Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)**

Кафедра автоматизации обработки информации (АОИ)

ПРОГРАММИРОВАНИЕ

Методические указания к лабораторным работам и
организации самостоятельной работы
для студентов направления
«Бизнес-информатика»
(уровень бакалавриата)

2018

Пермякова Наталья Викторовна
Программирование: Методические указания к лабораторным работам и организации самостоятельной работы для студентов направления «Бизнес-информатика» (уровень бакалавриата) / Н.В. Пермякова. — Томск, 2018. — 51 с.

Содержание

1 Введение

В методических указаниях к проведению лабораторных работ и организации самостоятельной работы по дисциплине «Программирование» собраны методические рекомендации для поддержки аудиторных занятий и самостоятельной работы студентов.

Целью проведения лабораторных работ и организации самостоятельной работы является формирование навыков структурного программирования для решения прикладных задач.

По окончанию обучения дисциплины «Программирование» студент должен:

— **знать** базовые структуры представления информации в компьютерных программах; алгоритмы сортировки массивов, способы оценки эффективности алгоритмов; машинные способы представления графов; алгоритмы генерации комбинаторных алгоритмов.

— **уметь** разрабатывать алгоритмы прикладных задач; выполнять осознанный выбор структуры представления данных в компьютерной программе;

— **владеть навыками** реализации и отладки программ на языке программирования Си.

Цикл лабораторных работ по дисциплине можно условно разделить на три группы. Первая группа лабораторных работ формирует навыки использования различных структур представления данных в программировании. Вторая группа лабораторных работ формирует навыки сравнения алгоритмов по эффективности. Третья группа лабораторных работ является компьютерным практикумом по темам, изученным в курсе дисциплины «Дискретная математика».

Самостоятельная работа студентов по дисциплине содержит несколько видов деятельности — проработка лекционного материала, подготовка к контрольным работам, подготовка к лабораторным работам, выполнение домашних заданий, подготовка к экзамену.

Необходимые базовые знания для дисциплины «Программирование» обеспечивают дисциплины «Информатика» и «Дискретная математика».

2 Методические указания к проведению лабораторных работ

2.1 Общие положения

Целью проведения лабораторных работ является формирование и развитие навыков применения знаний и умений, полученных в курсах «Информатика» и «Дискретная математика» при решении практических задач.

Основной формой проведения лабораторных работ является разработка алгоритма решения индивидуальной задачи и его программная реализация на языке Си. Процесс программной реализации включает в себя написание программы, отладку программы и тестирование программы.

К основным способам контроля формирования компетенций при выполнении лабораторных работ относятся индивидуальная защита выполненной работы, организация входного контроля знаний студентов по теоретическому материалу дисциплины, практическое применение которого осуществляется в ходе выполнения лабораторной работы.

Для получения максимальной оценки за лабораторную работу необходимо выполнить и защитить работу во время, отведенное для ее выполнения, согласно расписанию занятий. Допускается досрочное выполнение лабораторной работы по предварительной договоренности с преподавателем.

Выполнение всех лабораторных работ, предусмотренных рабочей программой дисциплины, является условием допуска к итоговому контролю изучения дисциплины — экзамену.

2.2 Лабораторная работа «Линейные динамические списки»

Цель работы: формирование навыков хранения обрабатываемой информации в динамических линейных списках.

Форма проведения: выполнение индивидуального задания.

Подготовка к выполнению лабораторной работы

Для выполнения лабораторной работы необходимо изучить теоретический материал, изложенный ниже.

Динамические структуры данных

При решении прикладных задач часто приходится использовать данные, размер и структура которых должны меняться в процессе работы. В этом случае значение объема выделяемой памяти выясняется только в процессе работы. В таких случаях применяют данные, которые представляют собой отдельные элементы, связанные с помощью ссылок.

Каждый элемент (узел) состоит из двух областей памяти: информационного поля и ссылок (рис. 1).

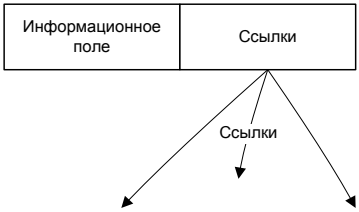


Рисунок 1 — Структура узла списка

Ссылочные данные хранят адреса других узлов этого же типа. В языке Си для организации ссылок используются переменные-указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в списке используются нулевой указатель — NULL.

Линейный список

В простейшем случае каждый узел содержит всего одну ссылку на следующий элемент. У последнего в списке элемента поле ссылки содержит нулевой указатель NULL. Чтобы не потерять информацию о списке, необходимо организовать хранение адрес первого узла списка в отдельной переменной. Первый узел списка будем называть «головой» списка (рис. 2).

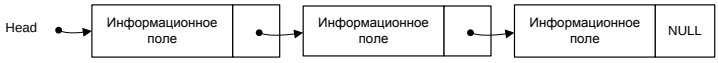


Рисунок 2 — Структура линейного списка

Для программной реализации необходимо объявить два новых типа данных — узел списка `Node` и указатель на него `PNode`. Узел представляет собой структуру, которая содержит два поля — целое число и указатель на такой же узел. Правилами языка Си допускается объявление:

```
struct Node {

int count; // информационное поле
Node *next;  // ссылка на следующий узел
};

typedef  Node *PNode; // тип данных: указатель на узел
```

В дальнейших приведенных примерах указатель *Head* указывает на начало списка, то есть, объявлен в виде: `PNode Head = NULL.` Такое определение первого элемента говорит о том, что в начале работы список пуст.

Создание элемента списка

Для добавления узла к списку необходимо выделить память под хранение узла и запомнить адрес выделенного блока памяти. Функция, создающая новый узел в памяти и возвращает его адрес может быть реализована следующим образом:

```
PNode CreateNode (int NewData)
{
PNode NewNode = new Node;  // указатель на новый узел
NewNode->count = NewData;  // запись информационного поля
                                  //нового узла
NewNode->next = NULL;  // следующего узла нет
return NewNode;  // результат функции — адрес созданного узла
}
```

После выполнения этой функции, узел необходимо добавить к списку (в начало, в конец или в середину).

Добавление узла

Добавление узла в начало списка

При добавлении нового узла `NewNode` в начало списка ссылка узла `NewNode` устанавливается на голову существующего списка (рис. 3)

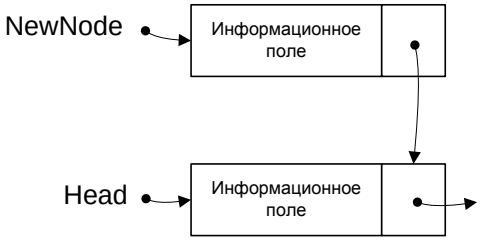


Рисунок 3 — Связь нового узла с «головой» списка

и изменяется переменная, в которой хранится «голова» (рис. 4).

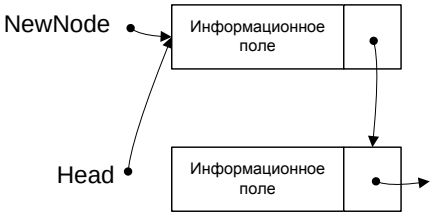


Рисунок 4 — Изменение адреса «головы» списка

По такой схеме работает функция `AddFirst`. Предполагается, что адрес начала списка хранится в `Head`. Обратите внимание, что здесь и далее адрес начала списка передается по ссылке, так как при добавлении нового узла он изменяется внутри процедуры.

```
void AddFirst (PNode *Head, PNode NewNode)
{
NewNode->next = Head;
*Head = NewNode;
}
```

Добавление узла после заданного

Предположим, что по условию задачи узел `NewNode` необходимо вставить после узла с заданным адресом `p`. Такая операция выполняется в два этапа:

1) установить ссылку нового узла на узел, следующий за данным;
2) установить ссылку данного узла `p` на `NewNode` (рис. 5).

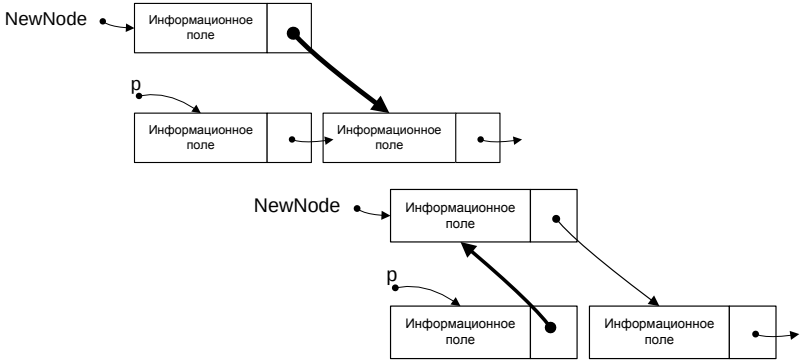


Рисунок 5 — Добавление узла перед заданным

```
void AddAfter (PNode p, PNode NewNode)
{
NewNode->next = p->next;
p->next = NewNode;
}
```

Добавление узла перед заданным

Такая схема добавления самая сложная. Проблема заключается в том, что в простейшем линейном списке (он называется односвязным, потому что связи направлены только в одну сторону) для того, чтобы получить адрес предыдущего узла, нужно пройти весь список сначала.

Задача сводится либо к вставке узла в начало списка (если заданный узел — первый), либо к вставке после заданного узла.

```
void AddBefore(PNode *Head, PNode p, PNode NewNode)
{
PNode q = *Head;
if (Head == p) {
AddFirst(Head, NewNode); // вставка перед первым узлом
return;
}
while (q && q->next!=p) // ищем узел, за которым следует p
q = q->next;
if (q) // если нашли такой узел,
AddAfter(q, NewNode); // добавить новый после него
}
```

Добавление узла в конец списка

Для решения задачи необходимо сначала найти последний узел, а затем воспользоваться процедурой вставки после заданного узла. Отдельно необходимо обработать случай, когда список пуст.

```
void AddLast(PNode *Head, PNode NewNode)
{
PNode q = *Head;
if (*Head == NULL) {  // если список пуст,
AddFirst(Head, NewNode);  // вставляем первый элемент
return;
}
while (q->next) q = q->next;  // ищем последний элемент
AddAfter(q, NewNode);
}
```

Проход по списку

Для обхода всего списка необходимо организовать проход с «головы» и, используя указатель `next`, продвигаться к следующему узлу.

```
PNode p = Head;  // начали с головы списка
while ( p != NULL ) {  // пока не дошли до конца
p = p->next;  // переходим к следующему узлу
}
```

Поиск узла в списке

Часто требуется найти в списке нужный элемент (его адрес или данные). Надо учесть, что требуемого элемента может и не быть, тогда просмотр заканчивается при достижении конца списка. Такой подход приводит к следующему алгоритму:

1) начать с головы списка;

2) пока текущий элемент существует (указатель — не равен NULL), проверить нужное условие и перейти к следующему элементу;

3) закончить, когда найден требуемый элемент или все элементы списка просмотрены.

Например, следующая функция ищет в списке элемент, соответствующий заданному значению (для которого поле count совпадает с заданным значением `NewNode`), и возвращает его адрес или NULL, если такого узла нет.

```
PNode Find (PNode Head, char NewWord[])
{
PNode q = Head;
while (q && strcmp(q->word, NewWord))
q = q->next;
return q;
}
```

Удаление узла

Эта процедура связана с поиском заданного узла по всему списку, так как необходимо поменять ссылку у предыдущего узла, а перейти к нему непосредственно невозможно. Если найден узел, за которым идет удаляемый узел, необходимо просто переставить ссылку (рис. 6).

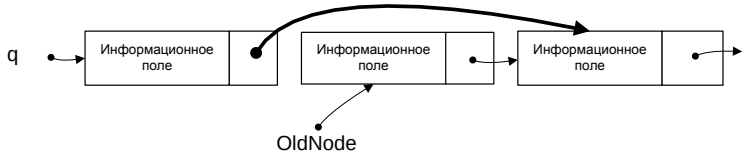


Рисунок 6 — Удаление узла

Отдельно обрабатывается случай, когда удаляется первый элемент списка.

```
void DeleteNode(PNode *Head, PNode OldNode)
{
PNode q = Head;
if (*Head == OldNode)
Head = OldNode->next; // удаляем первый элемент
else {
while (q && q->next != OldNode) // ищем элемент
q = q->next;
if ( q == NULL ) return; // если не нашли, выход
q->next = OldNode->next;
}
delete OldNode; // освобождаем память
}
```

Порядок выполнения работы

1. Самостоятельно разработать алгоритм индивидуального варианта.
2. Реализовать построенный алгоритм на языке Си.
3. Отладить и протестировать программу.
4. Защитить работу.

2.3 Лабораторная работа «Организация прямого доступа к двоичным файлам»

Цель работы: освоить навыки работы с двоичными файлами.

Форма проведения: выполнение индивидуального задания.

Рекомендации по подготовке к лабораторной работе: перед проведением занятия необходимо изучить теоретический материал, изложенный в [1]. Глава 9 пособия, стр. 156 — 166.

Порядок проведения работы

1. Получить индивидуальный вариант.
2. Написать, отладить и выполнить программу, создающую двоичный файл.
3. Написать программу, выполняющую задание, сформулированное в индивидуальном варианте.
4. Отладить и протестировать программу.
5. Защитить работу.

Пример организации прямого доступа

В двоичном файле записана размерность квадратной матрицы и сама вещественная матрица. Приведенная ниже программа на языке Си читает из файла элементы заданного столбца *k*. Решение оформлено в виде функций.

```
#include <stdio.h>
#include <stdlib.h>
// Создание двоичного файла с именем name,
// содержащем n*n вещественных чисел.
void create_file(char *name, int n){
  FILE *f = fopen(name,"wb");
  if (f==NULL) {
          printf("Ошибка создания файла. \n");
          system("pause");
```

```
          return ;}
// Запись в файл переменной n
fwrite(&n,sizeof(n),1,f);
int i;
// Запись  в файл элементов квадратной матрицы
  for( i=0;i<n*n;i++)
  {   float z = rand()%200/(rand()%100+1.)-rand()%70;
      fwrite(&z,sizeof(float),1,f);
   }   fclose(f);}
// Функция чтения файла с именем name
void read_file(char *name)
{ int m;
  FILE *f = fopen(name,"rb");
  if (f==NULL) {
          printf("Файл не найден. \n");
          system("pause");
          return ;}
  int i = 0;
// Чтение размерности матрицы.
  float z;
  fread(&m,sizeof(int),1,f);
 // Чтение всей матрицы и вывод ее на экран.
 while(!feof(f))  {
   if (i>=m&&i%m==0)
       printf("\n");
   if(fread(&z,sizeof(float),1,f)!=1) break;
   printf("%8.3f",z);
   i++;   }
   fclose(f);}
// Чтение элементов k-того столбца
void read_k(char *name, int k){
  int m;
  FILE *f = fopen(name,"rb");
  if (f==NULL) {
          printf("Файл не найден. \n");
          system("pause");
          return ;}
  float z;
  int i;
  fread(&m,sizeof(int),1,f);
```

```
  for( i=0;i<m;i++) {
    int l =i*m*sizeof(float) +
           k*sizeof(float)+sizeof(int);
    fseek(f, l,SEEK_SET);
    fread(&z,sizeof(z),1,f);
    printf("%8.3f",z);   }
   fclose(f);}

int main(int argc, char *argv[])
{
char Fname[30];
int n,m;
printf("Введите имя создаваемого файла: ");
scanf("%s",Fname);
printf("Введите количество строк матрицы: ");
scanf("%d",&n);
printf("Введите номер столбца: ");
scanf("%d",&m);
create_file(Fname,n);
printf("Матрица: \n");
read_file(Fname);
printf("Элементы столбца с номером %d \n",m);
read_k(Fname,m);
system("pause");
return 0;
}
```

2.4 Лабораторная работа «Простые сортировки на месте»

Цель работы: ознакомиться и реализовать простые методы сортировки — сортировки обменом, выбором, вставками, бинарными вставками. Исследовать сложность указанных алгоритмов сортировки.

Форма проведения: выполнение индивидуального задания.

Рекомендации по подготовке к лабораторной работе: для выполнения лабораторной работы необходимо ознакомиться с алгоритмами простых сортировок.

Сортировка обменом. Одним из простых методов сортировки на месте (т.е. при работе не требуется дополнительный объем памяти) является сортировка обменом или «пузырьковая» сортировка.

Суть алгоритма состоит в следующем: сравниваются пары рядом стоящих элементов массива, если первый элемент пары меньше второго, то элементы меняются местами. После первого просмотра массива самый большой элемент встает на свое место, а маленькие по значению элементы на один шаг продвигаются к началу массива. Отсюда метод и получил свое название: «легкие» элементы плавно «всплывают» к началу массива. «Тяжелые» элементы быстро «тонут», встают в конец массива.

После того, как все пары элементов массива просмотрены, массив еще не будет отсортирован, поэтому необходимо просмотреть пары элементов еще раз, и тогда еще один самый большой элемент встанет на свое место.

Если массив состоит из n элементов, то пар в таком массиве ровно *n-1.* На каждом шаге алгоритма самый большой элемент массива становится на свое место. Поэтому количество просматриваемых пар уменьшается с каждым шагом на единицу.

Таким образом, в алгоритме явно просматриваются два вложенных цикла. Внутренний цикл отвечает за просмотр пары элементов, количество шагов этого цикла зависит от внешнего цикла. Чем больше шагов выполнил внешний цикл, тем меньше пар просматривает внутренний цикл. Т.к. при выполнении одного шага внешнего цикла самый большой элемент становится на место, то количество шагов цикла равно количеству элементов массива -1. Однако, массив может быть уже отсортированным, до окончания работы внешнего цикла. Можно досрочно остановить выполнение цикла, если на каком-то *i*- том шаге во внутреннем цикле не произошло ни одного обмена.

Покажем применение сортировки на массиве *1 5 2 4 3.*

1: 1 2 4 3 ***5***

2: 1 2 3 ***4 5***

3: ***1 2 3 4 5***

После третьего шага алгоритм может закончить свою работу, так как не было проведено ни одного обмена.

Сортировка выбором. Сортировка выбором использует другой принцип упорядочивания элементов. На начальном шаге алгоритма выбирается минимальный элемент и ставится на место нулевого элемента. На последующих, *i*- тых шагах выбирается минимальный элемент среди элементов от *i*- того до *(n-1)*-го.

Рассмотрим на примере массива *1 5 2 4 3.*

1: ***1*** *5 2 4 3* Первый минимальный элемент уже стоит на своем месте.

2: ***1 2*** *5 4 3* Второй минимальный элемент — 2, поменяем его местами с 5.

3: ***1 2 3*** *4 5* Третий минимальный элемент — 3, поменяем его местами с 5.

4: ***1 2 3 4*** *5* Четвертый минимальный элемент стоит на своем месте.

После этого сортировка заканчивается, т.к. последний элемент в любом случае будет стоять на своем месте.

Сортировка вставками. Сортировка вставками упорядочивает массив, выполняя следующие действия — на начальном шаге алгоритма считаем, что последовательность из одного, первого элемента упорядоченная последовательность. Найдем место в этой последовательности для второго элемента. После этого упорядочены уже первые два элемента. Далее в текущей упорядоченной последовательности ищутся места для третьего, четвертого и т.д. элементов.

При программировании поиск места для вставляемого элемента лучше всего осуществить с конца упорядоченной последовательности.

Рассмотрим на примере массива *1 5 2 4 3.*

1: ***1 5*** *2 4 3* В упорядоченную последовательность добавляется 5.

2: ***1 2 5*** *4 3* В упорядоченную последовательность добавляется 2.

3: ***1 2 4 5*** *3* В упорядоченную последовательность добавляется 4.

4: ***1 2 3 4 5*** В упорядоченную последовательность добавляется 3.

Порядок выполнения работы

1. Получить индивидуальный вариант.
2. Составить алгоритм простой сортировки.
3. Реализовать алгоритм на языке Си, добавив в программу подсчет количества сравнений и перестановок, проведенных алгоритмом.
4. Найти среднее количество сравнений и перестановок, выполняемых программой для сортировки массивов из 100, 200, 300, 500, …, 10000 элементов, результаты сохраните в текстовом файле.
5. Построить графики зависимости сложности алгоритма (количество сравнений + количество перестановок) от количества обрабатываемых данных.
6. Сделать выводы по работе.

2.5 Лабораторная работа «Улучшенные методы сортировки»

Цель работы: ознакомиться и реализовать методы сортировок, оценка скорости которых не является квадратичной.

Форма проведения: выполнение индивидуального задания.

Рекомендации по подготовке к лабораторной работе: для выполнения лабораторной работы необходимо ознакомиться с сортировками — Хоара, Шелла, пирамидальной, подсчета.

Сортировка Шелла. Сортировка Шелла — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки **вставками**. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами.

Сортировка Шелла была названа в честь её изобретателя — Да Шелла, который опубликовал этот алгоритм в 1959 году.

Выбор длин промежутков

- Первоначально используемая Шеллом последовательность длин промежутков: $d_1 = N/2, d_i = d_{i-1}/2, d_k = 1$ в худшем случае, сложность алгоритма составит $O(n^2)$.
- Предложенная Хиббардом последовательность: $2^i - 1 \leq N, i \in N$. Такая последовательность шагов приводит к алгоритму сложностью $O(n^{3/2})$, массив шагов заполняется перед сортировкой.
- Предложенная Седжвиком последовательность: $d_i = 9 \cdot 2^i - 9 \cdot 2^{i/2} + 1$, если *i* четное и $d_i = 8 \cdot 2^i - 6 \cdot 2^{(i+1)/2} + 1$, если *i* нечетное. При использовании таких приращений средняя сложность алгоритма составляет: $O(n^{7/6})$, а в худшем случае порядка $O(n^{4/3})$. Массив приращений заполняется перед сортировкой. Последнее значение массива шаг[*s*-1], если 3*шаг[*s*] > N (если размер массива меньше 3-х шагов).
- Наиболее часто используемая последовательность шагов - d_i изменяется по правилу $d_{i+1} = (d_i - 1)/2$ (для массивов, содержащих более 500 элементов) и $d_{i+1} = (d_i - 1)/3$(для массивов, содержащих менее 500 элементов). За d_0 принимается число элементов массива. Метод заканчивает работу, когда d_i становится меньше 1.

Комбинированная сортировка (сортировка «расческой»). Комбинация сортировки обменом и сортировки Шелла. На каждом шаге срав-

ниваются значения отстоящие друг от друга на заданное значение шага $H_{i+1} = 8 \cdot H_i/11$, но такое сравнение происходит всего один раз. Как только значение смещения становится равным 1, выполняется сортировка до конца методом пузырька. За H_0 принимается число элементов массива.

Пирамидальная сортировка. Пирамида — это частично упорядоченное двоичное дерево, элементы которого расположены в узлах дерева по следующему правилу — каждый элемент родительского узла обязательно больше элементов, расположенных в дочерних узлах. На рис. 7 представлена пирамида из 15 элементов:

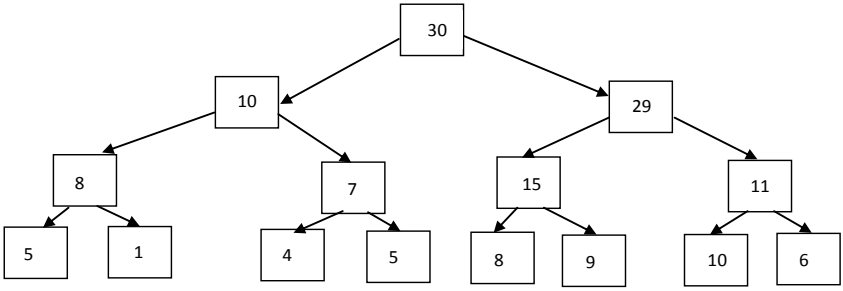


Рисунок 7 — Пирамидально упорядоченный массив

Элементы дерева легко представляются в виде массива — пусть родительский узел имеет индекс i, тогда дочерние узлы имеют индексы $2i$ и $2i + 1$. Рассмотренная пирамида может быть представлена массивом:

(30, 10, 29, 8, 7, 15, 11, 5, 1, 4, 5, 8, 9, 10, 6).

Т.к. корневой элемент элемент пирамиды всегда является максимальным элементом, то процесс пирамидальной сортировки можно описать следующим образом: поменять верхний элемент пирамиды с нижним элементом и рассматривать в дальнейшем не n элементов исходного массива, а $n - 1$ элемент. При выполнении этих действий нарушается правило расположения элементов в пирамиде, поэтому после обмена необходимо перестроить пирамиду с $n - 1$ элементами и далее, повторять два этих шага, пока пирамида не останется пустой. Таким образом, необходимо написать процедуру, строящую пирамиду для произвольного массива размерности *n*, далее алгоритм пирамидальной сортировки очень прост. Для формального описания алгоритма назовем процедуру построения пирамиды из массива размерностью *N KeyDown(N)* — т.к. элемент, находящийся в корне пирамиды может быть и не самым большим,

то необходимо опустить этот элемент на нижние уровни пирамиды, чтобы выполнялась частичная упорядоченность. Алгоритм может выглядеть следующим образом:

```
1. KeyDown(N,X); // Построение пирамиды на исходном массиве x.
2. X[1] ←→ X[N]; // Обмен первого элемента пирамиды с последним
3. L = N-1; // Изменение размерности пирамиды
4. Пока (L>1) // пока пирамида не пуста
        KeyDown(N-1,X);
        X[1] ←→ X[L];
        L = L-1;
5. Конец.
```

Рассмотрим процесс построения пирамиды на произвольном массиве (в скобках после элементов указаны индексы):

Элементы массива: (25(1), 11(2), 5(3), 11(4), 4(5), 8(6), 3(7), 28(8), 18(9), 10(10), 1(11), 5(12), 4(13), 2(14), 17(15)). Начальное расположение элементов массива в узлах пирамиды показано на рис. 8.

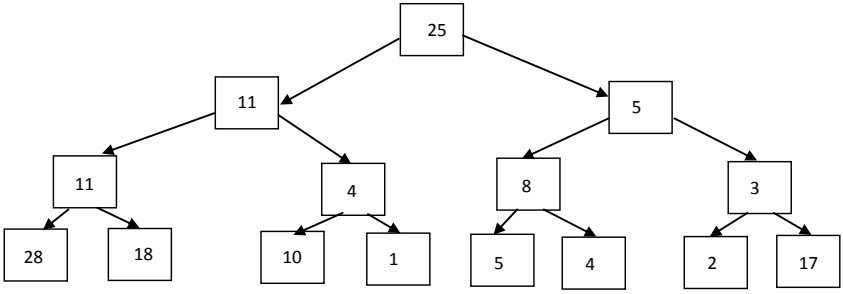


Рисунок 8 — Первый этап построения пирамиды

Размерность массива $N = 15$. Для элементов, находящихся на нижнем уровне не существует дочерних элементов, т.е. эти элементы могут не проверяться на выполнение правила пирамиды, индексы этих элементов от $N/2+1$ до N. Поэтому построение начинается с элемента с номером $N/2$, в рассматриваемом примере это x[7] = 3, сравним этот элемент с наибольшим из элементов x[14] и x[15], вторая нижняя пирамида остается без изменения, третья и четвертая пирамиды изменяются (см. рис. 9).

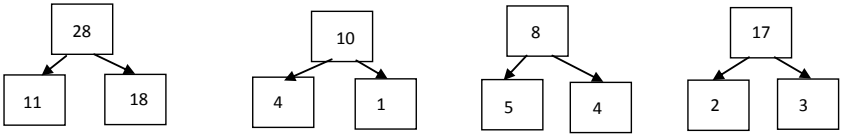


Рисунок 9 — Изменение порядка элементов нижнего уровня

Далее необходимо рассмотреть пирамиды с корневыми элементами во втором и третьем элементах (рис. 10):

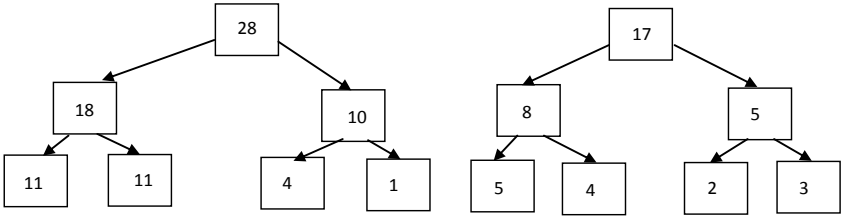


Рисунок 10 — Изменение пирамид среднего нижнего уровня

На рис. 11 изображен результат построения начальной пирамиды на массиве из 15 элементов.

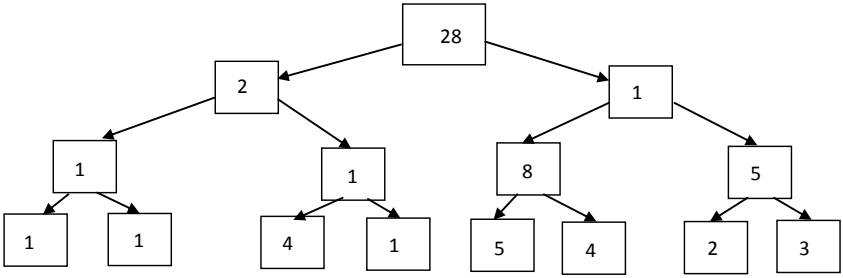


Рисунок 11 — Пирамидально упорядоченный массив

Очевидно, что процедура KeyDown(*N*,*X*) должна зависеть еще от одного параметра — номера элемента, для которого строится пирамида.

В общем случае для построения пирамиды с корнем в *L*-том элементе необходимо итеративно выполнить продвижение элемента по дереву вниз (при этом, элемент с номером *L* меняется местами с большим из своих потомков), продвижение элемента заканчивается, когда значение

элемента в позиции *L* становится больше значений элементов-потомков, или когда достигнут нижний уровень.

Полный алгоритм пирамидальной сортировки выглядит следующим образом:

```
1. L = (N/2) +1;
2. Пока L>1
        L = L-1;
        KeyDown(L,N,X);
3. N1 = N;
4. Пока N1>1
        V = x[1]; x[1] = x[N1]; x[N1] = v;
        N1 = N1 – 1;
        KeyDown(L,N1,X);
5. Конец.
```

Сортировка Хоара. Значение произвольного элемента, обычно центрального, принимается за значение опорного элемента. Организуется просмотр элементов массива. При движении по массиву слева направо ищется элемент больше или равный опорному. При движении справа налево ищется элемент меньше или равный опорному элементу. Найденные элементы меняются местами, далее продолжается встречный поиск. После этих действий массив окажется разделенным на две части. В первой части будут расположены элементы меньше либо равные опорному элементу, а справа - больше либо равные. Далее алгоритм рекурсивно выполняется для правой и левой частей.

Сортировка Хоара с выбором медианного элемента. Можно улучшить быструю сортировку, выбирая средний элемент таким образом, чтобы его значение было бы действительно близким к серединному значению массива. Для этого можно воспользоваться двумя стратегиями:

Выбор среднего значения осуществляется случайным образом (с использованием датчиков случайных чисел и информации о размерности массива). Т.к. разделяющий элемент выбирается при каждом вызове процедуры, случайный выбор может быть наиболее правильным и оградит от появления наихудшего случая — когда медианный элемент оказывается наименьшим или наибольшим.

Вторая стратегия состоит *в случайном выборе трех элементов*, по одному из начального, конечного и среднего интервалов сортируемого подмассива. Как разделяющий элемент используется среднее из этих трех чисел.

Порядок выполнения работы

1. Получить индивидуальный вариант.
2. Составить алгоритм простой сортировки.
3. Реализовать алгоритм на языке Си, добавив в программу подсчет количества сравнений и перестановок, проведенных алгоритмом.
4. Найти среднее количество сравнений и перестановок, выполняемых программой для сортировки массивов из 100, 200, 300, 500, …, 10000 элементов, результаты сохраните в текстовом файле.
5. Построить графики зависимости сложности алгоритма (количество сравнений + количество перестановок) от количества обрабатываемых данных.
6. Сравнить полученные результаты с результатами лабораторных работ «Простые сортировки на месте» и «Оптимизация простых сортировок».
7. Сделать выводы по работе.

2.6 Лабораторная работа «Практические задачи теории множеств»

Цель работы: применение на практике теоретических знаний дисциплины «Дискретная математика».

Форма проведения: выполнение индивидуального задания.

Рекомендации по подготовке к лабораторной работе

Для выполнения лабораторной работы необходимо изучить теоретический материал, изложенный ниже.

Определение булеана конечного множества

Булеаном множества *M* называется множество всех подмножеств множества *M*. Для конечного множества мощности *n* мощность булеана равна 2^n.

Алгоритмы генерации булеана конечного множества

Алгоритм генерации всех подмножеств n-элементного множества

В памяти компьютера целые числа представляются кодами в двоичной системе счисления, причем число $2^n - 1$ представляется кодом, содержащим *n* единиц.

Тогда, число 0 является представлением пустого множества, число 1 является представлением подмножества, состоящего из первого элемента и т.д..

Описанный ниже алгоритм перечисляет все двоичные коды чисел от 0 и до $2^n - 1$, т.е. все подмножества конечного множества мощности *n*.

1. Задать *n*, и множество *A*, состоящее из *n* элементов
2. Цикл $\left(i = \overline{0; 2^n - 1}\right)$
   2.1. *M* — пустое подмножество
   2.2. Цикл ($j = \overline{0; n-1}$)
      2.2.1. Получить значение j -го разряда числа i
      2.2.2. Если полученное значение равно 1, то включить в подмножество *M* элемент $A[j]$
   2.3. Конец цикла
   2.4. Вывести полученное подмножество *M*.
   2.5. Конец цикла
3. Конец

Алгоритм построения бинарного кода Грея

Описанный далее алгоритм генерирует последовательность всех подмножеств *n*-элементного множества таким образом, что каждое последующее множество получается из предыдущего добавлением или удалением одного элемента.

Код Грея называется так же отраженным кодом. Рассмотрим построение кода на примере *n= 4.*

Будем считать старшим разрядом нулевой разряд. Он может принимать значения 0 и 1.

000**0**

000**1**

Далее старший разряд первый, который принимает значения 1, а младший разряд (нулевой) принимает значения в обратном порядке от предыдущего:

00**1****1**

00**1****0.**

Далее аналогичным образом: (старший разряд выделен размером, отражаемая часть жирным шрифтом)

0**1****10**

0**1****11**

0**1****01**

0100
1100
1101
1111
1110
1010
1011
1001

Пронумеруем полученные наборы от 0 до 14.

0	0000	**7**	0100
1	0001	**8**	1100
2	0011	**9**	1101
3	0010	**10**	1111
4	0110	**11**	1110
5	0111	**12**	1010
6	0101	**13**	1011
		14	1001

В первом наборе инвертировался разряд с номером 0, во втором — разряд с номером 1, в третьем — разряд с номером 0, в четвертом разряд с номером 2 и т.д.. Разложим числа от 1 до 14 на простые множители и подсчитаем количество двоек в разложении числа. Демонстрация алгоритма представлена в табл. 1.

Таблица 1 — Отраженный код Грея

Число	Разложение	Количество двоек в разложении числа	Число	Разложение	Количество двоек в разложении числа
1	1	0	8	2*2*2*2	3
2	2	1	9	3*3	0
3	3	0	10	2*5	1
4	2*2	2	11	11	0
5	5	0	12	2*2*3	2
6	2*3	1	13	13	0
7	7	0	14	2*7	1

1. Задать A — множество из n элементов.
2. Задать M = [000..] — подмножество булеана.
3. Вывести M;
4. Цикл $\left(i = \overline{1;2^n - 1}\right)$
4.1. Найти k — количество двоек в разложении числа i ;
4.2. Если $M[k] = 0$ То {$M[k] = 1$
Иначе $M[k] = 0$;
4.3. Вывести M;
5. Конец цикла
6. Конец

Реализация кода Грея с помощью стека

1. СТЕК ← пустой стек
2. Цикл ($j = \overline{n-1;0}$)
    2.1. $g_j = 0$
    2.2. СТЕК ← j
3. Конец цикла
4. Печать $(g_{n-1}, g_{n-2}, \ldots, g_0)$
5. Пока (СТЕК не пуст);
    5.1. СТЕК → a
    5.2. $g_a := \overline{g_a}$ {Инвертировать g_a }
    5.3.Печать $(g_{n-1}, g_{n-2}, \ldots, g_0)$
    5.4. Цикл ($j = \overline{a-1;0}$)
        5.4.1. СТЕК ← j
    5.5. Конец цикла
6. Конец цикла

Порядок выполнения работы

1. Получить индивидуальное задание.
2. Составить алгоритм решения задания и реализовать его графическое представление.
3. Составить программу на языке Си.
4. Выполнить отладку и тестирование программы.
5. Защитить работу.

2.7 Лабораторная работа «Генерация комбинаторных объектов»

Цель работы: ознакомление с алгоритмами, генерирующими комбинаторные объекты и их программная реализация.

Форма проведения: выполнение индивидуального задания.

Рекомендации по подготовке к лабораторной работе: для выполнения лабораторной работы необходимо изучить теоретический материал, изложенный ниже.

Генерация сочетаний

Генерация сочетаний в лексикографическом порядке

Будем рассматривать в качестве множества $X = \{1,2,\ldots,n\}$. Требуется сгенерировать все подмножества мощности $k, (0 \le k \le n)$ множества *X*.

Определим отношение лексикографического порядка ($\prec$) следующим образом. Пусть $a = (a_1, a_2, \ldots, a_n)$, $b = (b_1, b_2, \ldots, b_m)$. Будем говорить, что набор *a* предшествует набору *b*: $a \prec b \Leftrightarrow \exists r \ge 1 : a_r < b_r$ и $\forall i = \overline{1, r-1}$; $a_i = b_i$.

Будем рассматривать сочетания *k* элементов из множества *X* как вектор $(c_1, c_2, \ldots, c_k)$, компоненты которого расположены в порядке возрастания слева направо (т.е. $c_i < c_{i+1}$ для любого *i*). Начиная с сочетания $(1,2,\ldots,k)$, следующие будем строить, просматривая текущее справа налево, чтобы найти самый первый элемент, не достигший максимального значения; этот элемент увеличим на единицу, а всем элементам справа от него присвоим номинальные наименьшие значения.

Лексикографический порядок порождения сочетаний не является алгоритмом с минимальными изменениями.

1. $c_0 := -1$
2. Цикл ($i := \overline{1, k}$)
   2.1. $c_i := i$
3. Конец цикла
4. $j := 1$
5. Пока ($j \ne 0$)
   5.1.Печать $(c_1, c_2, \ldots, c_k)$
   5.2. $j := k$
   5.3. Пока ($c_j = n - k + j$)

5.3.1 $j := j-1$

5.4. Конец цикла

5.5. $c_j := c_j + 1$

5.6.Цикл $(i := \overline{j+1,k})$

5.6.1. $c_i := c_{i-1} + 1$

5.7. Конец цикла

6. Конец цикла

7. Конец

Генерация сочетаний с помощью кодов Грея

При генерации сочетаний из *n* элементов по *k* наименьшим возможным изменением при переходе от текущего сочетания к следующему является замена одного элемента другим. В терминах Грея это означает, что мы хотим выписать все *n*-разрядные кодовые слова, содержащие ровно *k* единиц, причем последовательные наборы отличаются ровно в двух разрядах (в одном из разрядов 0 заменяется на 1, а в другом — 1 на 0).

Пусть $G(n)$ — двоично-отраженный код Грея, а $G(n,k)\ (0 \le k \le n)$ — последовательность кодовых слов ровно с *k* единицами:

$$G(n,k)^T = \left(G(n,k)_1, G(n,k)_2, \ldots, G(n,k)_{C_n^k}\right)^T.$$

Эту последовательность можно рекурсивно определить следующим образом:

$$G(n,0) = (0\,0\ldots0);$$

$$G(n,n) = (1\,1\ldots1);$$

$$G(n,k) = \begin{pmatrix} 0 & G(n-1,k) \\ 1 & \overline{G(n-1,k-1)} \end{pmatrix}, \qquad (1)$$

где 0 — вектор-столбец размерности $C_{n-1}^k \times 1$, состоящий из нулей;

1 — вектор-столбец размерности $C_{n-1}^{k-1} \times 1$, состоящий из единиц;

$G(n-1,k)$ — матрица $C_{n-1}^k \times (n-1)$ кодовых слов, содержащих ровно *k* единиц;

$\overline{G(n-1,k-1)}$ — матрица $C_{n-1}^{k-1} \times (n-1)$ кодовых слов, содержащих ровно $k-1$ единиц, причем кодовые слова записаны в порядке, обратном порядку $G(n-1,k-1)$ ($\overline{G}$ — «перевернутая» матрица *G*).

На рисунке 12 приведен пример построения кодовых слов Грея для генерации сочетаний из 4 элементов по 2.

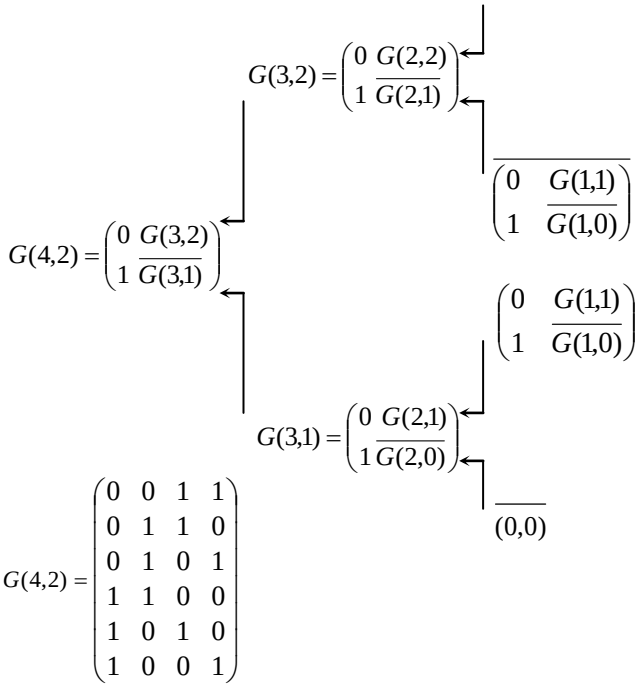


Рисунок 12 — Кодовые слова Грея для сочетаний из 4 по 2

Индукцией по n доказывается, что последовательность кодовых слов $G(n,k)$ получается удалением из кода Грея $G(n)$ всех кодовых слов с числом единиц, не равным k, причем в этой последовательности любые два соседних кодовых слов различаются только в двух позициях.

Генерация перестановок

Генерация перестановок в лексикографическом порядке

Будем рассматривать исходное множество $X=\{1,2,\ldots,n\}$, и в качестве начальной перестановки возьмем $\pi'=(1,2,\ldots,n)$. Условие окончания работы — порождение перестановки $\pi''=(n,n-1,\ldots,2,1)$, которая является последней в лексикографическом смысле среди всех перестановок множества X. Переход от текущей перестановки $\pi=(\pi_1,\pi_2,\ldots,\pi_n)$ к следующей за ней будем осуществлять таким образом:

1) просматривая перестановку π справа налево, ищем самую первую позицию i такую, что $\pi_i < \pi_{i+1}$ (если такой позиции нет, значит текущая подстановка $\pi = \pi''$ и процесс генерации завершается);

2) просматривая π от π_i слева направо, ищем наименьший из элементов π_j такой, что $\pi_1 < \pi_j (i < j)$;

3) меняем местами элементы π_i и π_j; затем все элементы $\pi_{i+1}, \pi_{i+2}, \ldots, \pi_n$ записываем в обратном порядке (т.е. меняем местами симметрично расположенные элементы π_{i+1+t} и π_{n-t}).

Пример. Пусть текущая перестановка π имеет вид $\pi = (3,5,7,6,4,2,1)$. На первом шаге найдены $\pi_i = 5, i = 2$; на втором — $\pi_j = 6, j = 4$; на третьем шаге меняем местами π_i и π_j: (3, 6, 7, 5, 4, 2, 1) и меняем местами элементы, начиная с третьей позиции: (3, 6, 1, 2, 4, 5, 7) — получили подстановку, следующую за текущей в лексикографическом порядке.

Генерация перестановок с помощью вложенных циклов

Будем говорить, что перестановка $\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ a_1 & a_2 & \ldots & a_n \end{pmatrix}$ является циклом длины k степени d, если ее элементы $a_i, i = \overline{1,k}$, получены из 1, 2, … , k циклическим сдвигом вправо на d позиций, остальные $n - k$ элементов стационарны. Например, подстановка $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 2 & 3 & 5 & 6 \end{pmatrix}$ является циклом длины 4 степени 1.

Алгоритм порождения подстановок с помощью вложенных циклов основан на следующей теореме.

Теорема 1. ***Любую подстановку*** π ***на множестве*** $X = \{1,2,\ldots,n\}$ ***можно представить в виде композиции***

$$\pi = \rho_n \circ \rho_{n-1} \circ \ldots \circ \rho_1 \qquad (2)$$

где ρ_i ***— циклическая подстановка порядка*** i.

Пример. Представим в виде (2) подстановку $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix}$, т.е. запишем

$$\pi = \rho_4 \circ \rho_3 \circ \rho_2 \circ \rho_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ a_1 & a_2 & a_3 & a_4 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ b_1 & b_2 & b_3 & 4 \end{pmatrix} \circ$$

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ c_1 & c_2 & 3 & 4 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ d_1 & 2 & 3 & 4 \end{pmatrix}.$$

Очевидно, последний цикл является тождественной подстановкой. Определим ρ_4 : т.к. $\rho_3(4) = 4, \rho_2(4) = 4,$ то $\rho_4(4) = 1$ следовательно

$$\rho_4 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \text{ — цикл порядка 4.}$$

Т.к. $\rho_2(3) = 3$, то $3 = \pi(1) = \rho_3(2) = \rho_3(\rho_4(1)) = \rho_2(2)$ и

$$\rho_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}.$$

Разложение подстановки π имеет вид:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix} \qquad (3)$$

Диаграмма композиции (3) приведена на рис. 13.

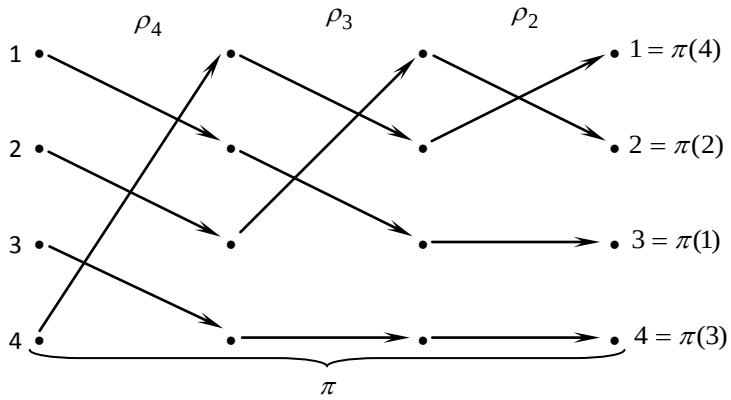


Рисунок 13 — Разложение в произведение вложенных циклов

Из теоремы 1 следует, что все перестановки можно получить систематическим перебором циклических сдвигов. В качестве начальной перестановки берем $\pi' = (1,2,\ldots,n)$ и сдвигаем на одну позицию вправо

все элементы до тех пор, пока вновь не получим π'; теперь сдвигаем циклически первые $n-1$ элементов и снова повторяем сдвиг всех n элементов на одну позицию до тех пор, пока не получим уже имеющуюся перестановку; сдвигаем циклически ее первые $n-2$ элементов… и т.д., пока не переберем все $n!$ перестановок. Ниже приведен алгоритм вложенных циклов.

1. Цикл ($i = \overline{1,n}$)
   1.1. $\pi_i := i$;
2. Конец цикла
3. $k := 0$
4. Пока ($k \neq 1$)
   4.1.Печать $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$
   4.2. $k := n$
   4.3. Сдвиг первых *k* элементов на одну позицию
   4.4. Пока ($\pi_k = k$ и $k > 0$)
      4.4.1. $k := k - 1$
      4.4.2. Сдвиг первых *k* элементов на одну позицию
   4.5. Конец цикла
5. Конец цикла
6. Конец

Этот алгоритм не является эффективным, т.к. на каждом шаге требует большого количества (не меньше n) транспозиций (транспозиция — обмен местами двух элементов).

Транспозиция соседних элементов

Описанные выше алгоритмы генерации перестановок не являются алгоритмами с минимальными изменениями. Минимальным изменением при переходе от текущей перестановки к следующей является транспозиция двух элементов. Дадим рекурсивное описание такого алгоритма.

Если $n = 1$, то существует единственная перестановка $\pi^{(1)} = (1)$. Пусть $n > 1$ и последовательность перестановок $\pi^{(1)}, \pi^{(2)}, \ldots, \pi^{(r)}, r = (n-1)!$ на множестве $(1,2,\ldots,n-1)$ построена. Для получения перестановок на множестве $(1,2,\ldots,n)$ будем вставлять элемент *n* на «промежутке» между элементами перестановки $\pi^{(i)}$ по следующему

правилу: если номер i подстановки $\pi^{(i)}$ — нечетное число, то элемент n вставляется в промежутки справа налево, если i — четное число, то элемент n вставляется в промежутки между элементами $\pi^{(i)}$ слева направо.

Пример генерации перестановки при $n = 4$ приведен на рисунке 14.

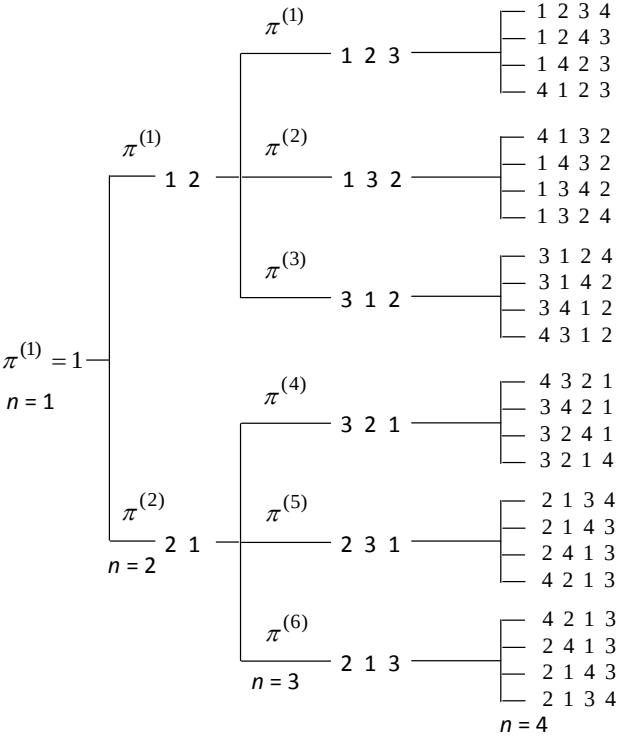


Рисунок 14 — Разложение в произведение вложенных циклов

Порядок выполнения работы

1. Получить индивидуальное задание.
2. Составить алгоритм решения задания и реализовать его графическое представление (блок-диаграмма, диаграмма Насси-Шнайдермана или псевдокод).
3. Составить программу на языке Си.
4. Выполнить отладку и тестирование программы.
5. Защитить работу.

2.8 Лабораторная работа «Машинное представление графов»

Цель работы: разработка и реализация алгоритмов преобразования различных форм представления графов.

Форма проведения: выполнение индивидуального задания.

Рекомендации по подготовке к лабораторной работе: для выполнения лабораторной работы необходимо изучить теоретический материал, изложенный ниже.

Машинные способы представления графов

Матрица смежности

Матрицей смежности неориентированного графа $G=(X,U)$, $|X|=n, |U|=m$ называется квадратная матрица $A[n\times n]$, элементы которой задаются по правилу:

$$a_{i,j}=\begin{cases}1, & \text{если вершина } x_i \text{ смежна вершине } x_j;\\ 0, & \text{если вершина } x_i \text{ не смежна вершине } x_j;\end{cases}$$

Матрицей смежности ориентированного графа $G=(X,U)$, $|X|=n, |U|=m$ называется квадратная матрица $A[n\times n]$, элементы которой задаются по правилу:

$$a_{i,j}=\begin{cases}1, & \text{если вершина } x_i \text{ смежна вершине } x_j;\\ -1, & \text{если вершина } x_j \text{ смежна вершине } x_i;\\ 0, & \text{если вершина } x_i \text{ не смежна вершине } x_j;\end{cases}$$

Матрица инцидентности

Матрицей инцидентности неориентированного графа $G=(X,U)$, $|X|=n, |U|=m$ называется матрица $B[n\times m]$, элементы которой задаются по правилу:

$$b_{i,j}=\begin{cases}1, & \text{если вершина } x_i \text{ инцидентна ребру } u_j;\\ 0, & \text{если вершина } x_i \text{ не инцидентна ребру } u_j.\end{cases}$$

Матрицей инцидентности ориентированного графа *G=(X,U)*, $|X|=n, |U|=m$ называется матрица $B[n \times m]$, элементы которой задаются по правилу:

$$a_{i,j} = \begin{cases} 1, & \text{если вершина } x_i \text{ начало дуги } u_j; \\ -1, & \text{если вершина } x_i \text{ конец дуги } u_j; \\ 2, & \text{если } u_j \text{ — петля при вершине } x_i; \\ 0, & \text{если вершина } x_i \text{ не инцидентна дуге } u_j. \end{cases}$$

Список ребер

Списком ребер неориентированного графа *G=(X,U)*, $|X|=n, |U|=m$ называются два массива $N[2m]$ и $K[2m]$, элементы которых задаются по правилу:

n_i – вершина, являющаяся началом ребра с номером *i.*

k_i – вершина, являющаяся концом ребра с номером *i.*

Списком ребер ориентированного графа *G=(X,U)*, $|X|=n, |U|=m$ называются два массива $N[m]$ и $K[m]$, элементы которых задаются по правилу:

n_i – вершина, являющаяся началом дуги с номером *i.*

k_i – вершина, являющаяся концом дуги с номером *i.*

Структура смежности

Структурой смежности графа *G=(X,U)*, $|X|=n, |U|=m$ называется массив линейных списков

$x_i : x_k, x_{k1}, .. x_{kz}$, $i = \overline{1,n}$,

где $x_k, x_{k1}, .. x_{kz}$ - все вершины, смежные вершине x_i.

Пример реализации структуры смежности представлен в приложении 1.

Порядок выполнения работы

1. Получить индивидуальный вариант.
2. Изучить теоретические аспекты лабораторной работы.
3. Изучить работу отладчика в среде DEV-CPP.

4. Разработать и реализовать на языке Си алгоритм решения предложенных задач.

5. Защитить работу, используя средства отладчика на тестовом примере с массивом из десяти элементов.

3 Методические указания для организации самостоятельной работы

3.1 Общие положения

Самостоятельная работа является важной составляющей в изучении дисциплины и состоит из следующих видов деятельности: проработка лекционного материала для подготовки к тестированию и контрольным работам, подготовка к лабораторным работам, выполнение домашних заданий.

Самостоятельная работа над теоретическим материалом направлена систематизацию и закрепление знаний, полученных на лекционных занятиях и на получение новых знаний по дисциплине, путем самостоятельного изучения тем.

Самостоятельная работа по подготовке к лабораторным работам направлена на изучение методического и теоретического материала по теме лабораторной работы.

Выполнение домашних заданий — полностью самостоятельная работа, направленная на получение навыков самостоятельного составления алгоритмов, реализацию программ, их дальнейшей отладки и тестирования.

3.2 Проработка лекционного материала, подготовка к контрольным работам и лабораторным работам

Проработка лекционного курса является одной из важных активных форм самостоятельной работы. Этот вид самостоятельной работы может быть организован следующим образом:

- прочитайте конспект лекции, согласуя Ваши записи с информацией на слайдах лекции;
- попробуйте выполнить самостоятельно примеры программ, разобранных на лекции;
- если в лекции рассматривался какой-либо алгоритм, попытайтесь выполнить этот алгоритм на тестовых данных без использования компь-

ютерной программы; такой способ проработки материалов лекции покажет, правильно ли Вы поняли идею алгоритма;

- изучите дополнительные учебные материалы, рекомендованные преподавателем;
- попытайтесь ответить на контрольные вопросы, которыми, как правило, заканчиваются разделы учебных пособий или учебников;
- если после выполненной работы Вы считаете, что материал освоен не полностью, сформулируйте вопросы и задайте их преподавателю.

Методические указания к ведению конспектов лекций. Лекции по дисциплине проводятся с использованием слайдов. Но это не означает, что лекцию можно просто слушать. Ведение конспектов значительно повышает качество последующей проработки лекционного материала. В силу специфики дисциплины на слайдах лекций очень много алгоритмов, кодов программ, примеров демонстрации работы изучаемых алгоритмов. Но этот материал может быть бесполезен, если Вы не делаете записи в течение лекции, потому что большинстве случаев, комментарии по представленным на слайдах примерам, лектор выполняет в устной форме.

Можно рекомендовать распечатывать слайды перед лекцией и вести конспект непосредственно на бумажном варианте слайд-презентации.

Одной из форм текущего мониторинга уровня знаний по дисциплине являются контрольные работы. Во время изучения дисциплины проводятся контрольные работы двух типов: тестовые опросы на лекции и контрольные работы, в которых студентам необходимо применить полученные знания на практике. Выполнение выше перечисленных действий поможет подготовиться и к выполнению контрольных работ. В приложении 2 указаны темы контрольных работ и приведены примерные варианты.

Самостоятельная работа по подготовке к лабораторным работам по дисциплине состоит в изучении методических материалов по темам соответствующих видов аудиторных занятий.

Рекомендуется перед выполнением лабораторной работы изучить лекционный и методический материал по теме занятия, ознакомиться с алгоритмами, реализацию которых необходимо выполнить во время проведения занятия. Обратите особое внимание на порядок выполнения работы. Поскольку конечным результатом всех лабораторных работ является компьютерная программа, самостоятельно разработайте структурную схему будущей программы, выполните заготовку проекта, подготовьте самостоятельно тестовые данные. Если при подготовке к занятию

остались нерешенные вопросы, обратитесь за консультацией к преподавателю.

3.3 Выполнение домашних заданий

3.3.1 Общие положения

Выполнение домашних заданий — это полностью самостоятельный вид деятельности студента. Темами домашних заданий являются темы дисциплины, не охваченные циклом лабораторных работ.

Выполнение домашнего задания состоит в написании программы по индивидуальному варианту. Обучающиеся самостоятельно разрабатывают алгоритм решения задания, реализуют разработанный алгоритм на языке программирования Си, отлаживают и тестируют написанную программу.

Защита домашнего задания проходит в сроки, установленные преподавателем. Процедура защиты представляет собой индивидуальное собеседование с преподавателем, примерный сценарий которого представлен ниже:

- комментирование студентом кода и логики написанной программы;
- ответы на вопросы преподавателя по коду и логике программы;
- комментирование студентом тестовых данных;
- демонстрация работы программы и пояснение полученных результатов.

3.3.2 Домашнее задание «Двунаправленные списки»

Домашнее задание состоит в реализации обработки данных, хранящихся в двунаправленном линейном списке. При выполнении домашнего задания рекомендуется изучить конспекты лекций, тема «Динамические структуры данных» и воспользоваться информацией из следующих источников:

Подбельский, В.В. Курс программирования на языке Си [Электронный ресурс] : учебник / В.В. Подбельский, С.С. Фомин. — Электрон. дан. — Москва : ДМК Пресс, 2012. — 384 с. — Режим доступа: https://e.lanbook.com/book/4148. — Загл. с экрана. Стр. 268 — 275.

Москвитина, О.А. Сборник примеров и задач по программированию [Электронный ресурс] : учебное пособие / О.А. Москвитина, В.С. Новичков, А.Н. Пылькин. — Электрон. дан. — Москва : Горячая линия-Телеком, 2014. — 245 с. — Режим доступа: https://e.lanbook.com/book/64090. — Загл. с экрана. Стр. 233 — 241.

Потопахин, В. Искусство алгоритмизации [Электронный ресурс] / В. Потопахин. — Электрон. дан. — Москва : ДМК Пресс, 2011. — 320 с. — Режим доступа: https://e.lanbook.com/book/1269. — Загл. с экрана. Стр. 224 — 228.

3.3.3 Домашнее задание «Поразрядные сортировки»

При выполнении этого домашнего задания студент должен самостоятельно ознакомится с принципами поразрядной сортировки, изучить принципы работы MSD-сортировки и LSD-сортировки. При реализации этих сортировок учесть, что наилучшим способом сортировки информации внутри одного разряда считается сортировка подсчетом. Описание алгоритмов можно найти в конспектах лекций и в следующих источниках:

Мещеряков, Р.В. Методы программирования [Электронный ресурс] : учебно-методическое пособие / Р.В. Мещеряков. — Электрон. дан. — Москва : ТУСУР, 2007. — 237 с. — Режим доступа: https://e.lanbook.com/book/11631. — Загл. с экрана. Стр. 147 — 155.

Потопахин, В. Искусство алгоритмизации [Электронный ресурс] / В. Потопахин. — Электрон. дан. — Москва : ДМК Пресс, 2011. — 320 с. — Режим доступа: https://e.lanbook.com/book/1269. — Загл. с экрана. Стр. 173 — 174.

3.3.4 Домашнее задание «Представление множеств упорядоченными списками»

Домашнее задание состоит в программировании операций над множествами — объединения, пересечения и проверки включения.

Для реализации этих алгоритмов изучите материал, представленный ниже.

Представление множеств упорядоченными списками

Если рассматриваемое множество не велико, то с точки зрения экономии памяти, множества достаточно часто представляются в виде списков элементов. Элемент списка в этом случае представляется записью с двумя полями: информационным и указателем на следующий элемент. Весь список описывается указателем на первый элемент.

Эффективная реализация операций над множествами, представленными в виде упорядоченных списков, основана на общем алгоритме типа слияния. Общая идея алгоритма типа слияния состоит в следующем: алгоритм параллельно просматривает два множества, представленных упорядоченными списками, причем на каждом шаге продвижение происходит в том множестве, в котором текущий элемент меньше.

Поверка включения слиянием

Даны проверяемые множества A и B, которые заданы указателями a и b. Если $A \subset B$, то функция возвращает 1, в противном случае 0.

1. *Pa = a; Pb = b;*
2. Пока (*Pa != NULL and Pb != NULL*)
   2.1. Если (*Pa.i<Pb.i*) // сравнение текущих информационных
   // полей.
   То вернуть 0 // элемент множества *A* отсутствует
   // в множестве *B*
   Иначе Если (*Pa.i >Pb.i*)
   То *Pb = Pb.n* // перейти к следующему
   // элементу в множестве *B*.
   Иначе *Pa:=Pa.n; Pb:=Pb.n* // Выполнить переход
   // в обоих множествах
3. Конец цикла
4. Если (*Pa = NULL)* То вернуть 1 // на выходе 1, если достигнут
   Иначе вернуть 0 // конец списка *A*.
5. Конец

Вычисление объединения слиянием

Даны объединяемые множества *A* и *B*, которые заданы указателями *a* и *b*.

1. *Pa = a; Pb = b; c = NULL;*
2. Пока (*Pa != NULL and Pb!=NULL*)
   2.1. Если (*Pa.i<Pb.i*) // сравнение информационных полей
   // текущих элементов
   То *d = Pa.i; Pa = Pa.n* // добавлению подлежит элемент
   // множества A
   Иначе Если (*Pa.i >Pb.i*)
   То *d = Pb.i; Pb = Pb.n* // добавлению подлежит
   // элемент множества B
   Иначе *d = Pb.i ; Pa = Pa.n; Pb = Pb.n*
   // в этом случае *Pa.i = Pb.i*, можно взять любой
   // элемент
   2.2. Добавить *d* в конец списка *c*.
3. Конец цикла
4. Если (*Pa != NULL*)
   То добавить в конец списка *c* оставшиеся элементы из *A*
5. Если (*Pb != NULL*)

То добавить в конец списка *c* оставшиеся элементы из *B*
6. Конец

Вычисление пересечения слиянием

Даны пересекаемые множества *A* и *B*, которые заданы указателями *a* и *b*.

1. *Pa = a; Pb = b; c = NULL;*
2. Пока (*Pa != NULL* and *Pb!=NULL*)
   2.1. Если (*Pa.i < Pb.i*) // сравнение текущих информационных
   // полей.
   То *Pa = Pa.n* // элемент множества A не принадлежит
   // пересечению
   Иначе Если (*Pa.i >Pb.i*)
   То *Pb = Pb.n* // элемент множества B не
   // принадлежит пересечению
   Иначе *d = Pb.i; Pa = Pa.n; Pb =Pb.n* // в этом случае
   // *Pa.i = Pb.i*, можно взять любой элемент
   2.2. Добавить *d* в конец списка *c*.
3. Конец цикла
4. Конец

3.3.5 Домашнее задание «Алгоритмы на графах»

Домашнее задание состоит в программной реализации алгоритмов поиска путей на графах. При выполнении этого домашнего задания необходимо ознакомиться с алгоритмами поиска путей на графе, построения остова графа и алгоритмами обходов графа. Ниже представлены некоторые алгоритмы на графах. Дополнительную информацию по теме домашнего задания можно получить в [2] (стр. 91 — 150) и [3], (стр. 200 — 275).

Алгоритмы обходов графа

Дан граф $G=(X,U)$, $|X|=n, |U|=m$.

1. Цикл ($i=\overline{1,n}$)
   1.1. *M[* x_i *]=0; //* Все вершины не отмечены
2. Конец цикла
3. Выбрать произвольную вершину $v=x_i \in X$
4. $v \to T$ // Записать выбранную вершину в структуру T

5. $M[v]=1$// Отмечаем вершину, как пройденную
6. Пока (*T* не пуста)
   6.1. $u \leftarrow T$ // извлекаем вершину из структуры
   6.2. Печать *u*
   6.3. Цикл(для всех вершин *w*, смежных с *u*)
      6.3.1. Если (*M[w]=0)* То $w \rightarrow T$; *M[w]=1*
   6.4. Конец цикла
7. Конец цикла
8. Конец

Если структуру Т определить как стек, то алгоритм обхода будет выполняться «в глубину». Если же Т определена как очередь, будет выполняться обход «в ширину».

Алгоритмы поиска путей на графах

Алгоритм Дейкстры

Алгоритм Дейкстры находит кратчайший путь между двумя заданными вершинами в орграфе.

Дан граф G=(X,U), $|X|=n, |U|=m$. Граф задан матрицей весов C, s— начальная вершина обхода, z —конечная вершина обхода.
На выходе алгоритма создаются два массива:

- *T* —если вершина *v* лежит на кратчайшем пути, то *T[v]*- длина кратчайшего пути от *s* к *v*.
- *H* —*H[v]* —вершина, непосредственно предшествующая *v* на кратчайшем пути.

1. Цикл($x=\overline{1,n}$)
   1.1. *T[x]*= ∞ // кратчайший путь не известен
   1.2. *M[x] = 0* // все вершины не отмечены
2. Конец цикла
3. *H[s]=0* // *s* ничего не предшествует
4. *T[s]=0* // кратчайший путь имеет длину 0
5. *M[s] = 1* // вершина *s* пройдена
6. *v = s* // зафиксируем текущую вершину
7. Цикл (1)
   7.1. Цикл(для $\forall u$ смежных с *v*)
      7.1.1. Если (*M[u]==0* и *T[u]>T[v]+C[v,u]*)
         То *T[u]=T[v]+C[v,u]; //* найден более
         // короткий путь из *s* в *u* через *v*

H[u]=v;

7.2. Конец цикла

7.3. $t=\infty$

7.4. *v=0*

7.5. Цикл($x=\overline{1,n}$)

7.5.1. Если (*M[x]==0* и *T[x]<t*)

То *v=x; t=T[x] //* вершина *v* заканчивает // кратчайший путь из *s*.

7.6. Конец цикла

7.7. Если (*v==0*) То «Нет пути из *s* в *z*»; Закончить выполнение цикла.

7.8. Если (*v==z*) То «Путь найден»;

Печать *T[v];*

Печать *H[v]*.

7.9. *M[v]=1, перейти на шаг 9.*

8. Конец цикла

9. Конец

Алгоритм Форда

Алгоритм Форда позволяет найти расстояние от заданной вершины *s* до всех вершин *T[v]=d(s,v)* ориентированного графа. Исходными данными для этого алгоритма является матрица весов дуг *C*. В отличие от алгоритма Дейкстры алгоритм может быть использован на графах с отрицательными длинами дуг.

Дан граф $G=(X,U)$, $|X|=n, |U|=m$. Граф задан матрицей весов *C*. *s* — вершина начала пути.

1. Цикл (для всех вершин *x* графа *G*)

1.1. *D[x]:=C[s,x];*

2. Конец цикла

3. *D[s]:=0;*

4. Цикл ($k=\overline{1,m-2}$)

// последовательно перебрать все ребра

4.1. Цикл (для всех вершин *v* графа, $v \neq s$)

// применить процесс поиска кратчайшего

//пути для всех ребер

4.1.1. Цикл (для всех вершин *u* графа, $u \neq v$)

4.1.1.1 Если (*D[v]>D[u]+C[u,v]*)

То *D[v]=D[u]+C[u,v]*

4.1.2. Конец цикла
4.2. Конец цикла
// закончить алгоритм досрочно
5. Если нет изменений в *D*, То *k=n-2*
6. Конец цикла
7. Печать *D.*
8. Конец

Алгоритм ближайшего соседа

Существует другой метод получения кратчайшего остовного дерева, который не требует ни сортировки ребер, ни проверки на цикличность на каждом шаге, - так называемый *алгоритм ближайшего соседа*. Просмотр начинается с некоторой произвольной вершины *a* в заданном графе. Пусть *(a,b)*- ребро с наименьшим весом, инцидентное *a*; ребро *(a,b)* включается в остов. Затем среди всех ребер, инцидентных либо *a*, либо *b*, выбираем ребро с наименьшим весом и включаем его в частично построенное дерево. В результате этого в дерево добавляется новая вершина, например, *c*. Повторяя процесс, ищем наименьшее ребро, соединяющее *a, b* или *c* с некоторой другой вершиной графа. Процесс продолжается до тех пор, пока все вершины из *G* не будут включены в дерево, то есть пока дерево не станет остовным.

Дан граф $G=(X,U)$, $|X|=n, |U|=m$. Граф задан матрицей весов *C*.

1. T —пустое множество ребер остовного дерева
2. a —произвольная вершина графа
3. $a \to T$
4. Пока ($T \neq X$)
   4.1. Найти ребро (u,v) с минимальным весом, такое, что $u \in T, v \notin T$.
   4.2. $v \to T$
5. Конец цикла
6. Печать T
7. Конец

Алгоритм Краскала

Алгоритм Краскала относится к семейству «жадных» алгоритмов. Алгоритм ищет кратчайший остов в связном графе $G=(X,U)$, $|X|=n, |U|=m$. Граф задан списком ребер U с длинами. На выходе алгоритма создается список T ребер кратчайшего остова.

1. T —пустой список
2. Упорядочить список U в порядке возрастания длин
3. $k = 1$ // номер рассматриваемого ребра
4. Цикл($i = \overline{1, m-1}$)
   4.1. Пока (добавление ребра $E[k]$ образует цикл в T)
      4.1.1. $k=k+1$
   4.2. Конец цикла
   4.3. $E[k] \to T$
5. Конец цикла
6. Печать T
7. Конец

Волновой алгоритм

Волновой алгоритм является алгоритмом поиска кратчайшего пути между двумя вершинами в неориентированном ненагруженном графе.

Пусть дан граф $G=(X,U)$, $|X| = n, |U| = m$. Вершина a —начало пути, вершина b —конец пути. Запишем вершину a во фронт волны нулевого уровня FW_0. Введем переменную $i = 0$. Далее найдем все вершины, смежные вершинам $v \in FW_i$ и ранее не пройденные. Запишем эти вершины во фронт волны следующего уровня FW_{i+1}. Увеличим i - $i = i + 1$. Если среди найденных вершин есть вершина b, то длина кратчайшего пути найдена и равна i, если нет, то процесс продолжается до тех пор пока не будет найдена конечная вершина пути, либо, пока не будут просмотрены все вершины графа. В этом случае пути из a в b нет.

1. Цикл($i = \overline{1, n}$)
   1.1. $M[x_i] = -1$ // отметим все вершины, как не пройденные
2. Конец цикла
3. $M[a]=0$; // вершина пройдена
4. $i=1$
5. $a \cup FW_{i-1}$
6. Пока ($b \notin FW_{i-1}$ И $\exists x \in X, M[x] = -1$)
   6.1. Цикл (для вершин $v \in X, M[v] = -1$ и смежных вершинам $\notin FW_{i-1}$)
      6.1.1. $v \cup FW_i$
   6.2. Конец цикла
   6.3. $i = i + 1$

7. Конец цикла
8. Если $b \in FW_{i-1}$ То «Путь найден», длина пути равна *i-1*.
   Иначе «Путь не существует»
9. Конец

Алгоритм Уоршалла

Алгоритм вычисления транзитивного замыкания для орграфа $G=(X,U)$, $|X|=n, |U|=m$. Граф задан матрицей смежности.

1. $S = R$ // вспомогательная матрица
2. Цикл($i = \overline{1,n}$)
   2.1. Цикл($j = \overline{1,n}$)
      2.1.1. Цикл($k = \overline{1,n}$)
         $T[i, j] = S[j,k] \vee S[j,i] \& S[i,k]$ // вычисление
                                                  //матрицы
                                                  // транзитивного
                                                  // замыкания
   2.2. Конец цикла
3. Конец цикла
4. Печать *T*.
5. Конец

Алгоритм построения эйлеровой цепи

Если граф имеет цикл содержащий все ребра графа по одному разу, то такой цикл называется эйлеровым циклом, а граф называется эйлеровым графом. Если граф имеет цепь (не обязательно простую), содержащую все вершины по одному разу, то такая цепь называется эйлеровой цепью, а граф называется полуэйлеровым графом.

Для того, чтобы в графе существовал эйлеров цикл граф должен быть связанным, для неориентированных графов число ребер в каждой вершине должно быть четным.

Дан эйлеров граф $G=(X,U)$, $|X|=n, |U|=m$, заданный структурой смежности. *Г[v]*- множество вершин, смежных с вершиной v.

1. *S* —пустой стек //стек для хранения вершин
2. Выбрать произвольную вершину $x \in X$
3. $x \to S$ // положить *x* в стек
4. Пока (Стек не пуст)

4.1. $v \leftarrow S$
4.2. $v \rightarrow S$
4.3. Если *Г[v]*- пустое множество То $v \leftarrow S$; Печать *v*
Иначе взять $u \in$ *Г[v];* // взять первую
// вершину, смежную *v*
$u \rightarrow S$
//удалить ребро *(v,u)*
Г[v]:=Г[v]\u;
Г[u]:=Г[u]\v;
5. Конец цикла
6. Конец

3.4 Подготовка к экзамену

Изучение дисциплины заканчивается экзаменом. Выполнение всех видов самостоятельных работ, лабораторных работ, посещение практических и лекционных занятий — гарантия успешной сдачи экзамена.

Для подготовки к экзамену рекомендуется повторить темы, вынесенные на экзамен. При подготовке обращайтесь не только к конспектам лекций, но и к рекомендованным преподавателем источникам. Организуйте план повторения материала таким образом, чтобы каждый день прорабатывать примерно одинаковый по объему материал. Изучая учебники и учебные пособия, отвечайте на контрольные вопросы. Прорешайте задачи примерного билета. Если после изучения материала Вы не смогли найти ответы на какие-либо вопросы — посетите консультацию перед экзаменом, кроме ответов на вопросы по теме экзамена на консультации освещаются организационные вопросы проведения экзамена время начала экзамена, время проведения экзамена, план проведения экзамена и т.д.

Пример экзаменационного билета
Билет 1

1. Опишите алгоритм пирамидальной сортировки. Продемонстрируйте работу алгоритм на массиве 9 3 4 6 1 8 2 7 5 0.
2. Запишите алгоритм сортировки обменом.
3. Напишите программу, реализующую алгоритм обхода графа в глубину. Граф задается матрицей смежности.

4 Рекомендуемые источники

1. Пермякова, Н. В. Информатика и программирование: Учебное пособие [Электронный ресурс] / Н. В. Пермякова — Томск: ТУСУР, 2016. — 188 с. — Режим доступа: https://edu.tusur.ru/publications/7678 .

2. Кузнецов, О.П. Дискретная математика для инженера [Электронный ресурс] : учебное пособие / О.П. Кузнецов. — Электрон. дан. — Санкт-Петербург : Лань, 2009. — 400 с. — Режим доступа: https://e.lanbook.com/book/220. — Загл. с экрана.

3. Асанов, М.О. Дискретная математика: графы, матроиды, алгоритмы [Электронный ресурс] : учебное пособие / М.О. Асанов, В.А. Баранский, В.В. Расин. — Электрон. дан. — Санкт-Петербург : Лань, 2010. — 368 с. — Режим доступа: https://e.lanbook.com/book/536. — Загл. с экрана.

ПРИЛОЖЕНИЕ 1

Пример программной реализации структуры смежности

```
#include <iostream.h>
#include <conio.h>

int main()
{
struct List {
  int Number;
  List *Next;
  };

List *Smegn;          // массив вершин
int n;                // количество вершин графа

cout << "Введите количество вершин графа: ";
cin >> n;
                      // Выделение памяти под массив вершин
Smegn = new List [n];
for (int i=0;i<n;i++)
  {
   Smegn[i].Number = i+1;
   Smegn[i].Next = NULL;
  }
                      // Ввод структуры смежности
cout << "Признак окончания ввода - 0" << endl;
for(i = 0;i<n;i++)
 {
 cout << "Вводите вершины смежные вершине " << i+1 << " :
";
 int d = 1;
 List* Cur = &Smegn[i];
 while (d!=0)
   {
       cout << "# вершины: ";
       cin >> d;
       if (d==0) continue;
       Cur->Next = new List;
```

```
                    // выделение памяти под новый элемент
      Cur = Cur->Next;
      Cur->Next = NULL;
      Cur->Number = d;    } }
                    // Печать структуры смежности
for (i=0;i<n;i++)
  {
   cout << Smegn[i].Number << ": "; // номер вершины
   List *Cur = &Smegn[i];
   if (Cur->Next == NULL) {cout << endl; continue;};
                    // Если смежных нет, то
                    //перейти на следующую вершину
   do               // пока не пройдены все смежные вершины,
                    //выводить на экран номера вершин
      {
       if (Cur->Next->Next == NULL) continue;
       Cur = Cur -> Next;
       cout << Cur->Number << ",";
      }
      while (Cur->Next->Next != NULL);
       Cur = Cur->Next;
       cout << Cur->Number << "." << endl;
                    // вывод последней вершины
  }
cout << endl;
                    // удаление структуры смежности из памяти.
for (i=0;i<n;i++)
{ List *Top;
 Top = &Smegn[i];
 List* Cur = Top->Next;
 delete Top;
 Top = Cur;
 while (Cur!=NULL)
  {
 Cur = Top->Next;
 delete Top;
 Top = Cur;  }
  delete [] Smegn;  }
}
```

ПРИЛОЖЕНИЕ 2

Темы контрольных работ

Темы и примерные варианты контрольных работ

1. Характеристики сортировок. Оценка сложности алгоритма

Вариант 1 Фамилия_____	

Является ли сортировка выбором устойчивой? Поясните, почему. (Приведите пример)	Найдите оценку временной сложности фрагмента программы:
	`int i = 2;`
	`int n = …`
	`while(i<=n){`
	`printf(“%d ”,i);`
	`i+=3;`
	`}`

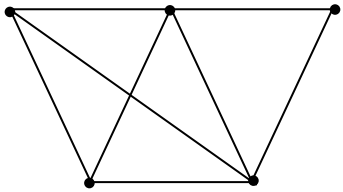
2. Улучшенные сортировки

   Вариант 1
   Фамилия_____
   Постройте начальную пирамиду на массиве 1 6 2 0 4 5 7 9 3.

3. Машинные способы задания графов

Вариант 1
Фамилия_____

1. Разметьте граф и постройте его матрицу смежности. Каким условиям должна удовлетворять матрица смежности неорграфа?



4. Алгоритмы на графах

Вариант 1
Фамилия_____

1. Продемонстрируйте алгоритм обхода «в глубину» по заданной матрице смежности графа. Обход начните с вершины с номером 0

0 1 0 1 1
1 0 1 1 1
0 1 0 0 1
1 1 0 0 1
1 1 1 1 0