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 ПРОДУКЦИИ В ЦЕХАХ ЗАВОДА»**

Выполнил: гр.324402 Цевелюк А.И.

Проверила: Новицкая Л.И.

Минск 2024

**РЕФЕРАТ**

ТЕМА: структура списка продукции в цехах завода / А.И. Цевелюк. – Минск : БГУИР, 2024, – п.з. – 57 с., рисунков – 12, источников – 5, приложений – 2.

*Объект исследования:* структура списка продукции в цехах завода.

*Предмет исследования:* программное обеспечение, обеспечивающее работу со структурой списка продукции в цехах завода.

*Цель* *курсовой работы*: разработка программного обеспечения, обеспечивающего работу со структурой списка продукции в цехах завода.

*Методология проведения работы*: в процессе разработки системы была разработана программная составляющая программного средства.

*Результаты работы*: выполнена постановка задачи и определены основные методы ее решения; описаны основные алгоритмы работы программного средства; выполнено тестирование программного средства, показавшее его соответствие функциональным требованиям, поставленным в задании на разработку. Программный продукт разработан на языке *C++* с применением *MS Visual Studio 2022*.

*Область применения результатов*: разработанная система может быть применена в таких сферах, как частные или государственные заводы. Она способствует эффективному управлению данными о продукции.

**СОДЕРЖАНИЕ**

[Введение 6](#_Toc167181861)

[1 Структуры и файлы 8](#_Toc167181862)

[1.1 Структуры 8](#_Toc167181863)

[1.2 Файлы 9](#_Toc167181864)

[2 Алгоритмы сортировки 11](#_Toc167181865)

[3 Алгоритмы поиска 16](#_Toc167181866)

[4 Пользовательские функции 22](#_Toc167181867)

[5 Описание работы программы 27](#_Toc167181868)

[Заключение 34](#_Toc167181869)

[Список использованных источников 35](#_Toc167181870)

[Приложение А (обязательное) Листинг кода 36](#_Toc167181871)

[Приложение Б (обязательное) Блок-схема работы программы 56](#_Toc167181872)

Введение

В современных условиях все сферы человеческой деятельности активно подвергаются процессам информатизации и компьютеризации. Не являются исключением и цеха заводов, где внедряются программные решения для управления списками продукции. Использование специальных программ контроля в цехах позволяет эффективно управлять производственными процессами, улучшать производительность труда сотрудников и оптимизировать внутренние операции.

С внедрением современных информационных технологий заводы получают возможность более точно и оперативно отслеживать информацию о производимой продукции. Это способствует улучшению качества управления ресурсами и повышению конкурентоспособности предприятия в целом.

Программы служат для систематического хранения, управления, обработки и извлечения значительных объемов информации. Они находят применение во многих сферах, выполняя различные функции, такие как:

* централизованное хранение данных: программы позволяют собирать данные в одном месте, облегчая доступ к ним и их управление;
* обеспечение безопасности: программы защищают хранимую информацию с помощью систем контроля доступа и методов шифрования;
* поддержка транзакций: программы обеспечивают выполнение транзакций, состоящих из нескольких шагов, гарантируя их правильное выполнение или откат в случае ошибки;
* управление данными: программы предоставляют возможности для организации и актуализации данных, включая их добавление, обновление и удаление;
* аналитика: программы упрощают проведение анализа данных через запросы, отчеты и средства визуализации;
* поиск и фильтрация: программы позволяют быстро находить и отбирать нужную информацию;
* масштабируемость и производительность: программы могут расширяться для обработки растущего объема данных и обеспечения доступа большого числа пользователей без значительного снижения производительности;
* восстановление данных: программы обеспечивают сохранность информации и ее восстановление после сбоев или ошибок системы с помощью резервного копирования и журналирования.

Взаимодействие с данными можно осуществлять двумя основными способами:

* изменения и поиск выполняются прямо в файле на диске с использованием методов работы с файловыми структурами. Это может замедлить обработку данных из-за медленного доступа к дискам, однако такой подход не ограничивает использование оперативной памяти;
* все данные или их часть загружается в массив структур для дальнейшей работы в оперативной памяти. Этот способ ускоряет обработку за счет более быстрого доступа к данным, однако требует больше памяти.

Изучение структур и алгоритмов сортировки и поиска играет ключевую роль при программе для контроля продукции в цехах завода. Сведения о продукции включают в себя наименование изделия, количество выпущенных единиц и номер цеха, в котором была произведена продукция.

Цель данного курсового проекта заключается в изучении методов сортировки и поиска данных о продукции в цехах завода, а также в разработке программного обеспечения на языке C++, соответствующего индивидуальному заданию. В рамках проекта будет реализована функциональность для вывода информации о продукции по каждому наименованию в порядке убывания их количества для заданного цеха.

Индивидуальное задание представляет собой следующее: для заданного цеха необходимо вывести изделия по каждому наименованию в порядке убывания их количества. Ключевым параметром в данной задаче является количество выпущенных изделий.

Автоматизация процессов управления данными о продукции в различных цехах завода значительно повышает эффективность работы предприятия и упрощает анализ производственных процессов. Ожидается, что разработанная программа существенно ускорит и упростит процесс анализа продукции в цехах завода.

1 Структуры и файлы

1.1 Структуры

В C++ структура представляет собой пользовательский тип данных, который позволяет объединять переменные различных типов под единым именем. Она используется для организации и хранения данных, создания более сложных типов данных и передачи информации между функциями. Для определения структуры в C++ используется ключевое слово struct, за которым следует имя структуры и определение ее членов в фигурных скобках. Доступ к полям структуры осуществляется с помощью оператора точка [1].

В данном проекте используется структура, описывающая отдельную продукцию в цехах завода. Ее формат представлен следующим образом:

struct production

{

std::string name; // Наименование продукции

unsigned amount; // Количество продукции

unsigned workshop\_number; // Номер цеха

unsigned id; // Идентификатор продукции

};

Эта структура обеспечивает удобное хранение информации о производимой продукции, идентифицируя каждый продукт по его наименованию, количеству, номеру цеха и идентификатору.

Для работы со структурами в данной курсовой работе используется вектор структур production. В C++ вектор представляет собой динамический массив, предоставляемый стандартной библиотекой шаблонов (STL). Вектор автоматически управляет своей памятью и способен изменять свой размер, добавляя или удаляя элементы. Он хранит элементы в непрерывной области памяти, подобно обычному массиву, что обеспечивает эффективный произвольный доступ к элементам [2].

Используемый вектор имеет вид:

std::vector<production> productions;

Таким образом, доступ к любому элементу в лице структуры легко осуществить с использованием оператора квадратных скобок ([]), характерного для вектора, а к любому полю этой структуры – с помощью оператора точки (.).

1.2 Файлы

Файл представляет собой набор данных или информации, который хранится на компьютере или другом устройстве. В контексте работы с продукциями, файлы могут содержать сведения о различных продуктах, их количестве, номерах цехов и идентификаторах. Файлы обычно имеют определенное расширение имени файла, указывающее на тип данных, содержащихся в файле. Эти файлы можно открывать, редактировать, перемещать, копировать и удалять с помощью различных программ и приложений [3].

В данной работе использован один файл, имеющий имя database и расширение .txt. В данный файл записывается информация о всех продукциях в формате таблицы, как показано на рисунке 1.1.

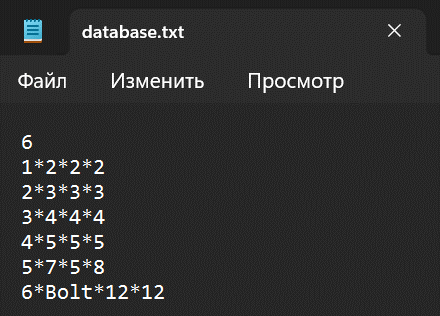


Рисунок 1.1 – Внешний вид файла database.txt.

В каждую колонку записывается по одному полю структуры: id, name, amount, workshop\_number. В первую строчку файла, в свою очередь, записывается количество продукции.

Для записи и чтения данных из файла в работе использован класс fstream из библиотеки fstream в C++. В отличие от классов ifstream и ofstream, которые представляют собой потоки ввода и вывода соответственно, fstream объединяет возможности обоих классов, предоставляя функционал для работы с файлами как для чтения, так и для записи.

Класс fstream используется для чтения и записи информации в файл. Он применяется, когда необходимо открыть файл для извлечения данных или сохранения информации. Этот класс предоставляет операции как ввода, так и вывода, что делает его универсальным для работы с файлами. Для его использования создается объект класса fstream, который ассоциируется с файлом на диске. После этого можно проводить чтение и запись данных в файл с использованием стандартных операций ввода-вывода.

Для чтения данных из файла используется метод fstream::open, который открывает файл в режиме чтения. После открытия файла данные могут быть извлечены стандартными методами ввода, аналогичными использованию std::cin.

Для записи данных в файл используется метод ofstream::open, который открывает файл в режиме записи. Запись данных осуществляется подобным образом, как вывод данных на экран через std::cout, но в данном случае информация направляется в файл.

При изменении данных о продукции, содержимое файла database.txt переписывается с помощью fstream. При выводе списка продукции на экран или перезапуске программы в режиме работы с последним созданным файлом, программа считывает содержимое файла с помощью fstream и записывает всё в вектор.

2 Алгоритмы сортировки

Сортировка – это процесс упорядочивания элементов массива или другого набора данных по определенному ключу, что облегчает последующий поиск. Сортировка считается устойчивой, если элементы с одинаковыми ключами сохраняют свой первоначальный порядок относительно друг друга после сортировки. Важное правило сортировки требует избегать использования дополнительных массивов и выполнять все изменения непосредственно в исходном массиве. Сортировки делятся на внутренние, которые выполняются в оперативной памяти, и внешние, которые осуществляются с использованием файлов [4].

Методы внутренней сортировки классифицируются по времени их выполнения. Эффективность сортировки часто оценивается по количеству операций сравнения и перестановки элементов. Существуют простые методы сортировки, которые подходят для небольших наборов данных (до 50 элементов), так как они быстро выполняются и легко реализуются. Для больших наборов данных (100 и более элементов) более сложные методы сортировки оказываются эффективнее, хотя и требуют более сложной реализации.

Ниже описаны наиболее известные простые методы сортировки.

Пузырьковая сортировка – это простой алгоритм, который многократно проходит по списку элементов, сравнивая соседние элементы попарно и меняя их местами, если они находятся в неправильном порядке. На каждом проходе самый большой (или самый маленький, в зависимости от порядка сортировки) элемент "всплывает" на своё место, как пузырёк в воде, что и дало название алгоритму. Эта сортировка была использована в курсовой работе. Ниже приведен код, который реализует эту сортировку:

void bubble\_sort\_by\_amount(const unsigned type)

{

for (int i = 0; i < static\_cast<int>(productions.size()); ++i)

{

for (int j = 0; j < static\_cast<int>(productions.size()) - i - 1; j++)

{

if (type == 1)

{

if (productions[j].amount > productions[j + 1].amount)

{

const production temp = productions[j];

productions[j] = productions[j + 1];

productions[j + 1] = temp;

}

}

else

{

if (productions[j].amount < productions[j + 1].amount)

{

const production temp = productions[j];

productions[j] = productions[j + 1];

productions[j + 1] = temp;

}

}

}

}

}

Сортировка методом выбора. Этот метод сортировки был использован в разработанной программе. В данном алгоритме на каждом шаге выбирается минимальный элемент из оставшихся несортированных элементов и помещается на текущую позицию. Сортировка выполняется пошагово, выбирая наименьший (или наибольший) элемент из несортированной части массива и меняя его местами с первым элементом этой части. Этот процесс повторяется до тех пор, пока весь массив не будет отсортирован. Описание алгоритма сортировки методом выбора обычно выглядит следующим образом:

* находим минимальный элемент в несортированной части массива;
* меняем местами этот минимальный элемент с первым элементом в несортированной части;
* сдвигаем границу между отсортированной и несортированной частями массива на один элемент вправо;
* повторяем предыдущие шаги для оставшейся несортированной части массива до полной сортировки.

Ниже приведен код, который реализует эту сортировку:

void selection\_sort\_by\_amount(const unsigned type)

{

for (int i = 0; i < static\_cast<int>(productions.size()) - 1; i++)

{

int m = i;

for (int j = i + 1; j < static\_cast<int>(productions.size()); j++)

{

if (type == 1)

{

if (productions[j].amount < productions[m].amount) m = j;

}

else

{

if (productions[j].amount > productions[m].amount) m = j;

}

}

const production temp = productions[m];

productions[m] = productions[i];

productions[i] = temp;

}

}

Сортировка вставками – это метод, который упорядочивает массив, поочередно вставляя каждый элемент на его правильное место. Алгоритм начинается с первого элемента и последовательно проходит по массиву, вставляя каждый элемент в соответствующее место в уже отсортированной части массива.

Шейкерная сортировка – это модификация пузырьковой сортировки, которая улучшает производительность за счет перемещения элементов в обе стороны. В отличие от стандартной пузырьковой сортировки, где элементы сдвигаются только в одном направлении, шейкерная сортировка проходит по массиву в обоих направлениях, что ускоряет процесс сортировки.

К более продвинутым методам сортировки относятся:

* сортировка Шелла, предложенная Д. Шеллом в 1959 году. Этот метод улучшает сортировку вставками за счет сравнения и перемещения элементов на определенные расстояния друг от друга, которые постепенно уменьшаются до тех пор, пока массив не будет окончательно отсортирован;
* сортировка кучей, известная как HeapSort, разработанная Дж. Уильямсом в 1964 году. Этот алгоритм использует структуру данных "куча", представляющую собой бинарное дерево, где каждый узел имеет значение большее или равное значениям его потомков. Максимальный элемент удаляется из кучи и помещается в конец массива, после чего куча перестраивается, чтобы следующий по величине элемент оказался в корне. Этот процесс повторяется, пока весь массив не будет отсортирован;
* быстрая сортировка или QuickSort, созданная Ч. Хоаром в 1962 году. Этот алгоритм, являющийся улучшенной версией пузырьковой сортировки, является одним из самых эффективных. В курсовой работе использован именно этот современный метод сортировки. Принцип работы QuickSort заключается в следующем: выбирается опорный элемент (обычно средний по позиции), затем массив делится на две части — элементы, меньшие опорного, и элементы, большие опорного. Далее сортировка рекурсивно применяется к каждой части, пока массив не будет полностью отсортирован.

Ниже приведен код, который реализует эту сортировку:

void quick\_sort\_by\_amount(const unsigned low, const unsigned high, const unsigned type)

{

if (low < high)

{

const unsigned x = productions[(low + high) / 2].amount;

unsigned i = low, j = high;

while (i <= j)

{

if (type == 1)

{

while (productions[i].amount < x) i++;

while (productions[j].amount > x) j--;

}

else

{

while (productions[i].amount > x) i++;

while (productions[j].amount < x) j--;

}

if (i <= j)

{

const production temp = productions[i];

productions[i] = productions[j];

productions[j] = temp;

i++;

j--;

}

}

if (low < j) quick\_sort\_by\_amount(low, j, type);

if (i < high) quick\_sort\_by\_amount(i, high, type);

}

}

Сравнение различных методов сортировки показывает, что при работе с массивами, содержащими более 100 элементов, пузырьковая сортировка оказывается наименее эффективной. В то время как QuickSort работает быстрее HeapSort в 2-3 раза и сортировки Шелла в 3-7 раз, что делает его предпочтительным выбором для больших объемов данных.

3 Алгоритмы поиска

Предположим, что имеется структура вида:

struct Ttype

{

type key; // ключевое поле типа type

// . . . другие компоненты структуры

} \*a; // указатель на динамический массив таких структур

Для поиска нужного элемента в массиве структур a (размер которого определяется при запуске программы и равен n) необходимо найти индекс i\_key, такой что a[i\_key].key = f\_key. Здесь key — это поле структуры, представляющее интерес, а f\_key — значение ключа, которое мы ищем и которое имеет тот же тип, что и key. После нахождения индекса i\_key можно получить доступ к другим полям искомой структуры, используя a[i\_key].

Рассмотрим методы линейного, бинарного и интерполяционного поиска, использованные в курсовой работе [5].

Когда отсутствует информация о структуре данных, применяется линейный поиск, который заключается в последовательном просмотре каждого элемента массива. Поиск останавливается либо при нахождении нужного ключа, либо при достижении конца массива, если ключ не найден. В случае отсутствия уникальности ключа необходимо проверять все элементы массива.

Ниже приведен код, который реализует линейый поиск в курсовой работе:

void find\_linear(const unsigned amount\_to\_find)

{

int code = 0;

for (const production& production : productions)

{

if (production.amount == amount\_to\_find)

{

code = 1;

std::cout << production.id << ' ';

}

}

if (code == 0)

std::cout << "there's no elements with such amount!\n";

else

std::cout << '\n';

}

Бинарный поиск (поиск делением пополам) – это эффективный алгоритм поиска элемента в отсортированном массиве. Данный метод был использован в разработанной программе. Бинарный поиск значительно сокращает время поиска, сравнивая целевой элемент с серединным элементом массива. Процесс поиска включает следующие шаги:

* определение середины отсортированного массива;
* сравнение целевого значения ключа с ключом серединного элемента;
* если ключи совпадают, поиск завершается;
* если целевое значение ключа меньше, чем ключ серединного элемента, поиск продолжается в левой половине массива;
* если целевое значение ключа больше, чем ключ серединного элемента, поиск продолжается в правой половине массива.

Эти шаги повторяются для нового подмассива до тех пор, пока не будет найден элемент с нужным ключом или подмассив не станет пустым.

Бинарный поиск достигает логарифмической временной сложности O(log n), что делает его гораздо быстрее линейного поиска, особенно на больших объемах данных.

Ниже приведен код, который реализует бинарный поиск в курсовой работе:

void find\_binary(const unsigned amount\_to\_find)

{

selection\_sort\_by\_amount(1);

int i\_key = 0, j = static\_cast<int>(productions.size()) - 1;

while (i\_key < j)

{

const int m = (i\_key + j) / 2;

if (productions[m].amount < amount\_to\_find) i\_key = m + 1;

else j = m;

}

if (productions[i\_key].amount != amount\_to\_find)

{

std::cout << "there's no elements with such amount!\n";

return;

}

std::cout << productions[i\_key].id << ' ';

for (int i = i\_key + 1; i < static\_cast<int>(productions.size()); i++)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

for (int i = i\_key - 1; i >= 0; i--)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

std::cout << '\n';

}

Интерполяционный поиск – это алгоритм, используемый для нахождения определенного значения в отсортированном массиве. Этот метод также был применен в разработанной программе. Интерполяционный поиск является улучшенной версией бинарного поиска и основывается на интерполяции, которая предсказывает местоположение искомого значения, используя его числовое значение.

Основная идея состоит в том, что данные в массиве воспринимаются как равномерно распределенные, и используется интерполяционная формула для определения позиции искомого элемента:

pos = lo + ((hi - lo) / (A[hi] - A[lo])) \* (x - A[lo])

где:

pos – предполагаемая позиция искомого элемента.

lo и hi – индексы первого и последнего элементов части массива, в которой выполняется поиск.

A[] – массив, в котором выполняется поиск.

x – значение, которое необходимо найти.

Процесс поиска включает следующие шаги:

* вычисление предполагаемой позиции pos с использованием интерполяционной формулы;
* сравнение значения в позиции pos с искомым значением x;
* если значения равны, поиск завершается;
* если значение в позиции pos меньше x, поиск продолжается в верхней части массива;
* если значение в позиции pos больше x, поиск продолжается в нижней части массива.

Процесс повторяется до тех пор, пока не будет найден искомый элемент или область поиска не станет пустой.

Интерполяционный поиск особенно эффективен для массивов с равномерно распределенными значениями и может достигать временной сложности близкой к O(log log n), что значительно быстрее, чем O(log n) у бинарного поиска. Однако при неоднородном распределении значений его эффективность может снижаться.

В данной курсовой работе был использован модифицированный вариант интерполяционного поиска: если ключ элемента с позицией pos меньше ключа x, то lo сдвигается на одну позицию вправо и поиск продолжается; в противном случае проверяются поля найденного элемента на соответствие условиям — при успешной проверке найденный элемент выводится на экран, затем lo сдвигается на одну позицию вправо и поиск продолжается. Такая модификация необходима, так как условие поиска нестрогое: может существовать несколько элементов с нужным количеством мест, подходящим временем и пунктом назначения.

Ниже приведен код, который реализует интерполяционный поиск в курсовой работе:

void find\_interpolation(const unsigned amount\_to\_find)

{

selection\_sort\_by\_amount(1);

int low = 0;

int high = static\_cast<int>(productions.size()) - 1;

while (low <= high && amount\_to\_find >= productions[low].amount && amount\_to\_find <= productions[high].amount)

{

if (low == high)

{

if (productions[low].amount == amount\_to\_find)

{

std::cout << productions[low].id << ' ';

for (int i = low + 1; i < static\_cast<int>(productions.size()); i++)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

for (int i = low - 1; i >= 0; i--)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

std::cout << '\n';

return;

}

std::cout << "there's no elements with such amount!\n";

return;

}

const int pos = low + (high - low) / static\_cast<int>((productions[high].amount - productions[low].amount)) \* static\_cast<int>((amount\_to\_find - productions[low].amount));

if (productions[pos].amount == amount\_to\_find)

{

std::cout << productions[pos].id << ' ';

for (int i = pos + 1; i < static\_cast<int>(productions.size()); i++)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

for (int i = pos - 1; i >= 0; i--)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

std::cout << '\n';

return;

}

if (productions[pos].amount < amount\_to\_find)

low = pos + 1;

else

high = pos - 1;

}

std::cout << "there's no elements with such amount!\n";

}

4 Пользовательские функции

В работе реализовано и использовано 25 пользовательских функций, не считая функцию main(). Ниже приведено объяснение назначения каждой функции.

inline void menu()- вызывается при заходе в main(). Выводит на экран главное меню программы. Выводит список всей продукции, а также следующие пункты: создать новую продукцию, удалить продукцию, очистить список продукций, сортировать по количеству продукции, сортировать по идентификатору, найти продукцию по количеству, выйти из программы. Основана на цикле до тех пор, пока пользователь не выберет пункт выйти из программы. Не возвращает никаких значений.

inline void clear\_input() – функция, которая создана для очистки буфера ввода. Вызывается в тех случаях, когда необходимо очистить буфер ввода (например, когда был введён строковый параметр в числовую переменную и др.). Работает следующим образом:

inline void clear\_input()

{

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

}

Не возвращает никаких значений.

inline void checked\_int\_input(int& input) – функция, которая отвечает за правильный ввод переменных целочисленного типа данных. Вызывается в случае того, когда необходимо ввести переменную. Принимает в себя ссылку на переменную, которую необходимо ввести. Работает следующим образом:

inline void checked\_int\_input(int& input)

{

while (!(std::cin >> input))

{

clear\_input();

std::cout << "Enter valid input: ";

}

}

Не возвращает никаких значений.

inline void checked\_unsigned\_input(unsigned& input) – функция, которая отвечает за правильный ввод переменных неотрицательного целочисленного типа данных. Вызывается в случае того, когда необходимо ввести переменную. Принимает в себя ссылку на переменную, которую необходимо ввести. Работает следующим образом:

inline void checked\_unsigned\_input(unsigned& input)

{

int to\_unsigned;

checked\_int\_input(to\_unsigned);

if (to\_unsigned < 0)

to\_unsigned \*= -1;

input = static\_cast<unsigned>(to\_unsigned);

}

Не возвращает никаких значений.

inline void end\_of\_function() – функция, которая позволяет выполнить запрос нажатия любой клавиши и очистки консоли. Работает следующим образом:

inline void end\_of\_function()

{

system("pause");

system("cls");

}

Не возвращает никаких значений.

bool is\_empty() – функция, которая проверяет, является ли вектор продукции пустым. Возвращает true, если вектор пустой и false в обратном случае.

void create\_new\_production() – функция создания новой продукции. Запрашивает все необходимые поля для новой продукции, после чего добавляет новую продукцию в вектор. Не возвращает никаких значений.

void set\_new\_ids(const unsigned id\_to\_delete) – функция, которая является вспомогательной функцией при удалении продукции. На вход принимает значение идентификатора продукции, которая удаляется из списка. Работает следующим образом:

void set\_new\_ids(const unsigned id\_to\_delete)

{

for (production& production : productions)

if (production.id > id\_to\_delete)

production.id--;

}

Не возвращает никаких значений.

void print\_productions() – функция форматированного вывода хэдера и списка продукции на экран. Не возвращает никаких значений.

void write\_to\_file() – функция для записи вектора продукции в форматированном виде в файл. Все поля структуры записываются через ‘\*’, чтобы не было проблем с чтением структуры из файла. Работает следующим образом:

void write\_to\_file()

{

std::fstream file;

file.open("database.txt", std::ios::out);

file << productions.size() << '\n';

for (auto it = productions.cbegin(); it != productions.cend(); ++it)

file << it->id << '\*' << it->name << '\*' << it->amount << '\*' << it->workshop\_number << '\n';

file.close();

}

Не возвращает никаких значений.

void read\_from\_file() – функция для чтения заранее отформатированного файла структур в вектор. Эта функция и функция выше были созданы для сохранения данных о векторе продукции при закрытии программы для последующего перезахода. Не возвращает никаких значений.

void clear() – функция для очистки вектора продукции. Реализована следующим образом:

void clear()

{

if (is\_empty())

{

std::cout << "Your products list is empty!\n";

return;

}

productions.clear();

std::cout << "Successfully cleared!\n";

}

Не возвращает никаких значений.

void delete\_production() – функция для удаления продукции из списка. Запрашивает идентификатор продукции на удаление, после чего проверяет, существует ли такой идентификатор и, если существует, то удаляет продукцию с таким идентификатором и меняет идентификатор остальной продукции, чтобы избежать разрыва. Не возвращает никаких значений.

void bubble\_sort\_by\_amount(const unsigned type) – функция для сортировки вектора продукции по количеству методом пузырька. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Не возвращает никаких значений.

void selection\_sort\_by\_amount(const unsigned type) – функция для сортировки вектора продукции по количеству методом выбора. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Не возвращает никаких значений.

void quick\_sort\_by\_amount(const unsigned low, const unsigned high, const unsigned type) – функция для сортировки вектора продукции по количеству методом быстрой сортировки. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Также принимает в себя параметры нижнего индекса и верхнего индекса, которые существуют в связи с тем, что метод быстрой сортировки является рекурсивным. Не возвращает никаких значений.

void sort\_by\_amount() – функция для запроса информации о желаемой сортировки по количеству. Запрашивает информацию о том, сортировать ли вектор по возрастанию или по убыванию, а также о желаемом методе сортировки (пузырьком, прямым выбором или быстрой). Не возвращает никаких значений.

void bubble\_sort\_by\_id(const unsigned type) – функция для сортировки вектора продукции по идентификатору методом пузырька. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Не возвращает никаких значений.

void selection\_sort\_by\_id(const unsigned type) – функция для сортировки вектора продукции по идентификатору методом выбора. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Не возвращает никаких значений.

void quick\_sort\_by\_id(const unsigned low, const unsigned high, const unsigned type) – функция для сортировки вектора продукции по идентификатору методом быстрой сортировки. На вход принимает в себя значение типа сортировки, где 1 – по возрастанию, а 2 – по убыванию. Также принимает в себя параметры нижнего индекса и верхнего индекса, которые существуют в связи с тем, что метод быстрой сортировки является рекурсивным. Не возвращает никаких значений.

void sort\_by\_id() – функция для запроса информации о желаемой сортировки по идентификатору. Запрашивает информацию о том, сортировать ли вектор по возрастанию или по убыванию, а также о желаемом методе сортировки (пузырьком, прямым выбором или быстрой). Не возвращает никаких значений.

void find\_linear(const unsigned amount\_to\_find) – функция для проведения линейного поиска в векторе. На вход принимает значение количества, по которому нужно искать продукцию. В случае, если продукция была найдена, то выводит её на экран, в противном случае выводит на экран сообщение об отсутствии таковой. Не возвращает никаких значений.

void find\_binary(const unsigned amount\_to\_find) – функция для проведения бинарного поиска в векторе. На вход принимает значение количества, по которому нужно искать продукцию. В случае, если продукция была найдена, то выводит её на экран, в противном случае выводит на экран сообщение об отсутствии таковой. Не возвращает никаких значений.

void find\_interpolation(const unsigned amount\_to\_find) – функция для проведения интерполяционного поиска в векторе. На вход принимает значение количества, по которому нужно искать продукцию. В случае, если продукция была найдена, то выводит её на экран, в противном случае выводит на экран сообщение об отсутствии таковой. Не возвращает никаких значений.

void find() – функция для запроса информации о желаемом поиске по количеству. Запрашивает информацию о том, по какому значению количества нужно искать, а также о желаемом методе поиска (линейным, бинарным, интерполяционным). Не возвращает никаких значений.

5 Описание работы программы

Программа представляет собой систему управления списком продукции в цехах завода. Код программы содержит структуру production, которая описывает продукцию.

При запуске программы, на экран пользователя выводится меню, в котором он может работать с программой при помощи выбора одного из пунктов меню, а именно:

* создать новую продукцию;
* удалить продукцию;
* очистить список продукции;
* сортировка списка продукции по количеству;
* сортировка списка продукции по идентификатору;
* поиск элемента списка продукции по количеству;
* выход из программы.

Скриншот меню показан на рисунке 5.1.

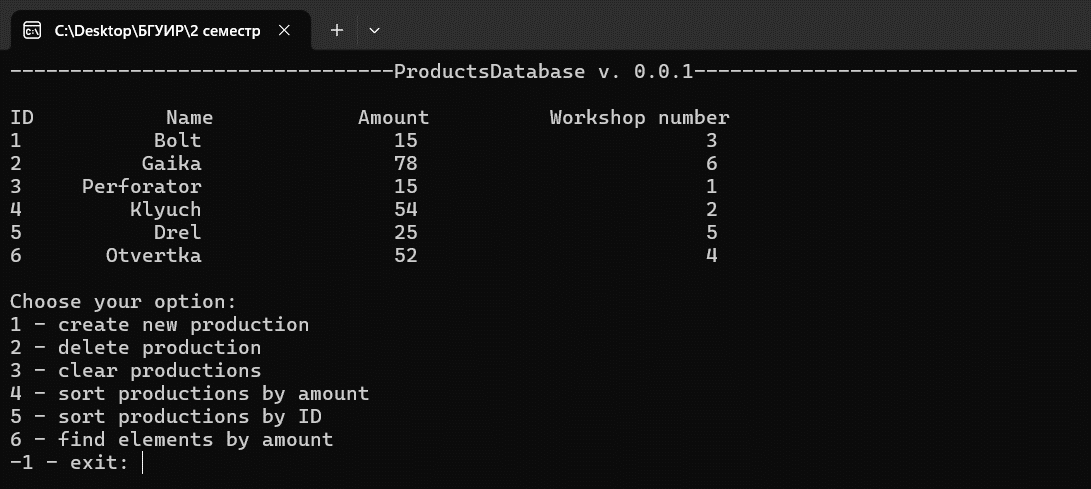


Рисунок 5.1 – Меню программы

При проведении каких-либо операций, изменяющих информацию о продукции, изменения производятся как в векторе структур (в окне приложения соотв.), так и в текстовом файле, содержащем информацию, необходимую для вывода, считывания и сохранения информации, посредством переписывания файла.

Выбор пункта “Создать новую продукцию” запросит список данных, необходимых для создания новой структуры продукции (наименование, количество, номер цеха). Идентификатор присваивается продукции автоматически, основываясь на размере нынешнего вектора структур. Новая продукция автоматически добавляется в конец вектора, после чего пользователь возвращается в главное меню программы (см. рисунок 5.2).

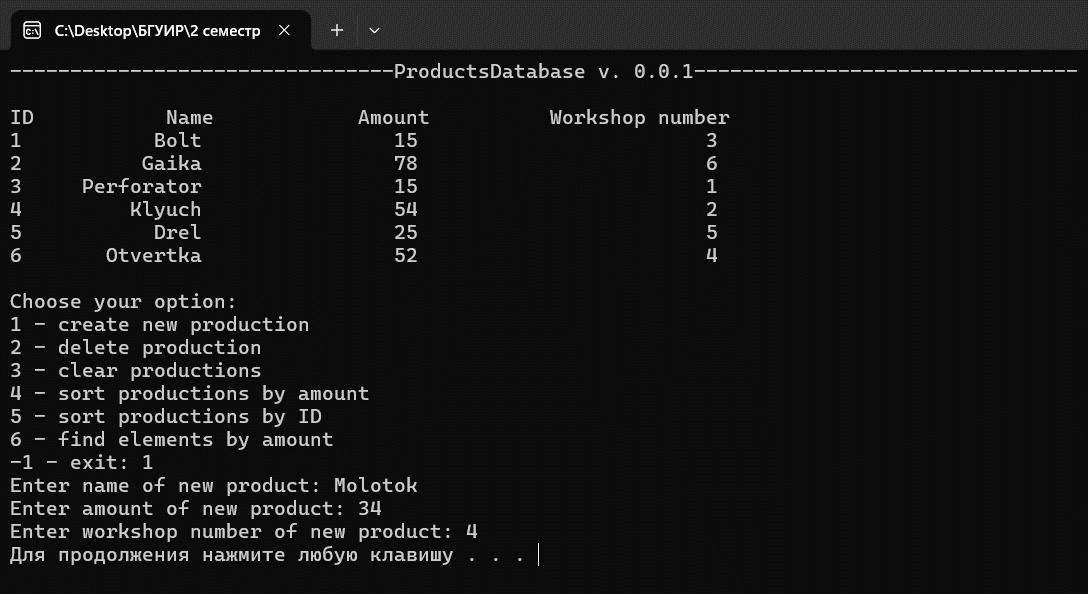


Рисунок 5.2 – Пункт “Создать новую продукцию”

При выборе пункта “Удалить продукцию”, пользователя попросят ввести идентификатор продукции, которую он хочет удалить. После этого продукция будет удалена, а идентификаторы остальной продукции будут обновлены. Список продукции обновится автоматически. Если список продукции пустой, то пользователю выведется сообщение об этом. По завершении пользователь вернётся обратно в главное меню (см. рисунок 5.3).

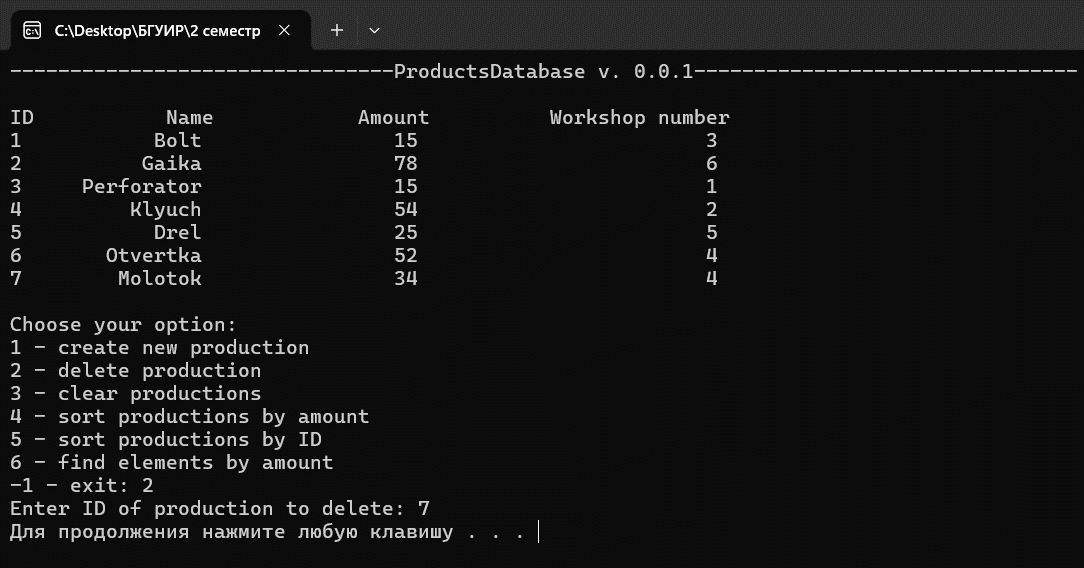


Рисунок 5.3 – Пункт “Удалить продукцию”

Если пользователь выбирает пункт “Очистить список продукции”, то весь список продукции станет пустым. Если список продукции был и так пустым, то пользователю выведется сообщение об этом. После очищения списка продукции пользователя вернёт обратно в главное меню (см. рисунок 5.4).

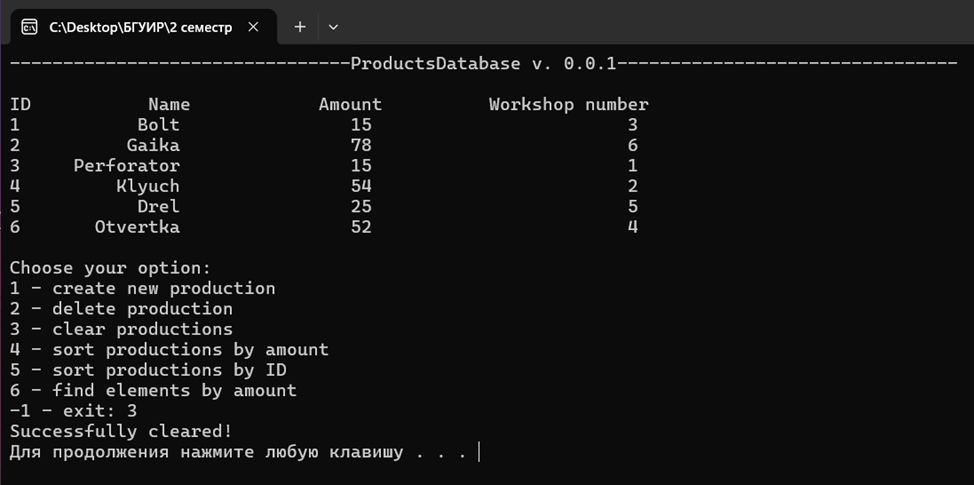


Рисунок 5.4 – Пункт “Очистить список продукции”

При выборе пункта “Сортировка списка продукции по количеству” ему будет предложено выбрать способ сортировки, а также её тип (по возрастанию, по убыванию). Всего существует 3 способа сортировки, а именно:

* сортировка пузырьком;
* сортировка прямым выбором;
* быстрая сортировка.

После выбора способа и типа сортировки будет произведена сортировка списка продукции по количеству в зависимости от выбранных параметров, а пользователя вернёт обратно в главное меню (см. рисунки 5.5 и 5.6).

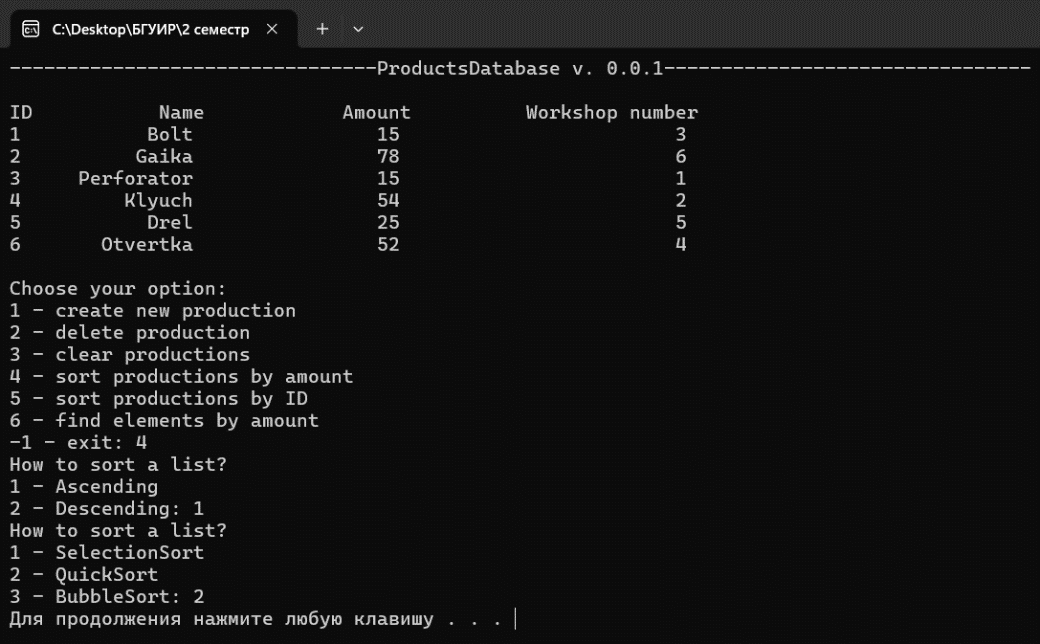


Рисунок 5.5 – Пункт “Сортировка списка продукции по количеству”

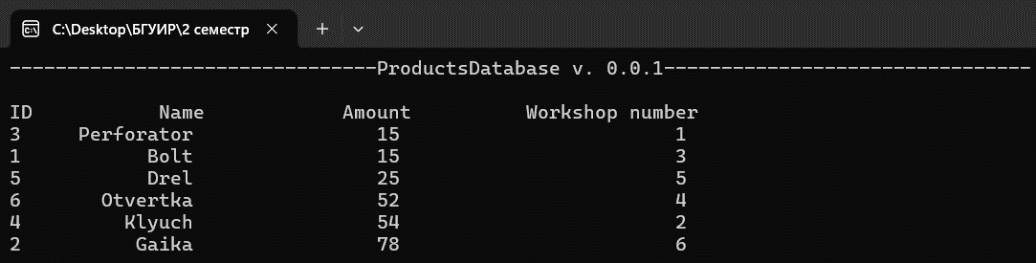


Рисунок 5.6 – Отсортированный список по количеству

При выборе пункта “Сортировка списка продукции по идентификатору”, то ему будет предложено выбрать способ сортировки, а также её тип (см. пункт “ Сортировка списка продукции по количеству”).

После выбора способа и типа сортировки будет произведена сортировка списка продукции по идентификатору в зависимости от выбранных параметров, а пользователя вернёт обратно в главное меню (см. рисунки 5.7 и 5.8).

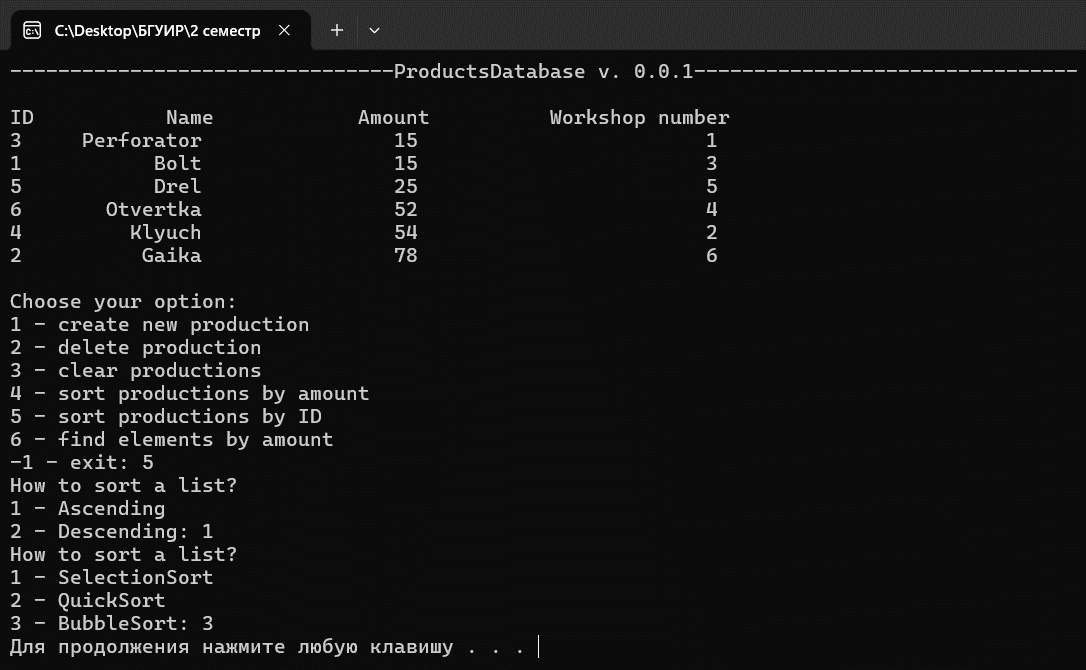


Рисунок 5.7 – Пункт “Сортировка списка продукции по идентификатору”

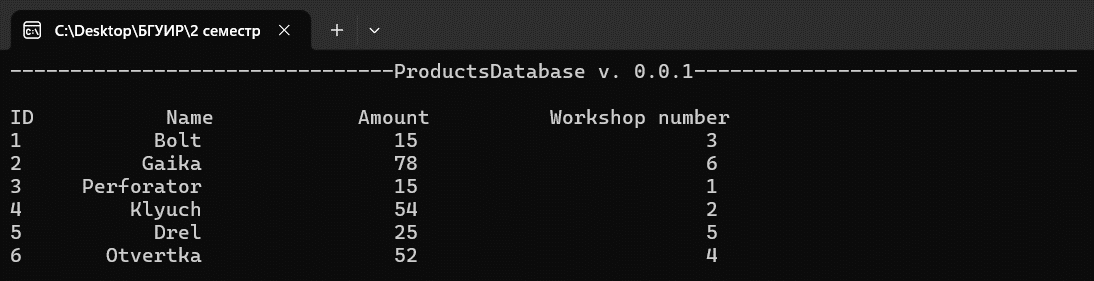


Рисунок 5.8 – Отсортированный список по идентификатору

При выборе пункта “Поиск элемента списка продукции по количеству”, пользователь должен ввести значение количества, по которому он хочет найти продукцию. Затем пользователю будет предложено выбрать один из следующих методов поиска:

* линейный поиск;
* двоичный поиск (список будет автоматически отсортирован по возрастанию, о чём будет выведено соответствующее сообщение);
* интерполяционный поиск (список будет автоматически отсортирован по возрастанию, о чём будет выведено соответствующее сообщение).

По окончании поиска на экран будет выведена вся продукция, которая была найдена по запросу пользователя. Если таковая продукция не была найдена, то будет выведено соответствующее сообщение. После этого пользователя перенесёт обратно в главное меню (см. рисунки 5.9 и 5.10).

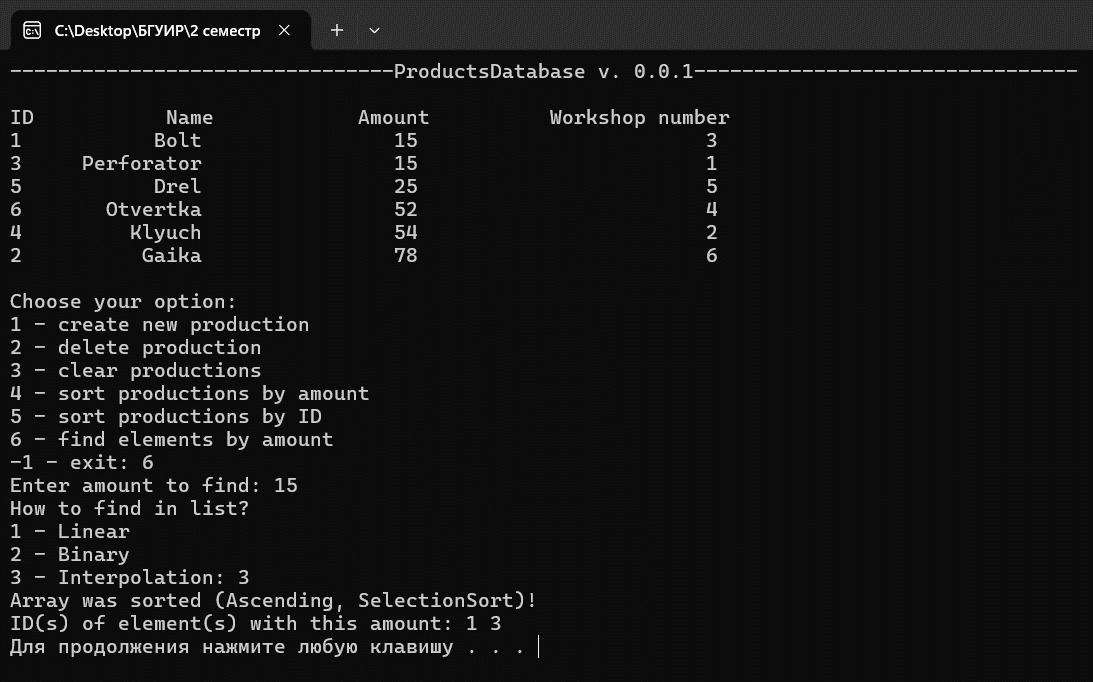


Рисунок 5.9 – Пункт “Поиск элемента списка продукции по количеству” (продукция была найдена)

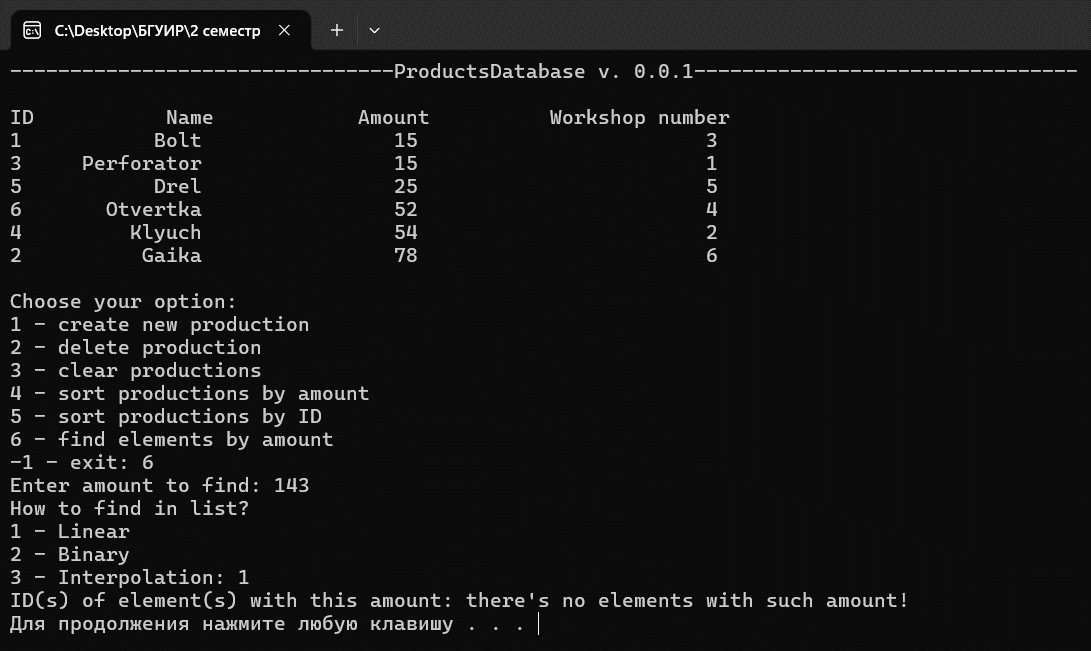


Рисунок 5.10 – Пункт “Поиск элемента списка продукции по количеству” (продукция не была найдена)

Пункт “Выход из программы” завершает работу программы.

Заключение

В итоге, создание программы потребовало выполнения ряда задач:

* интерпретировать требования задания для их реализации в коде;
* разработать модель программы с учетом необходимого функционала;
* написать код, реализующий все функции;
* отладить и протестировать работу программы.

Требования к проекту были тщательно изучены и проанализированы. На их основе был составлен перечень функциональных задач, которые необходимо реализовать. Далее была создана общая модель программы, описывающая её принцип работы и структуру кода. После этого началась работа над написанием кода на языке программирования C++.

При создании перечня необходимого функционала учитывались все требования задания. Эти требования были интерпретированы и адаптированы для их реализации в рамках программирования, затем они были разбиты на отдельные функции или блоки кода. После составления списка функциональных требований, была продумана их взаимосвязь и интерфейс работы программы. Код программы разрабатывался итерационно, что подразумевало создание работающей версии программы с минимальным функционалом и последующее поэтапное добавление новых функций. Каждый этап сопровождался тестированием корректности работы программы. Этот подход позволил проводить написание и отладку кода одновременно.

В результате выполнения курсового проекта и решения поставленных задач была создана программа на языке C++ для управления списком продукции в цехах завода. Программа включает функционал для просмотра, сортировки и редактирования информации о произведенных изделиях, что существенно упрощает и ускоряет работу с данными о производстве на предприятии.

Список использованных источников

[1] C++ | Структуры [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/5.8.php. – Дата доступа: 13.03.2024.

[2] Павловская, Т.А. C/C++. Программирование на языке высокого уровня: Учебник для вузов / Т.А. Павловская. – Санкт-Петербург : Питер, 2012. – 461 с.

[3] Что такое файл простыми словами и для чего он нужен [Электронный ресурс]. – Режим доступа: https://nardar.ru/articles/chto-takoe-fayl-prostymi-slovami-i-dlya-chego-on-nuzhen. – Дата доступа: 15.03.2024.

[4] 7 способов сортировки массивов на примере С++ с иллюстрациями [Электронный ресурс]. – Режим доступа: https://proglib.io/p/7-sposobov-sortirovki-massivov-na-primere-s-s-illyustraciyami-2022-04-20. – Дата доступа: 20.03.2024.

[5] Алгоритмы поиска в С++ [Электронный ресурс]. – Режим доступа: https://purecodecpp.com/algoritmy-v-c. – Дата доступа: 20.03.2024.

Приложение А  
(обязательное)  
Листинг кода

Файл “production.h”:

#pragma once

struct production

{

std::string name;

unsigned amount;

unsigned workshop\_number;

unsigned id;

production(const std::string& name, unsigned amount, unsigned workshop\_number, unsigned id);

};

void create\_new\_production();

void set\_new\_ids(unsigned id\_to\_delete);

void print\_productions();

void delete\_production();

bool is\_empty();

void write\_to\_file();

void read\_from\_file();

void clear();

void sort\_by\_amount();

void bubble\_sort\_by\_amount(unsigned type);

void selection\_sort\_by\_amount(unsigned type);

void quick\_sort\_by\_amount(unsigned low, unsigned high, unsigned type);

void bubble\_sort\_by\_id(unsigned type);

void sort\_by\_id();

void selection\_sort\_by\_id(unsigned type);

void quick\_sort\_by\_id(unsigned low, unsigned high, unsigned type);

void find();

void find\_linear(unsigned amount\_to\_find);

void find\_binary(unsigned amount\_to\_find);

void find\_interpolation(unsigned amount\_to\_find);

Файл “checked\_input.h”:

**Продолжение приложения А**

#pragma once

#include <iostream>

inline void checked\_int\_input(int& input);

inline void checked\_unsigned\_input(unsigned& input);

inline void clear\_input();

inline void clear\_input()

{

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

}

inline void checked\_int\_input(int& input)

{

while (!(std::cin >> input))

{

clear\_input();

std::cout << "Enter valid input: ";

}

}

inline void checked\_unsigned\_input(unsigned& input)

{

int to\_unsigned;

checked\_int\_input(to\_unsigned);

if (to\_unsigned < 0)

to\_unsigned \*= -1;

input = static\_cast<unsigned>(to\_unsigned);

}

Файл “production.cpp”:

#include <fstream>

#include <iomanip>

#include <string>

**Продолжение приложения А**

#include <vector>

#include "checked\_input.h"

#include "production.h"

std::vector<production> productions;

production::production(const std::string& name, const unsigned amount, const unsigned workshop\_number, const unsigned id)

{

this->name = name;

this->amount = amount;

this->workshop\_number = workshop\_number;

this->id = id;

}

bool is\_empty()

{

return static\_cast<int>(productions.size()) == 0;

}

void create\_new\_production()

{

std::string name;

unsigned amount, workshop\_number;

std::cout << "Enter name of new product: ";

std::getline(std::cin, name, '\n');

std::cout << "Enter amount of new product: ";

checked\_unsigned\_input(amount);

std::cout << "Enter workshop number of new product: ";

checked\_unsigned\_input(workshop\_number);

const unsigned id = static\_cast<unsigned>(productions.size()) + 1;

productions.emplace\_back(name, amount, workshop\_number, id);

clear\_input();

**Продолжение приложения А**

}

void set\_new\_ids(const unsigned id\_to\_delete)

{

for (production& production : productions)

if (production.id > id\_to\_delete)

production.id--;

}

void print\_productions()

{

std::cout << "--------------------------------ProductsDatabase v. 0.0.1--------------------------------\n\n";

if (is\_empty())

{

std::cout << "Your products list is empty!\n";

return;

}

std::cout << "ID" << std::setw(15) << "Name" << std::setw(18) << "Amount" << std::setw(25) << "Workshop number" << '\n';

for (const production& production : productions)

std::cout << production.id << std::setw(15) << production.name << std::setw(18) << production.amount << std::setw(25) << production.workshop\_number << '\n';

}

void write\_to\_file()

{

std::fstream file;

file.open("database.txt", std::ios::out);

file << productions.size() << '\n';

for (auto it = productions.cbegin(); it != productions.cend(); ++it)

file << it->id << '\*' << it->name << '\*' << it->amount << '\*' << it->workshop\_number << '\n';

**Продолжение приложения А**

file.close();

}

void read\_from\_file()

{

std::string buffer;

std::fstream file;

file.open("database.txt", std::ios::in);

std::getline(file, buffer);

const int productions\_size = std::stoi(buffer);

for (int i = 0; i < productions\_size; i++)

{

std::getline(file, buffer, '\*');

unsigned id = std::stoi(buffer);

std::getline(file, buffer, '\*');

std::string name = buffer;

std::getline(file, buffer, '\*');

unsigned amount = std::stoi(buffer);

std::getline(file, buffer, '\n');

unsigned workshop\_number = std::stoi(buffer);

productions.emplace\_back(name, amount, workshop\_number, id);

}

file.open("database.txt", std::ios::in);

}

void clear()

{

if (is\_empty())

{

std::cout << "Your products list is empty!\n";

return;

}

**Продолжение приложения А**

productions.clear();

std::cout << "Successfully cleared!\n";

}

void delete\_production()

{

if (is\_empty())

{

std::cout << "Your products list is empty!\n";

return;

}

unsigned id\_to\_delete;

std::cout << "Enter ID of production to delete: ";

checked\_unsigned\_input(id\_to\_delete);

if (id\_to\_delete > static\_cast<unsigned>(productions.size()))

{

std::cout << "There's no element with this ID!\n";

return;

}

for (auto it = productions.cbegin(); it != productions.cend(); ++it)

{

if (it->id == id\_to\_delete)

{

productions.erase(it);

set\_new\_ids(id\_to\_delete);

break;

}

}

}

void bubble\_sort\_by\_amount(const unsigned type)

{

for (int i = 0; i < static\_cast<int>(productions.size()); ++i)

{

for (int j = 0; j < static\_cast<int>(productions.size()) - i - 1; j++)

**Продолжение приложения А**

{

if (type == 1)

{

if (productions[j].amount > productions[j + 1].amount)

{

const production temp = productions[j];

productions[j] = productions[j + 1];

productions[j + 1] = temp;

}

}

else

{

if (productions[j].amount < productions[j + 1].amount)

{

const production temp = productions[j];

productions[j] = productions[j + 1];

productions[j + 1] = temp;

}

}

}

}

}

void selection\_sort\_by\_amount(const unsigned type)

{

for (int i = 0; i < static\_cast<int>(productions.size()) - 1; i++)

{

int m = i;

for (int j = i + 1; j < static\_cast<int>(productions.size()); j++)

{

if (type == 1)

{

if (productions[j].amount < productions[m].amount) m = j;

**Продолжение приложения А**

}

else

{

if (productions[j].amount > productions[m].amount) m = j;

}

}

const production temp = productions[m];

productions[m] = productions[i];

productions[i] = temp;

}

}

void quick\_sort\_by\_amount(const unsigned low, const unsigned high, const unsigned type)

{

if (low < high)

{

const unsigned x = productions[(low + high) / 2].amount;

unsigned i = low, j = high;

while (i <= j)

{

if (type == 1)

{

while (productions[i].amount < x) i++;

while (productions[j].amount > x) j--;

}

else

{

while (productions[i].amount > x) i++;

while (productions[j].amount < x) j--;

}

if (i <= j)

{

const production temp = productions[i];

productions[i] = productions[j];

**Продолжение приложения А**

productions[j] = temp;

i++;

j--;

}

}

if (low < j) quick\_sort\_by\_amount(low, j, type);

if (i < high) quick\_sort\_by\_amount(i, high, type);

}

}

void sort\_by\_amount()

{

if (is\_empty())

{

std::cout << "Your products list is empty!\n";

return;

}

unsigned sort\_parameter1 = 0, sort\_parameter2 = 0;

std::cout << "How to sort a list?\n1 - Ascending\n2 - Descending: ";

while (sort\_parameter1 != 1 && sort\_parameter1 != 2)

{

checked\_unsigned\_input(sort\_parameter1);

if (sort\_parameter1 != 1 && sort\_parameter1 != 2)

std::cout << "Enter valid input: ";

}

std::cout << "How to sort a list?\n1 - SelectionSort\n2 - QuickSort\n3 - BubbleSort: ";

while (sort\_parameter2 < 1 || sort\_parameter2 > 3)

{

checked\_unsigned\_input(sort\_parameter2);

if (sort\_parameter2 < 1 || sort\_parameter2 > 3)

std::cout << "Enter valid input: ";

}

switch(sort\_parameter2)

{

case 1:

selection\_sort\_by\_amount(sort\_parameter1);

**Продолжение приложения А**

break;

case 2:

quick\_sort\_by\_amount(0, static\_cast<unsigned>(productions.size()) - 1, sort\_parameter1);

break;

case 3:

bubble\_sort\_by\_amount(sort\_parameter1);

break;

default:

break;

}

}

void bubble\_sort\_by\_id(const unsigned type)

{

for (int i = 0; i < static\_cast<int>(productions.size()); ++i)

{

for (int j = 0; j < static\_cast<int>(productions.size()) - i - 1; j++)

{

if (type == 1)

{

if (productions[j].id > productions[j + 1].id)

{

const production temp = productions[j];

productions[j] = productions[j + 1];

productions[j + 1] = temp;

}

}

else

{

if (productions[j].id < productions[j + 1].id)

{

const production temp = productions[j];

productions[j] = productions[j + 1];

productions[j + 1] = temp;

**Продолжение приложения А**

}

}

}

}

}

void selection\_sort\_by\_id(const unsigned type)

{

for (int i = 0; i < static\_cast<int>(productions.size()) - 1; i++)

{

int m = i;

for (int j = i + 1; j < static\_cast<int>(productions.size()); j++)

{

if (type == 1)

{

if (productions[j].id < productions[m].id) m = j;

}

else

{

if (productions[j].id > productions[m].id) m = j;

}

}

const production temp = productions[m];

productions[m] = productions[i];

productions[i] = temp;

}

}

void quick\_sort\_by\_id(const unsigned low, const unsigned high, const unsigned type)

{

if (low < high)

{

const unsigned x = productions[(low + high) / 2].id;

**Продолжение приложения А**

unsigned i = low, j = high;

while (i <= j)

{

if (type == 1)

{

while (productions[i].id < x) i++;

while (productions[j].id > x) j--;

}

else

{

while (productions[i].id > x) i++;

while (productions[j].id < x) j--;

}

if (i <= j)

{

const production temp = productions[i];

productions[i] = productions[j];

productions[j] = temp;

i++;

j--;

}

}

if (low < j) quick\_sort\_by\_id(low, j, type);

if (i < high) quick\_sort\_by\_id(i, high, type);

}

}

void sort\_by\_id()

{

if (is\_empty())

{

std::cout << "Your products list is empty!\n";

return;

}

unsigned sort\_parameter1 = 0, sort\_parameter2 = 0;

std::cout << "How to sort a list?\n1 - Ascending\n2 - Descending: ";

**Продолжение приложения А**

while (sort\_parameter1 != 1 && sort\_parameter1 != 2)

{

checked\_unsigned\_input(sort\_parameter1);

if (sort\_parameter1 != 1 && sort\_parameter1 != 2)

std::cout << "Enter valid input: ";

}

std::cout << "How to sort a list?\n1 - SelectionSort\n2 - QuickSort\n3 - BubbleSort: ";

while (sort\_parameter2 < 1 || sort\_parameter2 > 3)

{

checked\_unsigned\_input(sort\_parameter2);

if (sort\_parameter2 < 1 || sort\_parameter2 > 3)

std::cout << "Enter valid input: ";

}

switch (sort\_parameter2)

{

case 1:

selection\_sort\_by\_id(sort\_parameter1);

break;

case 2:

quick\_sort\_by\_id(0, static\_cast<unsigned>(productions.size()) - 1, sort\_parameter1);

break;

case 3:

bubble\_sort\_by\_id(sort\_parameter1);

break;

default:

break;

}

}

void find\_linear(const unsigned amount\_to\_find)

{

int code = 0;

for (const production& production : productions)

{

if (production.amount == amount\_to\_find)

{

code = 1;

std::cout << production.id << ' ';

**Продолжение приложения А**

}

}

if (code == 0)

std::cout << "there's no elements with such amount!\n";

else

std::cout << '\n';

}

void find\_binary(const unsigned amount\_to\_find)

{

selection\_sort\_by\_amount(1);

int i\_key = 0, j = static\_cast<int>(productions.size()) - 1;

while (i\_key < j)

{

const int m = (i\_key + j) / 2;

if (productions[m].amount < amount\_to\_find) i\_key = m + 1;

else j = m;

}

if (productions[i\_key].amount != amount\_to\_find)

{

std::cout << "there's no elements with such amount!\n";

return;

}

std::cout << productions[i\_key].id << ' ';

for (int i = i\_key + 1; i < static\_cast<int>(productions.size()); i++)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

**Продолжение приложения А**

}

for (int i = i\_key - 1; i >= 0; i--)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

std::cout << '\n';

}

void find\_interpolation(const unsigned amount\_to\_find)

{

selection\_sort\_by\_amount(1);

int low = 0;

int high = static\_cast<int>(productions.size()) - 1;

while (low <= high && amount\_to\_find >= productions[low].amount && amount\_to\_find <= productions[high].amount)

{

if (low == high)

{

if (productions[low].amount == amount\_to\_find)

{

std::cout << productions[low].id << ' ';

for (int i = low + 1; i < static\_cast<int>(productions.size()); i++)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

for (int i = low - 1; i >= 0; i--)

**Продолжение приложения А**

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

std::cout << '\n';

return;

}

std::cout << "there's no elements with such amount!\n";

return;

}

const int pos = low + (high - low) / static\_cast<int>((productions[high].amount - productions[low].amount)) \* static\_cast<int>((amount\_to\_find - productions[low].amount));

if (productions[pos].amount == amount\_to\_find)

{

std::cout << productions[pos].id << ' ';

for (int i = pos + 1; i < static\_cast<int>(productions.size()); i++)

{

if (productions[i].amount != amount\_to\_find)

break;

std::cout << productions[i].id << ' ';

}

for (int i = pos - 1; i >= 0; i--)

{

if (productions[i].amount != amount\_to\_find)

break;

**Продолжение приложения А**

std::cout << productions[i].id << ' ';

}

std::cout << '\n';

return;

}

if (productions[pos].amount < amount\_to\_find)

low = pos + 1;

else

high = pos - 1;

}

std::cout << "there's no elements with such amount!\n";

}

void find()

{

if (is\_empty())

{

std::cout << "Your products list is empty!\n";

return;

}

std::cout << "Enter amount to find: ";

unsigned amount\_to\_find;

checked\_unsigned\_input(amount\_to\_find);

int find\_parameter = 0;

std::cout << "How to find in list?\n1 - Linear\n2 - Binary\n3 - Interpolation: ";

while (find\_parameter < 1 || find\_parameter > 3)

{

checked\_int\_input(find\_parameter);

if (find\_parameter < 1 || find\_parameter > 3)

std::cout << "Enter valid input: ";

}

if(find\_parameter != 1)

std::cout << "Array was sorted (Ascending, SelectionSort)!\n";

**Продолжение приложения А**

std::cout << "ID(s) of element(s) with this amount: ";

switch (find\_parameter)

{

case 1:

find\_linear(amount\_to\_find);

break;

case 2:

find\_binary(amount\_to\_find);

break;

case 3:

find\_interpolation(amount\_to\_find);

break;

default:

break;

}

}

Файл “menu.h”:

#pragma once

#include <iostream>

#include "production.h"

#include "checked\_input.h"

inline void end\_of\_function()

{

system("pause");

system("cls");

}

inline void menu()

{

int choice = 0;

read\_from\_file();

do

{

print\_productions();

choice = 0;

**Продолжение приложения А**

std::cout << "\nChoose your option:\n1 - create new production\n2 - delete production\n3 - clear productions\n4 - sort productions by amount\n5 - sort productions by ID\n6 - find elements by amount\n-1 - exit: ";

while ((choice < 1 || choice > 6) && choice != -1)

{

checked\_int\_input(choice);

if ((choice < 1 || choice > 6) && choice != -1)

std::cout << "Enter valid input: ";

}

switch (choice)

{

case 1:

clear\_input();

create\_new\_production();

end\_of\_function();

break;

case 2:

clear\_input();

delete\_production();

end\_of\_function();

break;

case 3:

clear();

end\_of\_function();

break;

case 4:

sort\_by\_amount();

end\_of\_function();

break;

case 5:

sort\_by\_id();

end\_of\_function();

break;

case 6:

find();

end\_of\_function();

break;

default:

std::cout << "Have a nice day!";

**Продолжение приложения А**

break;

}

write\_to\_file();

} while (choice != -1);

}

Файл “ConsoleApplication1.cpp”:

#include "menu.h"

int main()

{

menu();

return 0;

}

Приложение Б  
(обязательное)  
Блок-схема работы программы

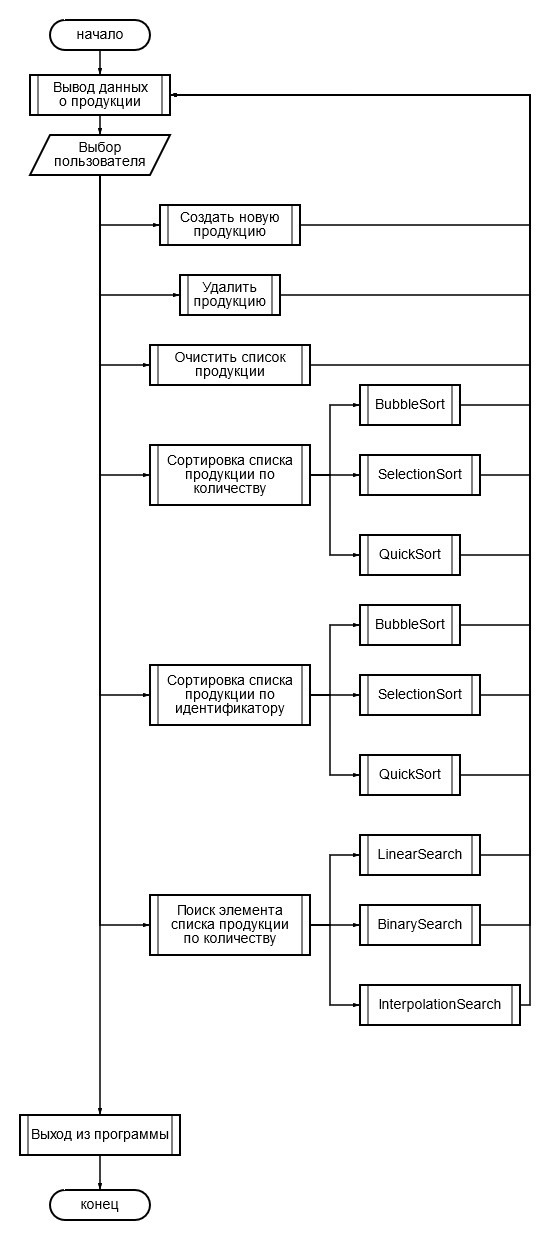


Рисунок Б.1 – Блок-схема работы программы

**ВЕДОМОСТЬ ДОКУМЕНТОВ**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Обозначение* | | | | | *Наименование* | | | | *Дополнитель-ные сведения* | |
|  | | | | | *Текстовые документы* | | | |  | |
|  | | | | |  | | | |  | |
| *БГУИР КП 1-40 05 01-12 102ПЗ* | | | | | *Пояснительная записка* | | | | *57 с.* | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  | | | | |  | | | |  | |
|  |  |  |  |  | ***БГУИР КП 1-40 05 01 12 102 ВД*** | | | | | |
|  |  |  |  |  |
|  |  |  |  |  |
| *Изм.* | *Л.* | *№ докум.* | *Подп.* | *Дата* | *Структура списка продукции в цехах завода* | *Лит* | | | *Лист* | *Листов* |
| *Разраб.* | | *Цевелюк* |  |  |  | *У* |  | *57* | *57* |
| *Пров.* | | *Новицкая* |  |  | *Кафедра ВМиП*  *гр. 324402* | | | | |
|  | |  |  |  |
|  | |  |  |  |
|  | |  |  |  |