4.3)

4.2. Метод поиска

4.2.1. Реализация метода поиска

```
list<int> adressSearch(string icol, bool wo){
    cout << "Введите текст/число для поиска: ";
    if (icol == "3"){
        string text;
        cin >> text;
        if (measureMap[text].size() == 0) {
            cout << "-------------------------------\n"
                 << "Ничего не найдено\n";
        };
        printTableArr(measureMap[text], wo);
        return measureMap[text];
    }
    int i = std::stoi(icol)-1;
    if (i > 2) i--;
    int text;
    cin >> text;
    if (maps[i][text].size() == 0) {
        cout << "-------------------------------\n"
             << "Ничего не найдено\n";
    };
    printTableArr(maps[i][text], wo);
    return maps[i][text];
}
```

Листинг 11 — *Метод адресного поиска*

Так как общие структуры для индексации не изменились с предыдущего решения сам алгоритм поиска тоже не изменился. Поэтому в основном все, что сказано в части 3.2 справедливо и тут.

4.2.2. Среднее количество сравнений

$$C = 0$$

4.2.3. Оценка времени поиска

На получение одного значения по ключу, в среднем, уходит около 30 тактов

4.2.4. Оценка занимаемой памяти

$$V_{\text{таб}} = \frac{1}{\text{к-т загрузки}} \times 5 \times v_{\text{int}} \times k = 40k \text{ байт}$$

4.3. Метод упорядочивания

Быстрая сортировка организована так: берется центральный элемент массива, а так же 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.

4.3.1. Реализация метода упорядочивания

```
void qSortI(vector<int>::iterator stbl, int k){
    int i = 0, j = k;
    int p;
    p = *(stbl+(k>>1));
    int temp;

    do {
        while (*(stbl + i) < p) i++;
        while (*(stbl + j) > p) j--;
        if (i <= j){
            temp = *(stbl + i);
            *(stbl + i) = *(stbl + j);
            *(stbl + j) = temp;
            i++;
            j--;
        }
    } while (i <= j);

    if ( j > 0 ) qSortI(stbl, j);
    if ( k > i ) qSortI(stbl + i, k-i);
}
```

Листинг 12 — *Сортировка Хоара для номеров записей*

```
void qSortUi(int nocol, vector<int>::iterator stbl, int k) {
    int i = 0, j = k;
    unsigned int p;
    p = tbl[*(stbl+(k>>1))].get(nocol);
    int temp;
    do {
        while (tbl[*(stbl + i)].get(nocol) < p) i++;
        while (tbl[*(stbl + j)].get(nocol) > p) j--;
        if (i <= j){
            temp = *(stbl + i);
            *(stbl + i) = *(stbl + j);
            *(stbl + j) = temp;
            i++;
            j--;
        }
    } while (i <= j);
    if ( j > 0 ) qSortUi(nocol, stbl, j);
    if ( k > i ) qSortUi(nocol, stbl + i, k-i);
}
void qSortStr(vector<int>::iterator stbl, int k){
    int i = 0, j = k;
    string p;
    p = tbl[*(stbl+(k>>1))].measure;
    int temp;
    do {
        while (tbl[*(stbl + i)].measure.compare(p) < 0) i++;
        while (tbl[*(stbl + j)].measure.compare(p) > 0) j--;
        if (i <= j){
            temp = *(stbl + i);
            *(stbl + i) = *(stbl + j);
            *(stbl + j) = temp;
            i++;
            j--;
        }
    } while (i <= j);
    if ( j > 0 ) qSortStr(stbl, j);
    if ( k > i ) qSortStr(stbl + i, k-i);
}
```

Листинг 13 — *Сортировка Хоара для целочисленного и строкового типа*

В данной реализации я решил создать 3 разные функции сортировки: для целочисленных, строквых полей и номеров записей в массиве.

4.3.2. Среднее количество сравнений

В данной сортировке используются 2 сравнения: сравнения с центральным элементом и сравнения индексов. Для начала посчитаем все для одного прохода. Тут n - кол-во обрабатываемых элементов, на первой итерации оно совпадает с k, на второй вдвое меньше и т.д.

В цикле произойдет от 1 до $\frac{n}{2}$ сравнений индексов, но по статистике в этот цикл входят повторно с 40% вероятностью (К. Дин «Простой анализ ожидаемого времени выполнения для рандомизированных алгоритмов “разделяй и властвуй”»). После цикла условия входа в рекурсию.

$$C_{\text{инд}} = 1 + 0.4 + 2 = 3.4$$

В цикле 2 раза войдем в цикл с $\frac{n}{4}$ в среднем значений, основной цикл выполняется

$$C_{\text{piv}} = 1.4 \times \left(\frac{n}{4} + \frac{n}{4}\right) = 1.4 \times \frac{n}{2} = 0.7n$$

$$C_{\text{общ}} = C_{\text{инд}} + C_{\text{piv}} = 3.4 + 0.7n$$

Так как мы постоянно разделяем массив на 2 всего функцию мы вызовем примерно $2k$ раз.

$$C_{\text{общ\_rec}} = (3.4 + 0.7n) \times 2k$$

Причем $n = \log_2 k$

$$C_{\text{общ\_rec}} = 6.8k + 2k \log_2 k$$

4.3.3. Оценка времени упорядочивания

Для начала посчитаем такты для одной функции, затем умножим на $2k$ это значение и подставим $n = \log_2 k$

Так как сама функция достаточно сложна подстичтаем только такты, учавствующие непосредственно в сравнениях, потому что остальные такты для всех методов, основанных на прямой перестановке будут примерно одинаковы.

$$T_{\text{инд}} = 4 \times C_{\text{инд}} = 13.6$$

$$T_{\text{piv}} = T_{\text{д}} \times C_{\text{piv}} = 30 \times 0.7n = 21n$$

$$T_{\text{общ}} = T_{\text{инд}} + T_{\text{piv}} = 13.6 + 21n$$

$$T_{\text{общ\_rec}} = 27.2k + 42k \log_2 k$$

4.3.4. Оценка используемой памяти

Функция рекурсивная, поэтому до окончания ее выполнения занимает некоторое место в стеке, но в сегменте данных она ничего не выделяет и работает только с текущим массивом, поэтому можно считать, что место используется только для хранения адреса массива, 2 индексов и серединного элемента. И так для каждой из $2k$ вызываемых рекурсивно функций.

$$V = 2k \times (4 + 2 \times 4 + 4) = 32k \text{ байт}$$

В результате получаем, что для организации рекурсии используется $32k$ байт стека. Но любая рекурсия может быть организована с помощью цикла, поэтому этой памятью можно принебречь.

4.4. Удаление маркировкой и сдвигом

4.4.1. Реализация метода удаления маркировкой и сдвигом

```
void removeMarkIndex(unsigned int index, bool wo){
    tbl[index].measure = "~";
    cout << "Строка " << index << " успешно удалена\n";
    evi++;
    if (evi == ev){
        int j = 0;
        for(int i = 0; i < k - j; i++){
            if (tbl[stbl[i+j]].measure == "~") {
                j++;
                clearHash(stbl[i]);
                tbl.erase(tbl.find(stbl[i]));
            }
            stbl[i] = stbl[i+j];
        }
        k-=ev;
        for(int i = 0; i < ev; i++) stbl.pop_back();
        cout << "Чистка мусора прошла успешно\n";
        evi = 0;
    }
    if (wo){
        cout << "[x] В главное меню\n";
        string outi;
        cin  >> outi;
    }
}
```

Листинг 14 — *Удаление маркировкой по индексу*

Глобальная переменная `evi` теперь считает количество удаленных элементов, если их теперь столько, сколько и выставленный предел все элементы одним циклом удаляются полностью. Иначе просто заменяем еденицу измерения «~» и элементы не отображаются (так написана функция отображения).

Идея с удалением по столбцам осталась такой же, поменялся только код реализации удаления. Удаление из хешей (для адресного поиска) тоже не изменилось, но ушло в отдельную функцию для лучшей читаймости кода.

4.4.2. Среднее количество сравнений

Сравнения требуются только при очистке и их ровно k.

$$C_{\text{уд}} = \frac{k}{\text{ev}}$$

4.4.3. Оценка времени удаления

Для начала разберем время без очистки: оно расходуется только на добавление «~».

$$T_{\text{уд}} = T_{\text{д}} + t_{++} + t_{=} = 30 + 2 + 2 = 34 \text{ такта}$$

При очистке мы проходим весь массив так же, как это делали каждый раз в основном решении.

$$T_{\text{оч}} = 2k$$

$$T_{\text{общ}} = T_{\text{уд}} + \frac{T_{\text{оч}}}{\text{ev}} = 34 + \frac{2k}{\text{ev}}$$

4.4.4. Оценка освобождаемой памяти

Понятно, что удаление будет самым быстрым, если `ev` будет большим, но с другой стороны, с каждым увеличением `evi` мы используем большее количество ненужной памяти. Одна запись в таблице занимает 42 байта. В массиве хранится индекс, который занимает 4 байта. В хешах адресного поиска хранится 20 байт индексов.

$$V_{\text{ос}} = (42 + 4 + 20) * ev = 66ev \text{ байт}$$

5. Таблица результатов и вывод

	Структура данных	Метод поиска	Метод упорядочения	Метод корректировки
Основной вариант				
Название	Массив записей	Адресный поиск	Сортировка пузырьком	Удаление смещением
Занимаемая/освобождаемая память (байт)	$\mathbf{23k}$	$\mathbf{40k}$	$\mathbf{31}$	$23(p) + 20$
Среднее количество сравнений	–	$\mathbf{0}$	$k \times (k-1)$	$\frac{47}{84}k$
Занимаемое время (такты)	$2/2i$	$\mathbf{30}$	$18k(k-1)$	$\frac{341}{84}k$
Альтернативный вариант				
Название	Массив индексов и хеш-таблица	Адресный поиск	Сортировка Хоара	Удаление маркировкой и смещением
Занимаемая/освобождаемая память (байт)	$42k$	$\mathbf{40k}$	32	$\mathbf{66ev}$
Среднее количество сравнений	–	$\mathbf{0}$	$\mathbf{6.8k + +2k \log_2 k}$	$\mathbf{\frac{k}{ev}}$
Занимаемое время (такты)	$\mathbf{32/30}$	$\mathbf{30}$	$\mathbf{27.2k + +42k \log_2 k}$	$\mathbf{32}$

Как видно из таблицы, альтернативный вариант выигравает основной по времени (во всех методах) и по освобождаемой памяти, но занимет больше места в оперативной памяти.

5.1. Вывод

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.

6. Приложения

В процессе разработки лабораторной работы для отладки методов были созданы два консольных приложения. Ниже приведен код этих приложений.

6.1. Основной вариант

```
#include "unordered_map"
#include "list"
#include "string"

#define max 300

using  std::cin,  std::cout,  std::unordered_map,  std::list,
std::string;

typedef unordered_map<unsigned int, list<int> > uimap;
typedef unordered_map<string, list<int> > smap;

struct norm {
    unsigned int detailCode;
    unsigned int materialCode;
    std::string measure;
    unsigned int workshopNum;
    unsigned int consumptionNorm;
    unsigned int no;

    unsigned int& get(int index) {
        switch (index) {
            case 0: return detailCode;
            case 1: return materialCode;
            case 3: return workshopNum;
            case 4: return consumptionNorm;
            case 5: return no;
            default: throw std::out_of_range("Invalid index");
        }
    }
};


#include "lab1ex.cpp"

norm tbl[max];
uimap maps[4] = {{},{},{},{}};
smap  measureMap = {};
int k = 0;
```

```
int getTableLength();
void getTable();
void getdata();

void mainMenu();

void adressSearchMenu();
list<int> adressSearch(string icol, bool wo);
template<typename tmap, typename t> void putmap(tmap &map, t key,
int value);

void bubbleSortMenu();
void bubbleSortUi(int nocol);
void bubbleSortStr();

void removeOffsetMenu();
void removeOffsetIndex(unsigned int index, bool wo);
void removeOffsetArray(list<int> indexes);

void printTable();
void printTableArr(list<int> arr, bool wo);


void clear() {
    std::cout << "\x1B[2J\x1B[H";
}


int getTableLength(){
    int res;
    cout << "Введите количество записей (min: 1, max: " << max <<
").....";
    cin  >> res;
    return res;
}

void getdata(){
    for (int i = 0; i < 40; i++) cout << "\n";
    clear();
    cout << "Лабораторная работа 1\n"
         << "Ввод данных\n"
         << "[1] Ввести вручную\n"
         << "[2] Использовать пример\n"
```

```
         << "[x] Выйти\n";
    string text;
    cin >> text;
    if (text == "x") {
        quit = true;
        return;
    }
    if (text == "1") {
        k = getTableLength();
        getTable();
        return;
    };
    k = 300;
    lab1ex(tbl);

    for (int i = 0; i < k; i++){
        putmap(maps[0],    tbl[i].detailCode,      i);
        putmap(maps[1],    tbl[i].materialCode,    i);
        putmap(measureMap, tbl[i].measure,         i);
        putmap(maps[2],    tbl[i].workshopNum,     i);
        putmap(maps[3],    tbl[i].consumptionNorm, i);
    }

    return;
}

void mainMenu(){
    clear();
    cout << "Лабораторная работа 1\n"
         << "[1] Адресный поиск\n"
         << "[2] Сортировка пузырьком\n"
         << "[3] Удаление сдвигом\n"
         << "[p] Вывести таблицу\n"
         << "[x] Выйти\n";
    string val;
    cin >> val;
    if (val == "1"){
        adressSearchMenu();
    }
    if (val == "2"){
        bubbleSortMenu();
```

```
    }
    if (val == "3"){
        removeOffsetMenu();
    }
    if (val == "p"){
        printTable();
    }
    if (val== "x"){
        quit = true;
    }
    return;
}

void adressSearchMenu(){
    clear();
    cout << "Лаболаторная работа 1\n"
         << "Адресный поиск\n"
         << "Выберите столбец, по которому необходимо искать:\n"
         << "[1] Код детали\n"
         << "[2] Код материала\n"
         << "[3] Единица измерения\n"
         << "[4] Номер цеха\n"
         << "[5] Норма расхода\n"
         << "[x] В главное меню\n";
    string icol;
    cin >> icol;
    if (icol == "x") return;
    adressSearch(icol, true);
}

list<int> adressSearch(string icol, bool wo){
    cout << "Введите текст/число для поиска: ";
    if (icol == "3"){
        string text;
        cin >> text;
        if (measureMap[text].size() == 0) {
            cout << "------------------------------\n"
                 << "Ничего не найдено\n";
        };
        printTableArr(measureMap[text], wo);
        return measureMap[text];
```

```
    }
    int i = std::stoi(icol)-1;
    if (i > 2) i--;
    int text;
    cin >> text;
    if (maps[i][text].size() == 0) {
        cout << "------------------------------\n"
             << "Ничего не найдено\n";
    };
    printTableArr(maps[i][text], wo);
    return maps[i][text];
}


void bubbleSortMenu(){
    clear();
    cout << "Лаболаторная работа 1\n"
         << "Сортировка пузырьком\n"
        << "Выберите столбец, по которому необходимо сортировать:
\n"
         << "[1] Код детали\n"
         << "[2] Код материала\n"
         << "[3] Единица измерения\n"
         << "[4] Номер цеха\n"
         << "[5] Норма расхода\n"
         << "[6] Номер записи\n"
         << "[x] В главное меню\n";
    string icol;
    cin >> icol;
    if (icol == "x") return;
    if (icol != "3"){
        int nocol = stoi(icol);
        nocol--;
        bubbleSortUi(nocol);
    } else {
        bubbleSortStr();
    }
    cout << "Отсортировано!\n"
         << "[x] В главное меню\n";
    cin >> icol;
    return;
}
```

```
void removeOffsetMenu(){
    clear();
    cout << "Лаболаторная работа 1\n"
         << "Удаление сдвигом\n"
        << "Выберите столбец, по которому необходимо искать данные
для удаления записей:\n"
         << "[i] Удалить по номеру записи\n"
         << "[1] Код детали\n"
         << "[2] Код материала\n"
         << "[3] Единица измерения\n"
         << "[4] Номер цеха\n"
         << "[5] Норма расхода\n"
         << "[x] В главное меню\n";
    string icol;
    cin >> icol;
    if (icol == "x") return;
    if (icol == "i") {
        cout << "Введите номер записи: ";
        cin  >> icol;
        removeOffsetIndex(stoi(icol), true);
        return;
    }
    removeOffsetArray(adressSearch(icol, false));
}

list<int>::const_iterator whereList(const list<int>& l, int a) {
    auto current = l.begin();
    while (*current != a && current != l.end())
        current++;
    return current;
}

unsigned int idxSearch(unsigned int idx){
    for (int i = 0; i < k; i++){
        if (tbl[i].no == idx) return i;
    }
    return 0;
}

void removeOffsetIndex(unsigned int index, bool wo){
    unsigned int tmp = index;
    index = idxSearch(index);
```

```
    int j = 0;
    for(int i = 0; i < 5 && j < 4; i++){
        if (i==2)i++;
        auto d = maps[j].find(tbl[index].get(i));
        if (d != maps[j].end()) {
            d->second.erase(whereList(d->second, tmp));
        }
        j++;
    }
    for (int i = index; i < k; ++i)
        tbl[i] = tbl[i + 1];
    cout << "Строка " << tmp << " успешно удалена\n";
    k--;
    if (wo){
        cout << "[x] В главное меню\n";
        string outi;
        cin  >> outi;
    }
}

void removeOffsetArray(list<int> indexes){
    list<int> tmp(indexes);
    for (auto i: tmp){
        removeOffsetIndex(i, false);
    }
    cout << "[x] В главное меню\n";
    string outi;
    cin  >> outi;
}

void bubbleSortUi(int nocol){
    for (int i = 0; i < k - 1; ++i) {
        for (int j = 0; j < k - i - 1; ++j) {
            if (tbl[j].get(nocol) > tbl[j + 1].get(nocol)) {
                norm temp = tbl[j];
                tbl[j] = tbl[j + 1];
                tbl[j + 1] = temp;
            }
        }
    }
}
```

```
void bubbleSortStr(){
    for (int i = 0; i < k - 1; i++) {
        for (int j = 0; j < k - 1 - i; j++) {
            if (tbl[j].measure.compare(tbl[j + 1].measure) > 0) {
                norm temp = tbl[j];
                tbl[j] = tbl[j + 1];
                tbl[j + 1] = temp;
            }
        }
    }
}

template<typename tmap, typename t> void putmap(tmap &map, t key,
int value){
    if (map.find(key) == map.end()){
        list<int> res = {};
        res.push_back(value);
        map[key] = res;
    } else {
        map[key].push_back(value);
    }
}

void getTable(){
    for (int i = 0; i < k; i++){
        tbl[i].no = i;
        cout << "-------------------------------\n";
        cout << "Запись............." << tbl[i].no <<"\n";
        cout << "Код детали........";
        cin  >> tbl[i].detailCode;
        cout << "Код материала.......";
        cin  >> tbl[i].materialCode;
        cout << "Единица измерения...";
        cin  >> tbl[i].measure;
        cout << "Номер цеха.........";
        cin  >> tbl[i].workshopNum;
        cout << "Норма расхода.......";
        cin  >> tbl[i].consumptionNorm;
        putmap(maps[0],    tbl[i].detailCode,      tbl[i].no);
        putmap(maps[1],    tbl[i].materialCode,    tbl[i].no);
        putmap(measureMap, tbl[i].measure,         tbl[i].no);
```

```
        putmap(maps[2],    tbl[i].workshopNum,     tbl[i].no);
        putmap(maps[3],    tbl[i].consumptionNorm, tbl[i].no);
    };
    cout << "-------------------------------\n";
}

void printTableArr(list<int> arr, bool wo){
    for (auto i: arr){
        unsigned int idx = idxSearch(i);
        cout << "-------------------------------\n"
           << "Запись.............." << tbl[idx].no            <<
"\n"
              << "Код детали........." << tbl[idx].detailCode
<< "\n"
             << "Код материала......." << tbl[idx].materialCode
<< "\n"
          << "Единица измерения..." << tbl[idx].measure       <<
"\n"
              << "Номер цеха........." << tbl[idx].workshopNum
<< "\n"
           << "Норма расхода......." << tbl[idx].consumptionNorm
<< "\n";
    };
    cout << "-------------------------------\n";
    if (wo){
        cout << "[x] Назад\n";
        string text;
        cin >> text;
    }
    return;
}

void printTable(){
    for (int i = 0; i < k; i++){
        cout << "-------------------------------\n"
            << "Запись.............." << tbl[i].no            <<
"\n"
           << "Код детали........." << tbl[i].detailCode     <<
"\n"
           << "Код материала......." << tbl[i].materialCode    <<
"\n"
           << "Единица измерения..." << tbl[i].measure       <<
"\n"
              << "Номер цеха........." << tbl[i].workshopNum
<< "\n"
```

```
            << "Норма расхода......." << tbl[i].consumptionNorm
<< "\n";
    };
    cout << "------------------------------\n";
    cout << "[x] Назад\n";
    string text;
    cin >> text;
    return;
}



int main(){
   getdata();
   while (!quit) mainMenu();
   return 0;
}
```

6.2. Альтернативный вариант

```
#include "unordered_map"
#include "list"
#include "string"

#define max 300

using std::cin, std::cout, std::unordered_map, std::list,
std::string;

typedef unordered_map<unsigned int, list<int> > uimap;
typedef unordered_map<string, list<int> > smap;

struct norm {
    unsigned int detailCode;
    unsigned int materialCode;
    std::string measure;
    unsigned int workshopNum;
    unsigned int consumptionNorm;
    unsigned int no;
```

```
    unsigned int& get(int index) {
        switch (index) {
            case 0: return detailCode;
            case 1: return materialCode;
            case 3: return workshopNum;
            case 4: return consumptionNorm;
            case 5: return no;
            default: throw std::out_of_range("Invalid index");
        }
    }
};


#include "lab1ex.cpp"

norm tbl[max];
uimap maps[4] = {{},{},{},{}};
smap  measureMap = {};
int k = 0;
bool quit = 0;




int getTableLength();
void getTable();
void getdata();

void mainMenu();

void adressSearchMenu();
list<int> adressSearch(string icol, bool wo);
template<typename tmap, typename t> void putmap(tmap &map, t key,
int value);

void bubbleSortMenu();
void bubbleSortUi(int nocol);
void bubbleSortStr();

void removeOffsetMenu();
void removeOffsetIndex(unsigned int index, bool wo);
void removeOffsetArray(list<int> indexes);

void printTable();
void printTableArr(list<int> arr, bool wo);
```

```
void clear() {
    std::cout << "\x1B[2J\x1B[H";
}


int getTableLength(){
    int res;
    cout << "Введите количество записей (min: 1, max: " << max <<
").....";
    cin  >> res;
    return res;
}

void getdata(){
    for (int i = 0; i < 40; i++) cout << "\n";
    clear();
    cout << "Лабораторная работа 1\n"
         << "Ввод данных\n"
         << "[1] Ввести вручную\n"
         << "[2] Использовать пример\n"
         << "[x] Выйти\n";
    string text;
    cin >> text;
    if (text == "x") {
        quit = true;
        return;
    }
    if (text == "1") {
        k = getTableLength();
        getTable();
        return;
    };
    k = 300;
    lab1ex(tbl);

    for (int i = 0; i < k; i++){
        putmap(maps[0],    tbl[i].detailCode,      i);
        putmap(maps[1],    tbl[i].materialCode,    i);
        putmap(measureMap, tbl[i].measure,         i);
        putmap(maps[2],    tbl[i].workshopNum,     i);
        putmap(maps[3],    tbl[i].consumptionNorm, i);
    }

    return;
}
```

```
void mainMenu(){
    clear();
    cout << "Лабораторная работа 1\n"
         << "[1] Адресный поиск\n"
         << "[2] Сортировка пузырьком\n"
         << "[3] Удаление сдвигом\n"
         << "[p] Вывести таблицу\n"
         << "[x] Выйти\n";
    string val;
    cin >> val;
    if (val == "1"){
        adressSearchMenu();
    }
    if (val == "2"){
        bubbleSortMenu();
    }
    if (val == "3"){
        removeOffsetMenu();
    }
    if (val == "p"){
        printTable();
    }
    if (val== "x"){
        quit = true;
    }
    return;
}

void adressSearchMenu(){
    clear();
    cout << "Лаболаторная работа 1\n"
         << "Адресный поиск\n"
         << "Выберите столбец, по которому необходимо искать:\n"
         << "[1] Код детали\n"
         << "[2] Код материала\n"
         << "[3] Единица измерения\n"
         << "[4] Номер цеха\n"
         << "[5] Норма расхода\n"
         << "[x] В главное меню\n";
    string icol;
    cin >> icol;
    if (icol == "x") return;
    adressSearch(icol, true);
}
```

```
list<int> adressSearch(string icol, bool wo){
    cout << "Введите текст/число для поиска: ";
    if (icol == "3"){
        string text;
        cin >> text;
        if (measureMap[text].size() == 0) {
            cout << "------------------------------\n"
                 << "Ничего не найдено\n";
        };
        printTableArr(measureMap[text], wo);
        return measureMap[text];
    }
    int i = std::stoi(icol)-1;
    if (i > 2) i--;
    int text;
    cin >> text;
    if (maps[i][text].size() == 0) {
        cout << "------------------------------\n"
             << "Ничего не найдено\n";
    };
    printTableArr(maps[i][text], wo);
    return maps[i][text];
}


void bubbleSortMenu(){
    clear();
    cout << "Лаболаторная работа 1\n"
         << "Сортировка пузырьком\n"
        << "Выберите столбец, по которому необходимо сортировать:
\n"
         << "[1] Код детали\n"
         << "[2] Код материала\n"
         << "[3] Единица измерения\n"
         << "[4] Номер цеха\n"
         << "[5] Норма расхода\n"
         << "[6] Номер записи\n"
         << "[x] В главное меню\n";
    string icol;
    cin >> icol;
    if (icol == "x") return;
```

```
    if (icol != "3"){
        int nocol = stoi(icol);
        nocol--;
        bubbleSortUi(nocol);
    } else {
        bubbleSortStr();
    }
    cout << "Отсортировано!\n"
         << "[x] В главное меню\n";
    cin >> icol;
    return;
}

void removeOffsetMenu(){
    clear();
    cout << "Лаболаторная работа 1\n"
         << "Удаление сдвигом\n"
        << "Выберите столбец, по которому необходимо искать данные
для удаления записей:\n"
         << "[i] Удалить по номеру записи\n"
         << "[1] Код детали\n"
         << "[2] Код материала\n"
         << "[3] Единица измерения\n"
         << "[4] Номер цеха\n"
         << "[5] Норма расхода\n"
         << "[x] В главное меню\n";
    string icol;
    cin >> icol;
    if (icol == "x") return;
    if (icol == "i") {
        cout << "Введите номер записи: ";
        cin  >> icol;
        removeOffsetIndex(stoi(icol), true);
        return;
    }
    removeOffsetArray(adressSearch(icol, false));
}

list<int>::const_iterator whereList(const list<int>& l, int a) {
    auto current = l.begin();
    while (*current != a && current != l.end())
        current++;
    return current;
}
```

```
unsigned int idxSearch(unsigned int idx){
    for (int i = 0; i < k; i++){
        if (tbl[i].no == idx) return i;
    }
    return 0;
}

void removeOffsetIndex(unsigned int index, bool wo){
    unsigned int tmp = index;
    index = idxSearch(index);
    int j = 0;
    for(int i = 0; i < 5 && j < 4; i++){
        if (i==2)i++;
        auto d = maps[j].find(tbl[index].get(i));
        if (d != maps[j].end()) {
            d->second.erase(whereList(d->second, tmp));
        }
        j++;
    }
    for (int i = index; i < k; ++i)
        tbl[i] = tbl[i + 1];
    cout << "Строка " << tmp << " успешно удалена\n";
    k--;
    if (wo){
        cout << "[x] В главное меню\n";
        string outi;
        cin  >> outi;
    }
}

void removeOffsetArray(list<int> indexes){
    list<int> tmp(indexes);
    for (auto i: tmp){
        removeOffsetIndex(i, false);
    }
    cout << "[x] В главное меню\n";
    string outi;
    cin  >> outi;
}
```

```
void bubbleSortUi(int nocol){
    for (int i = 0; i < k - 1; ++i) {
        for (int j = 0; j < k - i - 1; ++j) {
            if (tbl[j].get(nocol) > tbl[j + 1].get(nocol)) {
                norm temp = tbl[j];
                tbl[j] = tbl[j + 1];
                tbl[j + 1] = temp;
            }
        }
    }
}

void bubbleSortStr(){
    for (int i = 0; i < k - 1; i++) {
        for (int j = 0; j < k - 1 - i; j++) {
            if (tbl[j].measure.compare(tbl[j + 1].measure) > 0) {
                norm temp = tbl[j];
                tbl[j] = tbl[j + 1];
                tbl[j + 1] = temp;
            }
        }
    }
}

template<typename tmap, typename t> void putmap(tmap &map, t key,
int value){
    if (map.find(key) == map.end()){
        list<int> res = {};
        res.push_back(value);
        map[key] = res;
    } else {
        map[key].push_back(value);
    }
}

void getTable(){
    for (int i = 0; i < k; i++){
        tbl[i].no = i;
        cout << "------------------------------\n";
        cout << "Запись............." << tbl[i].no <<"\n";
        cout << "Код детали.........";
        cin  >> tbl[i].detailCode;
        cout << "Код материала.......";
        cin  >> tbl[i].materialCode;
        cout << "Единица измерения...";
```

```
        cin  >> tbl[i].measure;
        cout << "Номер цеха.........";
        cin  >> tbl[i].workshopNum;
        cout << "Норма расхода.......";
        cin  >> tbl[i].consumptionNorm;
        putmap(maps[0],    tbl[i].detailCode,      tbl[i].no);
        putmap(maps[1],    tbl[i].materialCode,    tbl[i].no);
        putmap(measureMap, tbl[i].measure,         tbl[i].no);
        putmap(maps[2],    tbl[i].workshopNum,     tbl[i].no);
        putmap(maps[3],    tbl[i].consumptionNorm, tbl[i].no);
    };
    cout << "-------------------------------\n";
}

void printTableArr(list<int> arr, bool wo){
    for (auto i: arr){
        unsigned int idx = idxSearch(i);
        cout << "-------------------------------\n"
           << "Запись............." << tbl[idx].no            <<
"\n"
              << "Код детали........." << tbl[idx].detailCode
<< "\n"
             << "Код материала......." << tbl[idx].materialCode
<< "\n"
          << "Единица измерения..." << tbl[idx].measure       <<
"\n"
              << "Номер цеха........." << tbl[idx].workshopNum
<< "\n"
           << "Норма расхода......." << tbl[idx].consumptionNorm
<< "\n";
    };
    cout << "-------------------------------\n";
    if (wo){
        cout << "[x] Назад\n";
        string text;
        cin >> text;
    }
    return;
}
```

```
void printTable(){
    for (int i = 0; i < k; i++){
        cout << "------------------------------\n"
             << "Запись.............." << tbl[i].no             <<
"\n"
            << "Код детали.........." << tbl[i].detailCode     <<
"\n"
            << "Код материала......." << tbl[i].materialCode   <<
"\n"
            << "Единица измерения..." << tbl[i].measure        <<
"\n"
            << "Номер цеха.........." << tbl[i].workshopNum    <<
"\n"
              << "Норма расхода......." << tbl[i].consumptionNorm
<< "\n";
    };
    cout << "------------------------------\n";
    cout << "[x] Назад\n";
    string text;
    cin >> text;
    return;
}



int main(){
   getdata();
   while (!quit) mainMenu();
   return 0;
}
```