Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет информационных технологий и управления

Кафедра вычислительных методов и программирования

Дисциплина: Основы алгоритмизации и программирования

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

«Структура списка изделий кондитерской фабрики»

БГУИР КП 6-05-0611-03 067 ПЗ

Студент: гр. 321702 Рублевская Е.А.

Руководитель: Семижон Е.А.

Минск 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Информация, как ключевой стратегический ресурс, играет огромную роль в современном мире. В наше время данные стали одним из самых ценных активов, и умение правильно и эффективно работать с ними стало важным навыком для успешного развития бизнеса, науки, образования и других сфер жизни. Современные технологии позволяют не только создавать и хранить большие объемы информации, но и обрабатывать её быстро и эффективно. Анализ данных, машинное обучение, искусственный интеллект – все это инструменты, которые помогают извлечь ценные знания из информации и принимать обоснованные решения.

Высокий уровень информатизации обеспечивает не только конкурентоспособность предприятий, но и повышает эффективность управления в различных сферах общественной жизни. Он позволяет автоматизировать процессы, улучшить коммуникацию, повысить качество услуг и товаров, а также оптимизировать ресурсы.

Таким образом, информатизация становится неотъемлемой частью успешного развития предприятий и организаций в целом. Рациональное использование информации и технологий помогает улучшить производительность труда, сократить издержки, увеличить доходы и обеспечить устойчивое развитие в условиях быстро меняющегося мира.

Так же не менее важную роль играет выбор IDE для разработки любой программы. Для своей работы я выбрала Clione – интегрирования среда разработки для языков программирования, разрабатываемая компанией JetBrains. CLion охватывает все основные шаги процесса разработки ПО для встраиваемых систем. Прямо в IDE можно создавать проекты, писать код на C, C++, Python и Assembly для самых разных тулчейнов и оборудования, а также выполнять отладку прошивок и внутрипроцессорных программ. Создавайте общие сессии, чтобы вместе с коллегами писать код, а также выполнять его ревью и отладку в реальном времени. CLion понимает даже самые сложные фрагменты кода: показывает выведенные типы, сигнатуры функций и документацию для сниппета, над которым вы работаете. CLion позволяет быстро переключаться между определениями и объявлениями, перемещаться по иерархии кода и находить использования символов в проектах любого размера.

1. СТРУКТУРЫ И ФАЙЛЫ

В информатике структура данных — программная единица, позволяющая хранить и обрабатывать данные, а также обеспечивающая их эффективное использование. Данные при этом должны быть однотипными или логически связанными.

При разработке программного обеспечения сложность реализации и качество работы программ существенно зависят не только от выбора алгоритма, но и от правильного выбора структур данных. Одни и те же данные можно сохранить в структурах, требующих различного объема памяти, а алгоритмы работы с разными структурами данных могут иметь различную эффективность. Структура данных, наиболее подходящая для решения конкретной задачи, позволяет выполнять большое количество различных операций, используя как можно меньший объем ресурсов.

Ни одна профессиональная программа сегодня не пишется без использования структур данных, поэтому многие из них содержатся в стандартных библиотеках современных языков программирования (например, STL для C++).

Структура данных представляет собой набор значений данных, отношения между ними, а также функции и (или) операции, которые могут быть применены к данным.

1.1.Классификация структур данных:

Линейные:

- массив;
- список;
- связанный список;
- стек;
- очередь;
- хэш-таблица.

Иерархические:

- двоичные деревья;
- n-арные деревья;

- иерархический список.

Сетевые:

- неориентированный граф;
- ориентированный граф.

Табличные:

- таблица реляционной базы данных;
- двумерный массив.

1.2.Основные функции структур данных:

1. **Хранение данных:**
    - Одной из основных функций структур данных является хранение данных. Структуры данных позволяют организовать данные таким образом, чтобы они занимали минимальное количество памяти и были легко доступны для обработки.
2. **Упорядочение данных:**
    - Некоторые структуры данных предназначены для упорядочения данных в определенном порядке, что упрощает поиск, сортировку и обработку данных.
3. **Быстрый доступ к данным:**
    - Структуры данных должны обеспечивать быстрый доступ к данным, чтобы операции чтения и записи были эффективными.
4. **Эффективность использования ресурсов:**
    - Хорошая структура данных должна быть эффективной в использовании ресурсов, таких как память и процессорное время.
5. **Поддержка операций:**
    - Структуры данных должны поддерживать различные операции, такие как добавление, удаление, поиск, сортировка и обновление данных.

2. АЛГОРИТМЫ СОРТИРОВКИ

Сортировка – это упорядочивание набора однотипных данных по возрастанию или убыванию.

В общем случае сортировку следует понимать как процесс перегруппировки заданного множества объектов в определенном порядке. Часто при сортировке больших объемов данных нецелесообразно переставлять сами элементы, поэтому для решения задачи выполняется упорядочивание элементов по индексам. То есть индексы элементов выстраивают в такой последовательности, что соответствующие им значения элементов оказываются отсортированными по условию задачи.

Сортировка применяется для облегчения поиска элементов в упорядоченном множестве. Задача сортировки одна из фундаментных в программировании.

Чаще всего при сортировке данных лишь часть их используется в качестве *ключа сортировки*. *Ключ сортировки* – это часть данных, определяющая порядок элементов. Таким образом, ключ участвует в сравнениях, но при обмене элементов происходит перемещение всей структуры данных. Например, в списке почтовой рассылки в качестве ключа может использоваться почтовый индекс, но сортируется весь адрес. При решении задач сортировок массивов ключ и данные совпадают.

Существует множество различных алгоритмов сортировки, каждый из которых имеет свои плюсы и минусы. Для оценки алгоритмов сортировки используются следующие критерии:

- Эффективность алгоритма, определяемая количеством сравнений и обменов элементов в процессе сортировки. Обмены требуют больше времени, чем сравнения. Некоторые алгоритмы имеют экспоненциальную зависимость времени работы, в то время как у других время работы логарифмически зависит от количества элементов.
- Время работы в лучшем и худшем случаях. Этот критерий важен для оценки производительности алгоритма в случае, если одно из крайних условий встречается часто. Некоторые алгоритмы могут иметь хорошее среднее время работы, но работать очень медленно в худшем случае.

- Поведение алгоритма сортировки. Естественное поведение алгоритма означает, что время сортировки минимально для уже упорядоченного списка элементов, увеличивается по мере неупорядоченности списка и достигает максимума, когда элементы расположены в обратном порядке. Оценка объема работы алгоритма основана на количестве сравнений и обменов.

Существуют различные методы сортировки массивов, которые различаются по скорости выполнения. Простые методы сортировки требуют квадратичное количество сравнений, в то время как быстрые методы сортировки требуют линейно-логарифмическое количество сравнений. Простые методы удобны для понимания основных принципов сортировки из-за своей простоты и краткости. Более сложные методы требуют меньшего количества операций, но сами операции более сложные, поэтому для небольших массивов простые методы могут быть более эффективными.

Простые методы сортировки можно разделить на три основные категории:

- Сортировка пузырьком (простой обмен элементов).
- Сортировка выбором (перебор элементов).
- Сортировка вставками (вставка и сдвиг элементов).

2.1.Сортировка методом "пузырька" (простого обмена)

Самый известный алгоритм – *пузырьковая сортировка* (*bubble sort*, сортировка методом пузырька или просто сортировка пузырьком). Его популярность объясняется интересным названием и простотой самого алгоритма.

Алгоритм попарного сравнения элементов массива в литературе часто называют "методом пузырька", проводя аналогию с пузырьком, поднимающимся со дна бокала с газированной водой. По мере всплывания пузырек сталкивается с другими пузырьками и, сливаясь с ними, увеличивается в объеме. Чтобы аналогия стала очевидной, нужно считать, что элементы массива расположены вертикально друг над другом, и их нужно так упорядочить, чтобы они увеличивались сверху вниз.

Алгоритм состоит в повторяющихся проходах по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов.

Проходы по массиву повторяются до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает – массив отсортирован. При проходе алгоритма элемент, стоящий не на своём месте, "всплывает" до нужной позиции. (см. Рисунок 2.1)

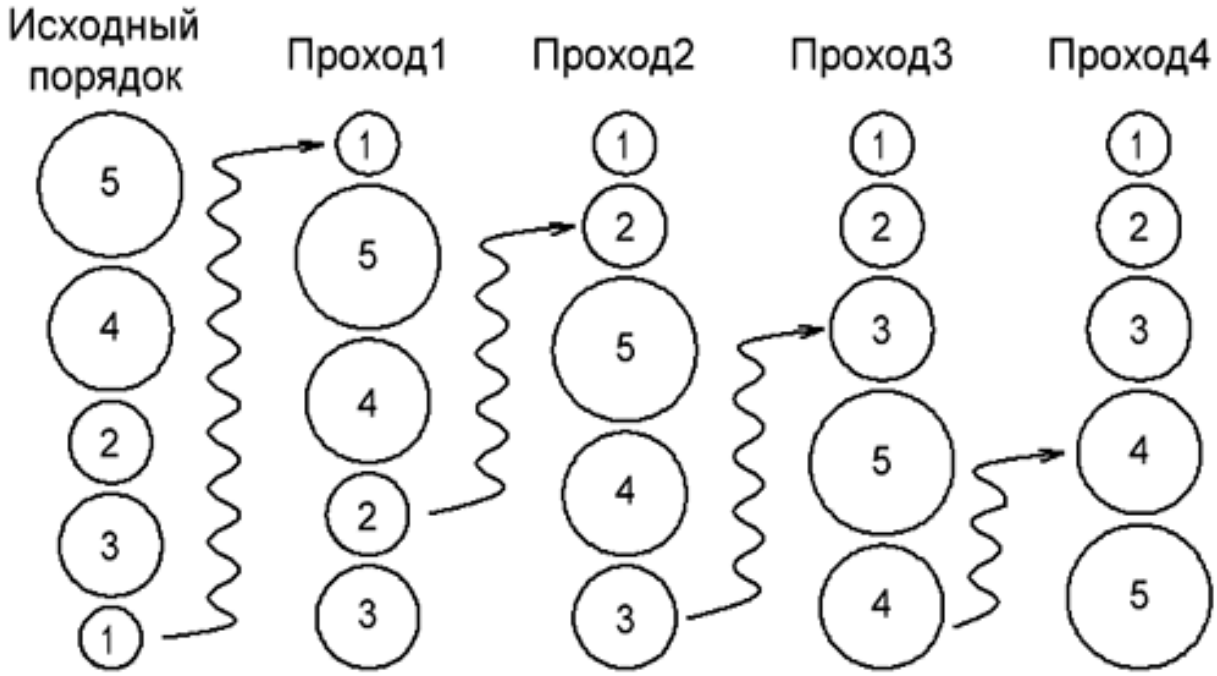

Рисунок 2.1 - метод пузырька

Описание функции сортировки методом пузырька

```
void BubbleSort (int k,int x[max])
{
int i,j,buf;
for (i=k-1;i>0;i--)
for (j=0;j<i;j++)
if (x[j]>x[j+1])
{
buf=x[j];
x[j]=x[j+1];
x[j+1]=buf;
}
}
```

2.2.Сортировка методом "шейкер"

Шейкер-сортировка является усовершенствованным методом пузырьковой сортировки.

Анализируя метод сортировки «пузырьком», можно отметить два обстоятельства:

- если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, ее можно исключить из рассмотрения.
- при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к модификациям в методе пузырьковой сортировки. (см. Рисунок 2.2)

- От последней перестановки до конца (начала) массива находятся отсортированные элементы. Учитывая данный факт, просмотр осуществляется не до конца (начала) массива, а до конкретной позиции. Границы сортируемой части массива сдвигаются на 1 позицию на каждой итерации.
- Массив просматривается поочередно справа налево и слева направо.
- Просмотр массива осуществляется до тех пор, пока все элементы не встанут в порядке возрастания (убывания).
- Количество просмотров элементов массива определяется моментом упорядочивания его элементов.

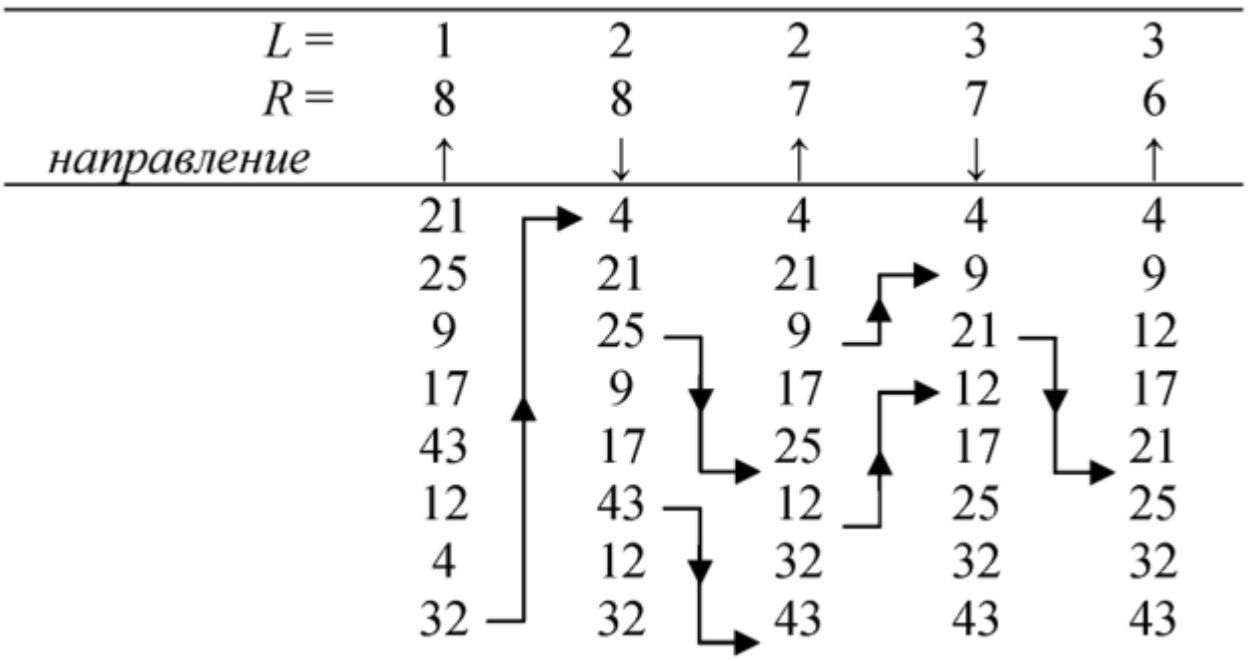

Рисунок 2.2 - шейкер-сортировка

Описание функции шейкер-сортировки

```
do
{
exchange = false;
for(i=k-1; i > 0; --i)
{
if(x[i-1] > x[i])
{
t = x[i-1];
x[i-1] = x[i];
x[i] = t;
exchange = true;
}
}
}
for(i=1; i < k; ++i)
{
if(x[i-1] > x[i])
{
t = x[i-1];
x[i-1] = x[i];
x[i] = t;
exchange = true;
}
}
```

2.3.Сортировка методом простого выбора (простой перебор)

Это наиболее естественный алгоритм упорядочивания. При данной сортировке из массива выбирается элемент с наименьшим значением и обменивается с первым элементом. Затем из оставшихся n - 1 элементов снова выбирается элемент с наименьшим ключом и обменивается со вторым элементом, и т.д.

Шаги алгоритма (см. Рисунок 2.3):

1. находим минимальное значение в текущей части массива;

2.производим обмен этого значения со значением на первой неотсортированной позиции;

3.далее сортируем хвост массива, исключив из рассмотрения уже отсортированные элементы.

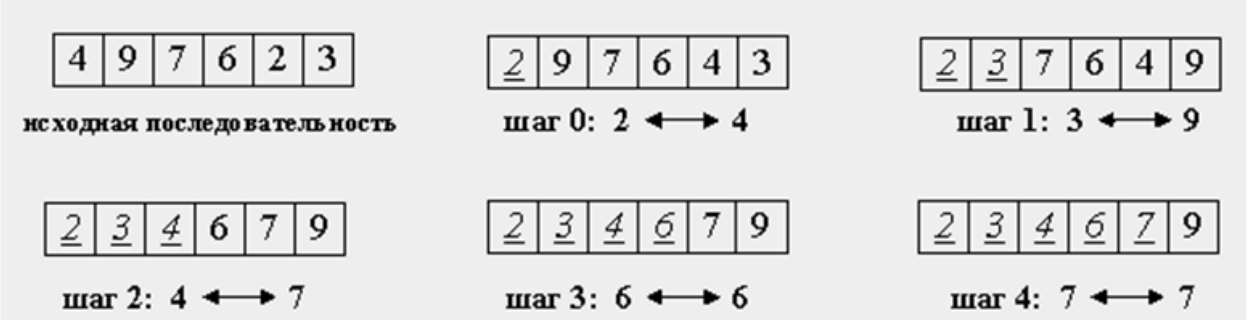

Рисунок 2.3 - метод простого выбора

Описание функции сортировки методом простого выбора

```
void SelectionSort (int k,int x[max])
{
int i,j,min,temp;
for (i=0;i<k-1;i++)
{
min=i; //устанавливаем начальное значение мин. индекса
for (j=i+1;j<k;j++) //ищем мин. индекс элемента
{
if (x[j]<x[min])
min=j; //меняем значения местами
}
temp=x[i];
x[i]=x[min];
x[min]=temp;
}
}
```

2.4. Сортировка методом простого включения

Хотя этот метод сортировки намного менее эффективен, чем сложные алгоритмы (такие как быстрая сортировка), у него есть ряд преимуществ:

- прост в реализации;
- эффективен на небольших наборах данных, на наборах данных до десятков элементов может оказаться лучшим;
- эффективен на наборах данных, которые уже частично отсортированы;
- это *устойчивый алгоритм* сортировки (не меняет порядок элементов, которые уже отсортированы);
- может сортировать массив по мере его получения;
- не требует временной памяти, даже под стек.

На каждом шаге алгоритма выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора. (см. Рисунок 2.4)

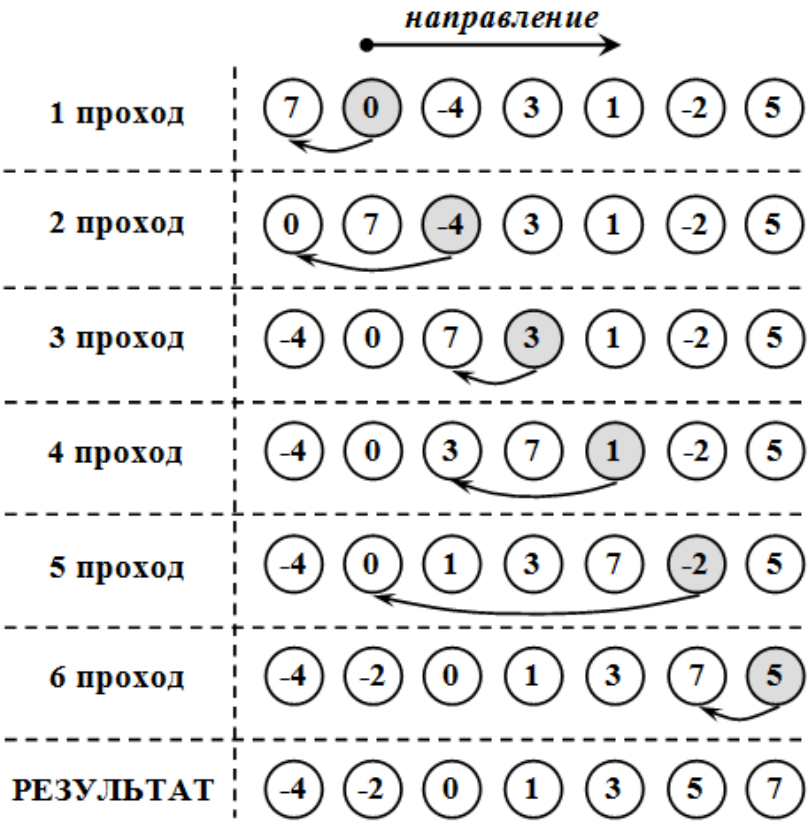

Рисунок 2.4 - метод простого включения

Описание функции сортировки методом простого включения:

```
void InsertSort (int k,int x[max])
{
int i,j, temp;
for (i=0;i<k;i++)
{
//цикл проходов, i - номер прохода
temp=x[i]; //поиск места элемента
for (j=i-1; j>=0 && x[j]>temp; j--)
x[j+1]=x[j];
//сдвигаем элемент вправо, пока не дошли
//место найдено, вставить элемент
x[j+1]=temp;
}
```

2.5.Сортировка методом Quicksort

Быстрая сортировка (англ. quicksort) – это метод сортировки значений в списке в последовательные списки с помощью повторяющейся процедуры. В методе быстрой сортировки выбирается значение из основного списка, которое называется опорным значением. Остальные значения разделяются на два списка:

Первый список содержит значения, которые меньше либо равны опорному значению. Эти значения располагаются слева от опорного значения.

Второй список содержит значения, которые больше опорного значения. Эти значения располагаются справа от опорного значения.

Метод быстрой сортировки повторяется для всех результирующих списков, пока не останется только одно значение или пустой список значений.После этого вы выбираете последнее одиночное значение, и если значение располагается слева от опорного значения, оно остается таким, пока вы не дойдете до первого опорного значения вверху. (см. Рисунок 2.5)

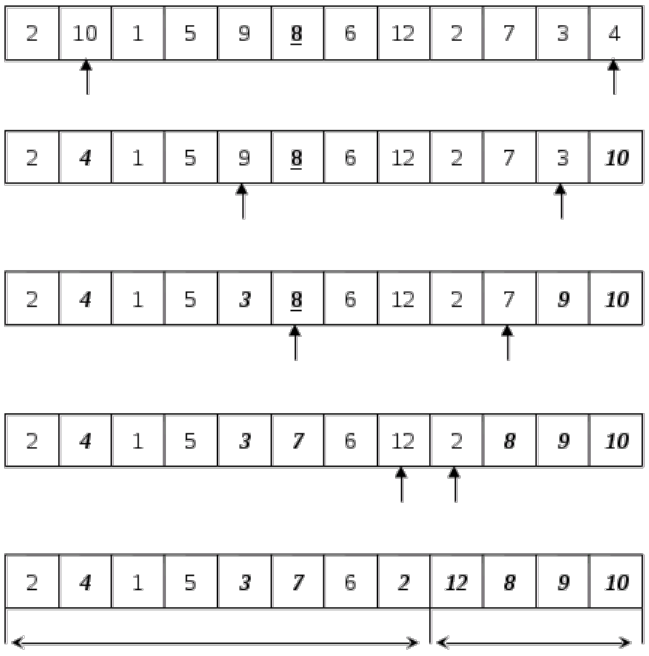

Рисунок 2.5 - быстрая сортировка

Описание функции сортировки Quicksort:

```
void qsort(int l, int r)
{
int w,x,i,j;
i=l;
j=r;
x=a[(l+r)/2];
while (i<=j)
{
while (      a[i]<x) i++;
while (x<a[j]) j--;
if (i<=j)
{
w=a[i]; a[i]=a[j]; a[j]=w;
i++; j--;
}
}
if (l<j) qsort(l,j);
if (i<r) qsort(i,r);
}
```

3. АЛГОРИТМЫ ПОИСКА

Одними из важнейших процедур обработки структурированной информации является поиск. Задача поиска привлекала большое внимание ученых (программистов) еще на заре компьютерной эры. С 50-х годов началось решение проблемы поиска элементов, обладающих определенным свойством в заданном множестве.

У каждого алгоритма есть свои преимущества и недостатки. Поэтому важно выбрать тот алгоритм, который лучше всего подходит для решения конкретной задачи.

3.1.Последовательный поиск

Последовательный (линейный) поиск – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

Идея этого метода заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы множества (например, слева направо). Если в ходе просмотра множества будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, алгоритм должен выдать отрицательный результат.

Алгоритм последовательного поиска (см. Рисунок 3.1):

Шаг 1. Полагаем, что значение переменной цикла i=0.

Шаг 2. Если значение элемента массива x[i] равно значению ключа key, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу. В противном случае значение переменной цикла увеличивается на единицу i=i+1.

Шаг 3. Если i<k, где k – число элементов массива x, то выполняется Шаг 2, в противном случае – работа алгоритма завершена и возвращается значение равное -1.

При наличии в массиве нескольких элементов со значением key данный алгоритм находит только первый из них (с наименьшим индексом).

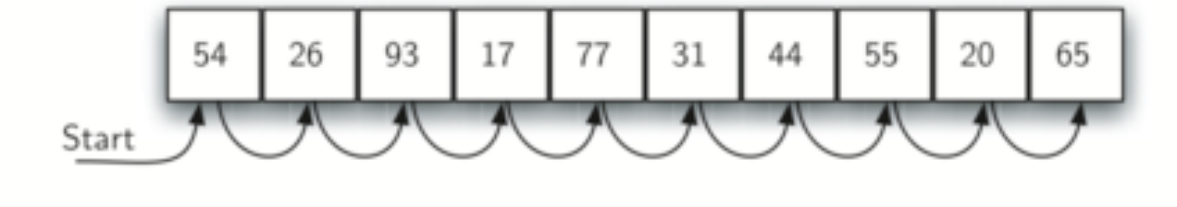

Рисунок 3.1 - последовательный поиск

Описание функции линейного (последовательного поиска):

```
int p_lin1(int a[],int  n, int x)
{
for(int i=0; i < n; i++)
if (a[i]==x) return i;
return -1;
}
int p_lin2(int a[],int  n, int x)
{
a[n]=x;
int i=0;
while (a[i]!=x) i++;
if (i==n) return -1;
else return i;
}
```

3.2.Поиск делением пополам

Двоичный(бинарный) поиск — алгоритм поиска элемента в отсортированном массиве.

Двоичный поиск можно использовать только в том случае, если есть массив, все элементы которого упорядочены (отсортированы). Бинарный поиск не используется для поиска максимального или минимального элементов, так как в отсортированном массиве эти элементы содержатся в начале и в конце массива соответственно, в зависимости от тога как отсортирован массив, по возрастанию или по убыванию. Поэтому алгоритм бинарного поиска применим, если необходимо найти некоторый ключевой элемент в массиве. То есть организовать поиск по ключу, где ключ — это определённое значение в массиве. (см. Рисунок 3.2)

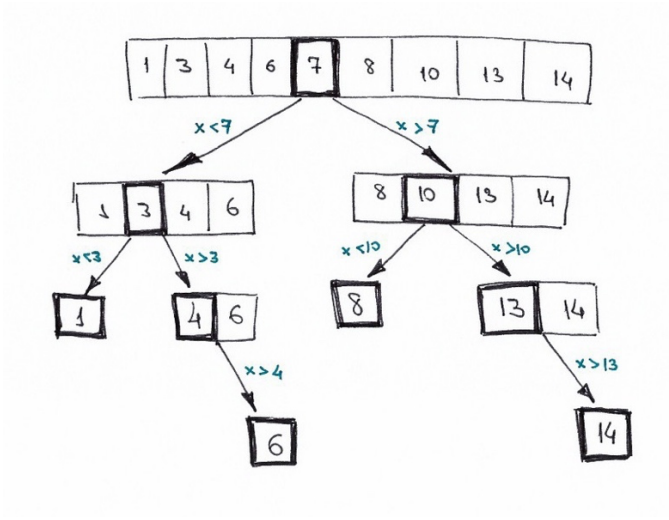

Рисунок 3.2 - бинарный поиск

Описание функции бинарного поиска

```
int p_dv(int a[], int  n, int x)
{
int i=0, j=n-1, m;
while(i<j)
{
m=(i+j)/2;
if (x > a[m])
i=m+1;
else j=m;
}
if (a[i]==x) return i;
else return -1;
}
```

3.3.Интерполяционный поиск

Интерполирующий поиск, напоминает двоичный поиск, за исключением того, что вместо деления области поиска на две части, интерполирующий поиск производит оценку новой области поиска по расстоянию между ключом поиска и текущим значением элемента. Иными словами, двоичный поиск учитывает лишь знак разности между ключом поиска и текущим значением, а интерполирующий поиск еще учитывает и модуль этой разности и по данному значению производит предсказание позиции следующего элемента для проверки. (см. Рисунок 3.3)

По скорости поиска интерполирующий поиск превосходит двоичный. Также существенным отличием от двоичного является то, что с помощью алгоритма интерполирующего поиска можно искать не только числовые значения, но и, к примеру, текстовую информацию.

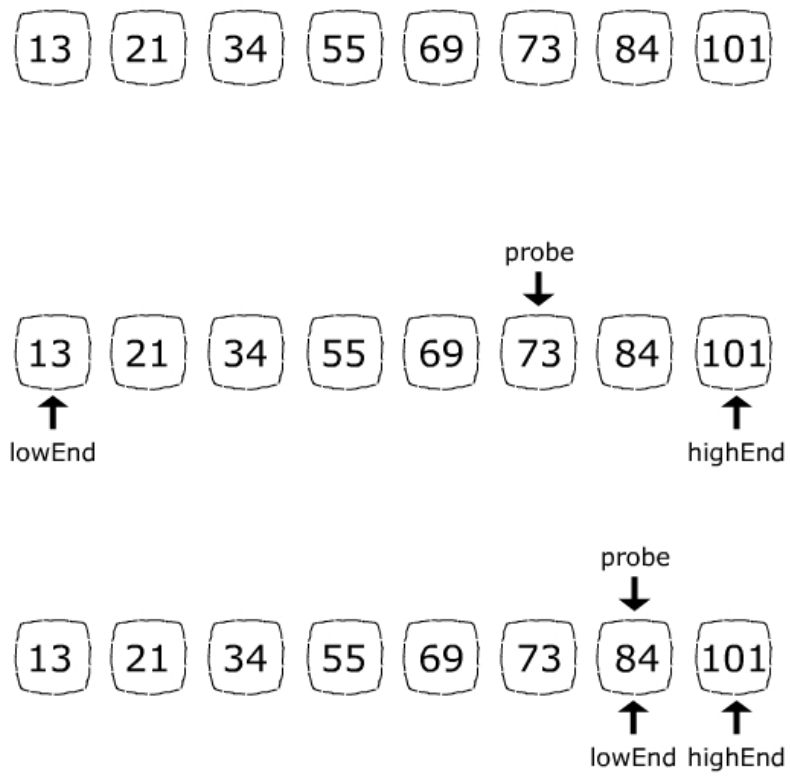

Рисунок 3.3 - интерполяционный поиск

Описание функции интерполяционного поска

```
int p_dv(int a[], int  n, int x)
{
int i=0, j=n-1, m;
while(i<j)
{
if (a[i]==a[j])
if (a[i]==x) return i;
else
return -1;
m=i+(j-i)*(x-a[i])/(a[j]-a[i]);
if (a[m]==x) return m;
else
if (x > a[m]) i=m+1;
else
j=m-1;
}
return -1;
}
```

4. ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ

В моём курсовом проекте были разработаны пользовательские функции работы с файлами, записи и редактирования данных, а также функции с применением вышеупомянутых алгоритмов сортировки и поиска. Однако программа начинает работать с вывода меню, где пользователь может выбрать то, что его интересует.

4.1. Функция для сортировки выбором selectionSort().

Эта функция принимает вектор **products**, содержащий объекты типа **Product**, и значение **price**, которое представляет собой цену продукта, который нужно найти. Функция выполняет линейный поиск вектора **products** и выводит на экран название продукта, если продукт с заданной ценой найден. Функция начинает поиск, перебирая элементы вектора **products** с помощью цикла **for**. Для каждого продукта в векторе проверяется, равна ли цена продукта **price** цене, переданной в качестве аргумента. Если найден продукт с нужной ценой, выводится сообщение о том, что продукт найден, и указывается его название. Переменная **found** устанавливается в значение **true**, чтобы указать, что продукт был найден. Если после завершения цикла переменная **found** остается равной **false**, это означает, что продукт с указанной ценой не был найден во всем векторе. В этом случае выводится сообщение о том, что продукт не найден.

Эта функция реализует простой линейный поиск вектора **products** на основе цены и выводит результат поиска на экран.

```
void selectionSort(vector<Product>& products)
{
    int n = products.size();
    for (int i = 0; i < n - 1; i++)
    {
        int minIndex = i;
        for (int  = i + 1; j < n; j++)
        {
            if (products[j].price < products[minIndex].price)
            {
                minIndex = j;
            }
        }
        swap(products[i], products[minIndex]);
    }
}
```

4.2.Функция создания необходимых файла createFile().

Эта функция принимает строку **filename** в качестве аргумента и создает файл с указанным именем. Она использует объект **ofstream**, который предоставляет функциональность для записи данных в файл. Сначала функция пытается открыть файл с указанным именем для записи. Если файл не удается открыть, она выводит сообщение об ошибке в поток стандартного вывода ошибок **cerr** и завершает выполнение функции, не создавая файл. Если файл успешно открыт для записи, выводится сообщение об успешном создании файла в поток стандартного вывода cout. Затем файл закрывается вызовом метода **close** объекта **file**.

```
void createFile(const string& filename)
{
    ofstream file(filename);
    if (!file.is_open())
    {
        cerr << "Ошибка создания файла!" << endl;
        return;
    }
    cout << "Файл создан успешно." << endl;
    file.close();
}
```

4.3.Функция просмотра содержимого файла viewFile ().

Эта функция принимает строку filename в качестве аргумента и открывает файл с указанным именем для чтения. Она использует объект ifstream, который предоставляет функциональность для чтения данных из файла. Если файл не удается открыть для чтения, функция выводит сообщение об ошибке в поток стандартного вывода ошибок cerr и завершает выполнение функции. Если файл успешно открыт для чтения, функция начинает считывать строки из файла поочередно с помощью функции **getline**. Каждая считанная строка предполагается содержащей информацию о продукте в формате *"название тип количество цена статус"*. Для разбора каждой строки используется объект **stringstream**, который разбивает строку на отдельные части с помощью оператора >>. Полученные данные записываются в объект типа Product. Далее, после успешного разбора строки, информация о продукте выводится на экран с помощью потока стандартного вывода **cout**. После завершения чтения содержимого файла, файл закрывается вызовом метода **close** объекта **file**.

Эта функция позволяет просмотреть содержимое файла, предполагая, что каждая строка в файле представляет отдельный продукт с указанными характеристиками.

```
void viewFile(const string& filename)
{
    ifstream file(filename);
    if (!file.is_open())
    {
        cerr << "Ошибка открытия файла!" << endl;
        return;
    }
    string line;
    cout << "Содержимое файла:" << endl;
    while (getline(file, line))
    {
        stringstream ss(line);
        Product product;
        ss >> product.name >> product.type >> product.quantity
>> product.price;
        getline(ss, product.status);
        cout << product.name << " | " << product.type << " | "
<< product.quantity << " | " << product.price << " | " <<
product.status << endl;
    }
    file.close();
}
```

4.4.Функция добавления данных в файл addData ().

Эта функция принимает строку **filename** в качестве аргумента и открывает файл с указанным именем для добавления данных в конец файла. Она использует объект **ofstream**, который предоставляет функциональность для записи данных в файл, с флагом **ios::app**, чтобы добавлять данные в конец файла. Если файл не удается открыть для записи, функция выводит сообщение об ошибке в поток стандартного вывода ошибок **cerr** и завершает выполнение функции. Затем функция запрашивает у пользователя данные о продукте: название, тип, количество, цена и статус. Для ввода названия продукта используется функция **getline**, чтобы обеспечить возможность ввода названия, содержащего пробелы. После получения всех данных о продукте, они записываются в файл с помощью объекта **file**, с использованием форматирования. Каждый параметр разделяется пробелом, а название продукта заключается в кавычки. Затем файл закрывается вызовом метода **close** объекта **file**, и выводится сообщение об успешном добавлении данных в файл.

Эта функция позволяет пользователю добавлять данные о продуктах в конец файла с определенным форматом.

```
void addData(const string& filename)
{
    ofstream file(filename, ios::app);
    if (!file.is_open())
    {
        cerr << "Ошибка открытия файла!" << endl;
        return;
    }
    Product product;
    cout << "Введите данные продукта:" << endl;
    cout << "Название: ";
    cin.ignore();
    getline(cin, product.name);
    cout << "Тип: ";       getline(cin, product.type);
    cout << "Количество: "; cin >> product.quantity;
    cout << "Цена: ";      cin >> product.price;
    cin.ignore();
    cout << "Статус: ";  getline(cin, product.status);
    file << "\"" << product.name << "\" "
         << product.type << " "
         << product.quantity << " "
         << product.price << " "
         << product.status << endl;

    file.close();
    cout << "Данные добавлены в файл." << endl;
}
```

4.5.Функция загрузки для загрузки данных loadProductsFromFile().

Эта функция принимает строку **filename** в качестве аргумента и загружает данные о продуктах из файла с указанным именем. Она возвращает вектор объектов типа **Product**, содержащий информацию о продуктах, считанную из файла. Сначала функция создает пустой вектор **products**, который будет содержать информацию о продуктах, считанную из файла. Затем функция открывает файл с указанным именем для чтения с помощью объекта **ifstream**. Если файл не удается открыть, функция выводит сообщение об ошибке в поток стандартного вывода ошибок **cerr** и возвращает пустой вектор **products**. После успешного открытия файла, функция начинает считывать строки из файла поочередно с помощью функции **getline**. Для каждой считанной строки создается новый объект **stringstream**, чтобы разбить строку на отдельные части. Затем данные о продукте считываются из **stringstream** с помощью оператора >> и сохраняются в объект типа Product. После этого оставшаяся часть строки,

содержащая статус продукта, считывается с помощью функции **getline**. Полученный объект Product добавляется в конец вектора **products** с помощью метода **push_back**. После завершения чтения содержимого файла, файл закрывается вызовом метода **close** объекта **file**, и вектор **products** с данными о продуктах возвращается из функции.

Эта функция позволяет загружать информацию о продуктах из файла и возвращать её в виде вектора объектов типа **Product**.

```
vector<Product> loadProductsFromFile(const string& filename)
{
    vector<Product> products;
    ifstream file(filename);
    if (!file.is_open())
    {
        cerr << "Ошибка открытия файла!" << endl;
        return products;
    }
    string line;
    while (getline(file, line))
    {
        stringstream ss;
        ss << line;
        Product product;
        ss >> product.name >> product.type >> product.quantity
>> product.price;
        ss.ignore();
        getline(ss, product.status);
        products.push_back(product);
    }
    file.close();
    return products;
}
```

4.6.Функция сохранения данных о продуктах saveProductsToFile ().

Эта функция принимает строку **filename** в качестве имени файла и вектор **products**, содержащий объекты типа Product, и сохраняет данные о продуктах в файл с указанным именем.

Сначала функция открывает файл с указанным именем для записи с помощью объекта **ofstream**. Если файл не удается открыть, функция выводит сообщение об ошибке в поток стандартного вывода ошибок **cerr** и завершает выполнение.

Затем функция проходит по всем элементам вектора **products** с помощью цикла **for**. Для каждого продукта в векторе данные о нем записываются в файл в виде строки, разделенной пробелами. Каждый параметр продукта (название, тип, количество, цена, статус) выводится в файл с помощью оператора **<<**. После записи данных о продукте в файл переходится на новую строку с помощью **endl**. После записи всех данных о продуктах в файл, файл закрывается вызовом метода **close** объекта **file**.

Эта функция позволяет сохранить данные о продуктах из вектора **products** в файл с определенным форматом для последующего использования или хранения.

```
void saveProductsToFile(const string& filename, const
vector<Product>& products)
{
    ofstream file(filename);
    if (!file.is_open())
   {
        cerr << "Ошибка открытия файла!" << endl;
        return;
    }
    for (const Product& product : products)
    {
        file << product.name << " "
             << product.type << " "
             << product.quantity << " "
             << product.price << " "
             << product.status
             << endl;
    }
    file.close();
}
```

4.7.Функция линейного поиска linearSearch ().

Эта функция принимает вектор **products**, содержащий объекты типа **Product**, и значение **price**, которое представляет собой цену продукта, который нужно найти. Функция выполняет линейный поиск вектора **products** и выводит на экран название продукта, если продукт с заданной ценой найден. Функция начинает поиск, перебирая элементы вектора **products** с помощью цикла **for**. Для каждого продукта в векторе проверяется, равна ли цена продукта **price** цене, переданной в качестве аргумента. Если найден продукт с соответствующей ценой, выводится сообщение о том, что продукт найден, и указывается его название. Переменная **found** устанавливается в значение **true**, чтобы указать, что

продукт был найден. Если после завершения цикла переменная **found** остается равной **false**, это означает, что продукт с указанной ценой не был найден во всем векторе. В этом случае выводится сообщение о том, что продукт не найден.

Эта функция реализует простой линейный поиск вектора **products** на основе цены и выводит результат поиска на экран.

```
void linearSearch(const vector<Product>& products, double
price)
{
    bool found = false;
    for (int i = 0; i < products.size(); i++)
   {
        if (products[i].price == price)
       {
            cout << "Продукт найден: " << products[i].name <<
endl;
            found = true;
        }
    }
    if (!found)
     {
        cout << "Продукт не найден." << endl;
     }
}
```

4.8. Функция для разбиения QuickSort partition() (Дополнительная функция).

Эта функция выполняет часть алгоритма быстрой сортировки (**quicksort**). Она принимает на вход вектор **products**, индексы **low** и **high**, указывающие на границы сегмента массива, который нужно разделить.

1. Выбирается опорный элемент. В данном случае опорным элементом является цена продукта с индексом **high**.

2. Инициализируется переменная i равная **low - 1**. Эта переменная будет использоваться для индексирования элементов, меньших или равных опорному.

3. Затем цикл **for** проходит через сегмент массива от **low** до **high - 1**. Для каждого элемента сравнивается его цена с ценой опорного элемента. Если цена текущего элемента меньше или равна цене опорного элемента, то переменная **i** увеличивается на **1**, а затем происходит обмен местами элементов **products[i]** и **products[j]**.

4. После завершения цикла все элементы меньшие или равные опорному будут находиться слева от **products[i]**, а элементы большие опорного — справа. Затем опорный элемент перемещается на позицию **i + 1** (следующая после последнего элемента, меньшего или равного опорному).

5. Функция возвращает индекс, на котором находится опорный элемент после разделения.

Эта функция выполняет один шаг алгоритма разделения массива на две части относительно опорного элемента и возвращает индекс, по которому массив разделен.

```
int partition(vector<Product>& products, int low, int high)
{
    double pivot = products[high].price;
    int i = (low - 1);
    for (int j = low; j < high; j++)
 {
        if (products[j].price <= pivot)
        {
            i++;
            swap(products[i], products[j]);
        }
    }
    swap(products[i + 1], products[high]);
    return (i + 1);
}
```

4.9.Функция сортировки quicksort().

Этот рекурсивная функция быстрой сортировки (**quickSort**), которая сортирует вектор **products** в порядке возрастания цены продуктов.

1. Условие **if (low < high)** проверяет, что сегмент массива, который нужно сортировать, содержит более одного элемента. Если это условие выполняется, то есть если **low** меньше **high**, выполняется сортировка.

2. Функция **partition** вызывается для разделения сегмента массива на две части относительно опорного элемента. После выполнения этой функции опорный элемент будет находиться на своем правильном месте в массиве, а элементы меньше опорного будут находиться слева от него, а больше — справа.

3. Рекурсивные вызовы **quickSort** выполняются для левой и правой половин массива. Первый рекурсивный вызов **quickSort(products, low, pi - 1)** сортирует левую половину массива от **low** до позиции **pi - 1**, т.е. до элемента, предшествующего опорному элементу. Второй вызов **quickSort(products, pi + 1, high)** сортирует правую половину массива от позиции **pi + 1** до **high**, т.е. от элемента, следующего за опорным элементом, до конца.

4. Рекурсивные вызовы продолжаются, пока размер сегмента массива, который нужно сортировать, не станет равным нулю или одному, что означает, что весь массив уже отсортирован.

```
void quickSort(vector<Product>& products, int low, int high)
{
    if (low < high)
    {
        int pi = partition(products, low, high);
        quickSort(products, low, pi - 1);
        quickSort(products, pi + 1, high);
    }
}
```

4.10. Функция двоичного поиска binarySearch().

Эта функция выполняет бинарный поиск заданной цены **price** в упорядоченном по возрастанию векторе продуктов **products**. Она возвращает вектор индексов элементов, у которых цена соответствует заданной.

1. Создается пустой вектор **indices**, который будет содержать индексы элементов с заданной ценой.

2. Запускается цикл **while**, который продолжает выполнение, пока значение **high** не станет меньше значения **low**. Это означает, что в рассматриваемом сегменте массива еще есть элементы для поиска.

3. Внутри цикла вычисляется индекс **mid** элемента, находящегося посередине рассматриваемого сегмента массива.

4. Если цена элемента с индексом **mid** равна заданной цене **price**, выполняется следующее:

- Ищутся все элементы с заданной ценой слева от **mid**. Последовательно уменьшая индекс **left**, добавляем найденные индексы в вектор **indices**.

- Затем ищутся все элементы с заданной ценой справа от **mid**. Последовательно увеличивая индекс **right**, добавляем найденные индексы в **indices**.
- Возвращается вектор **indices** с индексами элементов, у которых цена соответствует заданной.

5. Если цена элемента с индексом **mid** больше заданной цены **price**, переменная **high** обновляется на **mid - 1**, чтобы продолжить поиск в левой половине массива. В противном случае, переменная **low** обновляется на **mid + 1**, чтобы продолжить поиск в правой половине массива.

6. Если элемент с заданной ценой не найден в рассматриваемом сегменте массива, функция возвращает пустой вектор **indices**.

Эта функция предоставляет возможность выполнять бинарный поиск элементов с заданной ценой в упорядоченном по возрастанию векторе продуктов и возвращает вектор индексов найденных элементов.

```
vector<int> binarySearch(const vector<Product>& products,
double price, int low, int high)
{
    vector<int> indices;
    while (high >= low)
    {
        int mid = low + (high - low) / 2;
        if (products[mid].price == price)
        {
            int left = mid;
            while (left >= low && products[left].price ==
price)
           {
                indices.push_back(left);
                left--;
            }
            int right = mid + 1;
            while (right <= high && products[right].price ==
price)
            {
                indices.push_back(right);
                right++;
            }
            return indices;
        }
        if (products[mid].price > price)
        {
            high = mid - 1;
        } else
```

```
        {
            low = mid + 1;
        }
    }
    return indices;
}
```

4.11. Функция для вывода информации о продуктах по типу и статусу printProductsByTypeAndStatus().

Эта функция принимает вектор **products**, содержащий объекты типа **Product**, а также строки **type** и **status**, и выводит на экран информацию о продуктах, у которых тип и статус соответствуют переданным значениям.

1. Функция начинает проходить по всем элементам вектора **products** с помощью цикла **for**.

2. Для каждого продукта в векторе проверяется условие: если тип продукта (**product.type**) равен переданному значению **type** И статус продукта (**product.status**) равен переданному значению **status**, то выполняется следующее:

3. Информация о продукте выводится на экран с помощью потока стандартного вывода **cout**. Выводится название продукта (**product.name**), тип продукта (**product.type**), количество (**product.quantity**), цена (**product.price**) и статус (**product.status**), разделенные вертикальными чертами (|).

4. После каждой строки с информацией о продукте выводится символ новой строки **endl**.

Эта функция обеспечивает вывод информации о продуктах, у которых тип и статус соответствуют переданным значениям, на экран.

```
void printProductsByTypeAndStatus(const vector<Product>&
products, const string& type, const string& status)
{
    for (const Product& product : products)
    {
        if (product.type == type && product.status == status)
        {
        cout << product.name << " | " << product.type << " | "
<< product.quantity << " | " << product.price << " | "
<< product.status << endl;
        }
    }
```

```
}
```

4.12. Функция удаления элементов deleteProducktsByName().

Эта функция удаляет продукты с указанным именем из файла, используя вектор **products** для временного хранения данных, а затем сохраняет обновленный список продуктов в файл.

1. Сначала функция загружает данные о продуктах из файла с помощью функции **loadProductsFromFile**, сохраняя их в вектор **products**.

2. Затем используется алгоритм стандартной библиотеки C++**, remove_if**, чтобы удалить продукты с указанным именем из вектора **products**. Для этого используется лямбда-функция, которая возвращает **true**, если имя продукта соответствует указанному имени **productName**. Функция **remove_if** перемещает все элементы, не соответствующие условию, в конец вектора и возвращает итератор на первый элемент, который должен быть удален. Затем вызывается метод **erase,** чтобы удалить элементы, которые были перемещены в конец вектора.

3. Обновленный вектор **products** с удаленными продуктами сохраняется в тот же файл с помощью функции **saveProductsToFile.**

4. На экран выводится сообщение о том, сколько продуктов с указанным именем было удалено из файла.

Эта функция предоставляет возможность удалить из файла продукты с определенным именем, если они там присутствуют.

```
void deleteProductsByName(const string& filename, const
string& productName)
{
    vector<Product> products = loadProductsFromFile(filename);
    products.erase(remove_if(products.begin(), products.end(),
    [&productName](const Product& product)
    {
     return product.name == productName;
    }), products.end());

    saveProductsToFile(filename, products);
    cout << "Products with name '" << productName << "'
deleted (if found)." << endl;
}
```

4.13. Основная функция main().

Она представляет собой меню, где пользователь может выбирать различные действия для работы с файлом. Программа выполняет создание файла, запись в файл, добавление информации, удаление, просмотр информации, линейный поиск, линейную сортировку, бинарный поиск, вывод информации о продуктах по типу и статусу, а также реализует алгоритм быстрой сортировки и функцию обратной сортировки. После выполнения каждой операции программа ожидает нажатия клавиши для продолжения.

```
int main()
{
    string filename =
"/Users/katusarublevsk/Desktop/BSUIR/ОАИП/Mycursach/cmake-
build-debug/products.txt";
    int choice;
    while (true)
    {
        cout << "\nМеню:\n";
        cout << "1. Создать файл\n";
        cout << "2. Просмотр файла\n";
        cout << "3. Добавить данные\n";
        cout << "4. Удаление элементов\n";
        cout << "5. Сортировка выбором\n";
        cout << "6. Сортировка QuickSort\n";
        cout << "7. Двоичный поиск\n";
        cout << "8. Вывести продукты по типу и статусу\n";
        cout << "9. Линейный поиск\n";
        cout << "0. Выход\n";
        cout << "Ваш выбор: ";
        cin >> choice;
        if (choice == 0)
        {
            break;
        }
        switch (choice)
        {
            case 1: createFile(filename); break;
            case 2: viewFile(filename); break;
            case 3: addData(filename); break;
            case 4:
            {
                string productName;
                cin.ignore();
                cout << "Выберите продукт, который хотите
удалить: "; getline(cin, productName);
deleteProductsByName(filename, productName);
```

```
                break;
            }
            case 5:
            {
                vector<Product> products =
loadProductsFromFile(filename);
             selectionSort(products);
                saveProductsToFile(filename, products);
       cout << "Данные отсортированы выбором." << endl;
                break;
            }
            case 6:
            {
                vector<Product> products =
loadProductsFromFile(filename);
                quickSort(products, 0, products.size() - 1);
                saveProductsToFile(filename, products);
     cout << "Данные отсортированы QuickSort." << endl;
                break;
            }
            case 7:
            {
    vector<Product> products = loadProductsFromFile(filename);
                quickSort(products, 0, products.size() - 1);
                double price;
          cout << "Введите цену для поиска: "; cin >> price;
 vector<int> indices = binarySearch(products, price, 0,
products.size() - 1);
             if (!indices.empty())
             {
  cout << "Найдены продукты с ценой " << price << ":" << endl;
              for (int index : indices)
              {
  cout << "Продукт найден: " << products[index].name << endl;
              }
              }
              else
              {
              cout << "Продукт не найден." << endl;
              }
                break;
             }
            case 8:
            {
                string type, status;
                cin.ignore();
                cout << "Введите тип продукта: ";
                getline(cin, type);
                cout << "Введите статус продукта: ";
                getline(cin, status);
  vector<Product> products = loadProductsFromFile(filename);
```

```
  printProductsByTypeAndStatus(products, type, status);
                break;
            }
            case 9:
            {
                double price;
     cout << "Введите цену для поиска: "; cin >> price;
    vector<Product> products = loadProductsFromFile(filename);
                linearSearch(products, price);
                break;
            }
            default: cout << "Неверный выбор!" << endl;
        }
    }
    return 0;
}
```

5. ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

Моя программа представляет собой меню для работы с файлом, где пользователь может выполнять различные операции для управления данными в файле. Она позволяет пользователю:

1.Создавать новый файл для хранения данных.
2.Записывать в файл новую информацию.
3.Добавлять новые данные в конец файла без удаления существующей информации.
4.Удалять определенные данные из файла.
5.Просматривать содержимое файла для получения информации.
6.Осуществлять поиск определенного элемента в файле путем последовательного просмотра каждого элемента.
7.Сортировать элементы в файле в порядке возрастания или убывания.
8.Осуществлять эффективный поиск элемента в отсортированном файле путем деления области поиска пополам.
9.Выполнять сортировку, которая разделяет массив на более мелкие части для последующей сортировки.
10.Сортировать элементы по убыванию элемента в файле.
11.Осуществять поиск по заданному продукту и его статусу.

Программа предоставляет пользователю инструменты для структурирования, хранения и обработки данных в файле. Благодаря разнообразным функциям, пользователь может легко добавлять новую информацию, редактировать существующие данные, искать нужные элементы, сортировать данные по различным критериям и удалять ненужные записи.

Эти возможности помогают пользователю эффективно управлять информацией, обеспечивая быстрый доступ к необходимым данным и облегчая процесс обработки информации. Благодаря широкому спектру функций программа становится мощным инструментом для работы с данными, который помогает организовать информацию таким образом, чтобы пользователь мог эффективно выполнять свои задачи и достигать поставленных целей.

ЗАКЛЮЧЕНИЕ

Программа предоставляет пользователю инструменты для структурирования, хранения и обработки данных в файле. Благодаря разнообразным функциям, пользователь может легко добавлять новую информацию, редактировать существующие данные, искать нужные элементы, сортировать данные по различным критериям и удалять ненужные записи.

Эти возможности помогают пользователю эффективно управлять информацией, обеспечивая быстрый доступ к необходимым данным и облегчая процесс обработки информации. Благодаря широкому спектру функций программа становится мощным инструментом для работы с данными, который помогает организовать информацию таким образом, чтобы пользователь мог эффективно выполнять свои задачи и достигать поставленных целей.

СПИСОК ЛИТЕРАТУРЫ

1. Информационные технологии в управлении. Учебное пособие. – М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2008. – 336с.
2. Структуры данных и алгоритмы. : Пер. с англ. : Уч. пос. — М. : Издательский дом "Вильямс", 2007. - 400 с. : ил. — Парал. тит. англ.
3. Адельсон-Вельский Г. М., Ландис Е. М. (1962). Один алгоритм организации информации. Докл. АН СССР, 146, с. 263-266.
4. Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1974). The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass. (Русский перевод: Ахо А., Хоркрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М., "Мар", 1979.)
5. Зуев.К.И Структуры данных / Зуев.К.И [Электронный ресурс] // Профильное обучение. Информатика 10 класс: [сайт]. — URL: http://profil.adu.by/mod/book/tool/print/index.php?id=3573 (дата обращения: 10.05.2024).
6. НАУЧНЫЕ ИССЛЕДОВАНИЯ МОЛОДЫХ УЧЁНЫХ: сборник статей XII Международной научно-практической конференции. – Пенза: МЦНС «Наука и Просвещение». – 2021. – 252 с.
7. Б. Керниган, Д. Ритчи Язык программирования Си. – AT&T Bell Labs, 1978. – 343 с.
8. Таненбаум Э. Остин Т. Cho Архитектура компьютера. – Питер – 2013. – 320 с.
9. Г.В. Ваныкина, Т.О. Сундукова Структуры и алгоритмы компьютерной обработки данных [Текст] / Г.В. Ваныкина, Т.О. Сундукова — 1. — Новосибирск: ИНТИУИТ, 2011 — 219 с.
10. НОУ ИНИТ. Структуры и алгоритмы компьютерной работы [Электронный ресурс] // intuit.ru: [сайт]. - URL: https://intuit.ru/studies/courses/648/504/lecture/11466
11. Лафоре Р. Объектно-ориентированное программирование в C++. Классика Computer Science. 4 изд. — СПб.: Питер, 2018. — 928 с.

ПРИЛОЖЕНИЕ А:

Весь код выглядит так:

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <sstream>
using namespace std;
struct Product {
    string name;
    string type;
    int quantity;
    double price;
    string status;
};
void createFile(const string& filename)
{
    ofstream file(filename);
    if (!file.is_open())
    {
        cerr << "Ошибка создания файла!" << endl;
        return;
    }
    cout << "Файл создан успешно." << endl;
    file.close();
}
void viewFile(const string& filename)
{
    ifstream file(filename);
    if (!file.is_open())
    {
        cerr << "Ошибка открытия файла!" << endl;
        return;
    }
    string line;
    cout << "Содержимое файла:" << endl;
    while (getline(file, line))
    {
        stringstream ss(line);
        Product product;
        ss >> product.name >> product.type >> product.quantity
>> product.price;
        getline(ss, product.status);
        cout << product.name << " | " << product.type << " | "
<< product.quantity << " | " << product.price << " | " <<
product.status << endl;
    }
    file.close();
}
```

```
void addData(const string& filename)
{
    ofstream file(filename, ios::app);
    if (!file.is_open())
    {
        cerr << "Ошибка открытия файла!" << endl;
        return;
    }
    Product product;
    cout << "Введите данные продукта:" << endl;
    cout << "Название: ";
    cin.ignore();
    getline(cin, product.name);
    cout << "Тип: ";      getline(cin, product.type);
    cout << "Количество: "; cin >> product.quantity;
    cout << "Цена: ";     cin >> product.price;
    cin.ignore();
    cout << "Статус: ";  getline(cin, product.status);
    file << "\"" << product.name << "\" "
         << product.type << " "
         << product.quantity << " "
         << product.price << " "
         << product.status << endl;

    file.close();
    cout << "Данные добавлены в файл." << endl;
}
vector<Product> loadProductsFromFile(const string& filename)
{
    vector<Product> products;
    ifstream file(filename);
    if (!file.is_open())
    {
        cerr << "Ошибка открытия файла!" << endl;
        return products;
    }
    string line;
    while (getline(file, line))
    {
        stringstream ss;
        ss << line;
        Product product;
        ss >> product.name >> product.type >> product.quantity
>> product.price;
        ss.ignore();
        getline(ss, product.status);
        products.push_back(product);
    }
    file.close();
    return products;
}
```

```
void saveProductsToFile(const string& filename, const
vector<Product>& products)
{
    ofstream file(filename);
    if (!file.is_open())
   {
        cerr << "Ошибка открытия файла!" << endl;
        return;
    }
    for (const Product& product : products)
    {
        file << product.name << " "
             << product.type << " "
             << product.quantity << " "
             << product.price << " "
             << product.status
             << endl;
    }
    file.close();
}
void linearSearch(const vector<Product>& products, double
price)
{
    bool found = false;
    for (int i = 0; i < products.size(); i++)
   {
        if (products[i].price == price)
       {
            cout << "Продукт найден: " << products[i].name <<
endl;
            found = true;
        }
    }
    if (!found)
     {
        cout << "Продукт не найден." << endl;
     }
}
int partition(vector<Product>& products, int low, int high)
{
    double pivot = products[high].price;
    int i = (low - 1);
    for (int j = low; j < high; j++)
 {
        if (products[j].price <= pivot)
        {
            i++;
            swap(products[i], products[j]);
        }
    }
    swap(products[i + 1], products[high]);
```

```
    return (i + 1);
}
void quickSort(vector<Product>& products, int low, int high)
{
    if (low < high)
    {
        int pi = partition(products, low, high);
        quickSort(products, low, pi - 1);
        quickSort(products, pi + 1, high);
    }
}
vector<int> binarySearch(const vector<Product>& products,
double price, int low, int high)
{
    vector<int> indices;
    while (high >= low)
    {
        int mid = low + (high - low) / 2;
        if (products[mid].price == price)
        {
            int left = mid;
            while (left >= low && products[left].price ==
price)
           {
                indices.push_back(left);
                left--;
            }
            int right = mid + 1;
            while (right <= high && products[right].price ==
price)
            {
                indices.push_back(right);
                right++;
            }
            return indices;
        }
        if (products[mid].price > price)
        {
            high = mid - 1;
        } else
        {
            low = mid + 1;
        }
    }
    return indices;
}
void printProductsByTypeAndStatus(const vector<Product>&
products, const string& type, const string& status)
{
    for (const Product& product : products)
    {
```

```
        if (product.type == type && product.status == status)
        {
        cout << product.name << " | " << product.type << " | "
<< product.quantity << " | " << product.price << " | "
<< product.status << endl;
        }
    }
}
void deleteProductsByName(const string& filename, const
string& productName)
{
    vector<Product> products = loadProductsFromFile(filename);
    products.erase(remove_if(products.begin(), products.end(),
    [&productName](const Product& product)
    {
     return product.name == productName;
    }), products.end());

    saveProductsToFile(filename, products);
 cout << "Products with name '" << productName << "'
deleted (if found)." << endl;
}
int main()
{
    string filename =
"/Users/katusarublevsk/Desktop/BSUIR/OAИП/Mycursach/cmake-
build-debug/products.txt";
    int choice;
    while (true)
    {
        cout << "\nМеню:\n";
        cout << "1. Создать файл\n";
        cout << "2. Просмотр файла\n";
        cout << "3. Добавить данные\n";
        cout << "4. Удаление элементов\n";
        cout << "5. Сортировка выбором\n";
        cout << "6. Сортировка QuickSort\n";
        cout << "7. Двоичный поиск\n";
        cout << "8. Вывести продукты по типу и статусу\n";
        cout << "9. Линейный поиск\n";
        cout << "0. Выход\n";
        cout << "Ваш выбор: ";
        cin >> choice;
        if (choice == 0)
        {
            break;
        }
        switch (choice)
        {
            case 1: createFile(filename); break;
            case 2: viewFile(filename); break;
```

```
            case 3: addData(filename); break;
            case 4:
            {
                string productName;
                cin.ignore();
                cout << "Выберите продукт, который хотите
удалить: "; getline(cin, productName);
deleteProductsByName(filename, productName);
                break;
            }
            case 5:
           {
                vector<Product> products =
loadProductsFromFile(filename);
             selectionSort(products);
                saveProductsToFile(filename, products);
       cout << "Данные отсортированы выбором." << endl;
                break;
            }
            case 6:
            {
                vector<Product> products =
loadProductsFromFile(filename);
                quickSort(products, 0, products.size() - 1);
                saveProductsToFile(filename, products);
     cout << "Данные отсортированы QuickSort." << endl;
                break;
            }
            case 7:
           {
    vector<Product> products = loadProductsFromFile(filename);
                quickSort(products, 0, products.size() - 1);
                double price;
          cout << "Введите цену для поиска: "; cin >> price;
 vector<int> indices = binarySearch(products, price, 0,
products.size() - 1);
             if (!indices.empty())
             {
  cout << "Найдены продукты с ценой " << price << ":" << endl;
              for (int index : indices)
              {
  cout << "Продукт найден: " << products[index].name << endl;
              }
              }
              else
              {
              cout << "Продукт не найден." << endl;
              }
                break;
            }
            case 8:
```

```
            {
                string type, status;
                cin.ignore();
                cout << "Введите тип продукта: ";
                getline(cin, type);
                cout << "Введите статус продукта: ";
                getline(cin, status);
  vector<Product> products = loadProductsFromFile(filename);
  printProductsByTypeAndStatus(products, type, status);
                break;
            }
            case 9:
            {
                double price;
     cout << "Введите цену для поиска: "; cin >> price;
   vector<Product> products = loadProductsFromFile(filename);
                linearSearch(products, price);
                break;
            }
            default: cout << "Неверный выбор!" << endl;
        }
    }
    return 0;
}
```

ПРИЛОЖЕНИЕ В: