Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Инженерно-экономический факультет

Кафедра экономической информатики

Н. А. Кириенко, Ю. Ю. Петрович, А. А. Ефремов

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

В ДВУХ ЧАСТЯХ
ЧАСТЬ 2

РЕАЛИЗАЦИЯ АЛГОРИТМОВ НА ЯЗЫКЕ СИ В ИНТЕГРИРОВАННОЙ СРЕДЕ РАЗРАБОТКИ VISUAL STUDIO

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальностей 6-05-0611-01 «Информационные системы и технологии», 7-06-0611-04 «Электронная экономика», 6-05-0611-07 «Цифровой маркетинг»

Минск БГУИР 2026

УДК [004.021+004.42](076.5)
ББК 32.973.2я73
К43

Р е ц е н з е н т ы:

кафедра информационных технологий в экологии и медицине
учреждения образования
«Международный государственный экологический
институт имени А. Д. Сахарова»
Белорусского государственного университета
(протокол № 5 от 26.12.2024);

заведующий лабораторией синтеза технических систем
государственного научного учреждения
«Объединенный институт проблем информатики
Национальной академии наук Беларуси»
доктор технических наук С. В. Медведев

Кириенко, Н. А.

К43 Основы алгоритмизации и программирования. Лабораторный практикум. В 2 ч. Ч. 2 : Реализация алгоритмов на языке Си в интегрированной среде разработки Visual Studio : пособие / Н. А. Кириенко, Ю. Ю. Петрович, А. А. Ефремов. – Минск : БГУИР, 2026. – 110 с. : ил.
ISBN 978-985-543-829-9 (ч. 2).

Содержит описания лабораторных работ, выполняемых студентами, осваивающими дисциплину «Основы алгоритмизации и программирования». Представлены лабораторные работы по темам части 2 пособия «Реализация алгоритмов на языке Си в интегрированной среде разработки Visual Studio». В каждой работе представлен краткий теоретический материал, подробная инструкция по разработке приложения, варианты индивидуальных заданий.

Первая часть была издана в 2024 г.

УДК [004.021+004.42](076.5)
ББК 32.973.2я73

ISBN 978-985-543-829-9 (ч. 2)
ISBN 978-985-543-735-3

Содержание

Введение

Дисциплина «Основы алгоритмизации и программирования» является основной базовой частью цикла дисциплин по информационным технологиям, изучаемых студентами специальностей общего высшего образования 6-05-0611-01 «Информационные системы и технологии», 7-06-0611-04 «Электронная экономика», 6-05-0611-07 «Цифровой маркетинг» на протяжении всего курса обучения в БГУИР.

Дисциплина состоит из двух частей и изучается в течение двух семестров. Данное пособие является логическим продолжением части 1 пособия [1], которое обеспечивает методическими материалами первую часть дисциплины «Использование языка Си в интегрированной среде разработки Visual Studio». Настоящее пособие раскрывает темы, изучаемые во втором семестре курса «Основы алгоритмизации и программирования», и называется «Реализация алгоритмов на языке Си в интегрированной среде разработки Visual Studio».

Целью преподавания данной дисциплины является формирование у студентов основных понятий и навыков использования алгоритмов работы со сложными структурами данных и создания программных комплексов на языке программирования Си в операционной среде Windows на базе изученных алгоритмов.

Задачей изучения части 2 пособия «Реализация алгоритмов на языке Си в интегрированной среде разработки Visual Studio» является знакомство с основными алгоритмами работы со сложными структурами данных: сортировка и поиск данных, использование бинарных деревьев, графов, механизм хеширования.

Особенностью изучения дисциплины для студентов данных направлений специальности является рассмотрение тем и задач, наиболее тесно связанных с экономическими, экономико-математическими, логистичекими проблемами.

В процессе изучения дисциплины студенты прослушивают курс лекций и выполняют серию лабораторных работ. В пособии представлено восемь лабораторных работ, даны подробные инструкции по их выполнению. Темы лабораторных работ охватывают наиболее важные разделы изучаемой дисциплины. В каждой работе приведен список индивидуальных заданий.

Лабораторная работа № 1.
Область видимости, время жизни переменных и классы памяти

Цель работы – изучить область видимости и время жизни переменных с использованием новых ключевых слов. Научиться выполнять разделение программы на разные файлы и устанавливать коммуникацию между ними.

Ход выполнения лабораторной работы должен быть отражен в отчете. Отчет должен содержать титульный лист, номер задания, коды программ, скриншоты с результатами выполнения программы.

Методические указания

Структура файла проекта и объявление переменных. Подробная информация по теме представлена в источнике [2]. Напомним, что Си-программа представляет собой определение функции main, которая для выполнения необходимых действий вызывает другие функции. Компилятор языка Си позволяет разбить программу на несколько отдельных частей (исходных файлов), оттранслировать каждую часть отдельно и затем объединить все части в один выполняемый файл при помощи редактора связей [2]. При такой структуре исходной программы функции, находящиеся в разных исходных файлах, могут использовать глобальные переменные. Все функции в языке Си по определению глобальные объекты и доступны из любых файлов. Например, если проект состоит из двух исходных файлов, как показано на рисунке 1.1, то функция main может вызывать любую из трех функций func1, func2, func3, а каждая из этих функций может вызывать любую другую. Для этого в каждом файле должны быть определены прототипы этих функций [2].

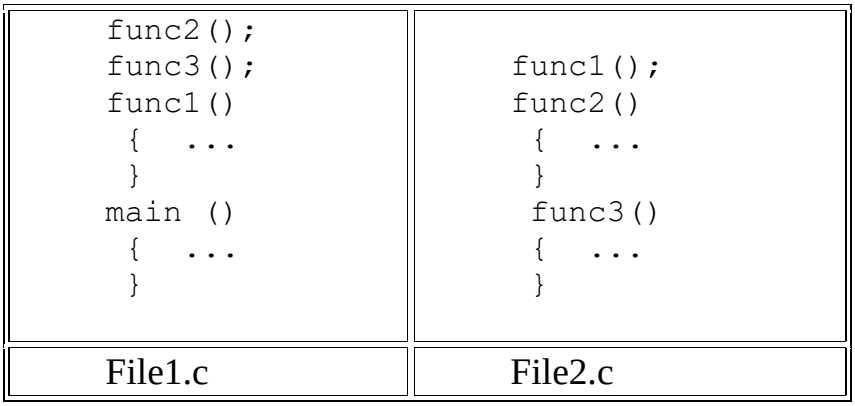


Рисунок 1.1 – Проект из двух исходных файлов

Для того чтобы функция могла выполнять какие-либо действия, она должна использовать переменные. В языке Си все переменные должны быть объявлены до их использования. Объявления устанавливают соответствие имени и атрибутов переменной, функции или типа. Определение переменной вы-

зывает выделение памяти для хранения ее значения. Класс выделяемой памяти определяется спецификатором класса памяти и определяет время жизни и область видимости переменной, связанные с понятием блока программы. В языке Си составным оператором считается последовательность операторов, заключенная в фигурные скобки. Блоком считается такой составной оператор, в котором присутствуют объявления переменных [2].

Определение функции состоит из тела функции, которое является блоком, и предшествующего телу заголовка функции (в который входят имя функции, типы возвращаемого значения и формальных параметров). Блоки могут включать в себя составные операторы, но не определения функций. Внутренний блок называется вложенным, а внешний блок – объемлющим.

Время жизни – это интервал времени выполнения программы, в течение которого программный объект (переменная или функция) существует. Время жизни переменной может быть локальным или глобальным. Переменная с глобальным временем жизни имеет выделенную для нее память и определенное значение на протяжении всего времени выполнения программы, начиная с момента объявления этой переменной. Переменная с локальным временем жизни имеет выделенную для нее память и определенное значение только во время выполнения блока, в котором эта переменная определена или объявлена. При каждом входе в блок для локальной переменной выделяется новая память, которая освобождается при выходе из блока [2].

Область видимости – это часть текста программы, в которой может быть использован данный объект. Объект считается видимым в блоке или в исходном файле, если в этом блоке или файле известны имя и тип объекта. Объект может быть видимым в пределах блока, исходного файла или во всех исходных файлах, образующих программу. Это зависит от того, на каком уровне объявлен объект: на внутреннем, т. е. внутри некоторого блока, или на внешнем, т. е. вне всех блоков. Если объект объявлен внутри блока, то он видим в этом блоке и во всех внутренних блоках. Если объект объявлен на внешнем уровне, то он видим от точки его объявления до конца данного исходного файла [2].

Спецификатор класса памяти в объявлении переменной может быть auto, register, static или extern. Если класс памяти не указан, то он определяется по умолчанию из контекста объявления. Объекты классов auto и register имеют локальное время жизни. Спецификаторы static и extern определяют объекты с глобальным временем жизни. При объявлении переменной на внутреннем уровне может быть использован любой из четырех спецификаторов класса памяти, а если он не указан, то подразумевается класс памяти auto. Переменная класса памяти auto имеет локальное время жизни и видна только в блоке, в котором объявлена. Память для такой переменной выделяется при входе в блок и освобождается при выходе из блока. При повторном входе в блок этой переменной может быть выделен другой участок памяти. Переменная класса памяти auto автоматически не инициализируется. Она должна быть проинициализирована явно при объявлении путем присвоения ей начального значения. Значение неинициализированной переменной с классом памяти auto считается неопределенным [2].

Спецификатор класса памяти register предписывает компилятору выделить память для переменной в регистре, если это представляется возможным. Использование регистровой памяти обычно приводит к сокращению времени доступа к переменной. Переменная, объявленная с классом памяти register, имеет ту же область видимости, что и переменная auto. Число регистров, которые можно использовать для значений переменных, ограничено возможностями компьютера, и в случае, если компилятор не имеет в распоряжении свободных регистров, то переменной выделяется память, как для класса auto. Класс памяти register может быть указан только для переменных с типом int или указателей с размером, равным размеру int [2].

Объявление переменной на внутреннем уровне со спецификатором класса памяти static обеспечивает возможность сохранить ее значение при выходе из блока и использовать его при повторном входе в блок. Такая переменная имеет глобальное время жизни и область видимости внутри блока, в котором она объявлена. В отличие от переменных класса auto, память для которых выделяется в стеке, для переменных класса static память выделяется в сегменте данных, и поэтому их значение сохраняется при выходе из блока [2].

Пример 1.1. Объявление переменной i на внутреннем уровне с классом памяти static.

```
//Исходный файл file1.c
main()
{...}
fun1()
{static int i=0; ...   }
//Исходный файл file2.c
fun2() {  static int i=0; ...   }
fun3() {  static int i=0; ...   }
```

В приведенном примере объявлены три разные переменные с классом памяти static, имеющие одинаковые имена i. Каждая из этих переменных имеет глобальное время жизни, но видима только в том блоке (функции), в котором она объявлена. Эти переменные можно использовать для подсчета числа обращений к каждой из трех функций.

Переменные класса памяти static могут быть инициализированы константным выражением. Если явной инициализации нет, то такой переменной присваивается нулевое значение. При инициализации константным адресным выражением можно использовать адреса любых внешних объектов, кроме адресов объектов с классом памяти auto, т. к. адрес последних не является константой и изменяется при каждом входе в блок. Инициализация выполняется один раз при первом входе в блок [2].

Переменная, объявленная локально с классом памяти extern, является ссылкой на переменную с тем же именем, определенную глобально в одном из исходных файлов программы. Цель такого объявления состоит в том, чтобы сделать определение переменной глобального уровня видимым внутри блока.

Пример 1.2 Объявление переменной i, являющейся именем внешнего массива длинных целых чисел, на локальном уровне.

```
//Исходный файл file1.c
main() { ...  }
fun1() {extern long i[]; ...  }
//Исходный файл file2.c
long i[MAX]={0};
fun2(){  ...   }
fun3(){  ...   }
```

Объявление переменной i[] как extern в приведенном примере делает ее видимой внутри функции fun1. Определение этой переменной находится в файле file2.c на глобальном уровне и должно быть только одно, в то время как объявлений с классом памяти extern может быть несколько.

Объявление с классом памяти extern требует при необходимости использовать переменную, описанную в текущем исходном файле, ниже по тексту программы, т. е. до выполнения ее глобального определения. Следующий пример иллюстрирует такое использование переменной с именем st.

Пример 1.3. Объявление переменной с классом памяти extern.

```
main() { extern int st[]; ...    }
static int st[MAX]={0};
fun1() {   ...   }
```

Объявление переменной со спецификатором extern информирует компилятор о том, что память для переменной выделять не требуется, т. к. это выполнено где-то в другом месте программы.

При объявлении переменных на глобальном уровне может быть использован спецификатор класса памяти static или extern, а также можно объявлять переменные без указания класса памяти. Классы памяти auto и register для глобального объявления недопустимы. Объявление переменных на глобальном уровне – это или определение переменных, или ссылки на определения, сделанные в другом месте программы [2]. Объявление глобальной переменной, которое инициализирует эту переменную (явно или неявно), является определением переменной. Определение на глобальном уровне может задаваться в следующих формах:

1. Переменная объявлена с классом памяти static. Такая переменная может быть инициализирована явно константным выражением или по умолчанию

нулевым значением. То есть обявления static int i = 0 и static int i эквивалентны, и в обоих случаях переменной i будет присвоено значение 0.

2. Переменная объявлена без указания класса памяти, но с явной инициализацией. Такой переменной по умолчанию присваивается класс памяти static. То есть объявления int i = 1 и static int i = 1 будут эквивалентны.

Переменная, объявленная глобально, видима в пределах остатка исходного файла, в котором она определена. Выше своего описания и в других исходных файлах эта переменная невидима (если только она не объявлена с классом extern). Глобальная переменная может быть определена только один раз в пределах своей области видимости. В другом исходном файле может быть объявлена другая глобальная переменная с таким же именем и с классом памяти static, конфликта при этом не возникает, т. к. каждая из этих переменных будет видимой только в своем исходном файле [2].

Спецификатор класса памяти extern для глобальных переменных используется в качестве ссылки на переменную, объявленную в другом месте программы, т. е. для расширения области видимости переменной. При таком объявлении область видимости переменной расширяется до конца исходного файла, в котором сделано объявление. В объявлениях с классом памяти extern не допускается инициализация, т. к. эти объявления ссылаются на уже существующие и определенные ранее переменные. Переменная, на которую делается ссылка с помощью спецификатора extern, может быть определена только один раз в одном из исходных файлов программы [2].

Объявление функций. Функции всегда определяются глобально. Они могут быть объявлены с классом памяти static или extern.

Правила определения области видимости для функций отличаются от правил видимости для переменных и состоят в следующем:

1. Функция, объявленная как static, видима в пределах того файла, в котором она определена. Каждая функция может вызвать другую функцию класса памяти static из своего исходного файла, но не может вызвать функцию, определенную с классом static в другом исходном файле. Разные функции класса памяти static, имеющие одинаковые имена, могут быть определены в разных исходных файлах, и это не ведет к конфликту.

2. Функция, объявленная с классом памяти extern, видима в пределах всех исходных файлов программы. Любая функция может вызывать функции класса памяти extern.

3. Если в объявлении функции отсутствует спецификатор класса памяти, то по умолчанию принимается класс extern. Все объекты класса памяти extern компилятор помещает в объектном файле в специальную таблицу внешних ссылок, которая используется редактором связей для разрешения внешних ссылок. Часть внешних ссылок порождается компилятором при обращениях к библиотечным функциям Си, поэтому для разрешения этих ссылок редактору связей должны быть доступны соответствующие библиотеки функций [2].

Приведем пример программы, текст которой размещен в двух файлах (для связи функций программы используется внешняя переменная i).

Пример 1.4. Использование внешней переменной i для связи функций программы.

```
// File1.c
#include <stdio.h>
extern int i;
void main(void)
{void next(void);
 i++;
 printf("B main i=%d\n",i);
 next();
}
int i=3; // Глобальная переменная
void next(void)
{void other(void);
 i++;
 printf("B next i=%d\n",i);
 other();
}
// File2.c
#include <stdio.h>
extern int i;
void other(void)
{ i++;
  printf("B other i=%d\n",i);
}
```

Результат:

```
B main i=4
B next i=5
B other i=6
```

Индивидуальные задания

В каждом задании предусмотреть следующие действия с массивом структурных переменных и оформить каждое из них в виде отдельной функции:

– ввод информации в массив структурных переменных из файла;

– вывод текущего состояния массива структурных переменных на экран;

– поиск элемента с заданным значением поля и вывод его на экран.

Все функции поместить в отдельные файлы, структурную переменную объявить как глобальную, счетчики циклов объявить как регистровые переменные, передачу значений осуществить через глобальные переменные, осуще-

ствить динамическое распределение памяти под массив структурных переменных. Выбор действия осуществить через меню с помощью оператора switch.

Варианты заданий:

1 Описать структуру, содержащую следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов). Дополнительное задание: вывести на дисплей фамилии и номера групп для всех студентов, имеющих оценки 8 и 9; если таких студентов нет, вывести соответствующее сообщение.

2 Описать структуру, содержащую следующие поля: название пункта назначения рейса; номер рейса; тип самолета. Дополнительное задание: вывести на экран номера рейсов и типы самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры; если таких рейсов нет, выести на дисплей соответствующее сообщение.

3 Описать структуру, содержащую следующие поля: фамилия и инициалы работника; название занимаемой должности; год поступления на работу. Дополнительное задание: вывести на дисплей фамилии работников, чей стаж работы в организации превышает значение, введенное с клавиатуры; если таких работников нет, вывести на дисплей соответствующее сообщение.

4 Описать структуру, содержащую следующие поля: название пункта назначения; номер поезда; время отправления. Дополнительное задание: вывести на экран информацию о поездах, направляющихся в пункт, название которого введено с клавиатуры; если таких поездов нет, вывести на дисплей соответствующее сообщение.

5 Описать структуру, содержащую следующие поля: название начального пункта маршрута; название конечного пункта маршрута; номер маршрута. Дополнительное задание: вывести на экран информацию о маршрутах, которые начинаются или кончаются в пункте, название которого введено с клавиатуры; если таких маршрутов нет, вывести на дисплей соответствующее сообщение.

6 Описать структуру, содержащую следующие поля: фамилия, имя; номер телефона; день рождения (массив из трех чисел). Дополнительное задание: вывести на экран информацию о человеке, номер телефона которого введен с клавиатуры; если такого нет, вывести на дисплей соответствующее сообщение.

7 Описать структуру, содержащую следующие поля: фамилия, имя; знак зодиака; день рождения (массив из трех чисел). Дополнительное задание: вывести на экран информацию о людях, родившихся под знаком, наименование которого введено с клавиатуры; если такого нет, вывести на дисплей соответствующее сообщение.

8 Разработать программу учета покупок в ювелирном магазине. Данные о покупках хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по стоимости ювелирного украшения.

9 Разработать программу учета жилищного фонда. Данные о жилищном фонде хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру жилищного договора.

10 Разработать программу учета стройматериалов. Данные о стройматериалах хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру договора.

11 Разработать программу учета посадок на участке в ботаническом саду. Данные об участках хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру участка.

12 Разработать программу расчета закупки сырья промышленного предприятия. Данные о закупках хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по типу сырья.

13 Разработать программу расчета прибыли от выполняемых работ по ремонту офиса многофилиального концерна. Данные о выполняемых работах хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по сумме выполненных работ.

14 Разработать программу расчета деталей, использованных при изготовлении какого-либо изделия. Данные о деталях хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по стоимости деталей, используемых в выбранном изделии.

15 Разработать программу расчета закупки сырья промышленного предприятия. Данные о закупках хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру накладной.

16 Разработать программу определения затрат рабочего времени на выполнение строительных работ. Данные о строительных работах хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру заказа.

17 Разработать программу определения пробега автомобиля на основе путевых листов. Данные о путевых листах хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру путевого листа.

18 Разработать программу определения величины таможенных сборов на базе контрактов коммерческой фирмы. Данные о таможенных сборах хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру контракта.

19 Разработать программу определения процента выхода годных изделий на основе актов приема ОТК. Данные о тестируемых партиях хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру заказа.

20 Разработать программу оценки экспорта фирмы. Данные об экспортных операциях хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру контракта.

21 Разработать программу оценки роста промышленного предприятия по данным за последние годы. Данные о финансовых отчетах хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по номеру финансового документа.

22 Разработать программу оценки продаж театральных билетов в зависимости от времени года. Данные о продажах хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по величине прибыли.

23 Разработать программу определения суммарной продажи проездных билетов за определенный месяц. Данные о продажах хранить в виде массива структур. Итоговая информация должна выводиться на экран в виде таблицы, данные в которой отсортированы по величине прибыли.

24 Разработать программу вывода упорядоченного по алфавиту списка студентов, предусмотрев ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

25 Разработать программу вывода упорядоченного по году рождения списка студентов, предусмотрев ввод исходной информации о пяти студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

26 Разработать программу вывода упорядоченного по году поступления списка студентов-отличников, предусмотрев ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

27 Разработать программу вывода анкетных данных студентов, сдавших сессию на 4 и 5, предусмотрев ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

28 Разработать программу вывода списка студентов, фамилии которых начинаются с буквы «Б», и их оценок по всем предметам. Предусмотреть ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

29 Разработать программу вывода анкетных данных отличников. Предусмотреть ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

30 Разработать программу вывода списка студентов, фамилии которых начинаются с буквы «А», и их дат рождения. Предусмотреть ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

31 Разработать программу вывода анкетных данных студентов, имеющих хотя бы одну оценку 3 за сессию. Предусмотреть ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

32 Разработать программу вывода списка студентов и их оценок, фамилии студентов начинаются с букв «В» и «Г». Предусмотреть ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

33 Разработать программу вывода фамилий и дат рождения студентов, не имеющих оценок 3. Предусмотреть ввод исходной информации о четырех студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

34 В программе вычислить общий средний балл всех студентов и вывести на экран список студентов со средними баллами выше общего среднего балла. Предусмотреть ввод исходной информации о пяти студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

35 Разработать программу вывода анкетных данных студентов, имеющих оценку 4 по физике и оценку 5 по высшей математике. Предусмотреть ввод исходной информации о пяти студентах: фамилия и инициалы; год рождения; год поступления в БГУИР; оценки за первый семестр по следующим предметам: физика, высшая математика, информатика.

Приложение 1

Программа состоит из двух файлов: f1.cpp и f2.cpp. Файл f1.cpp содержит описание главной функции, в которой реализовано меню. В меню происходит вызов основных функций, описание которых находится в файле f2.cpp. Кроме того, в файле f2.cpp объявлена структура [2].

```
// Файл f1.cpp
#include <iostream>
#include <cstdio>
#include <cstring>
#include <windows.h>
#include <clocale>

#include "f2.cpp"
extern int kol_zakazov;

int main()
{

SetConsoleCP(1251);
SetConsoleOutputCP(1251);
setlocale(LC_ALL, "RUS");

int choice;
do {cout << "1. Ввести информацию\n";
    cout << "2. Вывести на экран таблицу заказов\n";
    cout << "3. Вывести заказ по номеру\n";
    cout << "4. Сортировать\n";
    cout << "5. Завершить программу\n\n";
    cout << "Введите операцию: ";
    cin >> choice;
    switch(choice)
    {
      case 1: enter(); break;
      case 2: print(); break;
      case 3: read();  break;
      case 4: sort();  break;
      case 5: printf ("Программа завершена\n"); return 0;
      default: printf ("Неправильная операция\n"); break;
    }
```

```
  } while(choice != 5);
return 0;
}
// Файл f2.cpp
#include <iostream>
#include <cstdio>
#include <cstring>
#include <windows.h>
#include <clocale>

struct zakaz
{int nomer_zakaza;
 char nazvanie_siriy[20];
 int kol_vo;
 int stoimost;
 int itog_stoimost;
};
static zakaz data[20], temp[20];
int kol_zakazov;

extern void enter()
{register int a;
  cout << "Введите количество заказов: ";
  cin >> kol_zakazov;
  for(a=0; a<kol_zakazov; a++)
    {cout << "Введите номер заказа:";
     cin >> data[a].nomer_zakaza;
     cout << "Введите название товара: ";
     cin >> data[a].nazvanie_siriy;
     cout << "Введите количество: ";
     cin >> data[a].kol_vo;
     cout << "Введите стоимость: ";
     cin >> data[a].stoimost;
     data[a].itog_stoimost = data[a].kol_vo * data[a].stoimost;
    }
  cout << '\n';
}

extern void print()
{register int a;
 cout << '\n';
 cout << "#  Название  Количество  Стоимость  Итого\n";
 cout <<"*****************************************\n";
 for(a=0; a<kol_zakazov; a++)
  {printf("%d %6s %9d %10d %10d", data[a].nomer_zakaza,
           data[a].nazvanie_siriy, data[a].kol_vo,
           data[a].stoimost, data[a].itog_stoimost);
   cout << '\n';
  }
```

```
 cout << '\n';
}

extern int read()
{register int a;
 cout << "Введите номер заказа.\n";
 auto int choice; cin >> choice;
 for(a=0; a<kol_zakazov; a++)
   {if(choice == data[a].nomer_zakaza)
    {cout << "# Название  Количество  Стоимость  Итого\n";
     cout <<"*****************************************\n";
    printf("%d %6s %9d %10d %10d", data[a].nomer_zakaza, da-
    ta[a].nazvanie_siriy, data[a].kol_vo, data[a].stoimost, da-
    ta[a].itog_stoimost);
    cout << "\n\n";
    return 0;
    }
   }
   cout << "Заказа с заданным номером нет.";
}

extern void sort_nomer()
{register int a, b;
 cout << "Old";
 print();
 for(a=1; a<kol_zakazov; a++)
   for(b=kol_zakazov-1; b>=a; b--)
    {if(data[b-1].nomer_zakaza > data[b].nomer_zakaza)
       {temp[b] = data[b-1];
        data[b-1] = data[b];
        data[b] = temp[b];  }
       }
 cout << "New";
 print();
 }
extern void sort_nazvanie()
{register int a, b;
 cout << "Old";
 print();
 for(a=1; a<kol_zakazov; a++)
   for(b=kol_zakazov-1; b>=a; b--)
   {if(strcmp(data[b-1].nazvanie_siriy, data[b].nazvanie_siriy)>0)
     {temp[b] = data[b-1];
     data[b-1] = data[b];
     data[b] = temp[b];
    }
   }
```

```
 cout << "New";
 print();
 }
extern void sort_kol()
{register int a, b;
 cout << "Old";
 print();
 for(a=1; a<kol_zakazov; a++)
   for(b=kol_zakazov-1; b>=a; b--)
    {if(data[b-1].kol_vo > data[b].kol_vo)
        {temp[b] = data[b-1];
        data[b-1] = data[b];
        data[b] = temp[b];
        }
    }
 cout << "New";
 print();
 }
extern void sort_stoimost()
{register int a, b;
 cout << "Old";
 print();
 for(a=1; a<kol_zakazov; a++)
   for(b=kol_zakazov-1; b>=a; b--)
    {if(data[b-1]. stoimost > data[b]. stoimost)
        {temp[b] = data[b-1];
        data[b-1] = data[b];
        data[b] = temp[b];
        }
    }
 cout << "New";
 print();
}
extern void sort_itog_stoimost()
{register int a, b;
    cout << "Old";
    print();
    for(a=1; a<kol_zakazov; a++) for(b=kol_zakazov-1; b>=a; b--)
    {if(data[b-1].itog_stoimost > data[b].itog_stoimost)
        {      temp[b] = data[b-1];
            data[b-1] = data[b];
            data[b] = temp[b];     }
    }
    cout << "New";
    print();
 }
```

```
extern int sort()
{if(!data[0].nomer_zakaza)
{cout << "Введите информацию\n\n";
return 0; }
int choice;
do {cout << "\t1. Сортировать по номеру заказа\n";
    cout << "\t2. Сортировать по названию\n";
    cout << "\t3. Сортировать по количеству\n";
    cout << "\t4. Сортировать по  стоимости\n";
    cout << "\t5. Сортировать по итоговой стоимости\n";
    cout << "\t6. Завершить сортировку\n";
    cin >> choice;
    switch(choice)
     {case 1: sort_nomer(); break;
      case 2: sort_nazvanie(); break;
      case 3: sort_kol(); break;
      case 4: sort_stoimost(); break;
      case 5: sort_itog_stoimost(); break;
      case 6: break;
     }
  } while(choice != 6);
cout << "***Завершена сортировка***\n"; cout << '\n';
return 0;
}
```

Лабораторная работа № 2.
Алгоритмы сортировки и поиска данных

Цель работы – изучить алгоритмы сортировки и поиска данных и разработать программу для реализации этих алгоритмов.

Ход выполнения лабораторной работы должен быть отражен в отчете. Отчет должен содержать титульный лист, номер задания, коды программ, скриншоты с результатами выполнения программы.

Методические указания

Простые алгоритмы сортировки. Основная задача сортировки звучит так: дано множество элементов массива, которые можно сравнить по какому-то признаку и расположить в заданном порядке. Рассмотрим простые алгоритмы сортировки.

Сортировка пузырьком. Данная сортировка является одной из самых простых для понимания. Выполняется несколько проходов по массиву от начала и перебирая пары соседних элементов. Если первый элемент пары больше второго, то они меняются местами. Таким образом, меньшие элементы массива как бы «всплывают» в начало массива. Код ниже демонстрирует сортировку пузырьком.

Пример 2.1. Сортировка пузырьком.

```
void swap(int *v0, int *v1)// Функция для обмена значениями
{int tmp = *v0;
     *v0 = *v1;
     *v1 = tmp;
}
void BubbleSort(int *array, size_t size)
{for(size_t i = 0; i < size - 1; i++)
  {bool sorted = true;
   for(size_t j = 1; j < size - i; j++)
     {if(array[j] < array[j - 1])
     {
       sorted = false;
       swap(array[j], array[j - 1]);
     }
     }
     if(sorted == true) break;
  }
}
```

Среднее время работы сортировки пузырьком $O(n^2)$.

Сортировка вставками. Алгоритм данной сортировки просматривает все элементы по порядку от начала и сравнивает текущий элемент с предыдущими, уже отсортированными. Алгоритм ищет среди предыдущих отсортированных элементов место для вставки нового и продолжает поиск для последующих элементов. Ниже представлен код, выполняющий сортировку вставками.

Пример 2.2. Сортировка вставками.

```
void insertSort(int* a, int size)
{int x; int i, j;
for (i=1; i<size; i++) // Цикл проходов, i - номер прохода
 {x=a[i]; // Элемент, для которого нужно найти место в готовой
             //последовательности
  for (j=i-1; j>=0&&a[j]>x; j--)
    a[j+1] = a[j]; // Сдвигаем элемент направо, пока не дошли
             //до меньшего, вместо трех обменов один
  a[j+1] = x; // Место найдено, вставляем элемент
 }
}
```

Количество сравнений при сортировке методом вставки зависит от исходной упорядоченности массива.

Наилучший случай: $2(n - 1)$.

Средний случай: $1/4(n^2 + n)$.

Наихудший случай: $1/2(n^2 + n)$.

Количество перестановок не превышает $n^2 / 4$.

Сортировка выбором. Алгоритм сортировки выбором является одним из самых простых. Находим номер минимального значения в текущем списке. Производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции). Теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы.

Пример 2.3. Сортировка методом выбора.

```
void MinSort (int *array, int size)
{int i, j, k;
  if (size <= 1) return;
  for (i=0; i<size-1; i++) // Определяем i-й элемент
                           //отсортированной последовательности
    {for (k=i, j=i+1; j<size; j++)
      if (array[j] < array[k]) // Сравниваем i-й элемент  со всеми
                     //остальными до конца и находим минимальный
        k = j; // Запоминаем номер
  Swap(array+k, array+i); // Меняем местами минимальный элемент
                           // и элемент,с которого начинался цикл
    }}
```

Количество сравнений – $(n^2 – n) / 2$.

Количество перестановок:

– в наилучшем случае: $3(n – 1)$;

– в среднем случае: $n(\ln n + y)$, где y – константа Эйлера (примерно 0,58);

– в худшем случае: $n^2 / 4 + 3(n – 1)$.

Усовершенствованные методы сортировки. Рассмотрим методы Шелла и Хоара.

Сортировка методом Шелла. Метод Шелла является обобщением метода вставок и основан на том, что сортировка методом вставок выполняется очень быстро на почти отсортированном массиве данных. Он также известен как сортировка с убывающим шагом. В отличие от сортировки методом вставок при сортировке методом Шелла весь массив не сортируется одновременно: массив разбивается на отдельные сегменты, которые сортируются раздельно с помощью метода вставок. Основная идея алгоритма состоит в том, что на начальном этапе реализуется сравнение и, если требуется, перемещение далеко отстоящих друг от друга элементов. Интервал между сравниваемыми элементами (gap) постепенно уменьшается до единицы, что приводит к перестановке соседних элементов на последних стадиях сортировки (если это необходимо) [2].

Реализуем метод Шелла следующим образом. Начальный шаг сортировки примем равным n / 2, т. е. 1/2 от общей длины массива, и после каждого прохода будем уменьшать его в два раза. Каждый этап сортировки включает в себя проход всего массива и сравнение отстоящих на gap элементов. Проход с тем же шагом повторяется, если элементы переставлялись. Заметим, что после каждого этапа отстоящие на gap элементы отсортированы [2]. Рассмотрим пример.

Пример 2.4. Сортировка методом Шелла.

```
void SortShell(int *x1,int n1)
{int i1,j1;
 int gap;     // Шаг сортировки
 int Sorted;  // Флаг окончания этапа сортировки
 for(gap=n1/2; gap>0; gap/=2)// Начало сортировки
   do // Приращение уменьшаем в 2 раза для каждого do-while
     {Sorted = 0; // Сортировка пузырьком на расстоянии gap
      for(i1=0,j1=gap;j1<n1;i1++,j1++)
         if(*(x1+i1)>*(x1+j1))
           { swap((x1+i1),(x1+j1); Sorted = 1;
           }
     } while(Sorted);
}
```

В худшем случае сложность алгоритма Шелла при таком выборе приращений $O(n^2)$.

Предложенная Хиббардом последовательность (все значения $2^i - 1 \le n$, $i \in n$) приводит к алгоритму сложностью $O(n^{3/2})$.

Последовательность Сэджвика

$$inc[s] = \begin{cases} 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1, & \text{если s четно,} \\ 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1, & \text{если s нечетно} \end{cases}$$

приводит к алгоритму со средней сложностью $O(n^{7/6})$, в худшем случае сложность порядка $O(n^{4/3})$.

Предложенная Праттом последовательность (все значения $2^i \cdot 3^j \le n / 2$, $i, j \in n$) обеспечивает сложность алгоритма $O(n \log^2 n)$.

Сортировка методом Хоара. Сортировка методом Хоара (или быстрая сортировка) – это наиболее эффективный алгоритм внутренней сортировки. Его производительность зависит от выбора точки разбиения. При быстрой сортировке используется следующая стратегия:

1 Массив разбивается на меньшие подмассивы.

2 Подмассивы сортируются.

3 Отсортированные подмассивы объединяются.

Быструю сортировку можно реализовать несколькими способами, но цель каждого подхода заключается в выборе элемента данных и помещении его в правильную позицию (этот элемент называется точкой разбиения). Поместить элемент надо таким образом, чтобы все элементы слева от точки разбиения оказались меньше точки разбиения, а все элементы справа от точки разбиения оказались больше. Выбор точки разбиения и метод, используемый для разбиения массива, оказывают большое влияние на общую производительность реализации [2].

Рассмотрим один из вариантов реализации сортировки Хоара.

Пример 2.5. В самой процедуре сортировки сначала выберем средний элемент. Потом, используя переменные i и j, пройдемся по массиву, отыскивая в левой части элементы больше среднего, а в правой – меньше среднего. Два найденных элемента переставим местами. Будем действовать так, пока i не станет больше j. Тогда получим два подмножества, ограниченные по краям индексами l и r, а в середине – j и i. Если эти подмножества существуют (т. е. i < r и j > l), то выполним их сортировку [2].

```
void QuickSort (int *a, int left, int right)
{int i=left, j=right; // Инициализируем переменные левой и правой
                      //границами подмассива
int test=a[(left+right)/2]; // Выбираем в качестве элемента
                      //разбиения средний элемент массива
```

```
do
  {while (a[i] < test) i++; // Находим элемент, больший элемента
                            //разбиения
   while (a[j] > test) j--; // Находим элемент, меньший элемента
                                 //разбиения
   if (i<=j)
   {Swap(a+i, a+j); i++; j--;
    }
  } while(i <= j);
   // Рекурсивно вызываем алгоритм для правого и левого подмассива
if (i<right) QuickSort(a, i, right);
if (j>left) QuickSort(a, left, j);
}
```

Сложность алгоритма в лучшем случае: O(n log 2n). Сложность алгоритма в среднем случае: O(n log n). Сложность алгоритма в худшем случае: $O(n^2)$.

Алгоритмы поиска. Рассмотрим два простейших метода поиска элементов в массиве.

Последовательный поиск. Последовательный поиск предусматривает последовательный просмотр всех элементов массива в порядке их расположения, пока не найдется элемент, равный заданному образцу. Если достоверно неизвестно, что такой элемент имеется в списке, то необходимо следить за тем, чтобы поиск не вышел за пределы списка [2].

Пример 2.6. Пусть во входном массиве k задано пять целых чисел и ключ v. Необходимо найти среди элементов массива элемент, равный v, и его номер в массиве.

```
#include <stdio.h>
void main(void)
{
  setlocale(LC_ALL, "RUS");
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
  int k[5],v,i,count=0;
  puts("Введите элементы:");
  for (i=0; i<5; i++)
    scanf("%d",&k[i]);
  puts("Введите элемент для поиска");
  scanf("%d",&v);
  i=0;
  while(i<5)
    {if (k[i]==v)
```

```
      {printf("элемент - %d позиция - %d\n",v,(i+1));
       count++;
      }
     i++;
     }
  if(count==0) printf("элемент %d не найден\n",v);
}
```

Пример 2.7. Поиск элемента по индексу.

```
#include <stdio.h>
void main(void)
{
  setlocale(LC_ALL, "RUS");
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
  int k[5],v,i,count=0;
  puts("Введите элементы:");
  for (i=0; i<5; i++)
    scanf("%d",&k[i]);
  puts("Введите индекс элемента");
  scanf("%d",&v);
  if(v>=0&&v<5)
    printf("Элемент с индексом %d равен %d\n",v,k[v]);
  else
  printf("Элемент с индексом %d не существует\n",v);
}
```

Вычислительная сложность линейного поиска равна O(n) (обычный проход по всем элементам в поисках нужного).

Бинарный поиск. Предположим, у нас есть отсортированный массив. Тогда, остановившись на любом индексе, мы сможем понять, в каком диапазоне находится искомый элемент (в левом или правом). Суть заключается в рекурсивном делении массива на два подмассива и сравнении медианного элемента и искомого. Если медианный элемент больше искомого, то продолжаем поиск в левом диапазоне. Если же медианный меньше искомого, то ищем в правом диапазоне. Если они равны, то возвращаем индекс.

Сложность бинарного поиска в худшем и среднем случаях равна O(log n), в лучшем – O(1) (если обнаруживаем искомый элемент на первой итерации). У бинарного поиска есть недостаток – он требует упорядочивания данных по возрастанию. Сложность сортировки – не менее O(n log n). Поэтому если список короткий, используется все-таки линейный поиск.

Пример 2.8. Программа бинарного поиска.

```
int binarySearch(int *nums,int low, int high, int target)
{
 if (low > high) {// Пространство поиска исчерпано
```

```
  return -1;}
// Находим средний элемент и сравниваем его с образцом
 int mid = (low + high)/2;
 if (target == nums[mid]) {
    return mid; } // Элемент найден
 else
  if (target < nums[mid]) {
// Отбрасываем все элементы справа, включая средний
    return binarySearch(nums, low, mid - 1, target);
    }
  else {
// Отбрасываем все элементы слева, включая средний
    return binarySearch(nums, mid + 1, high, target);
    }
}
```

Пример 2.9. Вызов программы бинарного поиска.

```
#include <stdio.h>
void main(void)
#define num 10
{
  setlocale(LC_ALL, "RUS");
  SetConsoleOutputCP(1251);
  SetConsoleCP(1251);
  int k[num],v;
  puts("Введите элементы:");
  for (int i=0; i<num; i++)
    scanf("%d",&k[i]);
  puts("Введите элемент для поиска:\n");
  scanf("%d",&v);
  res=binarySearch(k,0,num-1,v);
  if (res>=0)
    printf("Индекс искомого элемента: %d\n", res);
  else
    printf("Элемент не найден\n");
}
```

Индивидуальные задания

Разработать программу для создания динамического массива из 100 случайных целых чисел. Организовать сортировку массива методом обмена, методом выбора, методом вставки, методом Шелла и методом Хоара. Организовать последовательный поиск заданного элемента в массиве и подсчитать количество повторений этого элемента. Организовать бинарный поиск заданного элемента в отсортированном массиве.

Управление в программе организовать через меню, структура которого следующая:

1 Создать массив.

2 Пузырьковая сортировка.

3 Сортировка методом выбора.

4 Сортировка методом вставки.

5 Сортировка методом Шелла.

6 Сортировка методом Хоара.

7 Последовательный поиск.

8 Бинарный поиск.

Результаты всех сортировок и поиска записать в файл.

Дополнительное задание. Разработать программу для сортировки массива структур различными методами. Задание структурного типа взять из лабораторной работы № 1.

Приложение 2

Пример, реализующий алгоритмы сортировки и запись результатов в файл [2].

```
#include <stdio.h>
#include <iostream.h>
void swap(int *x,int *y);
void BubbleSort (int *a1, int count); // Функция пузырьковой
                                      //сортировки
void MinSort (int *a1,int count); // Функция сортировки методом
                                  //выбора
void insertSort (int *a1,int count); // Функция сортировки
                                      //вставками
void SortShell (int *x,int count); // Функция сортировки методом
                                      //Шелла
void QuickSort (int *x,int ,int ); // Функция быстрой сортировки
void vvod(int *a1,int count); // Функция ввода значений массива
void out(int *a1,int count); // Функция вывода значений массива
void file_write(int *a1,int count);
FILE *file;
int step=0;

void main(void)
{
int *a,n;
file=fopen("result_sort.txt","w");
puts("Enter size of array:");
scanf("%d",&n);
a=new int[n];
puts("Enter elements of array");
vvod(a,n);
file_write(a,n);
```

```
step++;
puts("Sorted array by method \\Bubble Sort\\:");
BubbleSort (a,n);
  out(a,n);
file_write(a,n);
step++;
vvod(a,n);
puts("Sorted array by method \\Min element\\:");
MinSort (a,n);
out(a,n);
file_write(a,n);
step++;
vvod(a,n);
puts("Sorted array by method \\Insert Sort\\:");
  insertSort (a,n)
out(a,n);
file_write(a,n);
step++;
vvod(a,n);
puts("Sorted array by method \\Shell Sort\\:");
SortShell (a,n);
out(a,n);
file_write(a,n);
step++;
vvod(a,n);
puts("Sorted array by method \\Hoare Sort\\:");
QuickSort (a,0,n-1);
out(a,n);
file_write(a,n);
fclose(file);
  delete []a;
}

void swap(int *x,int *y)
{ int t;
  t=*x;
  *x=*y;
  *y=t;
}

void vvod(int *a1,int count)
{
for(i=0;i<count;i++)
cin>>*(a1+i);
}

void out(int *a1,int count)
{
for(i=0;i<count;i++)
    cout<<*(a1+i)<<" ";
cout<<endl;
}
```

```
void BubbleSort(int *array, size_t size)
{for(size_t i = 0; i < size-1; i++)
  {bool sorted = true;
   for(size_t j = 1; j < size-i; j++)
     {if(array[j] < array[j-1])
       {
        sorted = false;
        Swap(array+j, array+(j-1));
       }
     }
      if(sorted == true) break;
  }
}

void MinSort (int *array, int size)
{int i, j, k;
 if (size <= 1) return;
for (i=0; i<size-1; i++) // Определяем i-й элемент отсортированной
                         //последовательности
  {for (k=i, j=i+1; j<size; j++)
    if (array[j] < array[k]) // Сравниваем i-й элемент  со всеми
                     //остальными до конца и находим минимальный
      k = j; // Запоминаем номер
  Swap(array+k,array+i); // Меняем местами минимальный элемент и
                         //элемент, с которого начинался цикл
  }
}

void insertSort(int* a, int size)
{int x; int i, j;
for (i=1; i<size; i++) // Цикл проходов, i - номер прохода
 {x=a[i]; // Элемент, для которого нужно найти место в готовой
                                  //последовательности
  for (j=i-1; j>=0&&a[j]>x; j--)
    a[j+1] = a[j]; // Сдвигаем элемент направо, пока не дойдем
                   //до меньшего, вместо трех обменов один
  a[j+1] = x; // Место найдено, вставляем элемент
 }
}

void SortShell (int *x,int count)
{ int i,j;     // Две переменные цикла
  int gap;     // Шаг сортировки
  int sorted;  // Флаг окончания этапа сортировки
  for(gap=count/2;gap>0;gap/=2)// Начало сортировки
    do  { sorted = 0;
          for(i=0,j=gap;j<count;i++,j++)
            if(*(x+i)>*(x+j))
              { swap((x+i),(x+j));
                sorted = 1;

        }while(sorted);}
```

```
void QuickSort(int *a,int left,int right)
{int i=left, j=right; // Инициализируем переменные левой
                     //и правой границами подмассива
int test=a[(left+right)/2]; // Выбираем в качестве
                     //элемента разбиения средний элемент массива
do {
  while (a[i] < test) i++; // Нашли элемент, больший элемента
                           // разбиения
  while (a[j] > test) j--; // Нашли элемент, меньший элемента
                           //разбиения
  if (i<=j) {Swap(a+i, a+j); i++; j--;} // Меняем местами и
                                  //продолжаем поиск
  }while(i <= j);
if (i<right) QuickSort(a, i, right); // Рекурсивно вызываем
                           //алгоритм для правого подмассива
if (j>left) QuickSort(a, left, j); // Рекурсивно вызываем алгоритм
                           //для левого подмассива
}

void file_write(int *a1,int count, int step)
{ if(step==0)
    fprintf(file,"\t\tRESULTS OF SORTS\n");
  if(step==1)
    fprintf(file,"1.Bubble Sort\n");
  if(step==2)
    fprintf(file,"\n2.Min Sort\n");
  if(step==3)
    fprintf(file,"\n3.Insert Sort\n");
  if(step==4)
    fprintf(file,"\n4.Shell Sort\n");
  if(step==5)
    fprintf(file,"\n5.Hoare Sort\n");
  if(step!=0)
    for(i=0;i<count;i++) fprintf(file,"%d ",*(a1+i));
}
```

Лабораторная работа № 3.
Списки и основные операции над ними

Цель работы – изучить структуру данных «список». Рассмотреть внутреннюю структуру, виды списков, основные операции над списками. Разобрать приведенные примеры и выполнить самостоятельно задания.

Ход выполнения лабораторной работы должен быть отражен в отчете. Отчет должен содержать титульный лист, номер задания, коды программ, скриншоты с результатами выполнения программы.

Методические указания

Список – это упорядоченная коллекция (массив) значений определенного типа, которая строится на основе узлов, соединенных между собой. Каждый узел списка содержит в себе значение и указатель (или же индекс) на следующий узел этого списка. На рисунке 3.1 изображена схема списка.

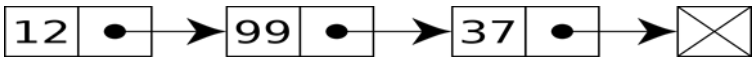


Рисунок 3.1 – Схема односвязного списка

Как видно из рисунка 3.1, узлы имеют указатели на следующий узел данного списка. Эта разновидность списка называется односвязным списком. Также существует двусвязный список, в котором каждый узел списка имеет указатель как на последующий узел, так и на предыдущий. Последний узел всегда имеет указатель на последующий элемент, который указывает на null. В двусвязном списке указатель на предыдущий элемент в первом узле также указывает на null. На рисунке 3.2 изображена схема организации двусвязного списка.

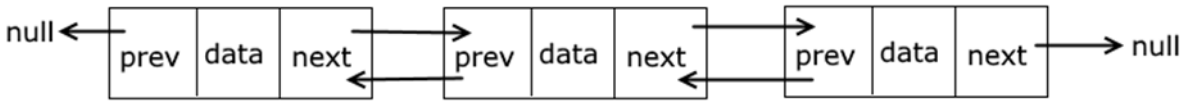


Рисунок 3.2 – Схема организации двусвязного списка

Также существует кольцевой тип списка. Кольцевой список отличается от обычного односвязного/двусвязного тем, что указатель на последующий узел последнего элемента указывает на первый узел списка. В двусвязном списке указатель на предыдущий элемент первого узла указывает на последний узел списка. На рисунке 3.3 изображена схема организации кольцевого списка.

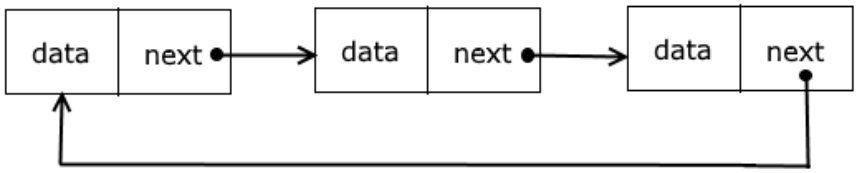


Рисунок 3.3 – Схема организации кольцевого списка

Стоит уточнить, что каждый узел списка должен получать память динамически, когда необходимо вставить элемент в список, и освобождать память, когда элемент удаляется из списка. Если заранее известен максимальный размер списка, хорошей практикой будет его реализация либо при помощи специального модуля, который выделит достаточное количество памяти (динамической или статической), либо при помощи массива узлов.

Операции над списками. Основными операциями, которые выполняются над списками, являются перебор элементов списка, поиск заданного элемента, вставка в список нового элемента, удаление элемента из списка. Выполнение этих операций основано на изменении указателей [2].

Создание списка. Эта операция предполагает определение указателя начала списка и присвоение ему значения NULL (т. е. никакого).

Перебор элементов списка. Эта операция выполняется для линейных списков очень часто и состоит в последовательном доступе к элементам списка – ко всем до конца списка либо до нахождения искомого элемента. Для каждого из перебираемых элементов осуществляется некоторая обработка его информационной части: сравнение с образцом, печать, модификация и пр.

Вставка элемента в список. На рисунке 3.4 мы видим, что указатель элемента Эл2 теперь указывает не на элемент Эл3, а на элемент Эл5. Указатель элемента Эл5 указывает на элемент Эл3. Логическая последовательность элементов будет следующей: Эл1, Эл2, Эл5, Эл3, Эл4. Подчеркнем еще раз, что в списке новый элемент Эл5 физически не обязательно помещается за элементом Эл2. Достаточно, чтобы он следовал за ним логически.

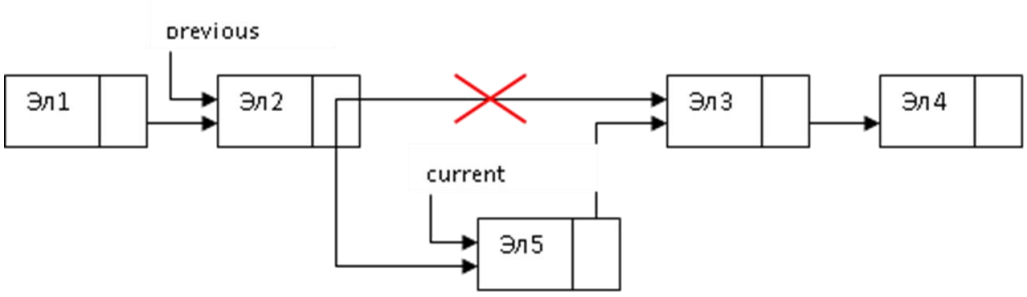


Рисунок 3.4 – Вставка элемента в список

Удаление элемента из списка. При удалении элемента Эл3 из списка (рисунок 3.5) прежде всего выполняется поиск этого элемента в списке, а затем его удаление. Теперь за элементом Эл5 следует элемент Эл4. Соответствующим образом изменился указатель элемента Эл5. Элемент Эл3 в списке теперь отсутствует, т. к. при просмотре списка, переходя от элемента к элементу в соответствии с указателями, мы на элемент Эл3 не попадаем. Следует отметить, что удаление элемента из списка и удаление из памяти – это не одно и то же. Элемент Эл3 будет находиться в памяти до тех пор, пока мы не удалим его явно с помощью оператора delete.

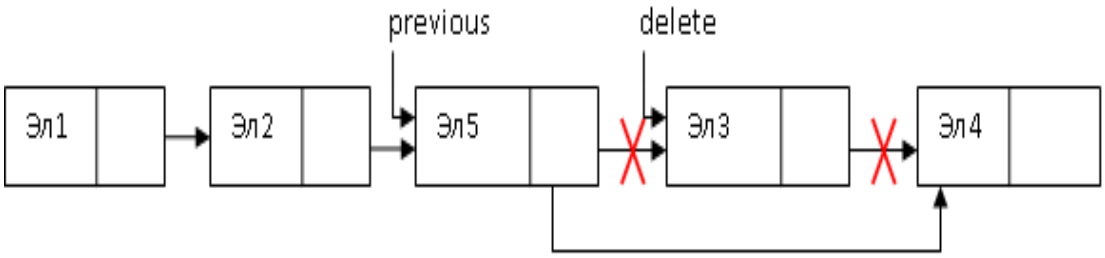


Рисунок 3.5 – Удаление элемента из списка

Организация односвязного списка с помощью массива структур. Элемент списка:

```
struct list {
  char book;
  list *next;
};
```

Память, отводимая под элемент списка, выделяется динамически, поэтому при реализации списков его длина ограничивается только доступным объемом памяти. Информационное поле представляет собой символьную переменную по имени book. Признаком последующего элемента списка является наличие поля next, а признаком последнего элемента списка является равенство значению NULL этого поля.

Следующая программа реализует приведенные ниже операции:

– создает пустой список;

– добавляет элементы в список в алфавитном порядке;

– удаляет элементы из списка;

– проверяет, является ли список пустым;

– выводит список на экран.

Пример 3.1. Организация односвязного списка с помощью массива структур.

```
#include <stdio.h>
#include <stdlib.h>
struct list
```

```
{char book;
 struct list *next;
};
typedef struct list ListNode;
typedef ListNode * ListNodePtr;
void insert (ListNodePtr*, char);
char del (ListNodePtr*, char);
int Empty (ListNodePtr);
void printList (ListNodePtr);
// Выводим инструкции
void instructions(void)
{
printf("Enter the choice:\n"
"1 insert an element into the list.\n"
"2 delete an element from the list.\n"
"3 end program.\n");
}
int main ()
{ ListNodePtr start = NULL;
  int choice; char elem;
  instructions();   // Выводим меню
  printf("?"); scanf("%d", &choice);
while (choice !=3){
switch (choice){
case 1:
/*Добавить значение в список*/
printf ("Enter an character: ");
scanf("\n%c", &elem);
insert(&start, elem);
printList (start);
break;
case 2: /*Удалить значение из списка*/
if (!Empty(start)){
  printf("Enter character to be deleted:");
  scanf("\n%c", &elem);
  if (del(&start,elem))
  {printf("%c deleted.\n",elem);
   printList(start);}
  else
   printf("%c not found.\n",elem);
  }
else printf("List is empty.\n");
     break;
default:
printf ("Invalid choice.\n");
instructions();
break;
}
printf("?");
scanf("%d", &choice);
}
printf("End of run.\n");
```

```
return 0;
}
// Вставляем в список новое значение в алфавитном порядке
void insert (ListNodePtr *s,char value)
{ListNodePtr newP, previous, current;
newP=(ListNodePtr)malloc(sizeof(ListNode));
if (newP!=NULL){/*есть ли место ?*/
newP->book =value; // Сформировали новый элемент
newP->next=NULL;
previous=NULL; // Сформировали указатели для поиска
current=*s;  // Начало списка
while (current!=NULL
&&value>current->book)
// пока не конец списка и value больше текущего элемента
 {previous=current; // Перемещаемся к следующему
  current=current->next;}
if(previous==NULL)
 { // Добавляем в начало списка
  newP->next=*s;
  *s=newP;
 }
else
 { // Добавляем перед большим
  previous->next=newP;
  newP->next=current;
 }
}
else
printf("%c not inserted.\n", value);
}
// Удаляем из списка заданное значение
char del (ListNodePtr *s, char value)
{
 ListNodePtr previous, current, temp;
 if (value==(*s)->book){
  temp=*s;
  *s=(*s)->next;  // Отсоединяем узел в начале списка
  free(temp); // Освободить память отсоединенного узла
  return value;
  }
 else{
  previous=*s;
  current=(*s)->next;
while (current!=NULL&&current->book!=value)
 { // Поиск удаляемого символа
  previous=current;
  current=current->next;
  }
if (current!=NULL){ // Переустанавливаем указатели
 temp=current;
 previous->next=current->next;
 free(temp);
```

```
 return value;}
}
 return '\0';
}
// Проверяем список на пустоту
int Empty(ListNodePtr s)
  return s==NULL; // Возвращаем 1 (true), если список пуст
// Выводим список на экран
void printList (ListNodePtr current)
{
 if (current==NULL)
  printf("The list is empty.\n\n");
 else {
  printf("The list is :\n");
  while (current!=NULL){
    printf("%c-->",
    current->book);
    current=current->next;
    }
  printf("NULL\n\n");
  }
}
```

Двунаправленный (двусвязный) список. В односвязном списке легко найти следующий элемент, но вернуться на предыдущий достаточно трудно. Для решения указанной проблемы используется другая структура – двусвязный список. Для такого списка в качестве ссылок используются два указателя – один на следующий элемент списка, другой – на предыдущий (рисунок 3.6).

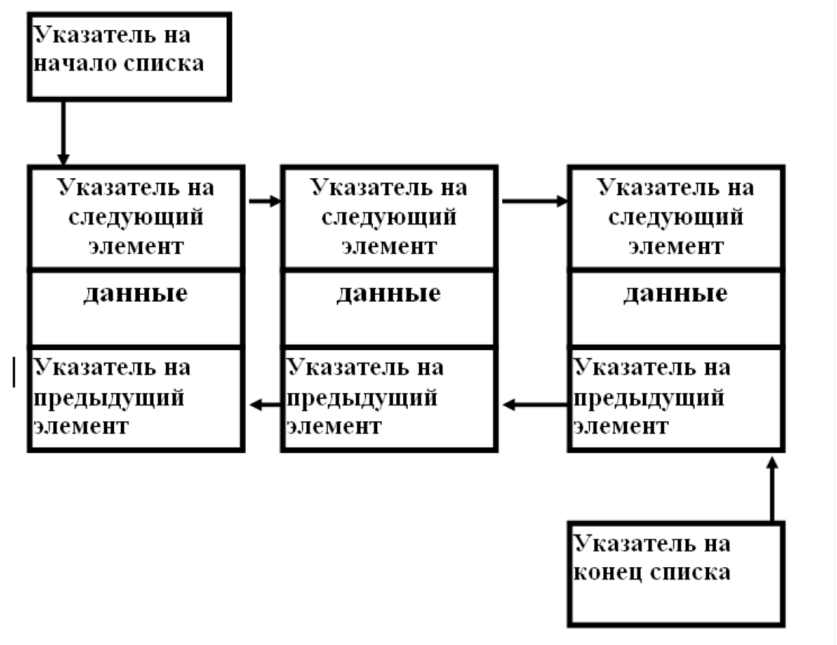


Рисунок 3.6 – Двунаправленный (двусвязный) список

Организацию двусвязных списков с помощью массива данных можно выполнить различными путями. Можно зарезервировать несколько массивов одинаковой размерности. В основном массиве хранится значащая информация, во вспомогательных – значения индексов следующих элементов списка и значения индексов предыдущих элементов списка, если список двусвязный [2]. В примере 3.2:

– основная информация представлена в массиве list;

– индексы следующих элементов – в массиве next;

– индексы предыдущих элементов – в массиве prev.

Пример 3.2. Организация двусвязного списка с помощью массива.

```
/* Создать список с помощью массива целых чисел. Элементы списка
в обратном порядке вывести на экран */
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

int main (void)
{int list[MAX]; int next[MAX]; int prev[MAX];
 int end=0; int begin=0;
 for (int i=0; i<MAX; i++) { // Инициализация
  list[i]=next[i]=prev[i]=0;}
 printf("Input the number of elements: ");
 int n=0; scanf("%d",&n);
 printf("Input elements of list: ");
 int count=0;
 for (i=0; i<n; i++) {scanf("%d",&list[i]);
  next[i]=i+1; // Ссылка на следующий элемент
  prev[i]=i-1; // Ссылка на предыдущий элемент
  count++; }
prev[0]=-1; next[count]=-1;
begin=0;
end=count-1;
printf("Elements of list from the end to begin: ");
int temp=end;
do {
  printf("%d ",list[temp]);
  temp=prev[temp]; // Переход к предыдущему элементу
  } while (prev[temp]!=-1);
printf("%d \n",list[temp]);
system("PAUSE");
return 0;
}
```

Индивидуальные задания

1 Создать список с помощью массива целых чисел. Элементы списка в обратном порядке вывести на экран.

2 Создать список с помощью массива целых чисел. Все четные элементы списка вывести на экран.

3 Создать список с помощью массива целых чисел. Все нечетные элементы списка вывести на экран.

4 Создать односвязный список с помощью массива целых чисел. Отсортировать элементы списка по возрастанию, задавая порядок чисел массивом индексов следующих элементов (next). В результате массив чисел остается без изменений, массив индексов переупорядочивается. Результирующий список вывести на экран.

5 Создать односвязный список с помощью массива целых чисел. Расположить в начале списка все четные элементы списка, задавая порядок чисел массивом индексов следующих элементов (next). В результате массив чисел остается без изменений, массив индексов переупорядочивается. Результирующий список вывести на экран.

6 Создать односвязный список с помощью массива целых чисел. Исключить из списка все нулевые элементы, задавая порядок чисел массивом индексов следующих элементов (next). В результате массив чисел остается без изменений, массив индексов переупорядочивается. Результирующий список вывести на экран.

7 Создать односвязный список с помощью массива целых чисел. Исключить из списка все нулевые элементы, задавая порядок чисел массивом индексов следующих элементов (next). В результате массив чисел остается без изменений, массив индексов переупорядочивается. Найти сумму всех четных элементов списка. Результирующий список и сумму вывести на экран.

8 Создать односвязный список с помощью массива целых чисел. Ввести с клавиатуры число и поместить его за пятым элементом списка, задавая порядок чисел массивом индексов следующих элементов (next). Результирующий список вывести на экран.

9 Создать односвязный список с помощью массива целых чисел. Ввести с клавиатуры число и поместить его перед тем элементом списка, который больше него. Результирующий список вывести на экран. Порядок чисел в списке задается массивом индексов следующих элементов (next).

10 Создать односвязный список с помощью массива целых чисел. Ввести с клавиатуры число, найти это число в списке и удалить. Результирующий список вывести на экран. Порядок чисел в списке задается массивом индексов следующих элементов (next).

11 Создать односвязный список с помощью массива целых чисел. Ввести с клавиатуры число, найти все элементы с этим числом в списке и удалить. Результирующий список вывести на экран. Порядок чисел в списке задается массивом индексов следующих элементов (next).

12 Создать односвязный список с помощью массива целых чисел. Поменять местами четные и нечетные элементы списка (рядом стоящие). Результирующий список вывести на экран. Порядок чисел в списке задается массивом индексов следующих элементов (next).

13 Создать односвязный список с помощью массива целых чисел. Сформировать новый список, в котором элементы следуют от конца к началу (последний элемент станет первым, предпоследний – вторым и т. д.). Результирующий список вывести на экран. Порядок чисел в списке задается массивом индексов следующих элементов (next).

14 Создать односвязный список с помощью массива целых чисел. Продублировать в списке первый, третий и пятый элементы. Результирующий список вывести на экран. Порядок чисел в списке задается массивом индексов следующих элементов (next).

15 Создать односвязный список с помощью массива целых чисел. Удалить в списке первый, третий и пятый элементы. Результирующий список вывести на экран. Порядок чисел в списке задается массивом индексов следующих элементов (next).

16 Создать список с помощью массива структур. Элементы списка в обратном порядке вывести на экран.

17 Создать список с помощью массива структур. Все четные элементы списка вывести на экран.

18 Создать список с помощью массива структур. Все нечетные элементы списка вывести на экран.

19 Создать односвязный список с помощью массива структур. Отсортировать элементы списка по возрастанию. Результирующий список вывести на экран.

20 Создать односвязный список с помощью массива структур. Один из элементов структуры – целое число. Расположить в начале списка все элементы списка с четными целыми числами. Результирующий список вывести на экран.

21 Создать односвязный список с помощью массива структур. Исключить из списка все элементы с нулевым целым числом. Результирующий список вывести на экран.

22 Создать односвязный список с помощью массива структур. Исключить из списка все элементы с нулевым целым числом. Найти сумму тех целочисленных элементов списка, которые являются четными. Результирующий список и сумму вывести на экран.

23 Создать односвязный список с помощью массива структур. Создать новый элемент списка, ввести с клавиатуры число и поместить его в целочисленное поле нового элемента. Разместить новый элемент за пятым элементом списка. Результирующий список вывести на экран.

24 Создать односвязный список с помощью массива структур. Создать новый элемент списка, ввести с клавиатуры число и поместить его в целочисленное поле нового элемента. Разместить новый элемент перед тем

элементом списка, целочисленное поле которого имеет большее значение. Результирующий список вывести на экран.

25 Создать односвязный список с помощью массива структур. Ввести с клавиатуры число, найти это число в списке (в целочисленном поле) и удалить соответствующий элемент списка. Результирующий список вывести на экран.

26 Создать односвязный список с помощью массива структур. Ввести с клавиатуры число, найти все элементы с этим числом (в целочисленном поле) в списке и удалить. Результирующий список вывести на экран.

27 Создать односвязный список с помощью массива структур. Поменять местами четные и нечетные элементы списка (рядом стоящие). Результирующий список вывести на экран.

28 Создать односвязный список с помощью массива структур. Сформировать новый список, в котором элементы следуют от конца к началу (последний элемент станет первым, предпоследний – вторым и т. д.). Результирующий список вывести на экран.

29 Создать односвязный список с помощью массива структур. Продублировать в списке первый, третий и пятый элементы. Результирующий список вывести на экран.

30 Создать односвязный список с помощью массива структур. Удалить в списке первый, третий и пятый элементы. Результирующий список вывести на экран.

31 Написать программу, содержащую процедуру, которая меняет местами первый и второй элементы непустого списка. Если элементы не найдены, то вывести на экран соответствующее сообщение.

32 Написать программу, содержащую процедуру, которая меняет местами первый и пятый элементы непустого списка. Если элементы не найдены, то вывести на экран соответствующее сообщение.

33 Написать программу, содержащую процедуру, которая меняет местами первый и последний элементы непустого списка. Если элементы не найдены, то вывести на экран соответствующее сообщение.

34 Написать программу, содержащую процедуру, которая вставляет новый элемент перед каждым вхождением заданного элемента. Если элементы не найдены, то вывести на экран соответствующее сообщение.

35 Написать программу, содержащую процедуру, которая вставляет новый элемент за каждым вхождением заданного элемента. Если элементы не найдены, то вывести на экран соответствующее сообщение.

36 Написать программу, содержащую подпрограмму, которая проверяет на равенство списки M1 и M2.

37 Написать программу, содержащую функцию, которая определяет, входит ли список M1 в список M2. Предполагается, что списки существуют.

38 Написать программу, содержащую подпрограмму, которая копирует в конец непустого списка M его первый элемент. Если элементы не найдены, то вывести на экран соответствующее сообщение.

39 Написать программу, содержащую подпрограмму, которая копирует в начало непустого списка M его последний элемент. Если элементы не найдены, то вывести на экран соответствующее сообщение.

40 Написать программу, содержащую процедуру, которая копирует в список M за каждым вхождением заданного элемента все элементы списка M1.

41 Написать программу, содержащую процедуру, которая объединяет два упорядоченных по возрастанию списка M1 и M2 в один упорядоченный по возрастанию список, построив новый список M.

42 Написать программу, содержащую процедуру, которая объединяет два упорядоченных по возрастанию списка M1 и M2 в один упорядоченный по возрастанию список, сменив соответствующим образом ссылки в M1 и M2.

43 Написать программу, содержащую функцию, которая проверяет, упорядочены ли элементы списка по алфавиту.

44 Написать программу сортировки существующего списка по алфавиту. В программе использовать подпрограммы.

45 Написать программу, которая создавала бы файл целых чисел, а затем формировала список целых чисел файла. Создать в конце списка элемент, содержащий сумму всех чисел файла. В программе использовать подпрограммы.

46 Написать программу, которая создавала бы файл целых чисел, а затем формировала список целых чисел файла. Создать список чисел, являющихся суммой соседних элементов. В программе использовать подпрограммы.

47 Написать программу, которая создавала бы текстовый файл, а затем формировала список строк файла. Создать список обратных строк. В программе использовать подпрограммы.

48 Написать программу, которая создавала бы текстовый файл, а затем формировала список строк файла. Создать отсортированный список строк. В программе использовать подпрограммы.

49 Написать программу, которая создавала бы файл комбинированного типа, а затем формировала список, используя какое-либо поле записи. Создать отсортированный список. В программе использовать подпрограммы.

50 Написать программу, которая создавала бы файл комбинированного типа, а затем формировала список элементов файла. Создать отсортированный по какому-либо полю список. В программе использовать подпрограммы.

51 Составить программу, которая принимает с клавиатуры названия городов, динамически отводит место в памяти под каждое название и строит из них связанный список, упорядоченный по алфавиту. По окончании формирования список городов вывести на экран монитора.

52 Составить список учебной группы, содержащий не менее 15 студентов. Указать для каждого студента оценки, полученные на последних четырех экзаменах. Разработать программу, которая принимает данные с клавиатуры о каждом студенте, строит односвязный список, а затем удаляет из списка элементы, относящиеся к неуспевающим студентам.

53 Информацию о величине экспорта и соответствующий номер контракта записать в двусвязный кольцевой список. Затем переместить данную информацию в двумерный динамический массив и осуществить поиск максимальной величины экспорта. На экран вывести искомую информацию.

54 Создать двусвязный список, содержащий следующую информацию: год и соответствующую численность населения. Программу организовать таким образом, чтобы на экран выводилась информация, численность населения в которой была больше введенного с клавиатуры значения.

55 Ввести с клавиатуры строку символов, формируя из ее элементов двусвязный список. Написать программу, которая формирует двусвязный список из входной строки. В поле данных каждого элемента списка записывается отдельный символ. В программе производится анализ первого символа входной строки: если это символ ***'A'***, то в конец списка добавляется еще один символ ***'A'***, иначе из списка исключаются все символы ***'A'***. Полученный результат выводится на экран.

Лабораторная работа № 4.
Очереди и основные операции над ними

Цель работы – познакомиться с созданием и обработкой одномерных массивов данных. Научиться использовать одномерные массивы в алгоритмах вычисления выражений, сортировки, поиска.

Ход выполнения лабораторной работы должен быть отражен в отчете. Отчет должен содержать титульный лист, номер задания, коды программ, скриншоты с результатами выполнения программы.

Методические указания

Очередь – структура данных, которая работает по принципу «первый вошел – первый вышел» (FIFO). Основное предназначение данной структуры данных – это возможность выстраивать некоторые операции или данные по-порядку и получать их в том порядке, в котором они пришли в очередь. При помощи очередей реализуются такие высокоуровневые структуры, как очередь сообщений и очередь событий. Основные операции над очередью – это операция вставки объекта в конец очереди (push) и операция получения и удаления объекта из начала очереди (pop).

Существует несколько разновидностей очереди, которые могут понадобиться в том или ином случае. Очередь можно реализовать на массиве (кольцевая очередь) или на списке (обычная очередь). Также существует очередь с приоритетом, однако она реализовывается на бинарных деревьях.

На рисунке 4.1 изображена схема работы очереди и ее операции. Операция enqueue – это push, а операция dequeue – pop.

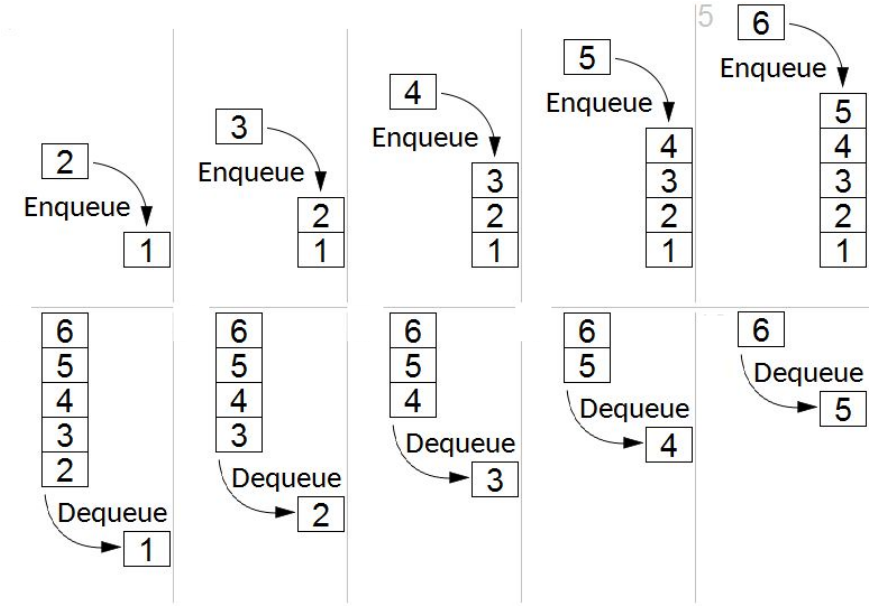


Рисунок 4.1 – Схема работы очереди

Как видно из рисунка 4.1, данные добавляются в конец очереди (tail), а извлекаются из головы очереди (head).

Очередь можно реализовать на массиве и на списке. Когда мы реализуем очередь на списке, нам необходимо использовать однонаправленный список. Операция push – это операция push_back в списке, а операция pop – pop_front в списке. На рисунке 4.2 показана схема работы очереди на двусвязном списке.

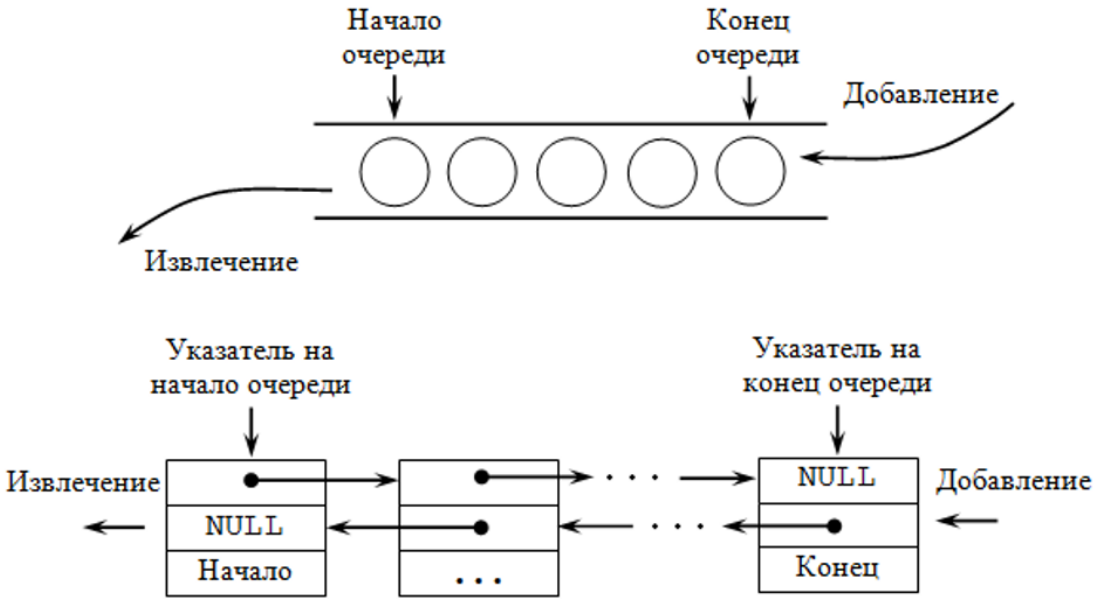


Рисунок 4.2 – Схема работы очереди на двусвязном списке

Очередь на массиве обычно реализуется как кольцевая очередь для того, чтобы не терять свободное место. На рисунке 4.3 изображена схема работы кольцевой очереди на массиве.

head = 0; tail = 10;

a	b	c	d	e	f	g	h	i	j
0	1	2	3	4	5	6	7	8	9

head = 5; tail = 10;

					f	g	h	i	j
0	1	2	3	4	5	6	7	8	9

head = 5; tail = 4;

k	l	m	n		f	g	h	i	j
0	1	2	3	4	5	6	7	8	9

Рисунок 4.3 – Схема работы кольцевой очереди

Как видно из рисунка 4.3, изначально мы имеем два индекса – начала (head) и конца (tail). Операция добавления элемента увеличивает значение tail и добавляет элемент в массив. Операция удаления увеличивает значение head и удаляет элемент из массива. Когда индекс начала или конца достигает конца массива, то необходимо передвинуть этот индекс на начало массива.

Основные операции над очередью:

– включение элемента;

– исключение элемента;

– определение размера очереди;

– очистка очереди;

– неразрушающее чтение.

Пример 4.1. Реализация очереди объектов структурного типа.

```
#include<stdio.h>
#include<stdlib.h>
struct queue{
  char data;
  struct queue *next;
};
typedef struct queue Queue;
typedef Queue *QueueNode;

// Прототипы функций
void printQueue(QueueNode);
int Empty(QueueNode head);
      {// Проверка пустоты очереди
        return head==NULL; }
char delqueue(QueueNode*, QueueNode*);
void enqueue(QueueNode*, QueueNode*, char);
void instructions(void);

int main()
{QueueNode head=NULL, tail=NULL;
 int choice;
 char item;
 instructions();
 printf("?");
 scanf("%d",&choice);
 while(choice!=3){switch(choice)
  {
   case 1:
   printf("Enter a character:");
   scanf("\n%c", &item);
   enqueue(&head, &tail, item);
   printQueue(head);
   break;
```

```
case 2:
  if(!Empty(head)){
    item=delqueue(&head, &tail);
    printf("%c has been dequeued.\n",item);
    }
  printQueue(head); break;
default:
   printf("Invalid choice.\n\n");
   instructions(); break;
  }
printf("?");
scanf("%d", &choice);
}
printf("End of run.\n");
return 0;
}
void instructions(void)
{
 printf("Enter your choice:\n"
  "1. add an item to the queue\n"
  "2. remove an item from the queue\n"
  "3. end\n");
}
void enqueue(QueueNode *head,QueueNode *tail, char val)
{// Добавление нового элемента в конец очереди
 QueueNode newP;
 newP = (QueueNode) malloc(sizeof(Queue));
 if(newP!=NULL)
   {newP->data=val;
    newP->next=NULL;
    if(Empty(*head))
      {*head = newP; // Очередь пуста
      } // Установим указатель нового элемента в начало очереди
    else
      {if((*head)->next==NULL)
         {(*head)->next=newP;
          (*tail)= newP;
         }
       else
         {(*tail)->next = newP;
          (*tail)= newP;
         }
      }
   }
 else
   printf("%c not inserted. No memory available.\n",val);
}

char delqueue(QueueNode *head, QueueNode *tail)
{ // Удаление элемента из начала очереди
 char val;
 QueueNode temp;
```

```
 val=(*head)->data;
 temp=*head;
 *head=(*head)->next;
 if(*head==NULL) *tail=NULL;
 free(temp);
 return val;
}

void printQueue(QueueNode current)
{ // Вывод содержимого очереди на экран
 if(current==NULL)
  printf("Queue is empty.\n\n");
 else
  {
  printf("The queue is:\n");
  while(current!=NULL)
   {
    printf("%c-->",current->data);
    current=current->next;
   }
  printf("NULL\n\n");
  }
}
```

Дек (от англ. deq – double ended queue, т. е. «очередь с двумя концами») – это такой последовательный список, в котором как включение, так и исключение элементов может осуществляться с любого из двух концов списка, как показано на рисунке 4.3.

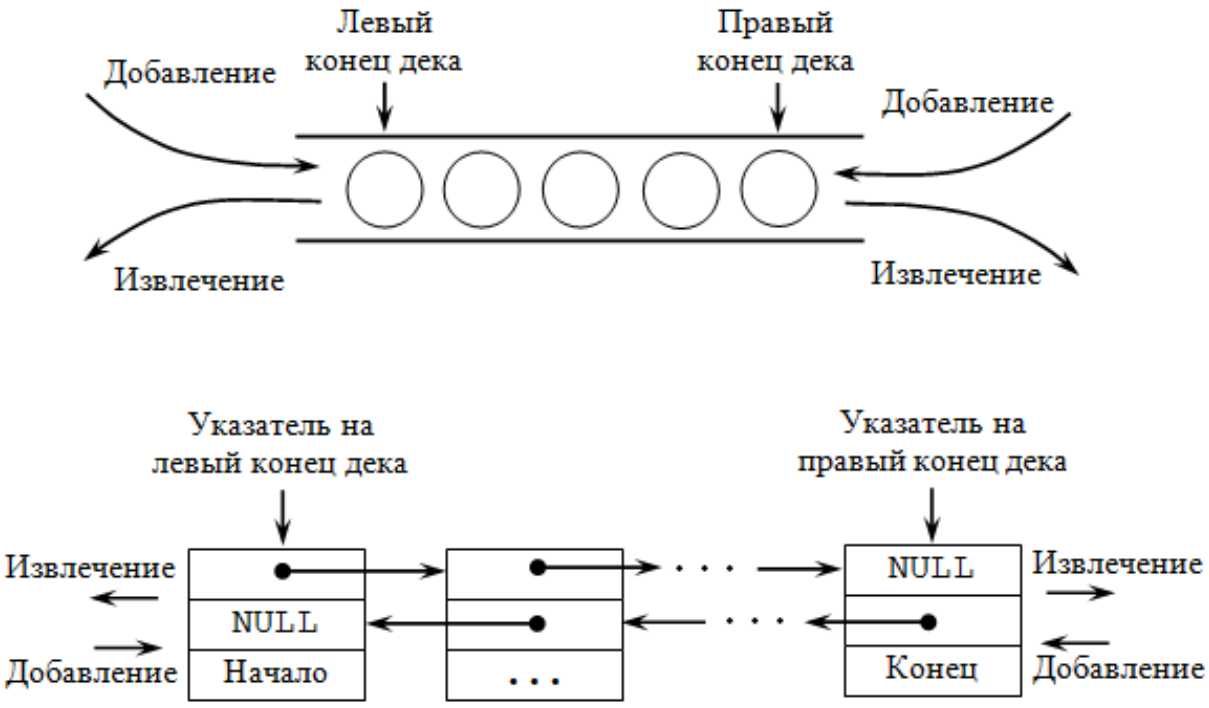


Рисунок 4.3 – Схема работы дека на двусвязном списке

Пример 4.2. Реализация дека с помощью двусвязного списка и класса deque.

```
struct Node {
    int data;
    Node *prev, *next;
     };
#include <bits/stdc++.h>
using namespace std;
// В организации дека участвуют два объекта:
// struct Node и class Deque
// Узел двусвязного списка
struct Node {
  int data;  // Данные
  Node *prev, *next;
  static Node* getnode(int data)
  { // Функция получения нового узла
   Node* newNode=(Node*)malloc(sizeof(Node));
   newNode->data = data;
   newNode->prev = newNode->next = NULL;
   return newNode;
   }
};
// Класс deque
class Deque {
  Node* front;
  Node* rear;
  int Size;
public:
  Deque()
  {front = rear = NULL;
   Size = 0;}
// Операции над деком
  void insertFront(int data);
  void insertRear(int data);
  void deleteFront();
  void deleteRear();
  int getFront(); int getRear();
  int size(); bool isEmpty();
  void erase();};
// Функция проверки пустоты дека
bool Deque::isEmpty() {return (front==NULL);}
// Функция определения количества элементов в деке
int Deque::size() { return Size; }
// Функция вставки элемента в начало дека
void Deque::insertFront(int data)
{Node* newNode = Node::getnode(data);
 if (newNode==NULL) cout << "OverFlow\n";
 else { if (front == NULL)
   rear=front=newNode;
        else {
```

```
            newNode->next = front;
            front->prev = newNode;
            front = newNode;
            }
   Size++;
   }}
// Функция вставки элемента в конец дека
void Deque::insertRear(int data)
{Node* newNode = Node::getnode(data);
 if (newNode == NULL) cout << "OverFlow\n";
 else {if (rear == NULL)
       front = rear = newNode;
       else {
            newNode->prev = rear;
            rear->next = newNode;
            rear = newNode;
        }
       Size++;
      }
}
// Функция удаления элемента с начала дека
void Deque::deleteFront()
{ if (isEmpty()) cout << "UnderFlow\n";
 else {
        Node* temp = front;
        front = front->next;
  // Если был только один элемент, то очередь теперь пуста
        if (front == NULL) rear = NULL;
        else front->prev = NULL;
        free(temp);
        Size--;
    }
}
// Функция удаления элемента с конца дека
void Deque::deleteRear()
{ if (isEmpty()) cout << "UnderFlow\n";
 else {
    Node* temp = rear;
    rear = rear->prev; // Отсоединяем узел
    if (rear == NULL) front = NULL;
    else rear->next = NULL;
    free(temp);
    Size--;
    }
}
// Функция получения элемента из начала дека
int Deque::getFront()
{ if (isEmpty()) return -1;
    return front->data;
}
// Функция получения элемента из конца дека
int Deque::getRear()
```

```
{ if (isEmpty()) return -1;
    return rear->data;
}
// Функция удаления всех элементов из дека
void Deque::erase()
{ rear = NULL;
  while (front != NULL) {
     Node* temp = front;
     front = front->next;
     free(temp);
    }
    Size = 0;
}
int main()
{Deque dq;
 cout << "Insert element '5' at rear end\n";
   dq.insertRear(5);
 cout << "Insert element '10' at rear end\n";
   dq.insertRear(10);
cout << "Rear end element: " << dq.getRear() << endl;
   dq.deleteRear();
cout << "After deleting rear element new rear"
     << " is: " << dq.getRear() << endl;
cout<<"Inserting element '15' at front end \n";
    q.insertFront(15);
cout << "Front end element: " << dq.getFront() << endl;
cout << "Number of elements in Deque: " << dq.size() << endl;
 dq.deleteFront();
cout << "After deleting front element new "
     << "front is: " << dq.getFront() << endl;
return 0;
}
```

Индивидуальные задания

1 Создать очередь для целых (положительных и отрицательных) чисел. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Ввести шесть элементов (положительных и отрицательных). Вывести два первых отрицательных элемента очереди.

2 Создать очередь для массива целых (положительных и отрицательных) чисел. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Ввести шесть элементов. При вводе чисел в очередь попадают только отрицательные элементы. Вывести все элементы очереди.

3 Создать очередь для целых (положительных и отрицательных) чисел. Максимальный размер очереди вводится с клавиатуры. Создать функции для

ввода, вывода и определения размера очереди. Ввести шесть элементов. Вывести два первых элемента очереди. Вывести размер оставшейся очереди.

4 Создать очередь для целых (положительных и отрицательных) чисел. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода, вывода и определения размера очереди. Ввести шесть элементов. Вывести элементы очереди до первого отрицательного (включительно). Вывести размер оставшейся очереди.

5 Создать очередь для символов. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Ввести эталонный символ. Вводить символы с клавиатуры в очередь до встречи эталонного. Вывести все элементы очереди.

6 Создать очередь для символов. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Вводить символы с клавиатуры в очередь. После ввода третьго символа в ответ на каждый вводимый выводить крайний левый символ.

7 Создать очередь для символов. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода, вывода и определения размера очереди. Ввести эталонный символ. Вводить символы с клавиатуры в очередь до встречи эталонного. Вывести размер очереди.

8 Создать очередь для символов. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Вводить символы с клавиатуры в очередь. В случае совпадения вводимого символа с последним элементом очереди выводить первый элемент очереди.

9 Создать очередь для символов. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода, вывода и определения размера очереди. Вводить символы с клавиатуры в очередь. В случае совпадения вводимого символа с последним элементом очереди выводить размер очереди.

10 Создать очередь для символов. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Добавлять символы с клавиатуры в очередь. В случае совпадения вводимого символа с последним элементом очереди удалять и выводить на экран все элементы очереди.

11 Создать очередь для символов. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Вводить символы с клавиатуры. В случае совпадения вводимого символа с последним элементом очереди в очередь его не добавлять, а удалять первый элемент.

12 Создать очередь для символов. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Вводить символы с клавиатуры. В случае совпадения вводимого символа с последним элементом очереди выводить размер очереди.

13 Создать очередь для целых чисел. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Ввести в очередь числа с клавиатуры. После этого перейти в режим, при котором при каждом вводе числа из очереди удаляется первый элемент, и если он совпадает с введенным числом, то он добавляется в очередь.

14 Создать очередь для целых чисел. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Ввести в очередь числа с клавиатуры. После этого перейти в режим, при котором при каждом вводе числа выводится первый элемент очереди, и если он не совпадает с введенным числом, то оно заносится в очередь.

15 Создать очередь для целых чисел. Максимальный размер очереди вводится с клавиатуры. Создать функции для ввода и вывода элементов очереди. Ввести в очередь числа с клавиатуры. После этого перейти в режим ввода, при котором перед добавлением элемента происходит удаление одного элемента.

16 Создать дек для целых (положительных и отрицательных) чисел. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Ввести шесть элементов (положительных и отрицательных). Вывести два первых правых отрицательных элемента дека.

17 Создать дек для массива целых чисел. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Ввести три элемента справа и три слева. При вводе чисел в дек попадают только отрицательные элементы. Вывести все элементы дека.

18 Создать дек для целых (положительных и отрицательных) чисел. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера дека. Ввести три элемента справа и три слева. Вывести по одному элементу справа и слева. Вывести размер оставшегося дека.

19 Создать дек для целых (положительных и отрицательных) чисел. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера дека. Ввести три элемента справа и три слева. Вывести элементы дека справа до первого отрицательного (включительно). Вывести размер оставшегося дека.

20 Создать дек для символов. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Ввести эталонный символ. Вводить символы в дек поочередно справа и слева до встречи эталонного. Вывести все элементы дека.

21 Создать дек для символов. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Вводить элементы в дек поочередно справа и слева. При этом в ответ на добавление элемента с противоположной стороны дека один элемент удаляется.

22 Создать дек для символов. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера де-

ка. Ввести эталонный символ. Вводить символы в дек поочередно справа и слева до встречи эталонного. Вывести размер дека.

23 Создать дек для символов. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Добавлять символы в дек поочередно справа и слева. В случае совпадения добавляемого символа с элементом на другом конце дека выводить его на экран.

24 Создать дек для символов. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера дека. Добавлять символы в дек поочередно справа и слева. В случае совпадения добавляемого символа с элементом на другом конце дека выводить размер дека.

25 Создать дек для символов. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Добавлять символы в дек поочередно справа и слева. В случае совпадения добавляемого символа с концом дека вывести на экран все элементы со стороны совпавшего.

26 Создать дек для символов. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Добавлять символы в дек поочередно справа и слева. В случае совпадения введенного символа с элементом соответствующего конца дека его не добавлять.

27 Создать дек для символов. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера дека. Ввести в дек символы с клавиатуры. Добавлять символы с клавиатуры. В случае совпадения вводимого символа с одним из концов дека выводить на экран его размер.

28 Создать дек для целых чисел. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. При каждом вводе числа слева удаляется элемент, и если он не совпадает с введенным числом, то введенное число добавляется справа.

29 Создать дек для целых чисел. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Ввести в дек числа с клавиатуры. После этого перейти в режим, при котором слева удаляется элемент, и если он совпадает с введенным числом, то введенное число добавляется справа, а иначе – слева.

30 Создать дек для плавающих чисел. Максимальный размер дека вводится с клавиатуры. Создать функции для ввода и вывода элементов дека. Ввести в дек числа с клавиатуры. После этого перейти в режим ввода, при котором перед занесением элемента происходит удаление элемента слева.

31 Дан текстовый файл. Проанализировав в программе содержимое файла, выбрать из него числа и занести в очередь. Вывести содержимое очереди на экран и посчитать сумму этих чисел. Решение в программе оформить через подпрограммы.

32 Дан текстовый файл. Проанализировав в программе содержимое файла, выбрать из него имена и занести в очередь. Вывести содержимое

очереди на экран и посчитать количество элементов образованной очереди. Решение в программе оформить через подпрограммы.

33 Проверить на равенство две очереди. Решение в программе оформить через подпрограммы.

34 Найти среди трех (четырех, пяти) очередей две одинаковые. Решение в программе оформить через подпрограммы.

35 Организовать три очереди с одинаковым количеством элементов, содержащие соответственно имена, отчества и фамилии людей. Составить очередь из элементов, содержащих наиболее полную информацию о людях, воспользовавшись уже созданными очередями и запросив какую-то дополнительную информацию. Решение в программе оформить через подпрограммы.

36 Создать файл символьного типа. Организовывая очереди по N элементов, создать файл слов по N символов в каждом. Решение в программе оформить через подпрограммы.

37 Создать файл целого типа. Проанализировав в программе содержимое файла, создать одну очередь однозначных чисел, а вторую – двузначных. Перемножить соответственные элементы двух очередей и организовать третью очередь. Результат вывести в текстовый файл. Решение в программе оформить через подпрограммы.

38 Используя очередь, проверить, какие строки текстового файла являются симметричными. Решение в программе оформить через подпрограммы.

39 Используя очередь, проверить на равенство два текстовых файла. Решение в программе оформить через подпрограммы.

40 Создать две очереди. Проверить, является ли одна из очередей частью другой. Решение в программе оформить через подпрограммы.

Лабораторная работа № 5.
Стеки и основные операции над ними

Цель работы – познакомиться со структурой данных «стек» и основными операциями над ним. Научиться реализовывать стек через линейный однонаправленный список.

Ход выполнения лабораторной работы должен быть отражен в отчете. Отчет должен содержать титульный лист, номер задания, коды программ, скриншоты с результатами выполнения программы.

Методические указания

Стек (от англ. *stack*) – структура данных, представляющая собой упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека (рисунок 5.1).

Притом первым из стека удаляется элемент, который был помещен туда последним, т. е. в стеке реализуется стратегия «последним вошел – первым вышел» (last-in, first-out – LIFO).

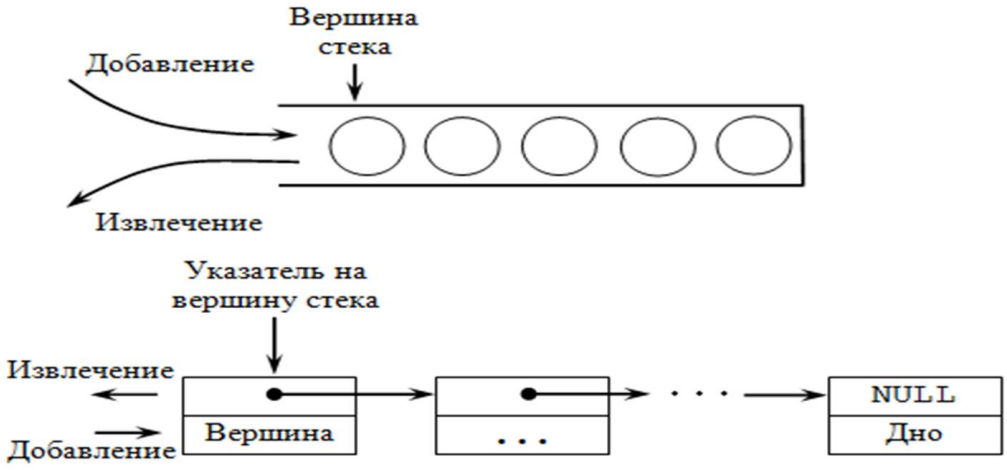


Рисунок 5.1 – Структура стека

Чаще всего стек реализуется с помощью списочной структуры. Каждый элемент стека содержит информационные поля и указатель на следующий элемент стека (см. рисунок 5.1). Указатель последнего элемента стека равен NULL, т. к. он не указывает ни на какой элемент. Для корректного задания стека необходимо определить указатель на вершину стека.

Основные операции над стеком – включение нового элемента (push) и исключение элемента из стека (pop).

Полезными могут быть также вспомогательные операции:

- определение текущего числа элементов в стеке;

– очистка стека;

– неразрушающее чтение элемента из вершины стека, которое может быть реализовано как комбинация основных операций:

```
x=pop(stack0);
push(stack0,x);
```

Программная реализация стека через линейный однонаправленный список. При реализации стека каждый новый узел необходимо создать динамически. Для организации стека необходимо хранить указатель вершины стека, который указывает на первый элемент списка. Если стек пуст, то списка не существует, и указатель принимает значение NULL.

Пример 5.1. Реализация стека через линейный однонаправленный список.

```
#include <stdio.h>
#include <stdlib.h>
// Узел связного списка
struct Node
{ int data;     // Целочисленные данные
  struct Node* next; // Указатель на следующий узел
};
int nodesCount;
// Функция для добавления элемента `x` в stack
int push(struct Node **top, int x) // top - указатель на стек
{struct Node* node = NULL;// node – указатель на новый узел
 node=(struct Node*)malloc(sizeof(struct Node));
 if (!node) // Достаточно ли памяти
  { printf("Heap Overflow\n");return -1;}
 printf("Inserting %d\n", x);
 node->data = x; // Устанавливаем данные в выделенном узле
 node->next = *top; // Устанавливаем указатель next нового узла
так,
                 //чтобы он указывал на текущий верхний узел списка
 *top = node; // Обновим указатель на вершину
 nodesCount += 1; // Увеличим размер stack на 1
}

// Функция для проверки, пуст stack или нет
int isEmpty(struct Node* top) {
    return top == NULL;}

// Функция для возврата верхнего элемента stack
int peek(struct Node *top)
{if (!isEmpty(top)) {// Проверка на пустой stack
        return top->data; }
 else { printf("The stack is empty\n");
        return -1;
      }
}
```

```
// Функция для возврата числа узлов stack
int size() { return nodesCount; // Глобальная переменная
           }

// Функция для извлечения элемента из вершины
int pop(struct Node** top)
{struct Node *node;
 if (*top == NULL) // Проверка на пустоту
   {printf("Stack Underflow\n");
    return -1;}
 int x = peek(*top); // Получаем данные верхнего узла
 printf("Removing %d\n", x);
 node = *top;
 *top = (*top)->next; // Обновляем верхний указатель, чтобы он
указывал
                      //на следующий узел
 nodesCount -= 1; // Уменьшаем размер stack на 1
 free(node); // Освобождаем выделенную память
 return x;
}

int main(void)
{struct Node* top = NULL;
 push(&top, 1);
 push(&top, 2);
 push(&top, 3);
 printf("The top element is %d\n", peek(top));
 pop(&top);
 pop(&top);
 pop(&top);
 if (isEmpty(top)) {
  printf("The stack is empty\n");
  }
 else {
  printf("The stack is not empty\n");
    }
 return 0;
}
```

Индивидуальные задания

1 Создать стек для целых чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Добавить шесть элементов. Удалить и вывести на экран два элемента.

2 Создать стек для целых (положительных и отрицательных) чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Ввести с клавиатуры шесть элементов. При вводе чисел в стек попадают только отрицательные элементы. Вывести все элементы стека.

3 Создать стек для целых чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера стека. Ввести с клавиатуры шесть элементов. Удалить два элемента. Вывести размер стека.

4 Создать стек для целых (положительных и отрицательных) чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера стека. Вводить с клавиатуры числа, причем в стек должны добавляться поочередно положительные и отрицательные числа.

5 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Ввести эталонный символ. Вводить символы с клавиатуры в стек до встречи эталонного. Вывести все элементы стека.

6 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Добавить символы с клавиатуры в стек. После добавления пятого символа перед добавлением удалять элемент из стека.

7 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера стека. Ввести эталонный символ. Вводить символы с клавиатуры в стек до встречи эталонного. Вывести размер стека.

8 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Добавлять символы с клавиатуры в стек. В случае совпадения вводимого символа с вершиной стека вытолкнуть и вывести на экран его значение.

9 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера стека. Вводить символы с клавиатуры в стек. В случае совпадения вводимого символа с вершиной стека вывести размер стека.

10 Создать два стека для символов. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры в первый стек. В случае совпадения вводимого символа с вершиной стека вводить во второй стек.

11 Создать два стека для символов. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры в стеки поочередно.

12 Создать стек для символов и стек для чисел. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры. Символ попадает в первый стек, а его численное представление – во второй.

13 Создать два стека для символов. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры. Прописные буквы попадают в первый стек, строчные – во второй, остальные символы пропускаются.

14 Создать два стека для символов. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры. Прописные буквы преобразуются в строчные и попадают в первый стек, строчные преобразуются в прописные и попадают во второй, остальные символы пропускаются.

15 Создать стек для целых чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры. Числовое представление символа попадает в стек.

16 Создать стек для целых чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Добавить шесть элементов. Удалить и вывести на экран два элемента. Задачу решить с использованием механизма указателей.

17 Создать стек для целых (положительных и отрицательных) чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Ввести шесть элементов. При вводе чисел в стек попадают только отрицательные элементы. Вывести все элементы стека. Задачу решить с использованием механизма указателей.

18 Создать стек для целых чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера стека. Ввести с клавиатуры шесть элементов. Удалить два элемента. Вывести размер стека. Задачу решить с использованием механизма указателей.

19 Создать стек для целых (положительных и отрицательных) чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера стека. Вводить с клавиатуры числа, причем в стек должны добавляться поочередно положительные и отрицательные числа. Задачу решить с использованием механизма указателей.

20 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Ввести эталонный символ. Вводить символы с клавиатуры в стек до встречи эталонного. Вывести все элементы стека. Задачу решить с использованием механизма указателей.

21 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Добавить символы с клавиатуры в стек. После добавления пятого символа перед добавлением следуюшего удалять элемент из стека. Задачу решить с использованием механизма указателей.

22 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера стека. Ввести эталонный символ. Вводить символы с клавиатуры в стек до встречи эталонного. Вывести размер стека. Задачу решить с использованием механизма указателей.

23 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Добавлять символы с клавиатуры в стек. В случае совпадения вводимого символа с вер-

шиной стека вытолкнуть его и вывести на экран ее. Задачу решить с использованием механизма указателей.

24 Создать стек для символов. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода, вывода и определения размера стека. Вводить символы с клавиатуры в стек. В случае совпадения вводимого символа с вершиной стека вывести размер стека. Задачу решить с использованием механизма указателей.

25 Создать два стека для символов. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры в первый стек. В случае совпадения вводимого символа с вершиной стека вводить во второй стек. Задачу решить с использованием механизма указателей.

26 Создать два стека для символов. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры в стеки поочередно. Задачу решить с использованием механизма указателей.

27 Создать стек для символов и стек для чисел. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры. Символ попадает в первый стек, а его численное представление – во второй. Задачу решить с использованием механизма указателей.

28 Создать два стека для символов. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры. Прописные буквы попадают в первый стек, строчные – во второй, остальные символы пропускаются. Задачу решить с использованием механизма указателей.

29 Создать два стека для символов. Максимальный размер стеков вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры. Прописные буквы преобразуются в строчные и попадают в первый стек, строчные преобразуются в прописные и попадают во второй, остальные символы пропускаются. Задачу решить с использованием механизма указателей.

30 Создать стек для целых чисел. Максимальный размер стека вводится с клавиатуры. Создать функции для ввода и вывода элементов стека. Вводить символы с клавиатуры. Числовое представление символа попадает в стек. Задачу решить с использованием механизма указателей.

31 Создать текстовый файл, содержащий текстовую и числовую информацию. Используя стек, создать другой текстовый файл, в котором числа были бы записаны в обратном порядке.

32 Создать текстовый файл, содержащий текстовую информацию. Используя стек, создать другой текстовый файл, в котором слова будут записаны в обратном порядке.

33 Создать текстовый файл, содержащий некоторую информацию. Используя стек, создать другой текстовый файл, в котором строки будут записаны в обратном порядке.

34 Создать текстовые файлы, содержащие один текстовую, а другой числовую информацию (количество слов и чисел должно быть одинаковым). Используя стек, создать другой текстовый файл, в котором числа и слова чередуются и записаны в обратном порядке.

35 Создать текстовые файлы, содержащие один – текстовую, а другой – числовую информацию (количество слов и чисел должно быть одинаковым). Используя стек, создать другой текстовый файл, в котором числа и слова чередуются, а порядок чисел и слов сохранен.

36 Создать текстовые файлы, содержащие один текстовую, а другой числовую информацию (количество слов и чисел может быть неодинаковым). Используя стек, создать другой текстовый файл, в котором числа и слова чередуются и записаны в обратном порядке («лишние» числа или слова записаны в конце файла).

37 В файле находится текст программы на Паскале. Используя стек, проверить правильность вложений циклов в этой программе.

38 В файле находится текст программы на Паскале. Используя стек, проверить правильность вложений операторных скобок (begin-end) в этой программе.

39 В файле записан текст, сбалансированный по круглым скобкам. Требуется для каждой пары соответствующих открывающей и закрывающей скобок напечатать номера их позиций в тексте, упорядочив пары номеров по возрастанию номеров позиций закрывающих скобок. Например, для текста a+(45-f(x)×(b-c)) надо напечатать 8 10, 12 16, 3 17.

41 Написать программу слияния двух стеков, содержащих возрастающую последовательность целых положительных чисел, в третий стек так, чтобы его элементы располагались также в порядке возрастания.

42 Написать программу формирования стека, куда помещаются целые положительные числа, вводимые с клавиатуры (процесс ввода должен прекращаться, как только среди вводимых чисел появляется отрицательное число, после этого программа должна вывести на экран содержимое стека в том же порядке, в котором они были введены).

Лабораторная работа № 6.
Бинарные деревья

Цель работы – познакомиться с бинарными деревьями, их концепцией и основными свойствами. Научиться представлять различные способы реализации бинарных деревьев, изучить оптимальные подходы по проектированию и использованию бинарных деревьев.

Ход выполнения лабораторной работы должен быть отражен в отчете. Отчет должен содержать титульный лист, номер задания, коды программ, скриншоты с результатами выполнения программы.

Методические указания

Бинарное дерево – это нелинейная структура данных, представляющая собой иерархическую структуру, в которой каждый узел может иметь не более двух дочерних узлов: левого и правого. Бинарные деревья широко используются в информатике для различных целей, таких, как:

– хранение иерархических данных, например файловой системы или организационной структуры;

– реализация алгоритмов поиска и сортировки;

– анализ и компрессия данных;

– искусственный интеллект и машинное обучение.

Формально бинарное дерево определяется как конечное множество узлов, удовлетворяющее следующим свойствам:

1 Наличие корневого и конечного узлов. Корневой узел – это самый верхний узел в бинарном дереве. Конечный узел может быть определен как узел без дочерних узлов.

2 Левый и правый дочерние узлы. Дочерние узлы – это узлы, которые являются непосредственными потомками родительского или корневого узла. Каждый узел в бинарном дереве может иметь не более двух дочерних узлов: левого и правого.

3 Отсутствие циклов. В дереве не может быть циклов, т. е. путей, которые начинаются и заканчиваются в одном и том же узле.

Цель данной работы заключается в разработке бинарного дерева на языке программирования Си, а также в написании функций, которые позволяют разработать структуру данных для создания узла бинарного дерева, содержащую данные и указатели на левого и правого потомков, а также выполнить следующие операции над деревом:

– создание дерева;

– добавление и удаление элементов в бинарном дереве;

– вывод на экран;

– подсчет максимальной суммы пути;

– обход дерева.

В языке программирования Си бинарные деревья могут быть реализованы с использованием структуры Node (пример 6.1), где каждый узел содержит значение данных и указатели левого и правого потомков. Это позволяет эффективно хранить и обрабатывать данные, осуществлять поиск, вставку и удаление элементов.

Внутри структуры Node определены три поля:

1 int data – это целочисленное значение, которое будет храниться в узле дерева.

2 struct Node* left – это указатель на левого потомка текущего узла. Каждый узел может иметь левого потомка, который также является узлом структуры Node.

3 struct Node* right – это указатель на правого потомка текущего узла. Каждый узел может иметь правого потомка, который также является узлом структуры Node.

После определения структуры Node используется ключевое слово typedef, которое позволяет создать псевдоним Node для этой структуры. Это делает использование структуры более удобным и позволяет обращаться к ней просто как к Node, без необходимости указывать ключевое слово struct.

Пример 6.1. Структура Node.

```
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
     } Node;
```

Рассмотрим функцию createNode (пример 6.2).

Пример 6.2. Функция createNode.

```
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value; // Значение value передается в поле
                           //data нового узла
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

Функция createNode принимает один аргумент типа int и возвращает указатель на узел Node. Функция принимает значение value, которое будет храниться в создаваемом узле. Функция malloc(sizeof(Node)) – здесь происходит выделение памяти для нового узла, sizeof(Node) возвращает размер структуры Node в байтах. Оператор newNode->left = NULL – левый потомок нового узла инициализируется значением NULL, т. к. при создании узла у него еще нет по-

томков. Оператор newNode->right = NULL – правый потомок нового узла также инициализируется значением NULL. Оператор return newNode – новый узел, который был создан и инициализирован данными, возвращается из функции как результат выполнения. Теперь этот узел может быть использован в дальнейшей работе с бинарным деревом. В примере 6.3 представлена функция вставки узла в дерево.

Пример 6.3. Функция insert принимает два аргумента: указатель на корневой узел root и значение value, которое нужно добавить в дерево. Функция не возвращает никакого значения (тип void).

```
void insert(Node* root, int value) {
    if (value < root->data) // Меньше ли значение value значения
                            //корневого узла root
      {
        if (root->left == NULL) // Пуста ли левая ветвь корневого
                                //узла
// Если левая ветвь пуста, то новый узел создается с помощью
//функции createNode(value) и добавляется как левый потомок
//корневого узла
         {
            root->left = createNode(value);
         }
        else {
            insert(root->left, value); // Вызывается рекурсивно
              //функция insert для левого потомка корневого узла
             }
      }
    else { // Если значение value больше или равно значению
//корневого узла, то новый узел добавляется в правую ветвь дерева.
//Здесь также используется аналогичная логика для проверки
//пустоты правой ветви и рекурсивного вызова функции insert для
//правого потомка корневого узла
        if (root->right == NULL) {
            root->right = createNode(value);
        }
        else {
            insert(root->right, value);
        }
    }
}
```

В примере 6.4 представлена функция вывода на экран бинарного дерева.

Пример 6.4. Функция printTree принимает два аргумента: указатель на корневой узел root и целочисленное значение level, представляющее текущий уровень вложенности узла. Функция не возвращает никакого значения (тип void).

```
void printTree(Node* root, int level) {
    if (root == NULL) // Является ли текущий узел root пустым
      {
        return;
       }
    printTree(root->right, level + 1); // Рекурсивный вызов функции
//printTree для правого потомка корневого узла с увеличением
//уровня на 1. Это позволяет спускаться по правой ветви дерева для
//вывода значений узлов
    for (int i = 0; i < level; i++) // Отступ (пробелы)
                //в зависимости от текущего уровня вложенности узла
      {
        printf("   ");
      }
    printf("%d\n", root->data); // Вывод значения данных текущего
   //узла. Значение узла выводится после отступов, чтобы
   //отобразить его на правильном уровне вложенности
    printTree(root->left, level + 1); // Рекурсивный вызов функции
   //printTree для левого потомка корневого узла с увеличением
   //уровня на 1. Это позволяет спускаться по левой ветви дерева
   //для вывода значений узлов
}
```

В примере 6.5 представлена функция поиска минимального элемента.

Пример 6.5. Функция findMin принимает указатель на корневой узел root в качестве аргумента и возвращает указатель на минимальный элемент.

```
// Поиск минимального элемента
struct Node* findMin(struct Node* root) {
    if (root == NULL) {
        // Если дерево пустое, возвращаем NULL
        return NULL;
    }

    if (root->left == NULL) {
        // Если левое поддерево пустое, текущий узел является ми-
нимальным
        return root;
    }

    // Рекурсивно ищем минимальный элемент в левом поддереве
    return findMin(root->left);
}
```

В примере 6.6 представлена функция удаления элемента из дерева.

Пример 6.6. Функция deleteElement принимает в качестве аргументов указатель на корневой узел root и значение удаляемого элемента. Функция возвращает указатель на новое состояние дерева.

```
struct Node* deleteElement(struct Node* root, int data) {
  if (root == NULL) {
     return NULL;
  }
  if (data < root->data) {
      root->left = deleteElement(root->left, data);
  }
  else if (data > root->data) {
      root->right = deleteElement(root->right, data);
      }
      else {
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        }
        else if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }
        else {
          struct Node* minRight = findMin(root->right);
          root->data = minRight->data;
          root->right =deleteElement(root->right, minRight->data);
        }
    }
  return root;
}
```

В примере 6.7 представлена функция обхода бинарного дерева в порядке возрастания значений.

Пример 6.7. Функция inOrderTraversal принимает указатель на корневой узел root в качестве аргумента и не возвращает никакого значения.

```
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    } }
```

В примере 6.8 представлена функция определения суммы узлов максимального пути.

Пример 6.8. Функция findMaxSumPath принимает указатель на корневой узел root в качестве аргумента и возвращает целочисленное значение – максимальную сумму пути.

```
int findMaxSumPath(Node* root)
{
    if (root == NULL) // Является ли текущий узел root пустым
  //(равным NULL)? Если это условие истинно, то функция возвращает
  //0, т. к. пустое поддерево не добавляет никакой суммы к пути
     {
        return 0;
     }
    int leftSum = findMaxSumPath(root->left);  // Рекурсивно
  //вызывается функция findMaxSumPath для левого потомка текущего
  //узла root, и результат сохраняется в переменной leftSum. Это
  //позволяет найти максимальную сумму пути в левом поддереве
    int rightSum = findMaxSumPath(root->right);  // Рекурсивно
  //вызывается функция findMaxSumPath для правого потомка текущего
  //узла root, и результат сохраняется в переменной rightSum.
  //Это позволяет найти максимальную сумму пути в правом поддереве
    return root->data + (leftSum > rightSum ? leftSum : rightSum);
  // Вычисляется максимальная сумма пути через текущий узел root.
  //Сумма равна значению данных текущего узла root, плюс
  //максимальная из сумм путей через левое и правое поддеревья
  //(leftSum и rightSum). Таким образом, выбирается путь
  //с максимальной суммой, который проходит через текущий узел
}
```

В примере 6.9 представлена главная функция программы.

Пример 6.9. Пример главной функции программы работы с бинарными деревьями.

```
int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    struct Node* root = NULL;
    int choice=1, value, sum;

while (choice!=6) {
  printf("1. Вставить элемент в дерево\n");
  printf("2. Определить максимальную сумму пути \n");
  printf("3. Удалить элемент из дерева \n");
  printf("4. Вывести все элементы дерева\n");
  printf("5. Вывести элементы в порядке возрастания\n");
```

```
  printf("6. Выйти из программы\n");
  printf("Введите выбор: ");
  scanf("%d", &choice);

  switch (choice) {
    case 1:
      printf("Введите значение для вставки: ");
      scanf("%d", &value);
      root=insert(root, value);
      break;
    case 2:
      sum = findMaxSumPath(root);
      printf("Максимальная сумма пути = %d", sum);
      break;
    case 3:
      printf("Введите значение для удаления: ");
      scanf("%d", &value);
      root=deleteElement(root, value);
      printf("Удален элемент %d\n", value);
      break;
    case 4:
      printf("Элементы дерева: ");
      printf("\n");
      printTree(root,1);
      printf("\n");
      break;
    case 5:
      printf("Элементы в порядке возрастания: ");
      printf("\n");
      inOrderTraversal(root);
      printf("\n");
      break;
    case 6:
      freeMemory(root); break;
    default:
       printf("Неверный выбор\n");
   }
  }
 return 0;
}
```

Индивидуальные задания

1 Написать функцию для проверки, является ли данное бинарное дерево симметричным (симметричный обход возвращает элементы в палиндромном порядке).

2 Написать функцию для определения высоты заданного узла в бинарном дереве.

3 Реализовать функцию для подсчета суммы всех значений в бинарном дереве.

4 Реализовать алгоритм проверки, является ли данное бинарное дерево симметричным относительно вертикальной оси (зеркально симметричным).

5 Реализовать функцию для поиска заданного значения в бинарном дереве. Функция должна возвращать True, если значение найдено, и False в ином случае.

6 Написать функцию, которая определяет, есть ли в бинарном дереве хотя бы два одинаковых элемента.

7 Написать функцию для определения глубины (максимальной высоты) бинарного дерева.

8 Реализовать алгоритм построения бинарного дерева из списка уникальных элементов.

9 Реализовать функцию для подсчета суммы всех значений в бинарном дереве.

10 Найти максимальный элемент бинарного дерева и количество повторений максимального элемента в данном дереве.

11 Реализовать алгоритм симметричного обхода бинарного дерева без использования рекурсии (с использованием стека или очереди).

12 Написать функцию для проверки, является ли данное бинарное дерево полным (каждый узел имеет либо два дочерних, либо является листом).

13 Реализовать алгоритм для поиска второго минимального значения в бинарном дереве.

14 Реализовать алгоритм для поиска k-го наименьшего значения в бинарном дереве.

15 Реализовать алгоритм для поиска наибольшего значения, меньшего заданного значения в бинарном дереве.

16 Ввести с клавиатуры последовательнось чисел с размерностью N. Создать дерево из N вершин, в котором каждая вершина (кроме корня) является правой дочерней вершиной. Каждой создаваемой вершине присваивать очередное значение из исходного набора. Вывести указатель на корень созданного дерева.

17 Ввести с клавиатуры последовательнось чисел с размерностью N. Создать дерево из N вершин, в котором каждая внутренняя вершина имеет только одну дочернюю вершину, причем правые и левые дочерние вершины чередуются (корень имеет левую дочернюю вершину). Каждой создаваемой вершине присваивать очередное значение из исходного набора. Вывести указатель на корень созданного дерева.

18 Ввести с клавиатуры последовательнось чисел с размерностью N. Создать дерево из N вершин, в котором каждая внутренняя вершина имеет только одну дочернюю вершину, причем если значение вершины является нечетным, то она имеет левую дочернюю вершину, а если четным, то правую. Каждой создаваемой вершине присваивать очередное значение из исходного набора. Вывести указатель на корень созданного дерева.

19 Ввести с клавиатуры последовательнось чисел с размерностью N (N – четное). Создать дерево из N вершин, в котором каждая левая дочерняя вершина является листом, а правая дочерняя вершина является внутренней. Для каждой внутренней вершины в начале создавать левую дочернюю вершину, а затем правую (если она существует); каждой создаваемой вершине присваивать значение из исходного набора. Вывести указатель на корень созданного дерева.

20 Ввести с клавиатуры последовательнось чисел с размерностью N (N – четное). Создать дерево из N вершин со следующей структурой: если вершина дерева является внутренней, то в случае, если она имеет нечетное значение, ее левая дочерняя вершина должна быть листом, а в случае, если она имеет четное значение, листом должна быть ее правая вершина. Для каждой внутренней вершины в начале создавать дочернюю вершину-лист, а затем дочернюю внутреннюю вершину (если данная вершина существует); каждой создаваемой вершине присваивать значение из исходного набора. Вывести указатель на корень созданного дерева.

21 Ввести с клавиатуры целое число N (> 0). Создать дерево, корень которого имеет значение N, а вершины обладают следующими свойствами. Вершина с четным значением K имеет левую дочернюю вершину со значением K/2 (символ «/» обозначает операцию деления нацело) и не имеет правой дочерней вершины. Вершина с нечетным значением K имеет две дочерние вершины: левую со значением K / 2 и правую со значением K – K/2. Вершина со значением 1 является листом. Вывести указатель на корень созданного дерева.

22 Ввести с клавиатуры целые положительные числа L, N (N > L) и последовательность из N чисел. Создать бинарное дерево глубиной L, содержащее вершины со значениями из исходного набора. Вершины добавлять к дереву в префиксном порядке, используя алгоритм, который для каждой вершины уровня, не превышающего L, вначале создает саму вершину со значением из исходного набора, затем ее левое поддерево соответствующей глубины, а затем ее правое поддерево. Если для заполнения дерева глубиной L требуется менее N вершин, то оставшиеся числа из исходного набора не использовать. Вывести указатель на корень созданного дерева.

23 Ввести с клавиатуры целое положительное число N (> 0) и последовательность из N чисел. Создать идеально сбалансированное бинарное дерево из N вершин с введенными значениями (т. е. дерево, для каждой вершины которого количество вершин в его левом и правом поддереве отличается не более чем на 1) и вывести указатель на его корень. Для создания дерева использовать рекурсивный алгоритм, который создает вершину дерева с очередным значением, после чего вызывается для создания левого поддерева с N/2 (символ «/» обозначает операцию деления нацело) вершинами и правого поддерева с N – 1 – N/2 вершинами.

24 Ввести с клавиатуры целое положительное число N (> 0). Создать идеально сбалансированное дерево из N вершин и вывести указатель на его корень. Значение каждой вершины положить равным уровню этой вершины

(например, корень дерева должен иметь значение 0, его дочерние вершины – значение 1 и т. д.). При формировании дерева использовать алгоритм, описанный в задании 22.

25 Дан указатель P1 на корень непустого дерева. Создать копию данного дерева и вывести указатель P2 на корень созданной копии.

26 Дан указатель P1 на корень непустого дерева. Удвоить значение каждой вершины дерева.

27 Дан указатель P1 на корень непустого дерева. Для каждой вершины дерева с четным значением уменьшить ее значение в два раза.

28 Дан указатель P1 на корень непустого дерева. Добавить 1 к значению каждого листа дерева и вычесть 1 из значения каждой внутренней вершины.

29 Дан указатель P1 на корень непустого дерева. Для каждой вершины дерева, имеющей две дочерние вершины, поменять местами значения дочерних вершин (т. е. значения их полей Data).

30 Дан указатель P1 на корень непустого дерева. Для всех внутренних вершин дерева поменять местами их левые и правые дочерние вершины (т. е. значения полей Left и Right).

Лабораторная работа № 7.
Хеширование

Цель работы – изучить методы использования механизма хеширования для сохранения (выборки) информации из массивов данных. Рассмотреть виды хеширования, основные операции с использованием хеширования.

Ход выполнения лабораторной работы должен быть отражен в отчете. Отчет должен содержать титульный лист, номер задания, коды программ, скриншоты с результатами выполнения программы.

Методические указания

Хеширование – хороший пример баланса между временем и объемом памяти. Если бы не было ограничения на объем используемой памяти, любой поиск можно было бы выполнить с помощью всего лишь одного обращения к памяти, просто используя ключ в качестве адреса памяти. Однако обычно этот идеальный случай недостижим, поскольку для длинных ключей может потребоваться огромный объем памяти. С другой стороны, если бы не было ограничений на время выполнения, можно было бы обойтись минимальным объемом памяти, пользуясь методом последовательного поиска. Хеширование представляет собой способ использования приемлемого объема как памяти, так и времени, и достижения баланса между этими двумя крайними требованиями. В частности, можно поддерживать любой баланс, просто меняя размер таблицы, а не переписывая код и не выбирая другие алгоритмы.

Хеширование – это разбиение общего (базового) набора уникальных ключей элементов данных на непересекающиеся наборы с определенным свойством. Хеширование (hashing – «перемешивание») – процесс создания ключей, определяющих местоположение данных в массиве. Ключи определяются с помощью хеш-функции.

Алгоритм хеширования определяет положение искомого элемента в массиве по значению его ключа, полученного хеш-функцией.

Пример данных – книга. В этом случае буквы алфавита могут быть приняты за ключи поиска, т. е. основным элементом алгоритма хеширования является ключ (key). В большинстве приложений ключ обеспечивает косвенную ссылку на данные.

Хеш-таблица и хеш-функции. Ключи (keys) определяют данные массива, и на их основании информация о данных записывается в таблицу, называемую хеш-таблицей.

Хеш-таблица – это структура данных, использующая хеш-функцию для хранения и быстрого поиска значений. Она представляет собой обычный массив с необычной реализацией, где каждый элемент массива называется ячейкой или слотом. Каждый слот содержит пару «ключ – значение». Ключ использует-

ся для вычисления хеша, который затем используется для определения индекса ячейки в массиве. Значение связано с ключом и сохраняется в соответствующей ячейке. Хеш-структуру считают обобщением массива, который обеспечивает быстрый прямой доступ к данным по индексу.

Хеш-функция – это функция, которая принимает на вход некоторые данные (например, ключ) и вычисляет хеш-код – числовое значение фиксированного размера. Хеш-функция должна быть быстрой и эффективной, чтобы обеспечить высокую производительность операций в хеш-таблице:

$$i = h(key),$$

где key – преобразуемый ключ; i – получаемый индекс таблицы, т. е. ключ отображается во множество целых чисел (**хеш-адреса**), которые впоследствии используются для доступа к данным.

Идеальная хеш-функция должна обладать следующими свойствами:

1 Детерминированность (для одного и того же входа хеш-функция всегда должна выдавать один и тот же хеш-код).

2 Равномерное распределение (хеш-функция должна равномерно распределять значения по всем возможным хеш-кодам, чтобы избежать коллизий – ситуаций, когда двум различным ключам соответствует один и тот же хеш-код).

3 Эффективность (вычисление хеш-кода должно быть быстрым).

Коллизия – это ситуация, когда двум разным ключам соответствует один и тот же хеш-код (рисунок 7.1). Это может произойти из-за ограниченного количества возможных хеш-кодов и бесконечного количества возможных ключей.

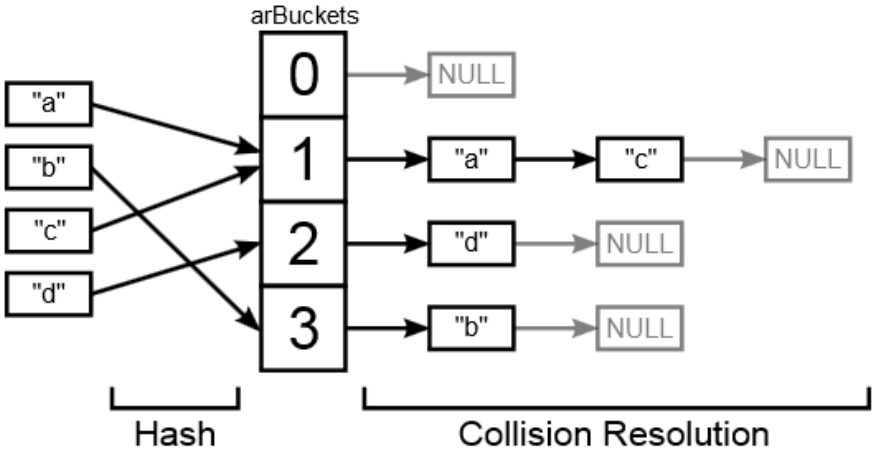


Рисунок 7.1 – Коллизия

Методы задания хеш-функций

Метод деления. Исходными данными являются: некоторый целый ключ key и размер таблицы m. Результатом данной функции является остаток от деления этого ключа на размер таблицы.

Пример 7.1. Получение ключа методом деления.

```
int h (int key, int m)
{
  return key % m;      // Значения
}
```

Рассмотрим примеры хеш-функций для хеш-таблицы размерностью m (рисунки 7.2 и 7.3).

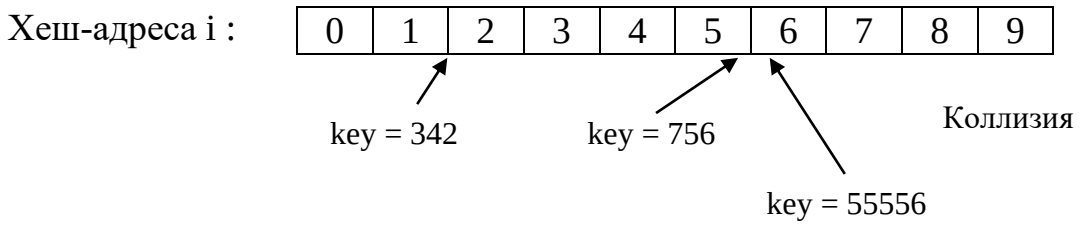


Рисунок 7.2 –Хеш-функция возвращает младшую цифру ключа для m = 10

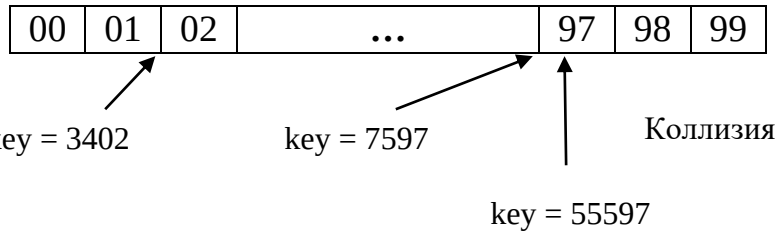


Рисунок 7.3 –Хеш-функция возвращает две младшие цифры ключа для m = 100

Метод середины квадрата. В нем ключ возводится в квадрат и в качестве индекса используются несколько средних цифр полученного значения. Например, ключом является целое 32-битное число, а хеш-функция возвращает средние 10 бит его квадрата.

Пример 7.2. Получение ключа методом квадрата.

```
int h(int key)
{
  key *= key;
  key>>= 11;          // Отбрасываем 11 младших бит
  return key % 1024;     // Возвращаем 10 младших бит
}
```

Аддитивный метод. В нем ключом является символьная строка. В хеш-функции строка преобразуется в целое суммированием всех символов и возвращается остаток от деления на m (обычно размер таблицы m = 256).

Пример 7.3. Получение ключа аддитивным методом.

```
int h(char *key, int m)
{
  int s = 0;
  while(*key) s += *key++;
  returns % m;
}
```

Коллизии возникают в строках, состоящих из одинакового набора символов, например, abc и cab. Данный метод можно несколько модифицировать, получая результат, суммируя только первый и последний символы строки-ключа.

Пример 7.4. Получение ключа модифицированным аддитивным методом.

```
int h (char *key, int m)
{
  int  len = strlen(key), s = 0;
  if(len < 2)    // Если длина ключа равна 0 или 1,
     s = key[0]; //возвращаем key[0]
  else
     s = key[0] + key[len-1];
  return s%m;
}
```

В этом случае коллизии будут возникать только в строках, например abc и amc.

Метод исключающего ИЛИ для ключей-строк (обычно размер таблицы m = 256). Этот метод аналогичен аддитивному, но в нем различаются схожие слова. Метод заключается в том, что к элементам строки последовательно применяется операция «исключающее ИЛИ».

В *мультипликативном методе* дополнительно используется случайное действительное число r из интервала [0, 1], тогда дробная часть произведения r*key будет находиться в интервале [0, 1]. Если это произведение умножить на размер таблицы m, то целая часть полученного произведения даст значение в диапазоне от 0 до m – 1.

Пример 7.5. Получение ключа мультипликативным методом.

```
int h(int key, int m)
{
  double r = key * rand();
  r = r - (int)r;   // Выделили дробную часть
  return r*m;
}
```

В общем случае при больших значениях m индексы, формируемые хеш-функцией, имеют большой разброс. Более того, математическая теория утверждает, что распределение получается более равномерным, если m является простым числом.

В рассмотренных примерах хеш-функция i = h(key) только определяет позицию, начиная с которой нужно искать (или первоначально – поместить в таблицу) запись с ключом key. Поэтому схема хеширования должна включать алгоритм разрешения коллизий (конфликтов), определяющий порядок действий, если позиция i = h(key) оказывается уже занятой записью с другим ключом.

Методы разрешения коллизий. Разрешить коллизии при хешировании можно двумя методами:

– методом *открытой адресации с линейным опробованием*;

– методом *цепочек*.

Метод открытой адресации с линейным опробованием. В методе открытой адресации с линейным опробованием используется простая стратегия разрешения коллизий: если ячейка, рассчитанная по хеш-коду ключа, уже занята другим элементом, то происходит последовательное смещение на одну ячейку вперед до тех пор, пока не будет найдена свободная ячейка для вставки элемента.

Например, предположим, у нас есть хеш-таблица размером 10 ячеек. При вставке элемента с ключом «A» с хеш-кодом 3 мы проверяем ячейку с индексом 3. Если она свободна, элемент помещается в нее. Если ячейка занята другим элементом, мы переходим к ячейке с индексом 4 и проверяем ее. Продолжаем этот процесс до тех пор, пока не найдем свободную ячейку.

При поиске элемента процесс аналогичен: мы вычисляем хеш-код ключа элемента и проверяем соответствующую ячейку. Если элемент с заданным ключом найден, возвращаем его. Если ячейка пуста, элемент отсутствует. Если ячейка занята другим элементом, мы переходим к следующей ячейке и продолжаем поиск.

Пример 7.6. Элемент данных.

```
struct ost
{
  int key;  // Ключ
  int info;  // Информация
} data;
```

Размер хеш-таблицы m = 5. Массив данных {44, 3}, {72, 4}, {93, 2}, {80, 1}, {70, 7}. Выберем хеш-функцию i = h(data) = data.key%10.

На основании исходных данных последовательно заполняем хеш-таблицу. Хеширование четырех ключей дает различные индексы (хеш-адреса):

i = 44 % 10 = 4;
i = 72 % 10 = 2;
i = 93 % 10 = 3;
i = 80 % 10 = 0;
i = 70 % 10 = 0;

Первая коллизия возникает между ключами 80 и 70 – место с индексом 0 занято. Поэтому просматриваем хеш-таблицу с целью поиска ближайшего свободного места, в данном случае это i = 1.

Таблица 7.1 – Хеш-таблица для метода линейного опробования

Хеш-адреса i	0	1	2	3	4
key	80	70	72	93	44
info	1	7	4	2	3

Метод цепочек. Метод цепочек представляет собой стратегию разрешения коллизий в хеш-таблицах, где каждая ячейка хеш-таблицы содержит связанный список или другую структуру данных, называемую цепочкой. Когда происходит коллизия, т. е. два или более элемента с одинаковыми хешами пытаются быть помещенными в одну ячейку, эти элементы добавляются в цепочку в этой ячейке.

В отличие от метода открытой адресации, где элементы хранятся непосредственно в ячейках хеш-таблицы, в методе цепочек каждая ячейка содержит ссылку на структуру данных, которая может быть изменяемой в длине, такую как связанный список или дерево. Элементы с одинаковыми хешами помещаются в одну и ту же ячейку в виде элементов списка или узлов дерева.

При вставке элемента с использованием метода цепочек вычисляется хеш-код ключа элемента, затем находится ячейка хеш-таблицы с соответствующим индексом. Если ячейка пуста, создается новая цепочка (связанный список или другая структура данных), и элемент помещается в эту цепочку. Если ячейка уже содержит цепочку, элемент добавляется в конец этой цепочки.

При поиске или удалении элемента сначала вычисляется хеш-код ключа элемента, затем находится соответствующая ячейка хеш-таблицы. Если ячейка пуста, элемент отсутствует. Если ячейка содержит цепочку, то выполняется операция поиска или удаления в этой цепочке.

Пример 7.7. Элемент данных для метода цепочек.

```
struct zap
{ int key;// Ключ
  int info;    // Информация
  zap *Next;  // Указатель на следующий элемент в списке
} data;
```

Размер хеш-таблицы m = 5. Массив данных {44, 3}, {72, 4}, {93, 2}, {80, 1}, {70, 7}. Выберем хеш-функцию i = h(data) = data.key%10. Хеширование первых четырех ключей, как и в предыдущем случае, дает различные индексы (хеш-адреса): 2, 0, 3 и 4. При возникновении коллизии новый элемент добавляется в конец списка с индексом 0.

Таблица 7.2 – Хеш-таблица метода цепочек

Хеш-адреса i	0	1	2	3	4
key	80		72	93	44
info	1		4	2	3
Next	Ссылка на элемент с ключом «70»		NULL	NULL	NULL

key	70
info	7
Next	NULL

Открытое и закрытое хеширование. В хеш-таблице с закрытой адресацией в случае коллизии элементы с одинаковыми хеш-кодами располагаются в виде списка или дерева (метод цепочек). В хеш-таблице с открытой адресацией в случае коллизии ссылки на элементы с одинаковыми хеш-кодами располагаются в этой же таблице (методы открытой адресации).

Реализация хеширования методом цепочек.

Пример 7.8. Создание хеш-таблицы со случайными целыми ключами и нахождение записи с минимальным ключом.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <locale.h>
#define TABLE_SIZE 25
// Структура для хранения записи в хеш-таблице
typedef struct Node {
    int key;
    struct Node* next;
} Node;
// Функция для создания новой записи
Node* createNode(int key) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->next = NULL;
    return newNode;
}
```

```
// Функция для вычисления хеша ключа
int hashFunction(int key) {
    return key % TABLE_SIZE;
}
// Функция для вставки записи в хеш-таблицу
void insert(Node* hashTable[], int key)
{
    int index = hashFunction(key);
    Node* newNode = createNode(key);
    if (hashTable[index] == NULL) {
        hashTable[index] = newNode;
       }
    else {
        Node* currentNode = hashTable[index];
        while (currentNode->next != NULL) {
            currentNode = currentNode->next;
        }
        currentNode->next = newNode;
    }
}
// Функция для поиска записи с минимальным ключом
Node* findMinKey(Node* hashTable[]) {
    Node* minNode = NULL;

    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i] != NULL) {
          if (minNode == NULL||hashTable[i]->key < minNode->key) {
                minNode = hashTable[i];
           }
        }
    }
    return minNode;
}
// Функция для вывода хеш-таблицы
void printHashTable(Node* hashTable[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Позиция %d:", i);
        Node* currentNode = hashTable[i];
        while (currentNode != NULL) {
            printf(" %d", currentNode->key);
            currentNode = currentNode->next;
        }
        printf("\n");
    }
}
int main() {
    setlocale(LC_ALL, "RUS");
    // Инициализация хеш-таблицы
    Node* hashTable[TABLE_SIZE];
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = NULL;
    }
```

```
    // Вставка случайных ключей в хеш-таблицу
    srand(time(NULL));
    for (int i = 0; i < TABLE_SIZE; i++) {
        int key = rand() % 1000; // Генерация случайного ключа
        insert(hashTable, key);
    }
    // Вывод хеш-таблицы
    printHashTable(hashTable);
    // Поиск записи с минимальным ключом
    Node* minNode = findMinKey(hashTable);
    if (minNode != NULL) {
        printf("Запись с минимальным ключом: %d\n", minNode->key);
    }
    else {
        printf("Хеш-таблица пуста.\n");
    }
    return 0;
}
```

Реализация хеширования методом открытой адресации с линейным опробованием.

Пример 7.9. Создание хеш-таблицы со случайными целыми ключами и нахождение записи с минимальным ключом.

```
#include<time.h>
#include<locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define TABLE_SIZE 25
// Структура для хранения записи в хеш-таблице
typedef struct {
    int key;
} Record;
// Функция для создания новой записи
Record* createRecord(int key) {
    Record* newRecord = (Record*)malloc(sizeof(Record));
    newRecord->key = key;
    return newRecord;
}
// Функция для вычисления хеша ключа
int hashFunction(int key) {
    return key % TABLE_SIZE;
}
// Функция для вставки записи в хеш-таблицу
void insert(Record* hashTable[], Record* record) {
    int index = hashFunction(record->key);
    int i = 0;
```

```
    while (hashTable[index] != NULL) {
        index = (index + 1) % TABLE_SIZE; // Линейное опробование
        i++;
// Если прошли по всей таблице и не нашли свободное место, выходим
        if (i == TABLE_SIZE) {
            printf("Хеш-таблица заполнена.\n");
            return;
        }
    }
    hashTable[index] = record;
}
// Функция для поиска записи с минимальным ключом
Record* findMinKey(Record* hashTable[]) {
    Record* minRecord = NULL;
    bool firstRecordFound = false;
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i] != NULL) {
            if (!firstRecordFound) {
                minRecord = hashTable[i];
                firstRecordFound = true;
            }
            else if (hashTable[i]->key < minRecord->key) {
                minRecord = hashTable[i];
            }
        }
    }
    return minRecord;
}
// Функция для вывода хеш-таблицы
void printHashTable(Record* hashTable[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Позиция %d:", i);
        if (hashTable[i] != NULL) {
            printf(" %d", hashTable[i]->key);
        }
        printf("\n");
    }
}
int main()
    {
    setlocale(LC_ALL, "rus");
    // Инициализация хеш-таблицы
    Record* hashTable[TABLE_SIZE];
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = NULL;
    }
    // Вставка случайных ключей в хеш-таблицу
    srand(time(NULL));
    for (int i = 0; i < TABLE_SIZE; i++) {
        int key = rand() % 1000; // Генерация случайного ключа
        Record* record = createRecord(key);
        insert(hashTable, record);
```

```
    }
    // Вывод хеш-таблицы
    printHashTable(hashTable);
    // Поиск записи с минимальным ключом
    Record* minRecord = findMinKey(hashTable);
    if (minRecord != NULL) {
      printf("Запись с минимальным ключом: %d\n", minRecord->key);
    }
    else {
        printf("Хеш-таблица пуста.\n");
    }
    return 0;
}
```

Индивидуальные задания

1 С помощью открытого хеширования создать хеш-таблицу со случайными ключами и вывести на экран все элементы в порядке убывания ключа.

2 С помощью закрытого хеширования создать хеш-таблицу со случайными ключами и вывести на экран все элементы в порядке возрастания ключа.

3 Создать хеш-таблицу со случайными целыми ключами в диапазоне от 1 до 75 и удалить из нее записи с четными ключами.

4 Создать хеш-таблицу со случайными целыми ключами в диапазоне от 135 до 245 и удалить из нее записи с нечетными ключами.

5 Подсчитать сколько элементов хеш-таблицы со случайными ключами не превышают среднее значение от всех ключей.

6 Создать хеш-таблицу со случайными ключами в диапазоне от 25 до 35 и определить наиболее часто повторяющийся ключ.

7 С помощью открытого хеширования создать хеш-таблицу со случайными ключами и вывести на экран все элементы в порядке возрастания ключа.

8 С помощью закрытого хеширования создать хеш-таблицу со случайными ключами и вывести на экран все элементы в порядке убывания ключа.

9 Создать хеш-таблицу со случайными целыми ключами и преобразовать ее в две таблицы. Первая должна содержать только положительные ключи, а вторая – отрицательные.

10 Создать хеш-таблицу из случайных целых чисел в диапазоне от –50 до 50 и из нее сделать еще две. В первую поместить записи с ключами, большими 5, а во второй – с меньшими 5.

11 С помощью открытого хеширования создать хеш-таблицу из случайных целых чисел и найти в ней элемент, содержащий максимальное значение ключа.

12 С помощью закрытого хеширования создать хеш-таблицу из случайных целых чисел и найти в ней номер списка, содержащего минимальное значение ключа.

13 Создать хеш-таблицу со случайными ключами в диапазоне от –25 до 25 и удалить из нее записи с ключами из диапазона $-5 < key < 5$.

14 Создать хеш-таблицу со случайными ключами в диапазоне от –50 до 50 и удалить из нее записи с ключами key < 5.

15 Создать хеш-таблицу со случайными целыми ключами в диапазоне от –10 до 10 и удалить из нее записи с отрицательными ключами.

16 Создать хеш-таблицу для хранения информации о студентах со случайными ключами (номерами студенческих билетов) в диапазоне от 1 до 1000 и удалить из нее записи о студентах с номерами билетов, оканчивающимися на четную цифру.

17 Создать хеш-таблицу для хранения информации о товарах в магазине со случайными ключами (ценами) в диапазоне от 1 до 100 и определить, какой товар имеет наибольшую стоимость.

18 Создать хеш-таблицу для хранения информации о городах с ключами (названиями) и найти все города, название которых начинается с буквы «М».

19 Создать хеш-таблицу для хранения информации о городах с ключами (названиями) и удалить из нее все города, название которых содержит букву «о».

20 Создать хеш-таблицу для хранения информации о книгах с ключами (названиями) и найти все книги, название которых содержит слово «король».

21 Создать хеш-таблицу для хранения информации о студентах с ключами (номерами зачетных книжек) в диапазоне от 1 до 1000 и вывести на экран список всех студентов с нечетными номерами зачетных книжек.

22 Создать хеш-таблицу для хранения информации о клиентах с ключами (номерами) в диапазоне от 100 до 999 и найти всех клиентов, у которых номер начинается с цифры 3.

23 Создать хеш-таблицу для хранения информации продуктах с ключами (ценой). Определить, сколько продуктов имеют цену выше средней.

24 Создать хеш-таблицу для хранения информации о клиентах банка с ключами (балансом на счете). Найти клиента с наибольшим балансом на счете.

25 Создать хеш-таблицу для хранения информации о студентах со случайными целыми ключами (номерами студенческих билетов) в диапазоне от 500 до 1500 и преобразовать ее в две таблицы. Первая должна содержать только трехзначные ключи, а вторая – четырехзначные.

26 Создать хеш-таблицу со случайными целыми ключами и определить, сколько элементов имеют четные значения ключей.

27 Создать хеш-таблицу со случайными целыми ключами и определить, сколько элементов имеют нечетные значения ключей.

28 Создать хеш-таблицу со случайными целыми ключами в диапазоне от –40 до +40 и преобразовать ее в две таблицы. Первая должна содержать только четные ключи, а вторая – нечетные.

29 Создать хеш-таблицу со случайными строковыми ключами и найти запись с наибольшей длиной ключа.

30 Создать хеш-таблицу со случайными строковыми ключами и найти запись с наименьшей длиной ключа.

31 Создать хеш-таблицу со случайными целыми ключами и удалить из нее записи с ключами, являющимися простыми числами.

32 Создать хеш-таблицу со случайными целыми ключами в диапазоне от –20 до 20 и удалить из нее все записи с четными положительными ключами.

33 Создать хеш-таблицу со случайными целыми ключами в диапазоне от –10 до 10 и удалить из нее все записи с нечетными отрицательными ключами.

34 Создать хеш-таблицу со случайными строковыми ключами и вывести на экран все элементы в порядке возрастания длины ключа.

35 Создать хеш-таблицу со случайными целыми ключами и вывести на экран все элементы в порядке убывания длины ключа.

36 Создать хеш-таблицу со случайными целыми ключами и значениями и вывести на экран все элементы в порядке возрастания значения ключа.

37 Создать хеш-таблицу со случайными целыми ключами и значениями и вывести на экран все элементы в порядке убывания значения ключа.

38 Создать хеш-таблицу со случайными целыми ключами в диапазоне от –50 до +50 и преобразовать ее в две таблицы. Первая должна содержать только положительные ключи, а вторая – отрицательные.

Лабораторная работа № 8.
Графы и алгоритмы поиска пути

Цель работы – ознакомиться со способами задания графов, основными операциями и алгоритмами работы с ними. Научиться использовать графовые модели при разработке программ.

Ход выполнения лабораторной работы должен быть отражен в отчете. Отчет должен содержать титульный лист, номер задания, коды программ, скриншоты с результатами выполнения программы.

Методические указания

Граф представляет собой абстрактную структуру, состоящую из вершин и ребер, соединяющих вершины. Вершины могут представлять объекты или сущности, а ребра – отношения или связи между ними. Графы могут быть ориентированными и неориентированными.

Ориентированный граф – граф, в котором ребра имеют направления и обозначаются стрелками. В ориентированном графе можно перемещаться вдоль ребра только в указанном направлении.

Неориентированный граф – граф, в котором ребра не имеют направления. На рисунке 8.1 показаны неориентированный и ориентированный графы. В неориентированном графе можно перемещаться вдоль ребра в любом из двух направлений. В ориентированном графе можно перемещаться вдоль ребра только в направлении стрелки.

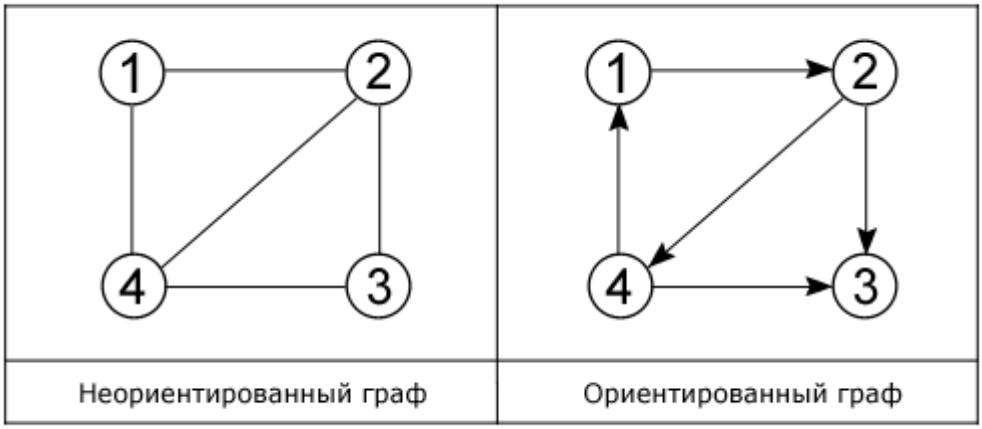


Рисунок 8.1 – Виды графов

Способы представления графа. Граф может быть представлен (сохранен) несколькими способами:

– матрица смежности;

– матрица инцидентности;

– список смежности (инцидентности);

– список ребер.

Использование двух первых методов предполагает хранение графа в виде двумерного массива (матрицы). Размер массива зависит от количества вершин и/или ребер в конкретном графе.

Матрица смежности для графа из N вершин хранится в виде двумерного массива размером N × N (рисунок 8.2). Вершины графа в этом случае задаются номерами (индексами строк и столбцов матрицы), а элемент matrix[i, j] отражает наличие дуги (единичное значение) между соответствующими вершинами или отсутствие дуги (нулевое значение).

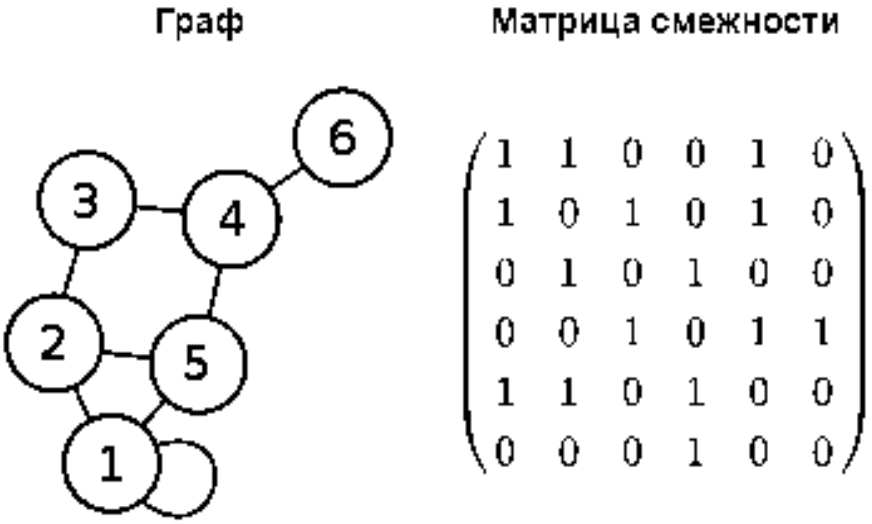


Рисунок 8.2 – Граф и его матрица смежности

Другой распространенный способ представления графов в коде – списки смежности (рисунок 8.3). Суть в том, что для каждой вершины создается список соседей.

Преимущества списка смежности:

– позволяет рационально использовать память;

– позволяет быстро перебирать соседей вершины;

– позволяет проверять наличие ребра и удалять его.

Недостатки списка смежности:

– при работе с насыщенными графами (с большим количеством ребер) будет низкое быстродействие программы;

– нет быстрого способа проверить, существует ли ребро между двумя вершинами;

– количество вершин графа должно быть известно заранее.

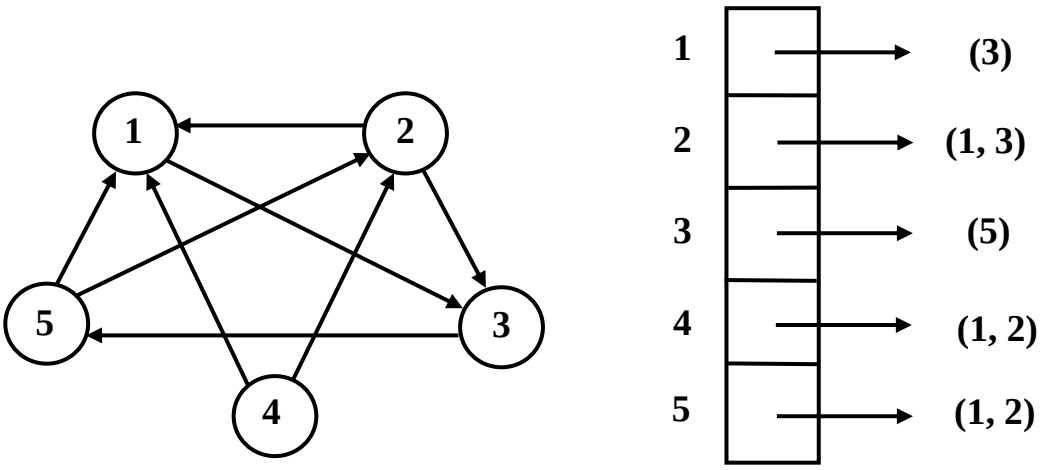


Рисунок 8.3 – Граф и его список смежности

Матрица инцидентности для графа из N вершин и ребер хранится в виде двумерного массива размером N × M (рисунок 8.4). Элемент матрицы matrix[i, j] отражает инцидентность ребра j вершине i, т. е. тот факт, что это ребро выходит или входит в нее. В неориентированном графе если вершина инцидентна ребру, то соответствующий элемент равен 1, в противном случае элемент равен 0. В ориентированном графе, если ребро выходит из вершины, то соответствующий элемент равен 1, если ребро входит в вершину, то соответствующий элемент равен –1, если ребро отсутствует, то элемент равен 0.

	a	**b**	**c**	**d**	**e**
1	1	1	0	0	0
2	1	0	1	1	0
3	0	0	0	1	1
4	0	1	1	0	1

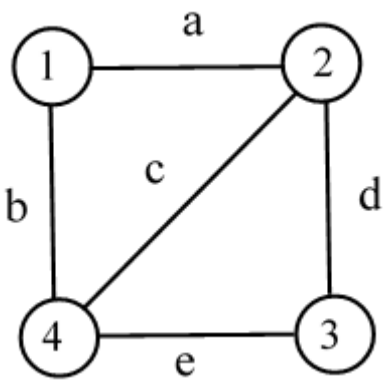


Рисунок 8.4 – Граф и его матрица инцидентности

Операции удаления ребра и вершины. При удалении ребра обе вершины остаются в графе, а связь между ними удаляется. При удалении ребра

ориентированного графа из списка смежности вершины, из которой исходит стрелка, удаляется связь с вершиной, в которую стрелка входит. При удалении ребра неориентированного графа из списков смежности обеих вершин удаляется связь со второй из них.

При удалении вершины из графа удаляются и все инцидентные ей ребра. Для удаления вершины ориентированного графа нужно удалить все ребра, исходящие из данной вершины (удалить связи, исходящие из списка смежности данной вершины), а также удалить все ребра, входящие в данную вершину (удалить соответствующие связи в списках смежности других вершин). А затем нужно удалить саму вершину из списка вершин графа. Для удаления вершины неориентированного графа нужно удалить все ребра, связанные с данной вершиной (удалить связи из списка смежности данной вершины и удалить соответствующие связи из списков смежности других вершин). А затем удалить саму вершину из списка вершин графа.

Алгоритмы обхода графов. Основными алгоритмами обхода графов являются:

– поиск в ширину;

– поиск в глубину.

Алгоритм поиска в глубину (Depth-First Search, DFS) предназначен для обхода графа и посещения всех вершин графа, исследуя каждую вершину как можно глубже, прежде чем переходить к следующей непосещенной вершине. Алгоритм DFS исследует граф, спускаясь как можно глубже от начальной вершины, пока не достигнет конечной точки или вершины без непосещенных соседей. Затем он возвращается назад и исследует другие пути.

Алгоритм DFS работает следующим образом:

1 Выбрать начальную вершину графа и пометить ее как посещенную.

2 Вывести или обработать текущую вершину.

3 Рекурсивно применить алгоритм DFS для каждой непосещенной смежной вершины. При этом для каждой смежной вершины нужно пометить смежную вершину как посещенную, и рекурсивно вызвать алгоритм DFS для смежной вершины.

4 Если все смежные вершины были посещены, возвратиться к предыдущей вершине и проверить, есть ли еще непосещенные смежные вершины для исследования.

5 Повторять шаги 3 и 4 до тех пор, пока не будут обработаны все вершины графа.

Алгоритм поиска в ширину (Breadth-First Search, BFS) предназначен для обхода графа, начиная с заданной вершины, и посещения всех вершин графа в порядке удаленности от начальной вершины. Алгоритм BFS использует принцип поиска в ширину, т. е. он посещает сначала все вершины, находящиеся на одном уровне от начальной вершины, а затем переходит к вершинам следующего уровня. При этом используется очередь для хранения вершин,

которые нужно посетить. Это гарантирует, что ближайшие соседние вершины будут обработаны раньше более отдаленных вершин.

Алгоритм BFS работает следующим образом:

1 Выбрать начальную вершину графа и пометить ее как посещенную.

2 Поместить начальную вершину в очередь.

3 Пока очередь не пуста, извлекать вершину из начала очереди, выводить извлеченную вершину.

4 Посещать все непосещенные смежные вершины извлеченной вершины и помечать их как посещенные.

5 Поместить посещенные вершины в конец очереди.

6 Если все вершины графа были посещены, алгоритм завершается.

Пример 8.1. Построение графа с помощью списка смежности, матрицы смежности и матрицы инцидентности. Удаление шестой вершины и ребер (0, 1), (2, 3), (3, 8).

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
struct AdjListNode {
// Структура, представляющая узел списка смежности
  int vertex;
  struct AdjListNode* next;
};
struct AdjList { // Структура, представляющая список смежности
  struct AdjListNode* head; // Указатель на головной узел списка
};
struct Graph { // Структура, представляющая граф
  int numVert;
  int numEdges;
  struct AdjList* array;// Массив списков смежности
  int **matrixSmezhn; // Матрица смежности
  int **matrixIncident; // Матрица инцидентности
};
// Функция для создания графа
struct Graph* createGraph(int numVert, num_edges) {
  struct Graph* graph=(struct Graph*)malloc(sizeof(struct Graph));
  graph->numVert = numVert;
  graph->numEdges = numEdges;
  graph->array=(struct AdjList*)malloc(numVert *sizeof(struct
                  AdjList));
  int i = 0;
  for (i = 0; i < numVert; ++i) {
    graph->array[i].head = NULL;
    }
  return graph;
}
// Функция для создания матрицы смежности
void createMatrixSmezhn(struct Graph* graph, int numVert) {
```

```
  graph->numVert = numVert;
  graph->matrixSmezhn =(int **)malloc(sizeof(int)*numVert);
  for (int i = 0; i < numVert; i++) {
     graph->matrixSmezhn[i] = (int *)malloc(sizeof(int)*numVert);
    for (int j = 0; j < numVert; j++) {
      *(*(graph->matrixSmezhn+i)+j) = 0;
      }
  }
}
// Функция для создания матрицы инцидентности
void createMatrixIncident(struct Graph* graph, int numVert, int
numEdges) {
   graph->numVert = numVert;
   graph->numEdges = numEdges;
   graph->matrixIncident=(int **)malloc(sizeof(int)*numVert);
   for (int i = 0; i < numVert; ++i) {
    graph->matrixIncident[i]=(int *)malloc(sizeof(int)*numEdges);
    for (int j = 0; j < numEdges; ++j) {
      graph->matrixIncident[i][j] = 0;
    }
  }
}
// Функция добавления нового узла в список
struct AdjListNode* newListNode(int vertex) {
  struct AdjListNode* newNode=(struct AdjListNode*)
         malloc(sizeof(struct AdjListNode));
  newNode->vertex = vertex;
  newNode->next = NULL;
  return newNode;
}
// Функция добавления ребра в список смежности
void addEdge(struct Graph* graph, int firstVertex, int secVertex)
{
  struct AdjListNode* newNode = newListNode(secVertex);
  newNode->next = graph->array[firstVertex].head;
  graph->array[firstVertex].head = newNode;
  newNode = newListNode(firstVertex);
  newNode->next = graph->array[secVertex].head;
  graph->array[secVertex].head = newNode;
}
// Функция добавления ребра в матрицу смежности
void addEdgeInMatrix(struct Graph* graph, int firstVertex, int
secVertex) {
  if (firstVertex >= 0 && firstVertex < graph->numVert && secVer-
tex >= 0 && secVertex < graph->numVert) {
    graph->matrixSmezhn[firstVertex][secVertex] = 1;
    graph->matrixSmezhn[secVertex][firstVertex] = 1;
  }
}
// Функция добавления ребра в матрицу инцидентности
void addEdgeInMIncident(struct Graph* graph, int firstVertex, int
secVertex, int edge) {
```

```
  graph->matrixIncident[firstVertex][edge] = 1;
  graph->matrixIncident[secVertex][edge] = 1;
}
// Функция удаления ребра из списка смежности
void removeEdge(struct Graph* graph, int firstVertex, int secVer-
tex) {
  struct AdjListNode* currentNode = graph-> array[firstVertex].head;
  struct AdjListNode* prevNode = NULL;
  while (currentNode != NULL && currentNode->vertex != secVertex)
  {
    prevNode = currentNode;
    currentNode = currentNode->next;
  }
  if (currentNode != NULL) {
        if (prevNode == NULL) {
            graph->array[firstVertex].head = currentNode->next;
        }
        else {
            prevNode->next = currentNode->next;
        }
        free(currentNode);
    }
    currentNode = graph->array[secVertex].head;
    prevNode = NULL;
    while (currentNode != NULL && currentNode->vertex != firstVer-
tex) {
        prevNode = currentNode;
        currentNode = currentNode->next;
    }
    if (currentNode != NULL) {
        if (prevNode == NULL) {
            graph->array[secVertex].head = currentNode->next;
        }
        else {
            prevNode->next = currentNode->next;
        }
        free(currentNode);
    }
}
// Функция удаления вершины из списка смежности
void removeVertex(struct Graph* graph, int vertex) {
    struct AdjListNode* currentNode = graph->array[vertex].head;
    struct AdjListNode* nextNode = NULL;
    while (currentNode != NULL) {
        nextNode = currentNode->next;
        free(currentNode);
        currentNode = nextNode;
    }
    graph->array[vertex].head = NULL;
    for (int i = 0; i < graph->numVert; ++i) {
      currentNode = graph->array[i].head;
      struct AdjListNode* prevNode = NULL;
```

```
      while (currentNode != NULL) {
        if (currentNode->vertex == vertex) {
          if (prevNode == NULL) {
             graph->array[i].head = currentNode->next;
          }
          else {
             prevNode->next = currentNode->next;
          }
              free(currentNode);
              break;
            }
            prevNode = currentNode;
            currentNode = currentNode->next;
        }
    }
}
// Функция вывода графа на консоль
void printGraph(struct Graph* graph) {
    int versh;
    for (versh = 0; versh < graph->numVert; ++versh) {
        struct AdjListNode* vertexes = graph->array[versh].head;
        printf("\n Список смежности для вершины %d\n head ",
versh);
        while (vertexes) {
            printf("-> %d", vertexes->vertex);
            vertexes = vertexes->next;
        }
    }
  printf("\n");
}
// Функция вывода матрицы смежности на консоль
void printMatrixSmezhn(struct Graph* graph) {
  for (int i = 0; i < graph->numVert; i++) {
    for (int j = 0; j < graph->numVert; j++) {
      printf("%d ", graph->matrixSmezhn[i][j]);
      }
  printf("\n");
    }
}
// Функция вывода матрицы инцидентности на консоль
void printIncidenceMatrix(struct Graph* graph) {
for (int i = 0; i < graph->numVert; ++i) {
  for (int j = 0; j < graph->numEdges; ++j) {
 printf("%d ", graph->matrixIncident[i][j]);
        }
        printf("\n");
    }
}
int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251);
```

```
int num_vershin = 5;
int num_edges=9;
// Создаем граф
struct Graph* graph= createGraph(num_vershin, num_edges);
// Создаем матрицу смежности графа
createMatrixSmezhn(graph, num_vershin);
// Создаем матрицу инцидентности графа
createMatrixIncident(graph, num_vershin, num_edges);
// Добавляем ребра в список смежности
addEdge(graph, 0, 1);
addEdge(graph, 0, 3);
addEdge(graph, 1, 2);
addEdge(graph, 3, 2);
addEdge(graph, 1, 4);
addEdge(graph, 2, 0);
addEdge(graph, 3, 4);
addEdge(graph, 4, 2);
addEdge(graph, 4, 0);
// Добавляем ребра в матрицу смежности
addEdgeInMatrix(graph, 0, 1);
addEdgeInMatrix(graph, 0, 3);
addEdgeInMatrix(graph, 1, 2);
addEdgeInMatrix(graph, 3, 2);
addEdgeInMatrix(graph, 1, 4);
addEdgeInMatrix(graph, 2, 0);
addEdgeInMatrix(graph, 3, 4);
addEdgeInMatrix(graph, 4, 2);
addEdgeInMatrix(graph, 4, 0);
// Добавляем ребра в матрицу инцидентности
addEdgeInMIncident(graph, 0, 1, 0);
addEdgeInMIncident(graph, 0, 3, 1);
addEdgeInMIncident(graph, 1, 2, 2);
addEdgeInMIncident(graph, 3, 2, 3);
addEdgeInMIncident(graph, 1, 4, 4);
addEdgeInMIncident(graph, 2, 0, 5);
addEdgeInMIncident(graph, 4, 0, 6);
addEdgeInMIncident(graph, 3, 4, 7);
addEdgeInMIncident(graph, 4, 2, 8);
printf("1. Вывести матрицу смежности для исходного графа\n");
printf("2. Вывести матрицу инцидентности для исходного графа\n");
printf("3. Вывести граф в виде списка\n");
printf("4. Удалить ребро (0,1)\n");
printf("5. Удалить вершину номер 4\n");
printf("6. Вывести итоговый граф\n");
printf("7. Выход \n");
int ch=0;
while (ch!=7) {
  scanf_s("%d",&ch);
  switch (ch)
    {
     case 1:
       printf("Матрица смежности: \n");
```

```
      printMatrixSmezhn(graph);
      break;
    case 2:
      printf("Матрица инцидентности: \n");
         printIncidenceMatrix(graph);
         break;
       case 3:
         printGraph(graph);
         break;
       case 4:
         removeEdge(graph, 0, 1);
         printf("Граф после удаления ребра");
         printGraph(graph);
         break;
       case 5:
         removeVertex(graph, 4);
         printf("Граф после удаления вершины");
         printGraph(graph);
         break;
       case 6:
         printf("Граф после удаления вершины и ребра:");
         printGraph(graph);
         break;
       case 7: return 0;
       default: printf("Неверный выбор\n");
           break;
       }
   }
  return 0;
}
```

Пример 8.2. Выполнение в графе поиска в глубину и поиска в ширину.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
struct AdjListNode { // Структура, представляющая узел списка
                     //смежности
  int vertex;
  struct AdjListNode* next;
};
struct AdjList { // Структура, представляющая список смежности
  struct AdjListNode* head; // Указатель на головной узел списка
};
struct Graph { // Структура, представляющая граф
  int numVert;
  int numEdges;
  struct AdjList* array;  // Массив списков смежности
  int **matrixSmezhn;     // Матрица смежности
  int **matrixIncident;   // Матрица инцидентности
};
```

```
// Функция для создания графа
struct Graph* createGraph(int numVert, int numEdges) {
  struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
  graph->numVert = numVert;
  graph->numEdges = numEdges;
  graph->array=(struct AdjList*)malloc(numVert *sizeof(struct AdjList));
  int i = 0;
  for (i = 0; i < numVert; ++i) {
    graph->array[i].head = NULL;
    }
  return graph;
}
// Функция для создания матрицы смежности
void createMatrixSmezhn(struct Graph* graph, int numVert) {
  graph->numVert = numVert;
  graph->matrixSmezhn=(int **)malloc(sizeof(int)*numVert);
  for (int i = 0; i < numVert; i++) {
    graph->matrixSmezhn[i] = (int *)malloc(sizeof(int)*numVert);
    for (int j = 0; j < numVert; j++) {
      *(*(graph->matrixSmezhn+i)+j) = 0;
      }
  }
}
// Функция добавления ребра в матрицу смежности
void addEdgeInMatrix(struct Graph* graph, int firstVertex, int
secVertex) {
  if (firstVertex >= 0 && firstVertex < graph->numVert && secVer-
tex >= 0 && secVertex < graph->numVert) {
    graph->matrixSmezhn[firstVertex][secVertex] = 1;
    graph->matrixSmezhn[secVertex][firstVertex] = 1;
  }
}
// Функция добавления ребра в список смежности
void addEdge(struct Graph* graph, int firstVertex, int secVertex)
{
  struct AdjListNode* newNode = newListNode(secVertex);
  newNode->next = graph->array[firstVertex].head;
  graph->array[firstVertex].head = newNode;
  newNode = newListNode(firstVertex);
  newNode->next = graph->array[secVertex].head;
  graph->array[secVertex].head = newNode;
}
// Функция вывода матрицы смежности на консоль
void printMatrixSmezhn(struct Graph* graph) {
  for (int i = 0; i < graph->numVert; i++) {
    for (int j = 0; j < graph->numVert; j++) {
      printf("%d ", graph->matrixSmezhn[i][j]);
    }
  printf("\n");
  }
}
```

```
// Функция вывода графа (списка смежности) на консоль
void printGraph(struct Graph* graph) {
  int versh;
  for (versh = 0; versh < graph->numVert; ++versh) {
    struct AdjListNode* vertexes = graph->array[versh].head;
    printf("\n Список смежности для вершины %d\n head ", versh);
    while (vertexes) {
      printf("-> %d", vertexes->vertex);
      vertexes = vertexes->next;
    }
  }
  printf("\n");
}
//Рекурсивная функция для обхода графа в глубину
void dfsUtil(Graph* graph, int vertex, bool visited[]) {
  visited[vertex] = true;
  printf("%d\n", vertex); // Выводим посещенную вершину.
// Рекурсивно посещаем все смежные вершины, если они еще
//не посещены
  for (int i = 0; i < graph->numVert; i++) {
    if (graph->matrixSmezhn[vertex][i] == 1 && !visited[i]) {
      dfsUtil(graph, i, visited);
    }
  }
}

// Обход графа в глубину (DFS)
void dfs(Graph* graph, int startVertex) {
  bool visited[9];
  // Инициализируем массив visited значением false
  for (int i = 0; i < graph->numVert; i++) {
    visited[i] = false;
  }
  // Вызываем рекурсивную функцию для обхода графа
  dfsUtil(graph, startVertex, visited);
}

// Обход графа в ширину (BFS)
void bfs(Graph* graph, int startVertex) {
  bool visited[9] = { false };
  int queue[9];
  int front = 0, rear = 0;
  visited[startVertex] = true;
  queue[rear++] = startVertex;
  while (front < rear) {
    int currentVertex = queue[front++];
    printf("%d ", currentVertex); // Выводим посещенную вершину
  // Перебираем все смежные вершины текущей вершины
    for (int i = 0; i < graph->numVert; i++) {
      if (graph->matrixSmezhn[currentVertex][i]==1&&!visited[i])
```

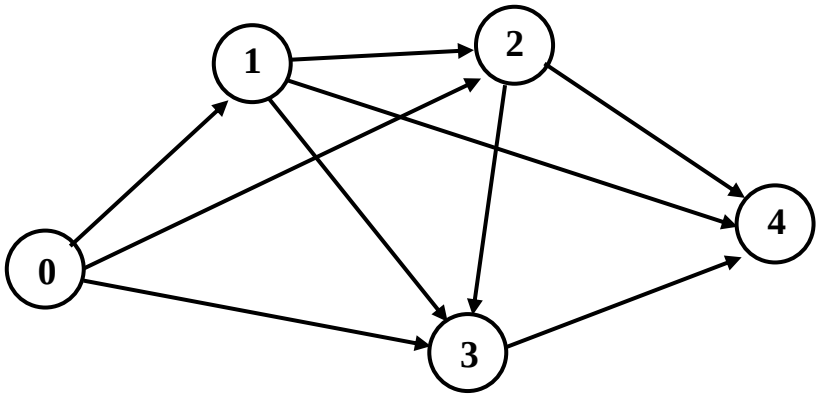
```
        {
         visited[i] = true;
         queue[rear++] = i;
        }
    }
  }
}

int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251);
int num_vershin = 9;
int num_edges=16;
  // Создаем граф
struct Graph* graph = createGraph(num_vershin, num_edges);
  // Создаем матрицу смежности графа
createMatrixSmezhn(graph, num_vershin);
  // Добавляем ребра в матрицу смежности
addEdgeInMatrix(graph, 0, 1);
addEdgeInMatrix(graph, 0, 6);
addEdgeInMatrix(graph, 0, 7);
addEdgeInMatrix(graph, 1, 2);
addEdgeInMatrix(graph, 1, 5);
addEdgeInMatrix(graph, 1, 6);
addEdgeInMatrix(graph, 2, 5);
addEdgeInMatrix(graph, 2, 3);
addEdgeInMatrix(graph, 3, 4);
addEdgeInMatrix(graph, 3, 5);
addEdgeInMatrix(graph, 3, 8);
addEdgeInMatrix(graph, 4, 5);
addEdgeInMatrix(graph, 4, 8);
addEdgeInMatrix(graph, 4, 6);
addEdgeInMatrix(graph, 6, 5);
addEdgeInMatrix(graph, 6, 7);
  // Добавляем ребра в список смежности
addEdge(graph, 0, 1);
addEdge(graph, 0, 6);
addEdge(graph, 0, 7);
addEdge(graph, 1, 2);
addEdge(graph, 1, 6);
addEdge(graph, 1, 5);
addEdge(graph, 2, 3);
addEdge(graph, 2, 5);
addEdge(graph, 3, 4);
addEdge(graph, 3, 5);
addEdge(graph, 3, 8);
addEdge(graph, 4, 5);
addEdge(graph, 4, 6);
addEdge(graph, 4, 8);
addEdge(graph, 5, 6);
addEdge(graph, 6, 7);
```

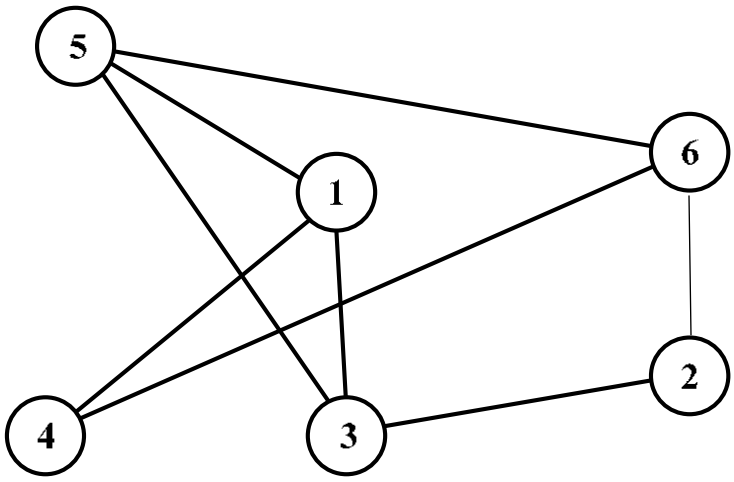
```
printMatrixSmezhn(graph);
printGraph(graph);
int startVertex;
int ch=0;
while (ch!=3) {
  printf("1. Поиск в глубину\n");
  printf("2. Поиск в ширину\n");
  printf("3. Выход \n");
  scanf("%d", &ch);
  switch (ch)
    {
     case 1:
       printf("Введите вершину, с которой начать обход: ");
       scanf("%d", &startVertex);
       printf("Результат обхода в глубину (DFS): \n");
       dfs(graph, startVertex);
       break;
     case 2:
       printf("Введите вершину, с которой начать обход: ");
       scanf("%d", &startVertex);
       printf("Результат обхода в ширину (BFS):\n ");
       bfs(graph, startVertex);
       printf("\n");
       break;
     case 3: return 0;
        default: printf("Неверный выбор\n");
            break;
        }
    }
  return 0;
  }
```

Индивидуальные задания

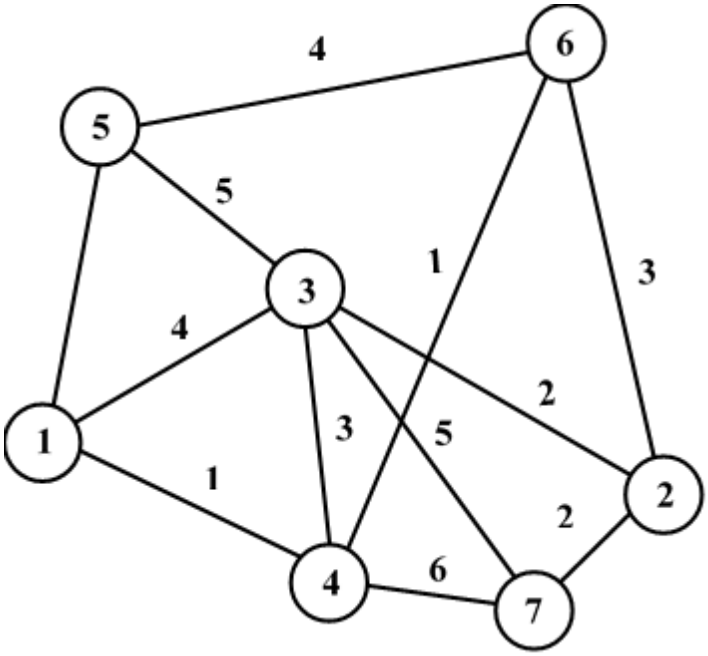
1 Построить граф согласно схеме, составить список смежности и матрицу смежности. Удалить вершину номер 3. Найти все вершины, достижимые из вершины 0, с помощью поиска в глубину.



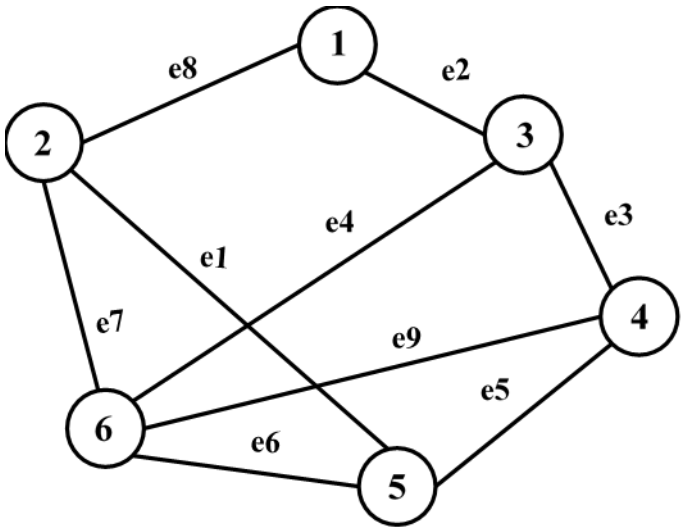
2 Построить граф согласно схеме, составить список смежности и матрицу смежности. Удалить вершину номер 4. Найти все вершины, достижимые из вершины 1, с помощью поиска в ширину.



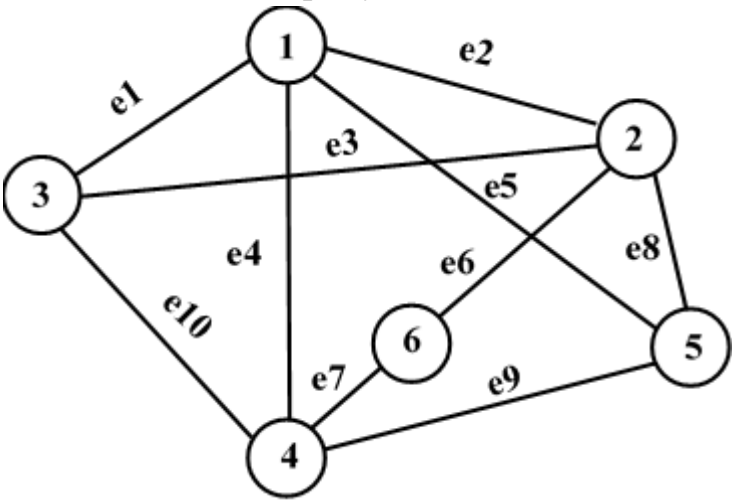
3 Построить граф согласно схеме, составить список смежности и матрицу смежности. Удалить все ребра с весом 5. Найти все вершины, достижимые из вершины 6, с помощью поиска в глубину.



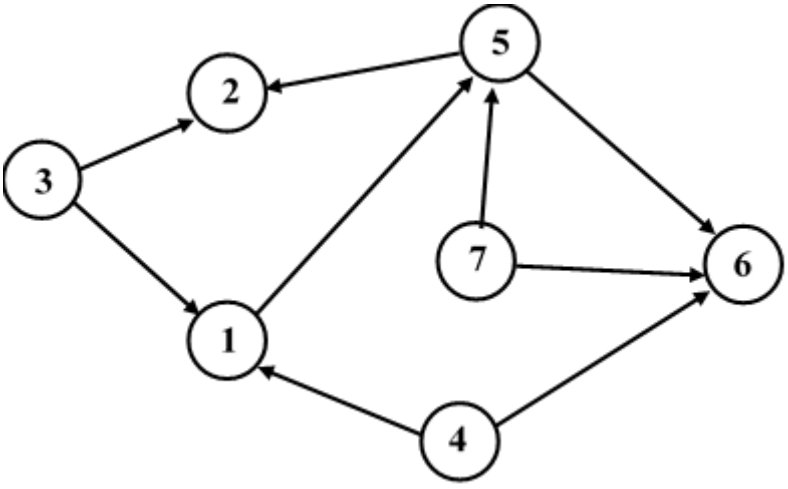
4 Построить граф согласно схеме, составить матрицу инцидентности и матрицу смежности. Удалить вершину номер 6. Найти все вершины, достижимые из вершины 4, с помощью поиска в глубину.



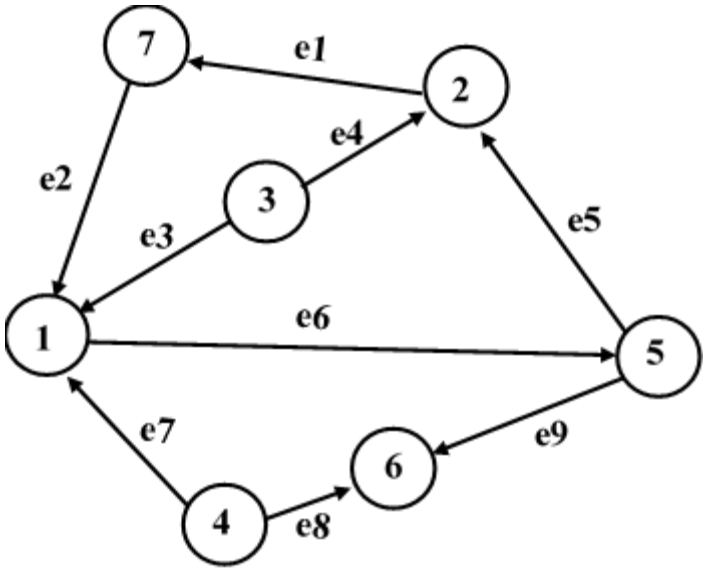
5 Построить граф согласно схеме, составить список смежности и матрицу инцидентности. Удалить ребра e3, e7, e10. Найти все вершины, достижимые из вершины 6, с помощью поиска в ширину.



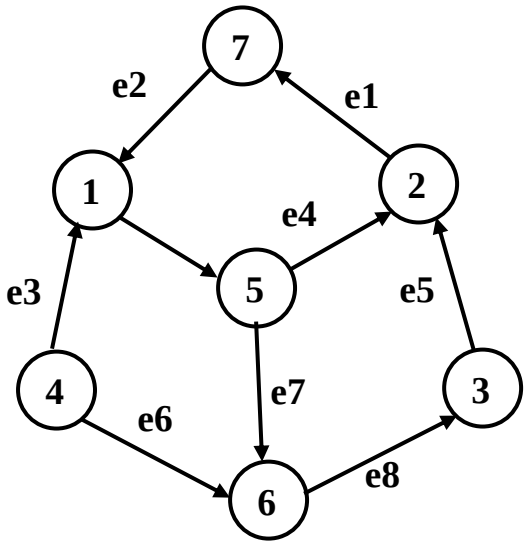
6 Построить граф согласно схеме, составить список смежности и матрицу смежности. Удалить вершину номер 1. Найти все вершины, достижимые из вершины 7, с помощью поиска в глубину.



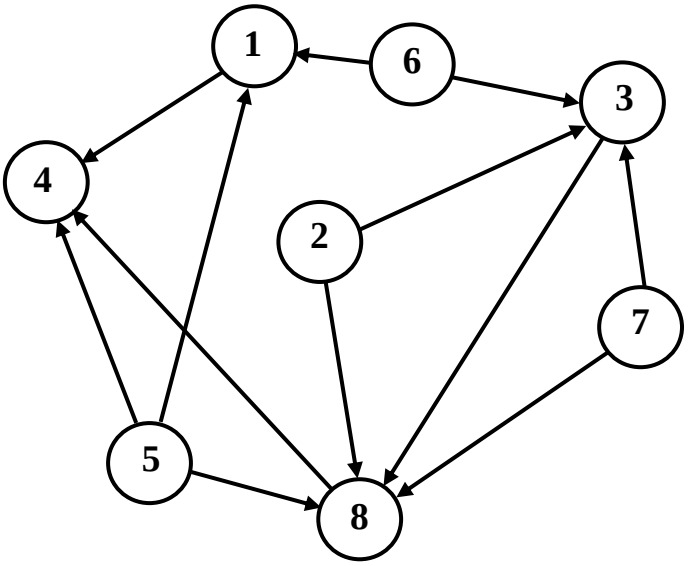
7 Построить граф согласно схеме, составить список смежности и матрицу инцидентности. Удалить ребро e3. Найти все вершины, достижимые из вершины 4, с помощью поиска в ширину.



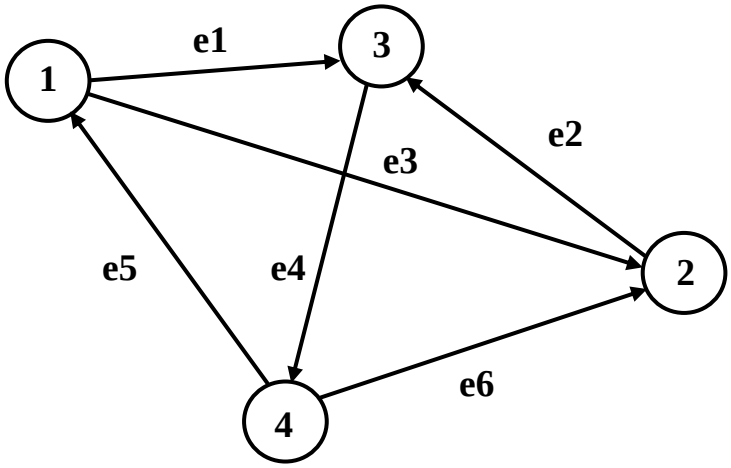
8 Построить граф согласно схеме, составить список смежности и матрицу смежности. Удалить ребро e2. Найти все вершины, достижимые из вершины 5, с помощью поиска в глубину.



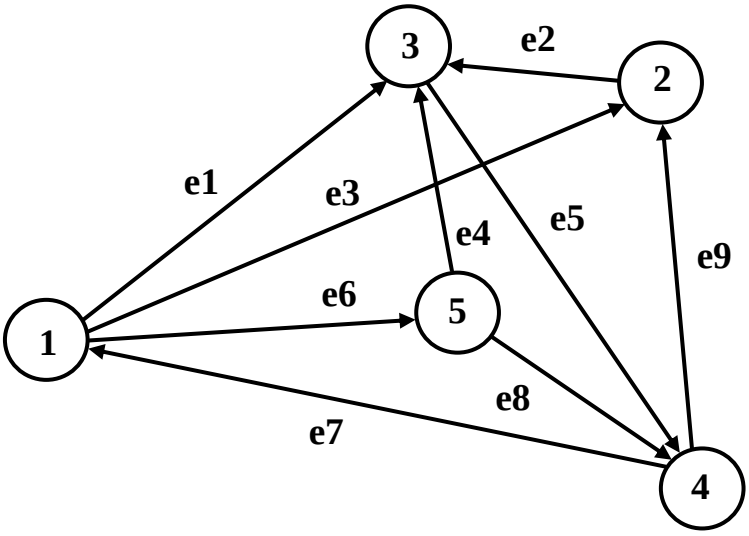
9 Построить граф согласно схеме, составить список смежности и матрицу смежности. Удалить вершины 2, 6. Найти все вершины, достижимые из вершины 5, с помощью поиска в ширину.



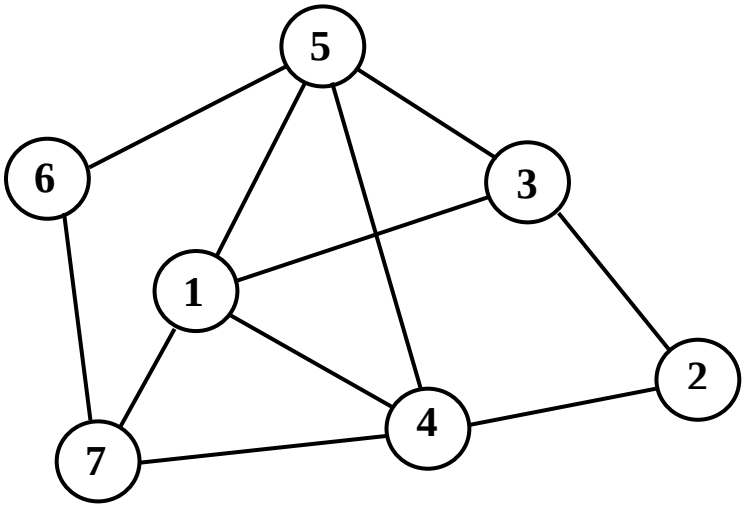
10 Построить граф согласно схеме, составить список смежности и матрицу инцидентности. Удалить ребра e1, e3. Найти все вершины, достижимые из вершины 4, с помощью поиска в глубину.



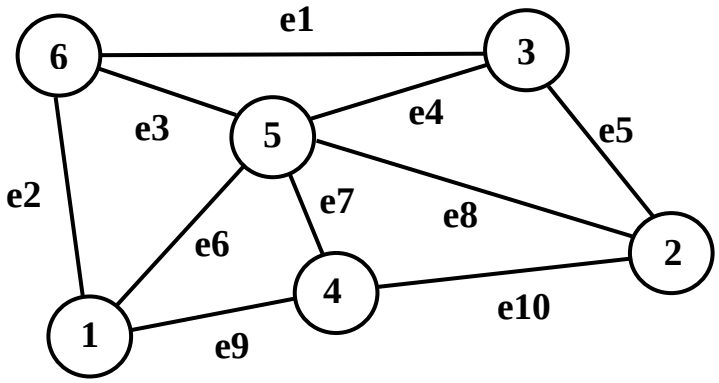
11 Построить граф согласно схеме, составить матрицу инцидентности и матрицу смежности. Удалить ребро e7. Найти все вершины, достижимые из вершины 1, с помощью поиска в ширину.



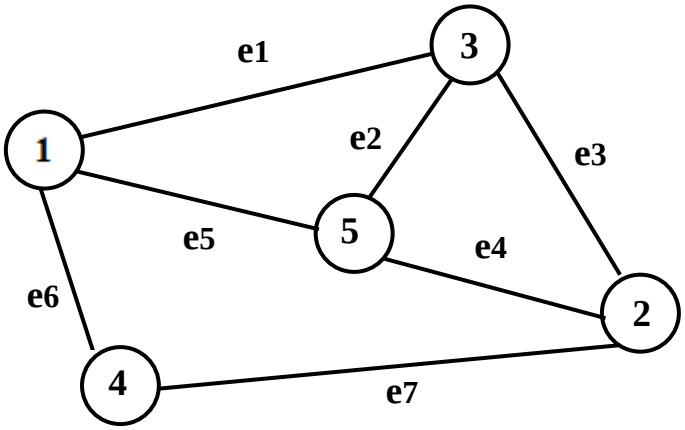
12 Построить граф согласно схеме, составить список смежности и матрицу смежности. Удалить вершину номер 1. Найти все вершины, достижимые из вершины 6, с помощью поиска в глубину.



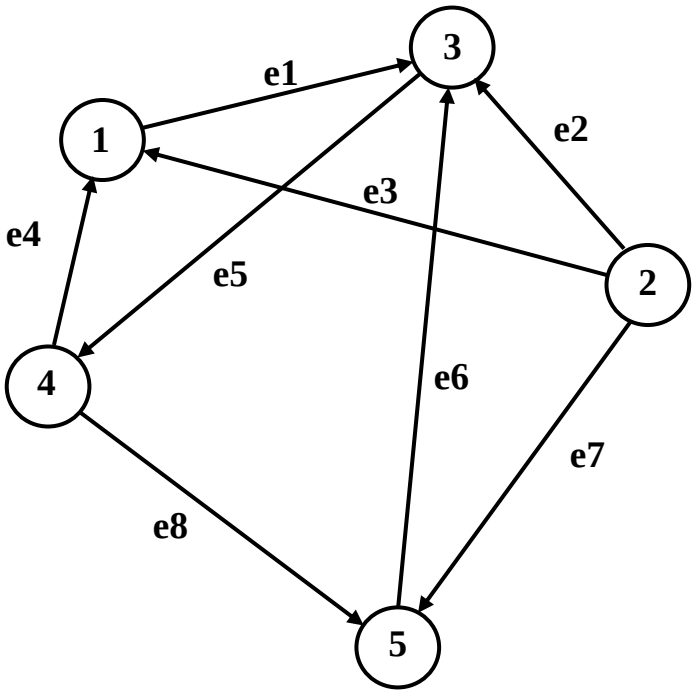
13 Построить граф согласно схеме, составить матрицу инцидентности и матрицу смежности. Удалить ребра e7, e9. Найти все вершины, достижимые из вершины 3, с помощью поиска в ширину.



14 Построить граф согласно схеме, составить список смежности и матрицу инцидентности. Удалить ребро e3. Найти все вершины, достижимые из вершины 5, с помощью поиска в глубину.

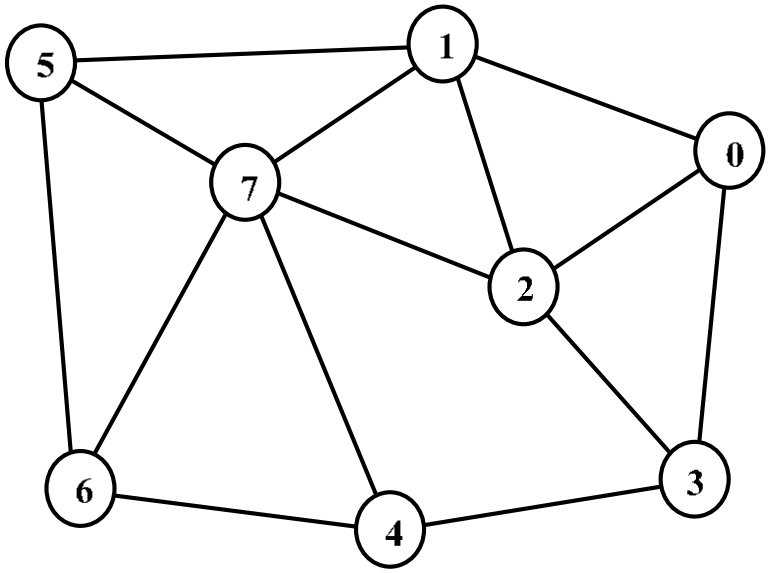


15 Построить граф согласно схеме, составить список смежности и матрицу смежности. Удалить ребра e3, e2. Найти все вершины, достижимые из вершины 4, с помощью поиска в ширину.



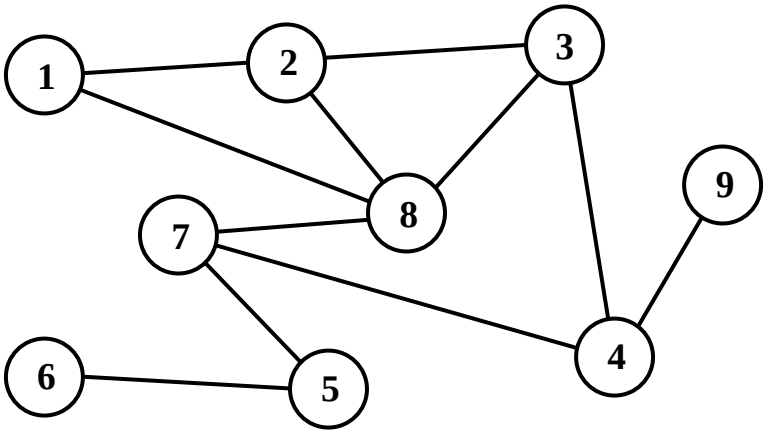
16 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 1, найти вершины между вершинами 2 и 5.



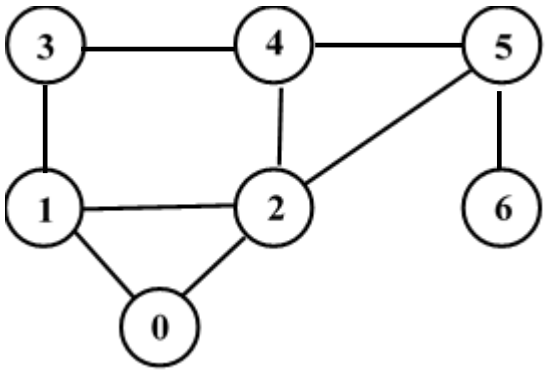
17 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 9, найти вершины между вершинами 1 и 4.



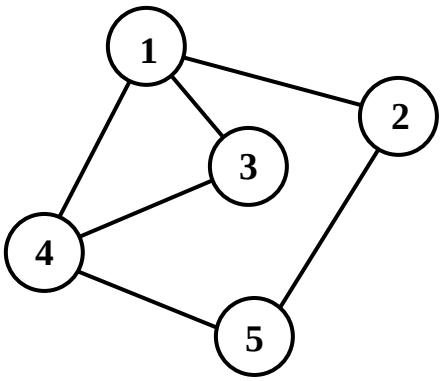
18 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 4, найти вершины между вершинами 6 и 0.



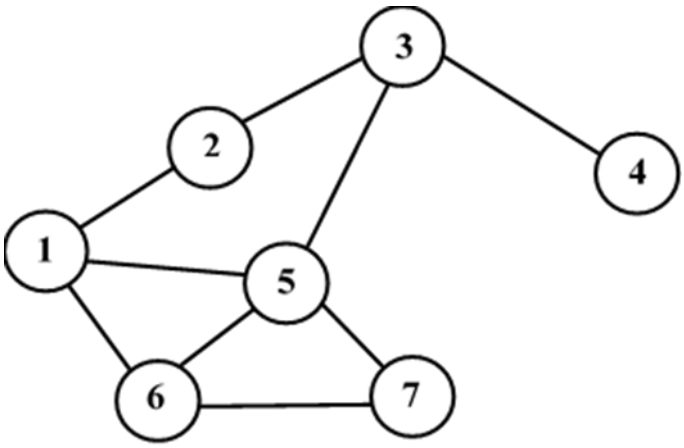
19 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 1, найти вершины между вершинами 5 и 3.



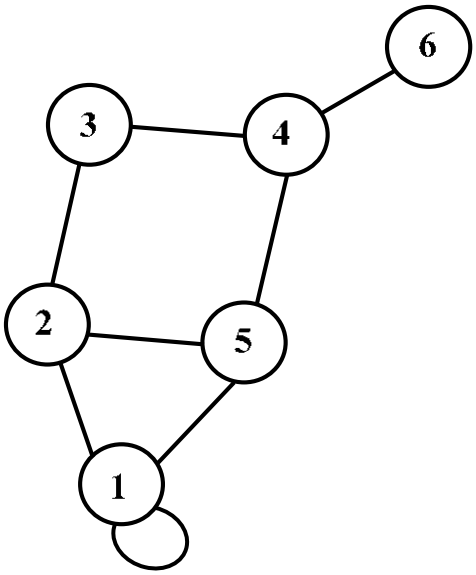
20 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 7, найти вершины между вершинами 3 и 6.



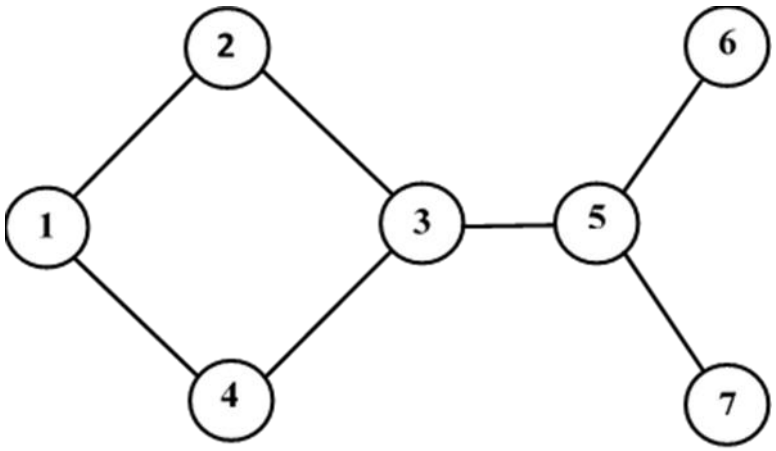
21 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 1, найти вершины между вершинами 2 и 4.



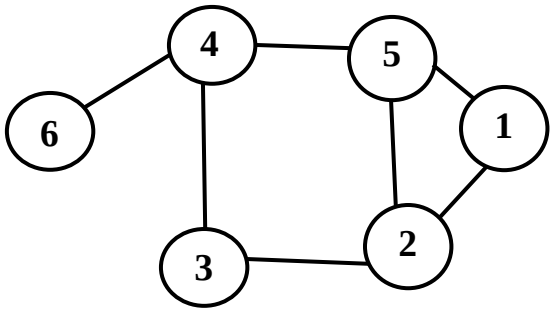
22 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 4, найти вершины между вершинами 2 и 7.



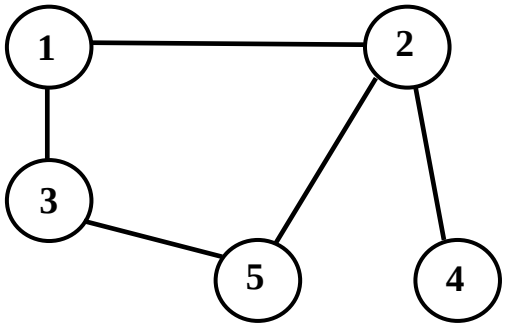
23 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 1, найти вершины между вершинами 3 и 5.



24 Построить граф согласно схеме. Построить матрицу смежности для данного графа. По запросу пользователя выполнить действия: удалить вершину; найти все вершины, лежащие на кратчайшем пути между заданными вершинами; вывести на экран результирующий граф.

Дополнительное задание: удалить вершину 5, найти вершины между вершинами 1 и 4.



Список использованных источников

1 Кириенко, Н. А. Основы алгоритмизации и программирования. Лабораторный практикум в 2 ч. Ч. 1 : Использование языка Си в интегрированной среде разработки Visual Studio : пособие / Н. А. Кириенко, Е. А. Полоско, А. А. Ефремов. – Минск : БГУИР, 2024. – 123 с.

2 Основы информатики и программирования : лаб. практикум для студ. спец. I-40 01 02-02 «Информационные системы и технологии в экономике» всех форм обуч. В 2 ч. Ч. 2 / Е. Н. Живицкая [и др.]. – Минск : БГУИР, 2007. – 108 с.

3 Седжвик, Р. Алгоритмы на C++. Анализ структуры данных. Сортировка. Поиск. Алгоритмы на графах / Р. Седжвик, Дж. Ван Вик Кристофер. – М. : Вильямс, 2019. – 1056 с.

4 Демидович, Е. М. Основы алгоритмизации и программирования. Язык Си / Е. М. Демидович. – СПб. : БХВ-Петербург, 2006. – 440 с.

5 Кнут, Д. Искусство программирования. В 3 т. Т. 1–3 / Д. Кнут. – М. : Диалектика-Вильямс, 2019. – 832 с.

6 Луцик, Ю. А. Основы алгоритмизации и программирования: язык Си : учеб.-метод. пособие / Ю. А. Луцик, А. М. Ковальчук, Е. А. Сасин. – Минск : БГУИР, 2015. – 169 с.

7 Котов, В. М. Структуры данных и алгоритмы. Теория и практика : учеб. пособие / В. М. Котов, Е. П. Соболевская. – Минск : БГУ, 2004. – 267 с.

8 Навроцкий, А. А. Основы алгоритмизации и программирования в среде Visual C++ : учеб.-метод. пособие / А. А. Навроцкий. – Минск : БГУИР, 2014. – 160 с. : ил.

9 Алгоритмы. Построение и анализ / Т. Кормен [и др.]. – М. : Вильямс, 2019. – 1328 с.

10 Культин, Н. Б. C/C++ в задачах и примерах / Н. Б. Культин. – СПб. : БХВ-Петербург, 2019. – 272 с.

11 Керниган, Б. Язык программирования C / Б. Керниган, Д. Ритчи. – 2-е изд., перераб. и доп. – М. : Вильямс, 2009. – 304 с.

Св. план 2025, поз. 40

Учебное издание

Кириенко Наталья Алексеевна
Петрович Юлия Юрьевна
Ефремов Андрей Александрович

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

В двух частях
Часть 2

РЕАЛИЗАЦИЯ АЛГОРИТМОВ НА ЯЗЫКЕ СИ В ИНТЕГРИРОВАННОЙ СРЕДЕ РАЗРАБОТКИ VISUAL STUDIO

ПОСОБИЕ

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *А. А. Лущикова*

Подписано в печать 22.01.2026. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 6,63. Уч.-изд. л. 6,7. Тираж 50 экз. Заказ 64.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск