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 ПРОДУКЦИИ ЦЕХОВ ЗАВОДА

БГУИР КР

Студент: гр. 321702 Сергиевич Д.П.

Руководитель: старший преподаватель Гуревич О.В.

Минск 2024

**СОДЕРЖАНИЕ**

Оглавление

[ВВЕДЕНИЕ 3](#_Toc165824680)

[СТРУКТУРЫ И ФАЙЛЫ 4](#_Toc165824681)

[Классификация структур данных 4](#_Toc165824682)

[Основные функции структур данных 5](#_Toc165824683)

[АЛГОРИТМЫ СОРТИРОВКИ 6](#_Toc165824684)

[**Сортировка методом "пузырька" (простого обмена)** 7](#_Toc165824685)

[**Сортировка методом "шейкер"** 8](#_Toc165824686)

[**Сортировка методом простого выбора (простой перебор)** 10](#_Toc165824687)

[**Сортировка методом простого включения** 11](#_Toc165824688)

[**Сортировка методом Quicksort** 12](#_Toc165824689)

[АЛГОРИТМЫ ПОИСКА 15](#_Toc165824690)

[**Последовательный поиск** 15](#_Toc165824691)

[**Поиск делением пополам** 16](#_Toc165824692)

[**Интерполяционный поиск** 17](#_Toc165824693)

[ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ 20](#_Toc165824694)

[**1.** **Функция вывода меню в консоль menu().** 20](#_Toc165824695)

[**2.** **Функция создания необходимых файлов createFile().** 20](#_Toc165824696)

[**3.** **Функция записи данных в файлы writeToFile().** 21](#_Toc165824697)

[**Функция добавления записи addInfo().** 21](#_Toc165824698)

# 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.

# 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 С++).

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

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.

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.

1. **Упорядочение данных:**

* Некоторые структуры данных предназначены для упорядочения данных в определенном порядке, что упрощает поиск, сортировку и обработку данных.

1. **Быстрый доступ к данным:**

* Структуры данных должны обеспечивать быстрый доступ к данным, чтобы операции чтения и записи были эффективными.

1. **Эффективность использования ресурсов:**

* Хорошая структура данных должна быть эффективной в использовании ресурсов, таких как память и процессорное время.

1. **Поддержка операций:**

* Структуры данных должны поддерживать различные операции, такие как добавление, удаление, поиск, сортировка и обновление данных.

# 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).

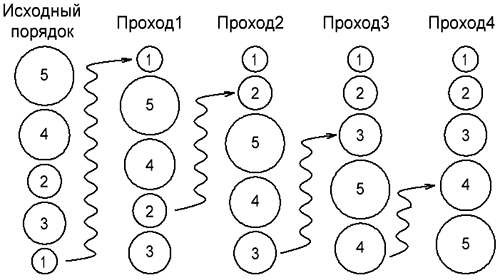
## **Сортировка методом "пузырька" (простого обмена)**

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:



**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;

}

}

## **Сортировка методом "шейкер"**

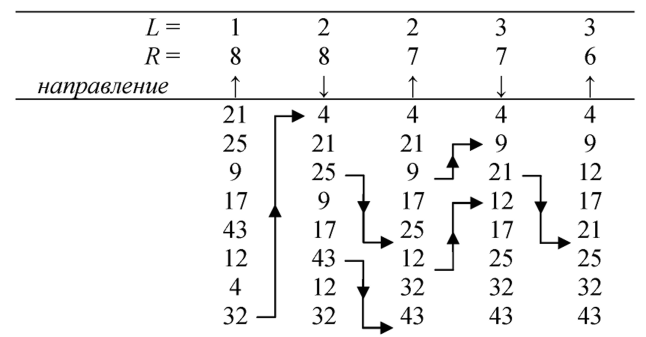
Шейкер-сортировка является усовершенствованным методом пузырьковой сортировки.

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.

* 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.



**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;

}

}

## **Сортировка методом простого выбора (простой перебор)**

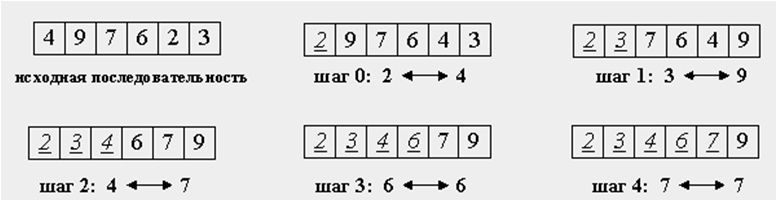
Это наиболее естественный алгоритм упорядочивания. При данной сортировке из массива выбирается элемент с наименьшим значением и обменивается с первым элементом. Затем из оставшихся n - 1 элементов снова выбирается элемент с наименьшим ключом и обменивается со вторым элементом, и т.д.

Шаги алгоритма:

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.



**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;

}

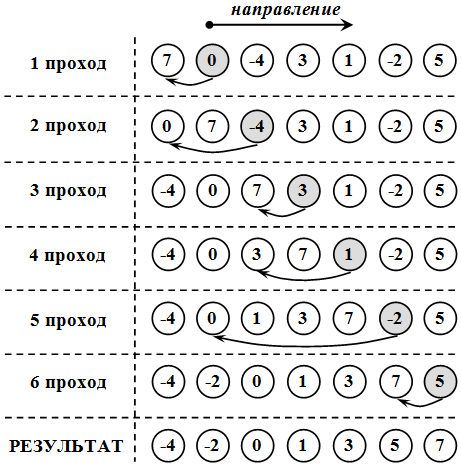
}

**Сортировка методом простого включения (сдвиг-вставка, вставками, вставка и сдвиг)**

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.



**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;

}

## **Сортировка методом Quicksort**

Быстрая сортировка (англ. quicksort) – это метод сортировки значений в списке в последовательные списки с помощью повторяющейся процедуры.

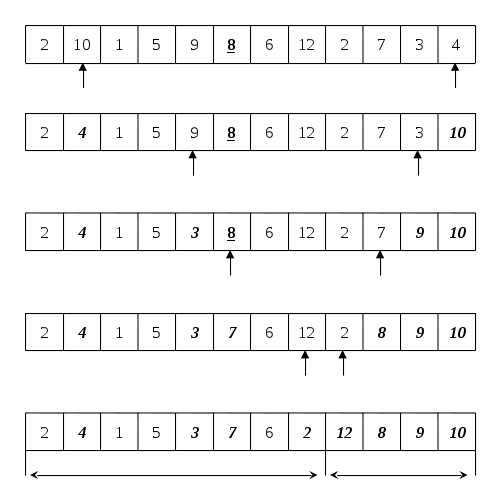
В методе быстрой сортировки выбирается значение из основного списка, которое называется опорным значением. Остальные значения разделяются на два списка:

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.



**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);}

# 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.

## **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.

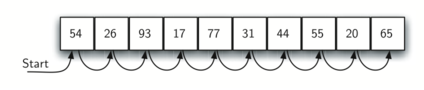
Алгоритм последовательного поиска:

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).



**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;

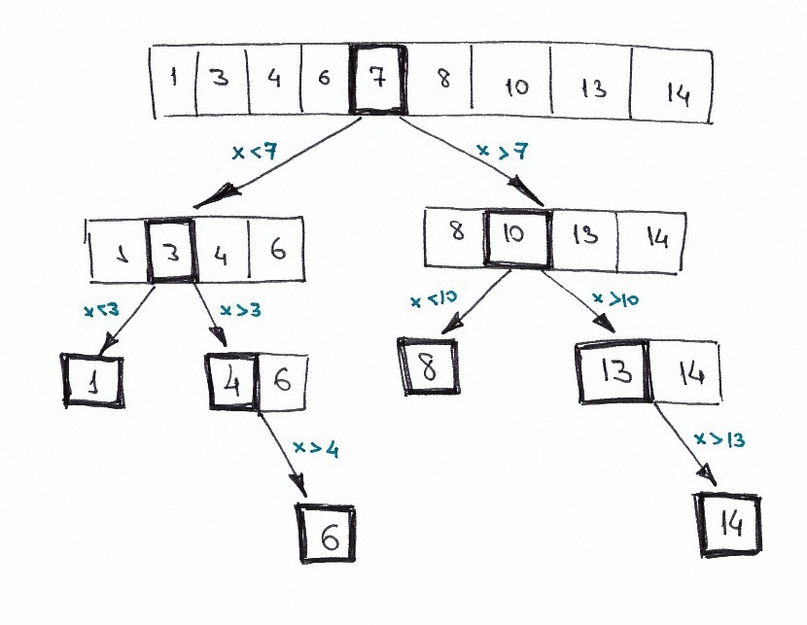
else return i;

}

## **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.



**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;

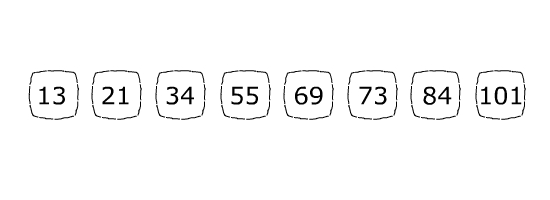
else return -1;

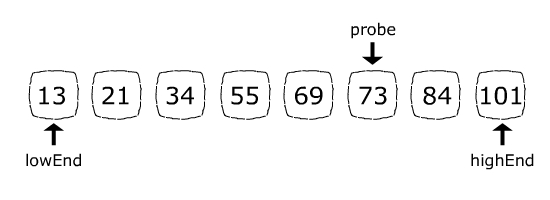
}

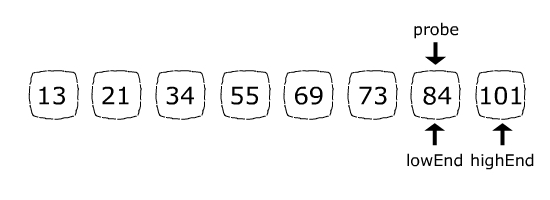
## **Интерполяционный поиск**

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.

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.







**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;

}

# 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.

## **Функция вывода меню в консоль menu().**

int menu()

{

cout << "Выберите: " << endl;

cout << "1. Создание файла " << endl;

cout << "2. Ввод данных " << endl;

cout << "3. Добавление данных " << endl;

cout << "4. Удаление данных " << endl;

cout << "5. Просмотр данных " << endl;

cout << "6. Линейный поиск " << endl;

cout << "7. Линейная сортировка " << endl;

cout << "8. Бинарный поиск " << endl;

cout << "9. Квиксорт " << endl;

cout << "10. Решение(вывод по убыванию) " << endl;

cout << "11. Выход " << endl;

int i;

cin >> i;

return i;

}

## **Функция создания необходимых файлов createFile().**

Эта функция открывает несколько файлов для записи, проверяет их доступность, и выводит соответствующие сообщения об успехе или ошибке.

void createFile() {

fi.open("okno.txt", ios::out | ios::app);

fl.open("Sorted.txt", ios::out | ios::app);

fr.open("Reshenie.txt", ios::out | ios::app);

f.open("ok.txt", ios::out);

fq.open("Quicksort.txt", ios::out | ios::app);

if (fi.is\_open()) {

if (f.is\_open()) {

f.close();

}

else {

cout << "Ошибка при создании файла" << endl;

}

fi.close();

cout << "Файл создан" << endl;

}

else {

cout << "Ошибка при создании файла" << endl;

}

}

## **Функция записи данных в файлы writeToFile().**

Данная функция открывает ранее созданные файлы и предлагает пользователю ввести количество изделий. Затем информация об каждом изделии (наименование, количество деталей, номер цеха) записывается в оба файла. После ввода всех данных файлы закрываются. Если возникла ошибка при открытии файлов, выводится сообщение об ошибке.

void writeToFile() {

int n, i;

fi.open("okno.txt", ios::out | ios::app);

f.open("ok.txt", ios::out | ios::app);

if (f.is\_open()) {

cout << "Введите количество изделий: " << endl;

cin >> n;

for (i = 0; i < n; i++)

{

zavod zv;

cout << "Введите наименование: ";

cin >> zv.name;

cout << "Введите количество деталей: ";

cin >> zv.kol\_vo;

cout << "Введите номер цеха: ";

cin >> zv.nomer;

fi << zv.name << " " << zv.kol\_vo << " " << zv.nomer << endl;

f << "Наименование: " << zv.name << endl;

f << "Количество деталей: " << zv.kol\_vo << endl;

f << "номер цеха: " << zv.nomer << endl;

f << "----------------------------------------------------------------" << endl;

}

f.close();

fi.close();

}

else {

cout << "Ошибка при открытии файла" << endl;

}

}

## **Функция добавления записи addInfo().**

Пользователю предлагается дополнить свои данные новой записью.

void addInfo() {

f.open("ok.txt", ios::out | ios::app);

fi.open("okno.txt", ios::out | ios::app);

if (f.is\_open()) {

zavod zv;

cout << "Введите name: ";

cin >> zv.name;

cout << "Введите kol\_vo: ";

cin >> zv.kol\_vo;

cout << "Введите nomer: ";

cin >> zv.nomer;

fi << zv.name << " " << zv.kol\_vo << " " << zv.nomer << endl;

f << "Наименование: " << zv.name << endl;

f << "Количество деталей: " << zv.kol\_vo << endl;

f << "Номер цеха: " << zv.nomer << endl;

f << "----------------------------------------------------------------" << endl;

f.close();

fi.close();

}

else {

cout << "Ошибка при открытии файла" << endl;

}

}

1. **Функция удаления записи delete().**

Пользователь имеет возможность отредактировать свои данные путём удаления записи.

void deletel()

{

ifstream fi("okno.txt");

ofstream f("ok.txt");

ofstream temp("temp.txt");

if (!fi.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

if (!f.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

int fi\_key;

cout << "Введите kol\_vo: " << endl;

cin >> fi\_key;

int kod = 0;

zavod zv;

while (fi >> zv.name >> zv.kol\_vo >> zv.nomer)

{

if (zv.kol\_vo == fi\_key)

{

kod = 1;

cout << "Позиция удалена." << endl;

}

else

{

// Записываем данные во временный файл, исключая удаляемую позицию

temp << zv.name << " " << zv.kol\_vo << " " << zv.nomer << endl;

f << "Наименование: " << zv.name << endl;

f << "Количество деталей: " << zv.kol\_vo << endl;

f << "Номер цеха: " << zv.nomer << endl;

f << "----------------------------------------------------------------" << endl;

}

}

fi.close();

f.close();

temp.close();

// Удаляем исходные файлы

remove("okno.txt");

// Заменяем исходный файл okno.txt временным файлом temp.txt

rename("temp.txt", "okno.txt");

if (kod == 0)

{

cout << "Не найдено введенного количества изделий." << endl;

}

}

1. **Функция просмотра (вывода) информации viewInfo().**

Данная функция предназначена для просмотра информации, которая была ранее записана в файл.

void viewInfo() {

ifstream inFile("ok.txt");

if (inFile.is\_open()) {

string line;

while (getline(inFile, line)) {

cout << line << endl;

}

inFile.close();

}

else {

cout << "Ошибка при открытии файла" << endl;

}

}

1. **Функция линейного поиска linposhyk().**

Функция ищет информацию об изделиях в файле по заданному количеству изделий и выводит результаты на экран. Если количество изделий совпадает с введенным пользователем, то выводится информация об изделии (наименование, количество и номер цеха). Если ни одного изделия с таким количеством не найдено, выводится соответствующее сообщение.

void linposhyk()

{

ifstream fi;

fi.open("okno.txt", ios::in);

if (!fi.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

int fi\_key;

cout << "Введите kol\_vo: " << endl;

cin >> fi\_key;

int kod = 0;

zavod zv;

while (fi >> zv.name >> zv.kol\_vo >> zv.nomer)

{

if (zv.kol\_vo == fi\_key)

{

kod = 1;

cout << "Найдено." << endl;

cout << "Наименование: " << zv.name << endl;

cout << "Количество изделий: " << zv.kol\_vo << endl;

cout << "Номер цеха: " << zv.nomer << endl;

cout << "—————————————————————-" << endl;

}

}

if (kod == 0)

{

cout << "Не найдено введенного количества изделий." << endl;

}

fi.close();

}

1. **Функция линейной сортировки lineynaysortirovka**().

Функция считывает данные из файла, сортирует их по количеству деталей и выводит отсортированные данные в консоль и в файл **"Sorted.txt"**.

void lineynaysortirovka() {

fi.open("okno.txt", ios::in);

fl.open("Sorted.txt", ios::out | ios::trunc);

if (!fi.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

if (!fl.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

vector<zavod> data;

zavod zv;

while (fi >> zv.name >> zv.kol\_vo >> zv.nomer) {

data.push\_back(zv);

}

fi.close();

// Линейная сортировка по полю 'kol\_vo'

for (int i = 0; i < data.size() - 1; i++) {

int m = i;

for (int j = i + 1; j < data.size(); j++) {

if (data[j].kol\_vo < data[m].kol\_vo) {

m = j;

}

}

// Обмен значениями

zavod temp = data[i];

data[i] = data[m];

data[m] = temp;

}

// Вывод отсортированных данных на консоль

for (const auto& item : data) {

cout << "Наименование: " << item.name << endl;

cout << "Количество деталей: " << item.kol\_vo << endl;

cout << "Номер цеха: " << item.nomer << endl;

cout << "----------------------------------------------------------------" << endl;

fl << "Наименование: " << item.name << endl;

fl << "Количество деталей: " << item.kol\_vo << endl;

fl << "Номер цеха: " << item.nomer << endl;

fl << "----------------------------------------------------------------" << endl;

}

fl.close();

}

1. **Функция сортировки и вывода информации по убыванию. (Применён метод линейной сортировки) reshenie().**

Данная функция считывает данные о запчастях из файла, сортирует их по количеству деталей в убывающем порядке и выводит отсортированные данные как на консоль, так и в файл **"Reshenie.txt"**.

void reshenie() {

fi.open("okno.txt", ios::in);

fr.open("Reshenie.txt", ios::out | ios::trunc);

if (!fi.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

if (!fr.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

vector<zavod> data;

zavod zv;

while (fi >> zv.name >> zv.kol\_vo >> zv.nomer) {

data.push\_back(zv);

}

fi.close();

// Линейная сортировка по полю 'kol\_vo' в убывающем порядке

for (int i = 0; i < data.size() - 1; i++) {

int m = i;

for (int j = i + 1; j < data.size(); j++) {

if (data[j].kol\_vo > data[m].kol\_vo) { // Измененное условие сравнения

m = j;

}

}

// Обмен значениями

zavod temp = data[i];

data[i] = data[m];

data[m] = temp;

}

// Вывод отсортированных данных на консоль

for (const auto& item : data) {

cout << "Наименование: " << item.name << endl;

cout << "Количество деталей: " << item.kol\_vo << endl;

cout << "Номер цеха: " << item.nomer << endl;

cout << "----------------------------------------------------------------" << endl;

fr << "Наименование: " << item.name << endl;

fr << "Количество деталей: " << item.kol\_vo << endl;

fr << "Номер цеха: " << item.nomer << endl;

fr << "----------------------------------------------------------------" << endl;

}

fr.close();

}

1. **Функция сортировки Quicksort quicksort(). (Дополнительно вспомогательная функция qssf()).**

Функция quicksort() реализует алгоритм быстрой сортировки для массива структур zavod, содержащего информацию о заводах. Она принимает на вход массив структур zv, левую границу сортировки l и правую границу сортировки r. Алгоритм быстрой сортировки разбивает массив на две части относительно опорного элемента, после чего рекурсивно сортирует каждую из частей.

Функция qssf() является вспомогательной функцией, которая объединяет чтение данных из файла, сортировку данных с помощью функции quicksort() и вывод отсортированных данных в консоль и файл.

void quicksort(zavod zv[30], int l, int r)

{

int l\_h = l;

int r\_h = r;

int take, take3;

char take2[50];

take = zv[l].kol\_vo;

take3 = zv[l].nomer;

strcpy\_s(take2, sizeof(take2), zv[l].name.c\_str());

while (l < r)

{

while ((zv[r].kol\_vo > take) && (l < r))

r--;

if (l != r)

{

zv[l] = zv[r];

l++;

}

while ((zv[l].kol\_vo < take) && (l < r))

l++;

if (l != r)

{

zv[r] = zv[l];

r--;

}

}

zv[l].kol\_vo = take;

zv[l].nomer = take3;

zv[l].name = take2;

take = l;

if (l\_h < take)

quicksort(zv, l\_h, take - 1);

if (r\_h > take)

quicksort(zv, take + 1, r\_h);

}

void qssf(){

fi.open("okno.txt", ios::in);

fq.open("Quicksort.txt", ios::out | ios::trunc); // Используем режим ios::trunc

if (!fi.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

if (!fi.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

if (!fq.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

zavod data[30];

int n = 0;

while (fi >> data[n].name >> data[n].kol\_vo >> data[n].nomer) {

n++;

}

fi.close();

quicksort(data, 0, n - 1);

// Вывод отсортированных данных на консоль

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

cout << "Наименование: " << data[i].name << endl;

cout << "Количество деталей: " << data[i].kol\_vo << endl;

cout << "Номер цеха: " << data[i].nomer << endl;

cout << "----------------------------------------------------------------" << endl;

fq << "Наименование: " << data[i].name << endl;

fq << "Количество деталей: " << data[i].kol\_vo << endl;

fq << "Номер цеха: " << data[i].nomer << endl;

fq << "----------------------------------------------------------------" << endl;

}

fq.close();

cout << "Сортировка завершена. Отсортированные данные записаны в файл Quick.txt." << endl;

}

1. **Функция бинарного поиска binaryposhyk().**

Функция реализует поиск детали завода по заданному количеству изделий с помощью линейного поиска в уже отсортированном массиве данных.

void binaryposhyk()

{

ifstream fi;

fi.open("okno.txt", ios::in);

if (!fi.is\_open())

{

cout << "Ошибка при открытии файла" << endl;

return;

}

vector<zavod> data; // Вектор для хранения данных из файла

zavod zv;

while (fi >> zv.name >> zv.kol\_vo >> zv.nomer)

{

data.push\_back(zv); // Заполнение вектора данными из файла

}

fi.close();

sort(data.begin(), data.end(), [](const zavod& a, const zavod& b) {

return a.kol\_vo < b.kol\_vo; // Сортировка по количеству изделий

});

int fi\_key;

cout << "Введите kol\_vo: " << endl;

cin >> fi\_key;

bool found = false;

for (const auto& item : data)

{

if (item.kol\_vo == fi\_key)

{

found = true;

cout << "Наименование: " << item.name << endl;

cout << "Количество изделий: " << item.kol\_vo << endl;

cout << "Номер цеха: " << item.nomer << endl;

cout << "----------------------------------------------------------------" << endl;

}

}

if (!found)

{

cout << "Не найдено введенного количества изделий." << endl;

}

}

1. **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, а также реализует алгоритм быстрой сортировки и функцию обратной сортировки. После выполнения каждой операции программа ожидает нажатия клавиши для продолжения и очищает экран.

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

setlocale(LC\_ALL, "RU");

while (true)

{

switch (menu())

{

case 1: createFile(); break;

case 2: writeToFile(); break;

case 3: addInfo(); break;

case 4: deletel(); break;

case 5: viewInfo(); break;

case 6: linposhyk(); break;

case 7: lineynaysortirovka(); break;

case 8: binaryposhyk(); break;

case 9: qssf(); break;

case 10: reshenie(); break;

case 11: return 0;

default:cout << "Неверный выбор" << endl;

}

puts("Press any key to continue");

system("pause"); system("cls");

}

return 0;

}

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.

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.