МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

А.А. МЕРСОВ, В.В. ФИЛАТОВ, А.М. РУСАКОВ

Методические указания по выполнению курсовой работы по языкам программирования (2 семестр)

Москва – 2019

УДК
ББК
Печатается по решению редакционно-издательского совета «МИРЭА – Российский технологический университет»

Рецензенты:

Мерсов А.А., Филатов В. В., Русаков А.М.

Методические указания по выполнению курсовой работы по языкам программирования / А.А. Мерсов, В. В. Филатов, А.М. Русаков – М.: МИРЭА – Российский технологический университет, 2019.

В методических указаниях изложен порядок выполнения курсовой работы по языкам программирования.

Рекомендуется студентам вузов, изучающим дисциплины «Языки программирования», «Дискретная математика», «Информатика» и др.

УДК
ББК

Содержание

Общие указания к выполнению курсовой работы

Практические работы выполняются с использованием персональных компьютеров. Указания по технике безопасности совпадают с требованиями, предъявляемыми к пользователю ЭВМ. Другие опасные факторы отсутствуют.

Цель курсовой работы

Цель курсовой работы по дисциплине «Языки программирования» состоит в закреплении и углублении знаний и навыков, полученных при изучении дисциплины. Курсовая работа предполагает выполнение задания повышенной сложности по проектированию, разработке и тестированию программного обеспечения, а также оформлению сопутствующей документации.

Последовательность выполнения

Общий план работы

Для выполнения курсовой работы требуется умение писать программы на алгоритмическом языке для выполнения предложенного варианта.

Таким образом, последовательность выполнения курсовой работы следующая:

1. Ознакомится необходимым алгоритмом согласно варианту задания.
2. В соответствии с заданным вариантом написать программу.
3. Описать соответствующий алгоритм.
4. Написать и отладить на компьютере программу
5. Привести контрольную распечатку.
6. Оформить отчет.

Работа считается выполненной только после оформления отчета, защиты и подписи преподавателя.

Последовательность написания программного обеспечения.

Шаг 1. Подготовка исходных данных для решения задачи.

Подготовить используя средство отрисовки графов **yEd Graph Editor** (загрузить можно по ссылке https://www.yworks.com/products/yed/ - лицензия GNU) 3 графа:

- **Граф 1:** 7 вершин и 7-14 дуг (необходимое количество дуг в зависимости от варианта)
- **Граф 2:** 15 вершин и 15-20 дуг (необходимое количество дуг в зависимости от варианта)
- **Граф 3:** 20 вершин и 20-40 дуг (необходимое количество дуг в зависимости от варианта)

Необходимо сохранить нарисованные графы в двух форматах:

GraphML — язык описания (иногда упоминается как отдельный формат файлов) графов на основе XML.

Trivial Graph Format («простой формат графов», сокр. TGF) — простой формат файлов, основанный на тексте, для описания графов.

При этом потребуется изобразить неориентированный граф в виде ориентированного графа, но с двунаправленными ребрами, как показано, например, на рисунке ниже.

Для графа 1 необходимо подтвердить полученный результат вручную поэтапно показав, как был получен именно этот результат.

Для графов 2 и 3 нужно привести скриншот работы программы. Необходимо, используя граф 2 и 3, показать наиболее сложные с точки зрения алгоритма случаи работы программы.

Изображения всех тестовых графов необходимо включить в отчет.

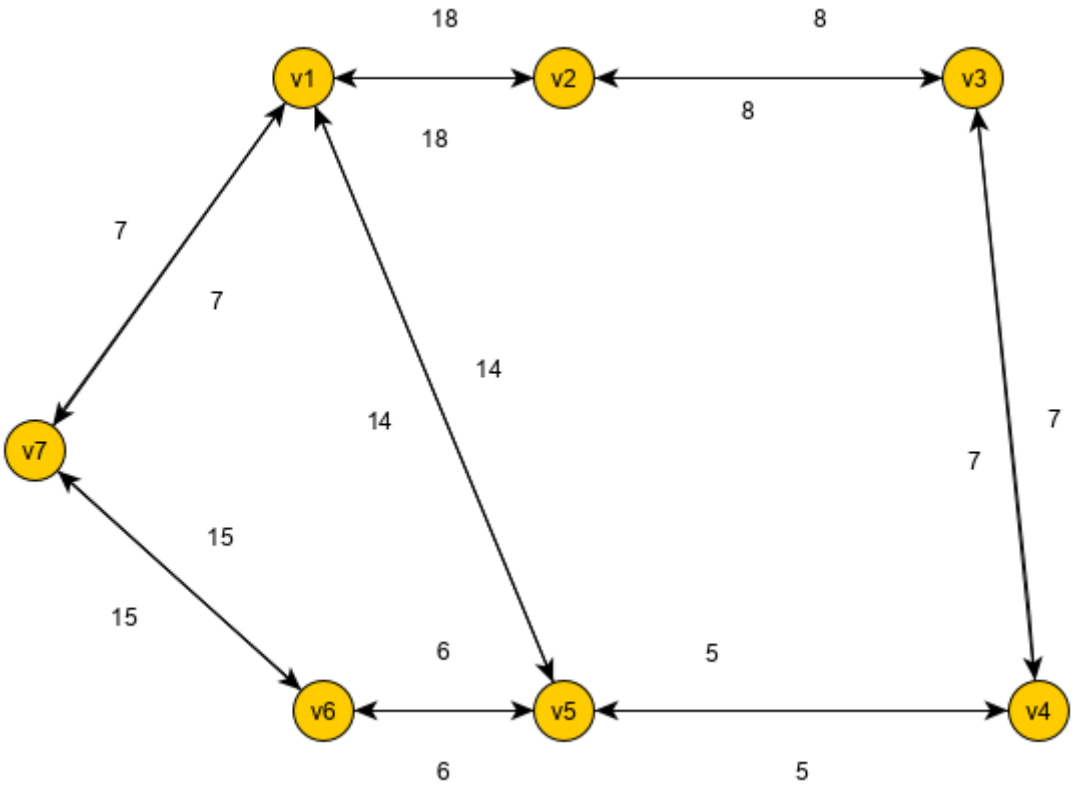


Рисунок. Пример графа.

Шаг 2. Реализация алгоритма чтения формата хранения графов TGF.

Студенту необходимо реализовать ввод данных графов через файлы в формате TGF.

Ниже приводится содержимое файла в формате TGF для изображенного выше графа:

```
1 v1
2 v2
3 v3
4 v5
5 v4
6 v6
7 v7
#
1 2 18
2 1 18
2 3 8
3 2 8
3 5 7
```

5 3 7
5 4 5
4 5 5
4 6 6
6 4 6
1 7 7
7 1 7
7 6 15
6 7 15
1 4 14
4 1 14

Таким образом, необходимо реализовать загрузку данных графа используя формат TGF, для разрабатываемой программы.

Шаг 3. Реализовать необходимый алгоритм в соответствии с заданным вариантом задания (необходимые материалы приведены в разделе необходимые сведения из теории).

Шаг 4. Адаптировать разработанный алгоритм на шаге 3 для заданного в варианте способа представления графа.

Шаг 5. Протестировать разработанное программное обеспечение.

Шаг 6. Оформить отчет по курсовой работе.

Отчет по курсовой работе

Отчет по курсовой работе оформляется в соответствии с требованиями, предъявляемыми к оформлению курсовых работ в вузе (Инструкция по организации и проведению курсового проектирования СМКО МИРЭА 7.5.1/04.И.05-18)*, и должен содержать:

1. Титульный лист (установленная форма Приложение №1 в *).
2. Наименование и цель работы (берется из рекомендаций по выполнению КР).
3. Исходные данные варианта задания (изображения).
4. Алгоритм решения задачи (сведения и .

5. Листинг программы с исходными и выходными данными (исходный код должен быть приведен в удобном для чтения формате, с необходимыми комментариями).
6. Анализ результатов (контрольная распечатка — скриншоты и граф 1,2 и 3).
7. Выводы.

Замечание: листы отчета должны быть скреплены.

К отчету также должен быть приложен цифровой носитель информации (диск или флэшка) со всеми необходимыми материалами для защиты курсовой работы.

Критерии определения оценки за курсовую работу

Общие критерии

Шкала 1. Оценка сформированности отдельных элементов компетенций

Обозначения		Формулировка требований к степени сформированности компетенции		
Цифр.	Оценка	Знать	Уметь	Владеть
1	Неуд.	Отсутствие знаний	Отсутствие умений	Отсутствие навыков
2	Неуд.	Фрагментарные знания	Частично освоенное умение	Фрагментарное применение
3	Удовл.	Общие, но не структурированные знания	В целом успешное, но не систематически осуществляемое умение	В целом успешное, но не систематическое применение
4	Хор.	Сформированные, но содержащие отдельные пробелы знания	В целом успешное, но содержащие отдельные пробелы умение	В целом успешное, но содержащее отдельные пробелы применение навыков
5	Отл.	Сформированные систематические знания	Сформированное умение	Успешное и систематическое применение навыков

Шкала 2. Комплексная оценка сформированности знаний, умений и владений

Обозначения		Формулировка требований к степени сформированности компетенции
Цифр.	Оценка	
1	Неуд.	Не имеет необходимых представлений о проверяемом материале
2	Удовл. или неуд. (по усмотрению	Знать на уровне о**риентирования**, представлений. Субъект учения знает основные признаки или термины изучаемого элемента содержания, их отнесенность к определенной науке, отрасли или

	преподавателя)	объектам, узнает их в текстах, изображениях или схемах и знает, к каким источникам нужно обращаться для более детального его усвоения.
3	Удовл.	Знать и уметь на **репродуктивном** уровне. Субъект учения знает изученный элемент содержания репродуктивно: произвольно воспроизводит свои знания устно, письменно или в демонстрируемых действиях.
4	Хор.	Знать, уметь, владеть на **аналитическом** уровне. Зная на репродуктивном уровне, указывать на особенности и взаимосвязи изученных объектов, на их достоинства, ограничения, историю и перспективы развития и особенности для разных объектов усвоения.
5	Отл.	Знать, уметь, владеть на **системном** уровне. Субъект учения знает изученный элемент содержания системно, произвольно и доказательно воспроизводит свои знания устно, письменно или в демонстрируемых действиях, учитывая и указывая связи и зависимости между этим элементом и другими элементами содержания учебной дисциплины, его значимость в содержании учебной дисциплины.

Индикаторные критерии оценки

Оценка	Индикаторные критерии оценки
Оценка «удовлетворительно»	• Реализована загрузка данных графа из файла в формате TGF. • Программа запускается и верно выполняет задание согласно варианту • Студент имеет представление о работе программы.
Оценка «хорошо»	Тоже что и на оценку «удовлетворительно» с применением следующих элементов: • Использование динамической памяти (операция new). • Функции добавления или удаления вершин и дуг в графе. • Функция изменения вершин и дуг (веса или направления) в графе.

	• Умение объяснить принципы работы разработанной программы.
Оценка «отлично»	Тоже что и на оценку «хорошо», дополнительно Использование классов и объектов для решения поставленной задачи согласно варианту с применением следующих элементов: • Конструкторов и деструкторов. • Друзья классов. • Наследование (простое или множественное). • Перегрузка операций. • Виртуальные функции. • Функция FIRST(*v*), которая возвращает ***индекс*** первой вершины в графе. • Функция NEXT(v, i), которая возвращает индекс вершины, смежной с вершиной v, следующий за индексом i. • Функция VERTEX(v, i), которая возвращает вершину с индексом i из множества вершин, смежных с v.

Сводная таблица критериев.

	Оценка «удовлетворительно»			Оценка "хорошо" тоже что и на оценку «удовлетворительно» с применением следующих элементов:				Оценка "отлично" тоже что и на оценку «хорошо», дополнительно: использование классов и объектов для решения поставленной задачи согласно варианту с применением следующих элементов:								
	Реализована загрузка данных графа из файла в формате TGF.	Программа запускается и верно выполняет задание согласно варианту	Студент имеет представление о работе программы.	Использование динамической памяти (операция new).	Функции добавления или удаления вершин и дуг в графе.	Функция изменения вершин и дуг (веса или направления) в графе.	Умение объяснить принципы работы разработанной программы.	Конструкторов и деструкторов.	Друзья классов.	Наследование (простое или множественное).	Перетрузка операций.	Виртуальные функции.	Функция FIRST(*v*), которая возвращает *индекс* первой вершины в графе.	Функция NEXT(v, i), которая возвращает индекс вершины, смежной с вершиной v, следующий за индексом i.	Функция VERTEX(v, i), которая возвращает вершину с индексом i из множества вершин, смежных с v.	Умение объяснить работу любого элемента разработанной программы.
балл за выполнение	10	10	20	5	10	10	40	10	10	10	10	15	10	10	10	60

Оценка	Балл	Набранное кол-во балов	Пояснения
оценка 3	не менее	40	максимальная сумма баллов за "оценка удовлетворительно"
оценка 4	не менее	88,75	сумма баллов за "оценка удовлетворительно" и 75% от суммы баллов за оценку "хорошо"
оценка 5	не менее	213,75	сумма баллов за "оценка удовлетворительно" , баллов за оценку "хорошо" и 75% от суммы баллов за оценку "отлично"

Варианты заданий

Вариант задания определяется по последним двум цифрам в зачетной книжке.

№	Алгоритм	Способ представления графа
1.	Дана матрица весов дуг. Определить и вывести все циклы в орграфе, заданной длины x (вводится с клавиатуры)	Матрица смежности
2.	Дана матрица весов дуг. Определить и вывести все циклы в орграфе, заданной длины x (вводится с клавиатуры)	Матрица инцидентности
3.	Дана матрица весов дуг. Определить и вывести все циклы в орграфе, заданной длины x (вводится с клавиатуры)	Список смежности
4.	Дана матрица весов дуг. Определить и вывести все циклы в орграфе, заданной длины x (вводится с клавиатуры)	Список дуг
5.	Дана матрица весов дуг. Определить ВСЕ (т.е. не обязательно самые короткие) незамкнутые пути в орграфе заданной длины x (вводится с клавиатуры).	Матрица смежности
6.	Дана матрица весов дуг. Определить ВСЕ (т.е. не обязательно самые короткие) незамкнутые пути в орграфе заданной длины x (вводится с клавиатуры).	Матрица инцидентности
7.	Дана матрица весов дуг. Определить ВСЕ (т.е. не обязательно самые короткие) незамкнутые пути в орграфе заданной длины x (вводится с клавиатуры).	Список смежности
8.	Дана матрица весов дуг. Определить ВСЕ (т.е. не обязательно самые короткие) незамкнутые пути в орграфе заданной длины x (вводится с клавиатуры).	Список дуг
9.	Транзитивная редукция ориентированного графа G = (V, E) определяется как произвольный граф G' — (V, E'), имеющий то же множество вершин, но с минимально возможным числом дуг (E’ $\not\subset$ E), транзитивное замыкание которого совпадает с транзитивным замыканием графа G, (причем если граф G ацикличен, то его транзитивная редукция единственна). Реализуйте программу транзитивной редукции графа.	Матрица смежности
10.	Транзитивная редукция ориентированного графа G = (V, E) определяется как произвольный граф G' — (V, E'), имеющий то же множество вершин, но с минимально возможным числом дуг (E’ $\not\subset$ E), транзитивное замыкание которого совпадает с транзитивным замыканием графа G, (причем если граф G ацикличен, то его транзитивная редукция единственна). Реализуйте программу транзитивной редукции графа.	Матрица инцидентности
11.	Орграф G' = (V, E') называется минимальным эквивалентным орграфом для орграфа G = (V, E), если E' — наименьшее подмножество множества E (E’ $\subseteq$ E) такое что транзитивные замыкания обоих орграфов G и G' совпадают (причем если граф G ацикличен, то для него существует только один минимальный эквивалентный орграф). Написать программу нахождения минимального эквивалентного орграфа.	Список смежности
12.	Орграф G' = (V, E') называется минимальным эквивалентным орграфом для орграфа G = (V, E), если E' — наименьшее подмножество множества E (E’ $\subseteq$ E) такое что транзитивные замыкания обоих орграфов G и G' совпадают (причем если граф G ацикличен, то для него существует только один минимальный эквивалентный	Список дуг

	орграф). Написать программу нахождения минимального эквивалентного орграфа.	
13.	Мостом графа G называется каждое ребро, удаление которого приводит к увеличению числа связных компонент графа. Представить алгоритм нахождения всех мостов графа	Матрица смежности
14.	Мостом графа G называется каждое ребро, удаление которого приводит к увеличению числа связных компонент графа. Представить алгоритм нахождения всех мостов графа	Матрица инцидентности
15.	Определить наличие всех циклов методом обхода в глубину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Список смежности
16.	Определить наличие всех циклов методом обхода в глубину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Список дуг
17.	Определить число сильно связных компонент в орграфе	Матрица смежности
18.	Определить число сильно связных компонент в орграфе	Матрица инцидентности
19.	Определить число сильно связных компонент в орграфе	Список смежности
20.	Определить число сильно связных компонент в орграфе	Список дуг
21.	Определить внешний диаметр не взвешенного неориентированного графа методом обхода в ширину. (Внешним диаметром графа будем называть наибольшее значение среди всех попарных наименьших расстояний между узлами). Вывести все пары узлов, образующие указанное значение и соответствующие **диаметральные цепи**.	Матрица смежности
22.	Определить внешний диаметр не взвешенного неориентированного графа методом обхода в ширину. (Внешним диаметром графа будем называть наибольшее значение среди всех попарных наименьших расстояний между узлами). Вывести все пары узлов, образующие указанное значение и соответствующие **диаметральные цепи**.	Матрица инцидентности
23.	Определить внешний диаметр не взвешенного неориентированного графа методом обхода в ширину. (Внешним диаметром графа будем называть наибольшее значение среди всех попарных наименьших расстояний между узлами). Вывести все пары узлов, образующие указанное значение и соответствующие **диаметральные цепи**..	Список смежности
24.	Определить внешний диаметр не взвешенного неориентированного графа методом обхода в ширину. (Внешним диаметром графа будем называть наибольшее значение среди всех попарных наименьших расстояний между узлами). Вывести все пары узлов, образующие указанное значение и соответствующие **диаметральные цепи**..	Список дуг
25.	Определить внешний радиус не взвешенного неориентированного графа методом обхода в ширину. (Внешним радиусом графа будем называть наибольшее	Матрица смежности

	среди кратчайших расстояние от центра до какого-либо узла.). Вывести значение, а также соответствующие ему цепи.	
26.	Определить внешний радиус не взвешенного неориентированного графа методом обхода в ширину. (Внешним радиусом графа будем называть наибольшее среди кратчайших расстояние от центра до какого-либо узла.) Вывести значение, а также соответствующие ему цепи.	Матрица инцидентности
27.	Определить внешний радиус не взвешенного неориентированного графа методом обхода в ширину. (Внешним радиусом графа будем называть наибольшее среди кратчайших расстояние от центра до какого-либо узла.) Вывести значение, а также соответствующие ему цепи.	Список смежности
28.	Определить внешний радиус не взвешенного неориентированного графа методом обхода в ширину. (Внешним радиусом графа будем называть наибольшее среди кратчайших расстояние от центра до какого-либо узла.) Вывести значение, а также соответствующие ему цепи.	Список дуг
29.	Определить внешний радиус взвешенного ориентированного графа методом обхода в ширину. (Внешним радиусом графа будем называть наибольшее среди кратчайших расстояние от центра до какого-либо узла.)	Матрица смежности
30.	Определить внешний радиус взвешенного ориентированного графа методом обхода в ширину. (Внешним радиусом графа будем называть наибольшее среди кратчайших расстояние от центра до какого-либо узла.) Вывести значение, а также соответствующие ему цепи.	Матрица инцидентности
31.	Определить внешний радиус взвешенного ориентированного графа методом обхода в ширину. (Внешним радиусом графа будем называть наибольшее среди кратчайших расстояние от центра до какого-либо узла.)	Список смежности
32.	Определить внешний радиус взвешенного ориентированного графа методом обхода в ширину. (Внешним радиусом графа будем называть наибольшее среди кратчайших расстояние от центра до какого-либо узла.)	Список дуг
33.	Определить наличие всех циклов методом обхода в глубину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Матрица смежности
34.	Определить наличие всех циклов методом обхода в глубину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Матрица инцидентности
35.	Определить наличие всех циклов методом обхода в глубину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Список смежности
36.	Определить наличие всех циклов методом обхода в	Список дуг

	глубину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	
37.	Определить наличие всех циклов методом обхода в ширину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Матрица смежности
38.	Определить наличие всех циклов методом обхода в ширину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Матрица инцидентности
39.	Определить наличие всех циклов методом обхода в ширину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Список смежности
40.	Определить наличие всех циклов методом обхода в ширину на орграфе. Вывести все циклы (варианты обхода, образующие циклы). Подсчитать их общее количество.	Список дуг
41.	Определить в орграфе сильно связные компоненты, подсчитать их число и вывести состав (номера узлов) каждой сильно связной компоненты.	Матрица смежности
42.	Определить в орграфе сильно связные компоненты, подсчитать их число и вывести состав (номера узлов) каждой сильно связной компоненты.	Матрица инцидентности
43.	В заданном неориентированном графе вывести все вершины – точки сочленения.	Матрица смежности
44.	В заданном неориентированном графе вывести все вершины – точки сочленения.	Матрица инцидентности
45.	Вывести на экран **все существующие** пути в ациклическом орграфе	Матрица смежности
46.	Вывести на экран **все существующие** пути в ациклическом орграфе	Матрица инцидентности
47.	Вывести на экран **все существующие** пути в ациклическом орграфе	Список смежности
48.	Вывести на экран **все существующие** пути в ациклическом орграфе	Список дуг
49.	Корнем ациклического орграфа называется вершина r такая, что существуют пути, исходящие из этой вершины и достигающие всех остальных вершин орграфа. Напишите программу, определяющую, имеет ли данный ациклический орграф корень и вывести его на экран.	Матрица смежности
50.	Корнем ациклического орграфа называется вершина r такая, что существуют пути, исходящие из этой вершины и достигающие всех остальных вершин орграфа. Напишите программу, определяющую, имеет ли данный ациклический орграф корень и вывести его на экран.	Матрица инцидентности
51.	Корнем ациклического орграфа называется вершина r такая, что существуют пути, исходящие из этой вершины и достигающие всех остальных вершин орграфа. Напишите программу, определяющую, имеет ли данный ациклический орграф корень и вывести его на экран.	Список смежности
52.	Корнем ациклического орграфа называется вершина r такая, что существуют пути, исходящие из этой вершины и достигающие всех остальных вершин орграфа. Напишите программу, определяющую, имеет ли данный ациклический орграф корень и вывести его на экран.	Список дуг

53.	Определить, есть ли какой-либо путь, проходящий через ВСЕ вершины орграфа, причем через вершину можно проходить только один раз, а начальная и конечная вершины не должны быть смежными, и вывести его на экран.	Матрица смежности
54.	Определить, есть ли какой-либо путь, проходящий через ВСЕ дуги орграфа, причем через дугу можно проходить только один раз, а начальная и конечная вершины не должны быть смежными, и вывести его на экран.	Матрица смежности
55.	Определить, есть ли какой-либо путь, проходящий через ВСЕ вершины орграфа, причем через вершину можно проходить только один раз, а начальная и конечная вершины должны совпадать, и вывести его на экран.	Матрица смежности
56.	Определить, есть ли какой-либо путь, проходящий через ВСЕ дуги орграфа, причем через дугу можно проходить только один раз, а начальная и конечная вершины должны совпадать, и вывести его на экран.	Матрица смежности
57.	Напишите программу, на входе которой вводятся две его вершины. Программа должна распечатывать все простые пути, ведущие от одной вершины к другой.	Матрица смежности
58.	Напишите программу, на входе которой вводятся две его вершины. Программа должна распечатывать все простые пути, ведущие от одной вершины к другой.	Матрица инцидентности
59.	Напишите программу, на входе которой вводятся две его вершины. Программа должна распечатывать все простые пути, ведущие от одной вершины к другой.	Список смежности
60.	Напишите программу, на входе которой вводятся две его вершины. Программа должна распечатывать все простые пути, ведущие от одной вершины к другой.	Список дуг
61.	Определить ВСЕ простые пути в орграфе.	Матрица смежности
62.	Определить ВСЕ простые пути в орграфе.	Матрица инцидентности
63.	Определить ВСЕ простые пути в орграфе.	Список смежности
64.	Определить ВСЕ простые пути в орграфе.	Список дуг
65.	Дана матрица весов дуг. Определить и вывести все циклы в орграфе, заданной длины x (вводится с клавиатуры)	Матрица смежности
66.	Дана матрица весов дуг. Определить и вывести все циклы в орграфе, заданной длины x (вводится с клавиатуры)	Матрица инцидентности
67.	Дана матрица весов дуг. Определить и вывести все циклы в орграфе, заданной длины x (вводится с клавиатуры)	Список смежности
68.	Дана матрица весов дуг. Определить и вывести все циклы в орграфе, заданной длины x (вводится с клавиатуры)	Список дуг
69.	Дана матрица весов дуг. Определить ВСЕ (т.е. не обязательно самые короткие) незамкнутые пути в орграфе заданной длины x (вводится с клавиатуры).	Матрица смежности
70.	Дана матрица весов дуг. Определить ВСЕ (т.е. не обязательно самые короткие) незамкнутые пути в орграфе заданной длины x (вводится с клавиатуры).	Матрица инцидентности
71.	Дана матрица весов дуг. Определить ВСЕ (т.е. не обязательно самые короткие) незамкнутые пути в орграфе заданной длины x (вводится с клавиатуры).	Список смежности
72.	Дана матрица весов дуг. Определить ВСЕ (т.е. не	Список дуг

	обязательно самые короткие) незамкнутые пути в орграфе заданной длины x (вводится с клавиатуры).	
73.	Транзитивная редукция ориентированного графа G = (V, E) определяется как произвольный граф G' — (V, E'), имеющий то же множество вершин, но с минимально возможным числом дуг (E' $\not\subset$ E), транзитивное замыкание которого совпадает с транзитивным замыканием графа G, (причем если граф G ацикличен, то его транзитивная редукция единственна). Реализуйте программу транзитивной редукции графа.	Матрица смежности
74.	Транзитивная редукция ориентированного графа G = (V, E) определяется как произвольный граф G' — (V, E'), имеющий то же множество вершин, но с минимально возможным числом дуг (E' $\not\subset$ E), транзитивное замыкание которого совпадает с транзитивным замыканием графа G, (причем если граф G ацикличен, то его транзитивная редукция единственна). Реализуйте программу транзитивной редукции графа.	Список смежности
75.	Орграф G' = (V, E') называется минимальным эквивалентным орграфом для орграфа G = (V, E), если E' — наименьшее подмножество множества E (E' $\subseteq$ E) такое что транзитивные замыкания обоих орграфов G и G' совпадают (причем если граф G ацикличен, то для него существует только один минимальный эквивалентный орграф). Написать программу нахождения минимального эквивалентного орграфа.	Список смежности
76.	Орграф G' = (V, E') называется минимальным эквивалентным орграфом для орграфа G = (V, E), если E' — наименьшее подмножество множества E (E' $\subseteq$ E) такое что транзитивные замыкания обоих орграфов G и G' совпадают (причем если граф G ацикличен, то для него существует только один минимальный эквивалентный орграф). Написать программу нахождения минимального эквивалентного орграфа.	Матрица смежности
77.	Мостом графа G называется каждое ребро, удаление которого приводит к увеличению числа связных компонент графа. Представить алгоритм нахождения всех мостов графа	Список смежности
78.	Мостом графа G называется каждое ребро, удаление которого приводит к увеличению числа связных компонент графа. Представить алгоритм нахождения всех мостов графа	Матрица смежности
79.	Определить k-связанность заданного неориентированного графа и вывести полученное число k на экран. (Граф называется k-связным, если между любой парой вершин v и w существует не менее k разных путей, таких, что, за исключением вершин v и w, ни одна из вершин, входящих в один путь, не входит ни в какой другой из этих путей).	Матрица смежности
80.	Определить k-связанность заданного неориентированного графа и вывести полученное число k на экран. (Граф называется k-связным, если между любой парой вершин v и w существует не менее k разных путей, таких, что, за	Список смежности

	исключением вершин v и w, ни одна из вершин, входящих в один путь, не входит ни в какой другой из этих путей).	
81.	Пусть дана сеть (узел a – исток, b–сток). Определить все разрезы сети.(на основе определения понятия разреза)	Матрица смежности
82.	Пусть дана сеть (узел a – исток, b–сток). Определить все разрезы сети. (на основе определения понятия разреза)	Список смежности
83.	Пусть дана сеть (узел a – исток, b–сток). Определить все разрезы сети. (на основе определения понятия разреза)	Матрица инцидентности
84.	Определить величину минимального разреза сети.	Матрица смежности
85.	Определить величину минимального разреза сети.	Список смежности
86.	Определить величину минимального разреза сети.	Матрица инцидентности
87.	Определить все непересекающиеся цепи между двумя произвольными узами графа.	Матрица смежности
88.	Определить все непересекающиеся цепи между двумя произвольными узами графа	Список смежности
89.	Определить все непересекающиеся цепи между двумя произвольными узами графа	Матрица инцидентности
90.	Методом обхода в ширину вычислить цикломатическую сложность графа	Матрица смежности
91.	Методом обхода в ширину вычислить цикломатическую сложность графа	Список смежности
92.	Методом обхода в ширину вычислить цикломатическую сложность графа	Матрица инцидентности
93.	Методом обхода в ширину вычислить цикломатическую сложность графа	Список дуг
94.	Методом обхода в глубину вычислить цикломатическую сложность графа	Матрица смежности
95.	Методом обхода в глубину вычислить цикломатическую сложность графа	Список смежности
96.	Методом обхода в глубину вычислить цикломатическую сложность графа	Матрица инцидентности
97.	Методом обхода в глубину вычислить цикломатическую сложность графа	Список дуг
98.	Определить минимальное число красок, которыми можно раскрасить граф и вывести пример такой раскраски.	Матрица смежности
99.	Определить минимальное число красок, которыми можно раскрасить граф и вывести пример такой раскраски.	Список смежности
100.	Определить минимальное число красок, которыми можно раскрасить граф и вывести пример такой раскраски.	Матрица инцидентности

Необходимые сведения из теории

Основные сведения из языков программирования

Массивы символов и строки

Массив символов

Символьная переменная (переменная типа `char`) - это величина размером в 1 байт, которая используется для представления литер и целых чисел в диапазоне от 0 до 255 или от -128 до 127, в зависимости от того, знаковая эта переменная или беззнаковая. Символьные константы заключаются в одинарные кавычки. Примеры символьных констант: 'a', '+', '1'.

Символьная константа `''` - это символ с нулевым значением, так называемый символ `NULL`. Вместо просто 0 часто используют запись `''`, чтобы подчеркнуть символьную природу используемого выражения.

Строковая константа, или строковый литерал - это нуль или более символов, заключенных в двойные кавычки, например,

```
"это строковая константа"
"" /*это пустая строка (нуль)*/.
```

Таким образом, в языке С строка (строковая константа) - это массив символов, заканчивающийся нулевым байтом. Поэтому памяти для строки требуется на один байт больше, чем число символов, расположенных между двойными кавычками.

Следующие инициализации массивов эквивалентны:

```
char* str="hello";
char str[6]={'h','e','l','l','o',''};
```

СИМВОЛЬНЫЕ СТРОКИ И ФУНКЦИИ НАД СТРОКАМИ

Всякий раз, когда компилятор встречается с чем-то заключённым в кавычки, он определяет это как строковую константу. Символыт'\0' записываются в последовательные ячейки памяти. Если необходимо включить кавычки в символьную строку, нужно поставить впереди \".

Строковые константы размещаются в области данных. Вся фраза в кавычках является указателем на место в памяти, где записана строка. Это аналогично имени массива, служащего указателем на адрес расположения массива. Для вывода символьной константы служит идентификатор %s.

char mas[] = "Это одна строка";

mas - адрес строки -> &m[0] *mas=='Э'.

Можно использовать указатель для создания строки.

char *str = "Таблица результатов";

char str[]= "Таблица результатов".

Сама строка размещается в области данных, а указатель инициализируется адресом.

void main(void){

char* mesg="Ошибка чтения";

```
  char* copy;
  copy = mesg; создается второй указатель на строку.
  printf ("%S", copy);
  printf ("%S", mesg);
}
 char* name;
 scanf ("%S", name);   //Так нельзя !!!  Указателю не присвоен адрес.
 char name [81];          //Нужно в начале определить массив
```

Массивы символьных строк

1. Строки в символьный массив можно вводить с клавиатуры.

```
char mas[80];
scanf("%S",mas);
```

2. Если требуется несколько строк, то организуем цикл ввода

```
char mas[4][81];
for (i=0; i<4; i++)
   scanf("%S", mas[i]);                    // &mas[i][0]
```

3. Можно сразу инициализировать в программе.

```
char m1[] = "Только одна строка";     // автоматически определяется
                                      // длина строки + 1 байт на '\0'.
```

4. Размер массива можно задать явно.

```
char m2[50] = "Только одна строка"; //18+1
```

5. `char m3[]={'c', 'm', 'p', 'o', 'k', 'a', '\o'};`

6. Инициализация массива строк:

```
char masstr[3][16]={"Первая строка",
                    "Вторая строка",
                    "Третья строка" };
```

```
*masstr[0]=='П';
*masstr[1]=='В';
*masstr[2]=='Т';
```

7. «Рваный массив» – это массив указателей на строки.

```
static char *masstr[3]= {"Первая строка",
                         "Вторая строка",
                         "Третья строка" };
```

В случае «рваного массива» длина строк разная и зря не расходуется память.

Массивы указателей

Можно определять массивы указателей

```
int* parray[5];                    //5 указателей на целые значения.
*parray[3] -              //3-й элемент массива.
char  *keywords[5]={"ADD",  "CHANGE",  "DELETE",  "LIST",  "QUIT"};
```

В памяти

```
        keyword[0] – адрес 10000  строка ADD\0   4б
        keyword[1] -          10004  CHANGE\0       7б
        keyword[2] -          10011  DELETE\0       7б
for (i=0; i<5; i++)
   printf("%S", keywords[i]);
char *key[3],**pt;                 //определение указателя на указатель
pt=key;
printf(«%s %d\n»,*pt,**pt);   //распечатывается первая строка и код
                                   //первой буквы
```

Функции, работающие со строками

Функции, определеные в заголовочном файле stdio.h.

1. Функция gets(char *) - вводит строку в массив с клавиатуры

```
char name[81];
gets(name);    // Берёт все символы до конца строки символ '\n'
                // отбрасывает и записывает '\0' и передаёт в  вызов программы.
```

2. Функция puts(char *) - выводит строку на экран

```
puts ("Я функция puts()");
char str1[]="Только одна строка";
puts(str1);  //Читает строку до встречи '\0'. Все строки
          //выводятся с новой строки.
```

3. Функция int getc(stdin) – вводит символ в переменную с клавиатуры.

4. Функция int putc(int c, stdout) – выводит символ на экран.

Стандартные библиотечные функции

Функции, определеные в заголовочном файле string.h.

1. int strlen(char *) - опредляет длинну строки без'\0';

```
int k = strlen(str1);
```

2. char * strcat(char *, char *) - объединяет две строки в одну.

```
strcat(str1, str2);        //результат в первом массиве.
```

```
void main (void){
   char str1[80]="Мой любимый цветок";
   char str2[10]="ромашка";
   if (strlen(str1)+strlen(str2)< 80-1)
```

```
        strcat(str1, str2);         //Чтобы копировать - необходимо проверить,
                                    //чтобы длины массива было достаточно на
                                    //две строки + ноль-байт.
}
```

3. char * strncat(char *, char *,int) - объединяет 1-ю строку и n-байтов второй строки в одну.

4. int strcmp(char *, char *) - сравнение строк.

```
# define ANSWER "YES"
void main(void){
        char try[10];
        gets(try);
        puts ("Вы студенты 203 группы?");
        while (strcmp(try, ANSWER)!=0){
           puts ("Попытайтесь ещё раз")
           gets (try);
        }
   puts("Верно");
}
```

Функция возвращает 0, если строки одинаковы. Сравнение идёт до признака конца строки - '\0', а не до конца массива, или до первого несравнения:

B-A возвращает 1, A-B возвращает -1.

5. int strncmp(char *, char *,int) - сравнение n байт у 2-х строк .

6. char * strcpy(char *, char *) - копирование строк

```
#define WORD  "Таблица результатов"
```

```
void main (void){
   char str1[30];    //Длина массива не проверяется
   strcpy(str1, WORD)
   puts(str1);
}
```

7. char * strтcpy(char *, char *,int) - копирование n байт строки
8. char *strdup(char *) – выделяет память и копирует строку.
9. char *strupr(char *) – преобразует строчные буквы в прописные.
10. char *strlwr(char *) – преобразует прописные буквы в строчные.
11. char *strrev(char *) – реверсирует строку.
12. char *strchr(char *, int) – устанавливает позицию первого вхождения символа
14. char *strrchr(char *, int c) – устанавливает позицию последнего вхождения символа c.
15. char *strstr(char *, char *) - устанавливает позицию первого вхождения подстроки str2 в строку str1.
16. int stricmp(char *, char *) – сравнивает не различая строчные и прописные буквы.

Преобразование символьных строк

Функции, определённые в заголовочном файле stdlib.h.

1. int atoi() - строку в целое.

```
double atof() - строку в число с плавающей точкой.
void main(void){
char num[10];
int val;
puts("Введите число");
gets(num);
val=atoi(num);   // обрабатывает до 1-го символа не являющегося
}                // цифрой
```

2. Существуют функции обратного преобразования числа в строку.

itoa(int val, char *str, int radix) - целое в строку, где

int val – число;

char *str – строка;

int radix – система счисления.

ltoa(long val, char *str, int radix) - с плавающей точкой в строку.

Функции, определённые в заголовочном файле ctype.h.

Выполняют преобразования только с буквами английского алфавита.

1. Преобразование строчной буквы в прописную - int toupper(int c)

2. Проверка буква прописная или нет - int isupper(int c)
3. Преобразование прописной буквы в строчную - int tolower(int c)
4. Проверка буква строчная или нет - int islower(int c)

```
#include <ctype.h>

void main(void){

  int ch; crit=0;          //Признак прописные или строчные буквы

  while ((ch=getche())!='\n'){

     if(crit==0){

          ch=isupper(ch) ? tolower(ch): ch;

          putchar(ch);

      }

     else{

          ch=islower(ch)? toupper(ch):ch;

          putchar (ch);

     }

  }
```

ССЫЛКИ

Ссылка - это псевдоним для другой переменной. Они объявляются при помощи символа &. Ссылки должны быть проинициализированы при объявлении, причем только один раз.

Тип "ссылка на type" определяется следующим образом:

type& имя_перем.

Ссылка при определении сразу же инициализируется. Инициализация ссылки производится следующим образом:

```
int i = 0;

int& iref = i;
```

Физически iref представляет собой постоянный указатель на int - переменную типа int* const. Ее значение не может быть изменено после ее инициализации. Ссылка отличается от указателя тем, что используется не как указатель, а как переменная, адресом которой она была инициализирована:

```
iref++;              //то же самое, что i++

int *ip = &iref;  //то же самое, что ip = &i;
```

Таким образом, iref становится синонимом переменной i.

При передаче больших объектов в функции с помощью ссылки он не копируется в стек и, следовательно, повышается производительность.

```
#include <iostream.h>

void incr (int&);

void main(void){

    int i = 5;

    incr(i);

    cout<< "i= " << i << "\n";

}

void incr (int& k){

    k++;

}
```

Поскольку ссылка это псевдоним, то при передаче объекта в функцию по ссылке внутри нее объект можно изменять. Ссылки не могут ссылаться на другие ссылки или на поле битов.

Не может быть массивов ссылок или указателей на ссылку.

Ссылка может использоваться для возврата результата из функции. Возвратить результат по ссылке - значит возвратить не указатель на объект и не его значение, а сам этот объект.

Работа с памятью

Указатели

Рассмотрим такую задачу: в файле записаны целые числа. Надо отсортировать их и записать в другой файл. Проблема заключается в том, что заранее неизвестно, сколько в файле таких чисел. В такой ситуации есть два варианта:

1. Выделить память с запасом, если известно, например, что этих чисел гарантированно не более 1000.
2. Использовать динамическое выделение памяти – сначала определить, сколько чисел в массиве, а потом выделить ровно столько памяти, сколько нужно.

Наиболее грамотным решением является второй вариант (использование динамических масси-

> **Указатель** – это переменная, в которой хранится адрес другой переменной или участка па- мяти.

вов). Для этого надо сначала поближе познакомиться с указателями.

Указатели являются одним из основных понятий языка Си. В такие переменные можно записывать адреса любых участков памяти, на чаще всего – адрес начального элемента динамического массива. Что же можно делать с указателями?

Объявить	`char *pC;`
	`int *pI;`
	`float *pF;`
		Указатели объявляются в списке переменных, но перед их именем ставится знак ***.** Указатель всегда указывает на переменную того типа, для которого он был объявлен. Например, **pC** может указывать на любой символ, **pI** – на любое целое число, а **pF** – на любое вещественное число.

Присвоить адрес	`int i, *pI;`
	`...`
	`pI = &i;`
		Такая запись означает: «записать в указатель **pI** адрес переменной **i**». Запись **&i** обозначает адрес переменной **i.**

Получить значение по этому адресу	`float f, *pF;`
	`...`
	`f = *pF;`
		Такая запись означает: «записать в переменную **f** то вещественное число, на которое указывает указатель **pF**». Запись ***pF** обозначает содержимое ячейки, на которую указывает **pF.**

Сдвинуть	`int *pI;`
	`...`
	`pI ++;`
	`pI += 4;`
	`-- pI;`
	`pI -= 4;`
		В результате этих операций значение указателя меняется особым способом: **pI++** сдвигает указатель на РАЗМЕР ЦЕЛОГО ЧИСЛА, то есть на 4 байта, а не на 1 как можно подумать. А запись **pI+=4** или **pI=pI+4** сдвигает указатель на 4 целых числа дальше, то есть на 16 байт.

Обнулить	`char *pC;`
	`...`
	`pC = NULL;`
		Если указатель равен нулевому адресу **NULL**, это обычно означает, что указатель недействительный. По нему нельзя ничего записывать (это вызывает сбой программы компьютера).

Вывести на экран	`int i, *pI;`
	`...`
	`pI = &i;`
	`printf("Адр.i =%p", pI);`
		Для вывода указателей используется формат **%p.**

Итак, что надо знать про указатели:

- указатель – это переменная, в которой записан адрес другой переменной;
- при объявлении указателя надо указать тип переменных, на которых он будет указывать, а перед именем поставить знак *****;
- знак **&** перед именем переменной обозначает ее адрес;
- знак ***** перед указателем в рабочей части программы (не в объявлении) обозначает значение ячейки, на которую указывает указатель;
- нельзя записывать по указателю, который указывает непонятно куда – это вызывает сбой программы, поскольку что-то стирается в памяти;
- для обозначения недействительного указателя используется константа **NULL**;
- при изменении значения указателя на **n** он в самом деле сдвигается к **n**-ому следующему числу данного типа, то есть для указателей на целые числа на **n*sizeof(int)** байт;
- указатель печатаются по формату **%p**.

Теперь вам должно быть понятно, что многие функции ввода типа **scanf** и **fscanf** в самом деле принимают в параметрах адреса переменных, например

```
   scanf ( "%d", &i);
```

Динамическое выделение памяти

Динамическими называются массивы, размер которых неизвестен на этапе написания программы. Прием, о котором мы будем говорить, относится уже не к стандартному языку Си, а к его расширению Си ++. Существуют и стандартные способы выделения памяти в языке Си (с помощью функций **malloc** и **calloc**), но они не очень удобны.

Следующая простейшая программа, которая использует динамический массив, вводит с клавиатуры размер массива, все его элементы, а затем сортирует их и выводит на экран.

```
#include <stdio.h>
main()
{
int N;      // размер массива (заранее неизвестен)
int *A;    // указатель для выделения памяти
printf ("Размер массива > ");  // ввод размера массива
scanf ("%d", &N);
A = new int [N];                  // выделение памяти
if ( A == NULL ) {                // если не удалось выделить
   printf("Не удалось выделить память");
   return 1;          // выход по ошибке, код ошибки 1
   }
for (i = 0; i < N; i ++ ) {     // дальше так же, как для массива
   printf ("\nA[%d] > ", i+1);
   scanf ("%d", &A[i]);
   }
 // здесь сортировка и вывод на экран
delete A;  // освободить память
}
```

Итак, мы хотим выделить в памяти место под новый массив целых чисел во время работы программы. Мы уже знаем его размер, пусть он хранится в переменной **N** (это число обязательно должно быть больше нуля)**.** Оператор выделения памяти **new** вернет нам адрес нового выделенного блока, и для работы с массивом нам надо где-то его запомнить. Вы уже знаете, что есть специальный класс переменных для записи в них адресов памяти – указатели.

- динамические массивы используются тогда, когда на момент написания программы размер массива неизвестен
- для того, чтобы работать с динамическим массивом, надо объявить указатель соответствующего типа (в нем будет храниться адрес первого элемента массива);

```
int *A;
```

- когда требуемый размер массива стал известен, надо использовать оператор **new** языка Си++, указав в скобках размер массива (в программе для краткости нет проверки на положительность **N**) ;

```
A = new int[N];
```

- **нельзя** использовать оператор **new** при отрицательном или нулевом **N**;
- после выделения памяти надо проверить, успешно ли оно прошло; если память выделить не удалось, то значение указателя будет равно **NULL**, использование такого массива приведет к сбой программы;
- работа с динамическим массивом, на который указывает указатель **A**, идет также, как и с обычным массивом размера **N** (помните, что язык Си не следит за выходом за границы массива);
- после использования массива надо освободить выделенную память, вызвав оператор

```
delete A;
```

- после освобождения памяти значение указателя не изменяется, но использовать его уже нельзя, потому что память освобождена;
- учитывая, что при добавлении числа к указателю он сдвигается на заданное число ячеек ЗАДАННОГО ТИПА, то следующие записи равносильны и вычисляют адрес **i**-ого элемента массива:

```
&A[i] и          A+i
```

Отсюда следует, что **A** совпадает с **&A[0]**, и поэтому имя массива можно использовать как адрес его начального элемента;

Ошибки, связанные с выделением памяти

Самые тяжелые и трудно вылавливаемые ошибки в программах на языке Си связаны именно с неверным использованием динамических массивов. В таблице перечислены наиболее тяжелые случаи и способы борьбы с ними.

Ошибка	Причина и способ лечения
Запись в чужую область памяти	Память была выделена неудачно, а массив используется. **Вывод:** надо всегда делать проверку указателя на **NULL.**
Повторное удаление	Массив уже удален и теперь удаляется снова.

указателя	**Вывод:** если массив удален из памяти, обнулите указатель – ошибка быстрее выявится.
Выход за границы массива	Запись в массив в элемент с отрицательным индексом или индексом, выходящим за границу массива
Утечка памяти	Неиспользуемая память не освобождается. Если это происходит в функции, которая вызывается много раз, то ресурсы памяти скоро будут израсходованы. **Вывод**: убирайте за собой «мусор» (освобождайте память).

Выделение памяти для матрицы

Для выделения памяти под одномерный массив целых чисел нам потребовался указатель на целые числа. Для матрицы надо выделить указатель на массив целых чисел, который объявляется как

```
int **A;
```

Но лучше всего сразу объявить новый тип данных - указатель на целое число. Новые типы объявляются директивой **typedef** вне всех процедур и функций (там же, где и глобальные переменные).

```
typedef int *pInt;
```

Этой строкой мы сказали компилятору, что любая переменная нового типа **pInt** представляет собой указатель на целое число или адрес массива целых чисел. К сожалению, место для матрицы не удается так же просто выделить в памяти, как мы делали это для одномерного массива.

Если написать просто

```
int M = 5, N = 7;
pInt *A;
A = new int[M][N];  // ошибочная строка
```

компилятор выдает множество ошибок. Связано это с тем, что ему требуется заранее знать длину одной строки, чтобы правильно расшифровать запись типа **A[i][j]**. Ниже рассмотрены три способа решения этой проблемы.

Известный размер строки

Если размер строки матрицы известен, а неизвестно только количество строк, можно поступить так: ввести новый тип данных – строка матрицы. Когда количество строк станет известно, с помощью оператора **new** выделяем массив таких данных.

```
typedef int row10[10]; // новы тип: массив из 10 элементов
```

```
main()
{
 int N;
 row10 *A;              // указатель на массив (матрица)
 printf ( "Введите число строк ");
 scanf ( "%d", &N );
 A = new row10[N];      // выделить память на N строк
```

```
    A[0][1] = 25;              // используем матрицу, как обычно
    printf("%d", A[2][3]);
    delete A;                  // освобождаем память
  }
```

Неизвестный размер строки

Пусть размеры матрицы **M** и **N** заранее неизвестны и определяются в ходе работы про- граммы. Тогда можно предложить следующий способ выделения памяти под новую матрицу. Поскольку матрицу можно рассматривать как массив из строк-массивов, объявим **M** указателей и выделим на каждый из них область памяти для одномерного массива размером **N** (то есть, на одну строку). Сами эти указатели тоже надо представить в виде динамического массива. Опре- делив требуемые размеры матрицы, мы выделяем сначала динамический массив указателей, а потом на каждый указатель – место для одной строки.

```
typedef int *pInt;  // новый тип данных: указатель на целое
main()
{
  int M, N, i;
  pInt *A;       // указатель на указатель
     // ввод M и N
  A = new pInt[M];    // выделить память под массив указателей
  for ( i = 0; i < M; i ++ ) // цикл по всем указателям
      A[i] = new int[N];     // выделяем память на строку i
   // работаем с матрицей A, как обычно
  for ( i = 0; i < M; i ++ ) // освобождаем память для всех строк
     delete A[i];
  delete A;                  // освобождаем массив указателей
}
```

В рассмотренном выше случае на каждую строку выделяется свой участок памяти. Можно по- ступить иначе: сначала выделим область памяти сразу на всю матрицы и запишем ее адрес в **A[0]**. Затем расставим указатели так, чтобы **A[1]** указывал на **N+1**-ый элемент с начала блока (начало строки 1), **A[2]** – на **2N+1**-ый (начало строки 2) и т.д. Таким образом, в памяти выде- ляется всего два блока – массив указателей и сама матрица.

```
typedef int *pInt;
main()
{
  int M, N, i;
  pInt *A;  // указатель на указатель
    // ввод M и N
  A = new pInt[M];           // память на массив указателей
```

```
   A[0] = new int [M*N];        // память для матрицы
   for ( i = 1; i < M; i ++ ) // расставляем указатели
      A[i] = A[i-1] + N;
     // работаем с матрицей
   delete A[0];                 // освобождаем матрицу
   delete A;                    // освобождаем указатели
}
```

Стандартные функции работы с файлами.

Теоретическая часть

Файл – это именованная область ячеек памяти, в которой хранятся данные одного типа. Файл имеет следующие характерные особенности:

- уникальное имя;
- однотипность данных;
- произвольная длина, которая ограничивается только емкостью диска.

Файлы бывают текстовыми и двоичными.

Текстовый файл – файл, в котором каждый символ из используемого набора хранится в виде одного байта (код, соответствующий символу). Текстовые файлы разбиваются на несколько строк с помощью специального символа "конец строки". Текстовый файл заканчивается специальным символом "конец файла".

Двоичный файл – файл, данные которого представлены в бинарном виде. При записи в двоичный файл символы и числа записываются в виде последовательности байт (в своем внутреннем двоичном представлении в памяти компьютера).

Все операции ввода-вывода реализуются с помощью функций, которые находятся в библиотеке C++. Библиотека C++ поддерживает три уровня ввода-вывода:

потоковый ввод-вывод;

ввод-вывод нижнего уровня;

ввод-вывод для консоли и портов (зависит от ОС).

Поток – это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику.

Функции библиотеки ввода-вывода языка C++, поддерживающие обмен данными с файлами на уровне потока, позволяют обрабатывать данные различных размеров и форматов, обеспечивая при этом буферизованный

ввод и вывод. Таким образом, поток представляет собой этот файл вместе с предоставленными средствами буферизации.

Чтение данных из потока называется **извлечением**, вывод в поток – **помещением** (включением).

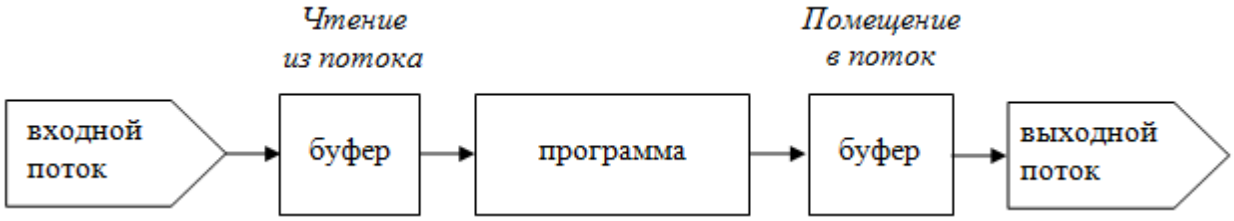


Рис. 1 Буферизация данных при работе с потоками

Для работы с файлом в языке C++ необходима ссылка на файл. Для определения такой ссылки существует структура **FILE**, описанная в заголовочном файле **stdio.h**. Данная структура содержит все необходимые поля для управления файлами, например: текущий указатель буфера, текущий счетчик байтов, базовый адрес буфера ввода-вывода, номер файла.

Функция открытия файла

При открытии файла (потока) в программу возвращается указатель на поток (файловый указатель), являющийся указателем на объект структурного типа FILE. Этот указатель идентифицирует поток во всех последующих операциях.

Например:

```
#include
.............
FILE *fp;
```

Для открытия файла существует функция **fopen**, которая инициализирует файл.

Синтаксис:

fp=fopen(ИмяФайла, РежимОткрытия);

где **fp** – указатель на поток (файловый указатель);

ИмяФайла – указатель на строку символов, представляющую собой допустимое имя файла, в которое может входить спецификация файла (включает обозначение логического устройства, путь к файлу и собственно имя файла);

РежимОткрытия – указатель на строку режима открытия файла.

Например:
fp=fopen("t.txt","r");

Существуют несколько режимов открытия файлов.

Поток можно открыть в текстовом (t) или двоичном (b) режиме. По умолчанию используется текстовый режим. В явном виде режим указывается следующим образом:

"r+b" или "rb" – двоичный (бинарный) режим;

"r+t" или "rt" – текстовый режим.

Функция закрытия файла

Открытые на диске файлы после окончания работы с ними рекомендуется закрыть явно. Это является хорошим тоном в программировании.

Синтаксис:

int fclose(УказательНаПоток);

Возвращает 0 при успешном закрытии файла и -1 в противном случае.

Открытый файл можно открыть повторно (например, для изменения режима работы с ним) только после того, как файл будет закрыт с помощью функции fclose().

Функция удаления файла

Синтаксис:

int remove(const char *filename);

Эта функция удаляет с диска файл, указатель на который хранится в файловой переменной filename. Функция возвращает ненулевое значение, если файл невозможно удалить.

Функция переименования файла

Синтаксис:

int rename(const char *oldfilename, const char *newfilename);

Функция переименовывает файл; первый параметр – старое имя файла, второй – новое. Возвращает 0 при неудачном выполнении.

Функция контроля конца файла

Для контроля достижения конца файла есть функция feof.

int feof(FILE * filename);

Функция возвращает ненулевое значение, если достигнут конец файла.

Функции ввода-вывода данных файла

1) Символьный ввод-вывод

Для символьного ввода-вывода используются функции:

int fgetc(FILE *fp);

где **fp** – указатель на поток, из которого выполняется считывание.

Функция возвращает очередной символ в формате int из потока fp. Если символ не может быть прочитан, то возвращается значение EOF.

int fputc(int c, FILE*fp);

где **fp** – указатель на поток, в который выполняется запись;

c – переменная типа int, в которой содержится записываемый в поток символ.

Функция возвращает записанный в поток fp символ в формате int. Если символ не может быть записан, то возвращается значение EOF.

Пример 1.

```
#include "stdafx.h"
#include “iostream”
using namespace std;
int main()
{
FILE *f;
int c;
char *filename="t.txt";
if ((f=fopen(filename,"r"))==0)
```

```
perror(filename);
else
while((c = fgetc(f)) !=EOF)
putchar(c);
//вывод с на стандартное устройство вывода
fclose(f);
system("pause");
return 0;
}
```

2) Строковый ввод-вывод

Для построчного ввода-вывода используются следующие функции:

char *fgets(char *s, int n, FILE *f);

где char *s – адрес, по которому размещаются считанные байты;

int n – количество считанных байтов;

FILE *f – указатель на файл, из которого производится считывание.

Прием байтов заканчивается после передачи n-1 байтов или при получении управляющего символа '\n'. Управляющий символ тоже передается в принимающую строку. Строка в любом случае заканчивается '\0'. При успешном завершении считывания функция возвращает указатель на прочитанную строку, при неуспешном – 0.

int fputs(char *s, FILE *f);

где char *s – адрес, из которого берутся записываемые в файл байты;

FILE *f – указатель на файл, в который производится запись.

Символ конца строки ('\0') в файл не записывается. Функция возвращает EOF, если при записи в файл произошла ошибка, при успешной записи возвращает неотрицательное число.

Пример 2. Построчное копирование данных из файла f1.txt в файл f2.txt.

```
#include"stdafx.h"
#include “iostream"
using namespace std;
#define MAXLINE 255 //максимальная длина строки
int main()
{
//копирование файла in в файл out
FILE *in, //исходный файл
*out; //принимающий файл
char buf[MAXLINE];
//строка, с помощью которой выполняется копирование
in=fopen("f1.txt","r");
//открыть исходный файл для чтения
out=fopen("f2.txt","w");
//открыть принимающий файл для записи
while(fgets(buf, MAXLINE, in)!=0)
//прочитать байты из файла in в строку buf
fputs(buf, out);
//записать байты из строки buf в файл out
fclose(in); //закрыть исходный файл
fclose(out);//закрыть принимающий файл
system("pause");
return 0;
}
```

Динамические структуры данных

Часто в серьезных программах надо использовать данные, размер и структура которых должны **меняться** в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы.

Например, надо проанализировать текст и определить, какие слова и в каком количество в нем встречаются, причем эти слова нужно расставить по алфавиту.

В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью **ссылок**.

данные ссылки

Каждый элемент (**узел**) состоит из двух областей памяти: **поля данных** и **ссылок**. Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке Си для организации ссылок используются переменные указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются **нулевые ссылки** (**`NULL`**).

Линейный список

В простейшем случае каждый узел содержит всего одну ссылку. Для определенности будем считать, что решается задача частотного анализа текста – определения всех слов, встречающихся в тексте и их количества. В этом случае область данных элемента включает строку (длиной не более 40 символов) и целое число.

Каждый элемент содержит также ссылку на **следующий** за ним элемент. У последнего в списке элемента поле ссылки содержит **`NULL`**. Чтобы не потерять список, мы должны где-то (в переменной) хранить адрес его первого узла – он называется «головой» списка. В программе надо объявить два новых типа данных – узел списка **`Node`** и указатель на него **`PNode`**. Узел представляет собой структуру, которая содержит три поля -

строку, целое число и указатель на такой же узел. Правилами языка Си допускается объявление

```
    struct Node {
        char  word[40];  // область данных
          int  count;
          Node  *next;     // ссылка на
следующий узел
     }; typedef Node *PNode;  // тип данных:
указатель на узел
```

В дальнейшем мы будем считать, что указатель **Head** указывает на начало списка, то есть, объявлен в виде

```
    PNode Head = NULL;
```

Первая буква «P» в названии типа **PNode** происходит от слова *pointer* – указатель (англ.) В начале работы в списке нет ни одного элемента, поэтому в указатель **Head** записывается нулевой адрес **NULL**.

Создание элемента списка

Для того, чтобы добавить узел к списку, необходимо создать его, то есть выделить память под узел и запомнить адрес выделенного блока. Будем считать, что надо добавить к списку узел, соответствующий новому слову, которое записано в переменной **NewWord**. Составим функцию, которая создает новый узел в памяти и возвращает его адрес. Обратите внимание, что при записи данных в узел используется обращение к полям структуры через указатель.

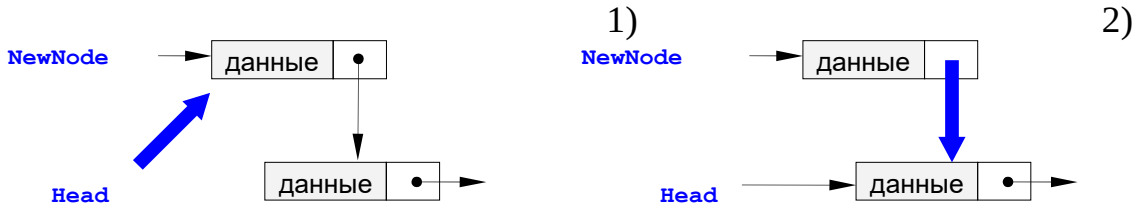
```
    PNode CreateNode ( char NewWord[] )
    {
       PNode NewNode = new Node; // указатель на
новый узел    strcpy(NewNode->word, NewWord); //
записать слово    NewNode->count = 1;
// счетчик слов = 1    NewNode->next = NULL;
// следующего узла нет    return NewNode;  //
результат функции – адрес узла }
```

После этого узел надо 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 надо 1) установить ссылку узла **NewNode** на голову существующего списка и 2) установить голову списка на новый узел.



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

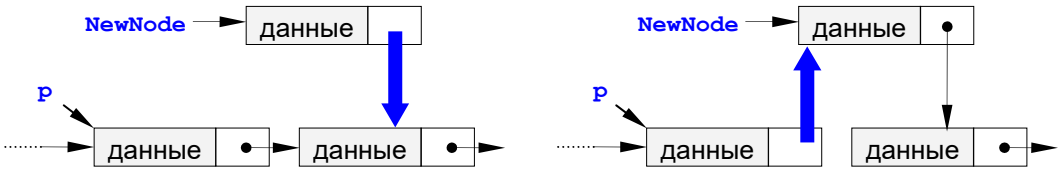
```
void AddFirst (PNode &Head, PNode NewNode)
{
  NewNode->next = Head;   Head
= NewNode;
}
```

Добавление узла после заданного

Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после узла с адресом **p**. Эта операция выполняется в два этапа:

1) установить ссылку нового узла на узел, следующий за данным; 2) установить ссылку данного узла **p** на **NewNode**.

1) 2)



Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла **p**, будет потерян адрес следующего узла.

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

Добавление узла перед заданным

Эта 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 сведется 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 }
```

Такая процедура обеспечивает «защиту от дурака»: если задан узел, не присутствующий в списке, то в конце цикла указатель **q** равен **NULL** и ничего не происходит.

Существует еще один интересный прием: если надо вставить новый узел **NewNode до** заданного узла **p**, вставляют узел **после** этого узла, а

потом выполняется обмен данными между узлами **NewNode** и **p**. Таким образом, по адресу **p** в самом деле будет расположен узел с новыми данными, а по адресу **NewNode** – с теми данными, которые были в узле **p**, то есть мы решили задачу. Этот прием не сработает, если адрес нового узла **NewNode** запоминается где-то в программе и потом используется, поскольку по этому адресу будут находиться другие данные.

Добавление узла в конец списка

Для решения задачи надо сначала найти последний узел, у которого ссылка равна **NULL**, а затем воспользоваться процедурой вставки после заданного узла. Отдельно надо 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 того, чтобы пройти весь список и сделать что-либо с каждым его элементом, надо начать 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
         // делаем что-нибудь с узлом p
       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 **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 слову (для которого поле **word** совпадает с заданной строкой **NewWord**), 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; }
```

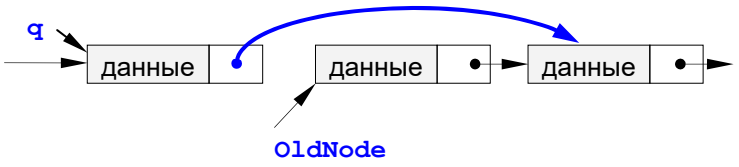
Вернемся к задаче построения алфавитно-частотного словаря. Для того, чтобы добавить новое слово в нужное место (в алфавитном порядке), требуется найти адрес узла, *перед* которым надо вставить новое слово. Это будет первый от начала списка узел, для которого «его» слово окажется «больше», чем новое слово. Поэтому достаточно просто изменить условие в цикле **while** в функции **Find**., учитывая, что функция **strcmp** возвращает «разность» первого и второго слова.

```
PNode FindPlace (PNode Head, char NewWord[])
{
   PNode q = Head;
   while (q && (strcmp(q->word, NewWord) > 0))
      q = q->next;
return q; }
```

Эта функция вернет адрес узла, перед которым надо вставить новое слово (когда функция **strcmp** вернет положительное значение), или **NULL**, если слово надо добавить в конец списка.

Удаление узла

Эта процедура также связана с поиском заданного узла по всему списку, так как нам надо поменять ссылку у предыдущего узла, а перейти к нему непосредственно невозможно. Если мы нашли узел, за которым идет удаляемый узел, надо просто переставить ссылку.



Отдельно обрабатывается случай, когда удаляется первый элемент списка. При удалении узла освобождается память, которую он занимал.

Отдельно рассматриваем случай, когда удаляется первый элемент списка. В этом случае адрес удаляемого узла совпадает с адресом головы списка **Head** и надо просто записать в **Head** адрес следующего элемента.

```
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;
```

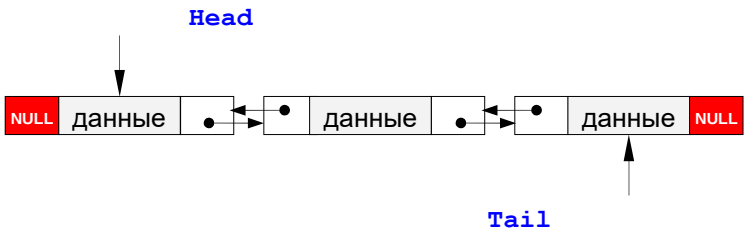
```
   if ( q == NULL ) return; // если не нашли, выход    q->next =
OldNode->next;
   }
delete OldNode;              // освобождаем память }
```

Барьеры

Вы заметили, что для рассмотренного варианта списка требуется отдельно обрабатывать граничные случаи: добавление в начало, добавление в конец, удаление одного из крайних элементов. Можно значительно упростить приведенные выше процедуры, если установить два барьера – фиктивные первый и последний элементы. Таким образом, в списке всегда есть хотя бы два элемента-барьера, а все рабочие узлы находятся между ними.

Двусвязный список

Многие проблемы при работе с односвязным списком вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная идея – хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Для доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списка (**Head**) и на «хвост» - последний элемент (**Tail**).



Каждый узел содержит (кроме полезных данных) также ссылку на **следующий** за ним узел (поле **next**) и предыдущий (поле **prev**). Поле **next** у последнего элемента и поле **prev** у первого содержат **NULL**. Узел объявляется так:

```
struct Node {
   char word[40];      // область данных    int
count;
   Node *next, *prev; // ссылки на соседние узлы
};
typedef Node *PNode;  // тип данных «указатель на узел»
```

В дальнейшем мы будем считать, что указатель **Head** указывает на начало списка, а указатель **Tail** – на конец списка:

```
PNode Head = NULL, Tail = NULL;
```

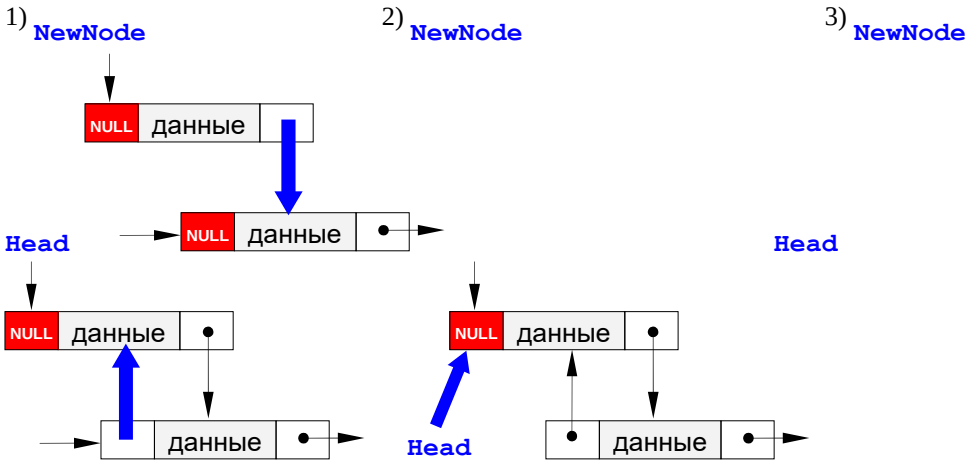
Для пустого списка оба указателя равны **NULL**.

Операции с двусвязным списком

Добавление узла в начало списка

При добавлении нового узла **NewNode** в начало списка надо

1) установить ссылку **next** узла **NewNode** на голову существующего списка и его ссылку **prev** в **NULL**;
2) установить ссылку **prev** бывшего первого узла (если он существовал) на **NewNode**; 3) установить голову списка на новый узел;
4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.



По такой схеме работает следующая процедура:

```
void AddFirst(PNode &Head, PNode &Tail, PNode NewNode)
{
   NewNode->next = Head;    NewNode->prev
= NULL;
   if ( Head ) Head->prev = NewNode;    Head
= NewNode;
   if ( ! Tail ) Tail = Head; // этот элемент – первый }
```

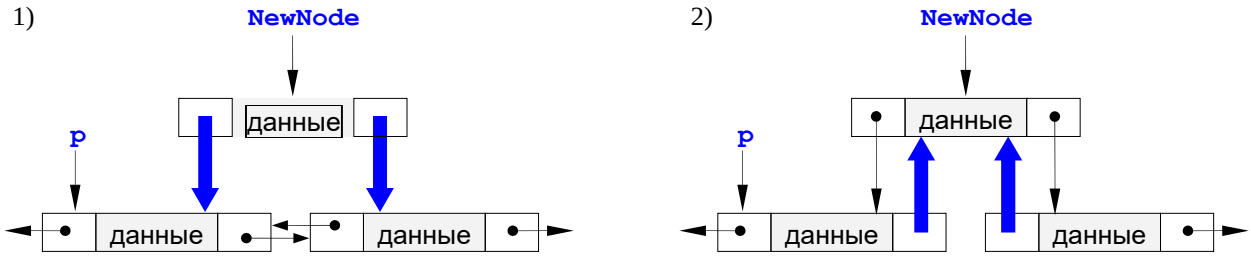
Добавление узла в конец списка

Благодаря симметрии добавление нового узла **NewNode** в конец списка проходит совершенно аналогично, в процедуре надо везде заменить **Head** на **Tail** и наоборот, а также поменять **prev** и **next**.

Добавление узла после заданного

Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после **p**. Если узел **p** является последним, то операция сводится к добавлению в конец списка (см. выше). Если узел **p** – не последний, то операция вставки выполняется в два этапа:

1) установить ссылки нового узла на следующий за данным (***next***) и предшествующий ему (***prev***);
2) установить ссылки соседних узлов так, чтобы включить **NewNode** в список.



Такой метод реализует приведенная ниже процедура (она учитывает также возможность вставки элемента в конец списка, именно для этого в параметрах передаются ссылки на голову и хвост списка):

```
    void AddAfter (PNode &Head, PNode &Tail,
                   PNode p, PNode NewNode)
    {
    if ( ! p->next )
     AddLast (Head, Tail, NewNode);  // вставка
в конец списка else {
       NewNode->next = p->next;  // меняем
ссылки нового узла    NewNode->prev = p;
       p->next->prev = NewNode;  // меняем
ссылки соседних узлов
   p->next = NewNode;
       }
```

```
}
```

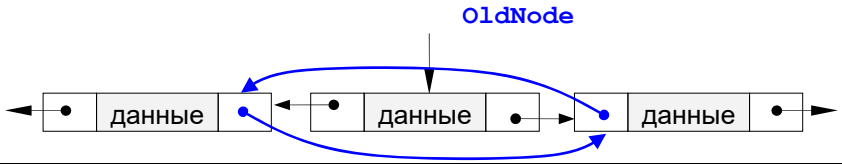
Добавление узла перед заданным выполняется аналогично.

Поиск узла в списке

Проход по двусвязному списку может выполняться в двух направлениях – от головы к хвосту (как для односвязного) или от хвоста к голове.

Удаление узла

Эта процедура также требует ссылки на голову и хвост списка, поскольку они могут измениться при удалении крайнего элемента списка. На первом этапе устанавливаются ссылки соседних узлов (если они есть) так, как если бы удаляемого узла не было бы. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке внизу. Отдельно проверяется, не является ли удаляемый узел первым или последним узлом списка.




```
void Delete(PNode &Head, PNode &Tail, PNode OldNode)
{
if (Head == OldNode) {
   Head = OldNode->next;     // удаляем первый элемент     if (
Head )
         Head->prev = NULL;
   else Tail = NULL;         // удалили единственный элемент
   } else
{
   OldNode->prev->next = OldNode->next;     if
( OldNode->next )
   OldNode->next->prev = OldNode->prev;
   else Tail = NULL;        // удалили последний элемент
   }
delete OldNode;
}
```

Циклические списки

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель **`next`** последнего элемента указывает на первый элемент, и (для двусвязных списков) указатель **`prev`** первого элемента указывает на последний. В таких списках понятие «хвоста» списка не имеет смысла, для работы с ним надо использовать указатель на «голову», причем «головой» можно считать любой элемент.

Парадигмы ООП – инкапсуляция, наследование, полиморфизм

Инкапсуляция - сведение кода и данных воедино в одном объекте, получившим название класс.

Наследование - наличие в языке ООП механизма, позволяющего объектам класса наследовать характеристики более простых и общих типов. Наследование обеспечивает как требуемый уровень общности, так и необходимую специализацию.

Полиморфизм - дословный перевод с греческого "много форм". В C++ полиморфизм реализуется с помощью виртуальных функций, которые позволяют в рамках всей иерархии классов иметь несколько версий одной и той же функции. Решение о том, какая именно версия должна выполняться в данный момент, определяется на этапе выполнения программы и носит название позднего связывания.

Существует несколько реализаций системы, поддерживающих стандарт C++, из которых можно выделить реализации Visual C++ (Microsoft) и Builder C++ (Inprise). Отличия относятся в основном к используемым библиотекам классов и средам разработки. В действительности в C++ программах можно использовать библиотеки языка C, библиотеки классов C++, библиотеки визуальных классов VCL (Builder C++), библиотеку MFC (Visual C++ и Builder C++).

Язык C++ является родоначальником множества объектно-ориентированных языков, таких как Java, C#, PHP и др.

Данное пособие предназначено для начинающих изучение технологии ООП для проектирования систем управления на основе C++.

Использование динамических массивов. Теоретические сведения

Объявление динамического массива

Массивы, создаваемые в динамической памяти, будем называть *динамическими* (размерность становится известна в процессе выполнения программы). При описании массива после имени в квадратных скобках задается количество его элементов (размерность), например **`int a[10]`**. Размерность массива может быть задана только константой или константным выражением.

При описании массив можно инициализировать, то есть присвоить его элементам начальные значения, например:

```
int a[10] = {1, 1, 2, 2, 5, 100};
```

Если инициализирующих значений меньше, чем элементов в массиве, остаток массива обнуляется, если больше — лишние значения не используются. Элементы массивов нумеруются с нуля, поэтому максимальный номер элемента всегда на единицу меньше размерности. Номер элемента указывается после его имени в квадратных скобках, например, **`a[0], a[3].`**

Если до начала работы программы неизвестно, сколько в массиве элементов, в программе следует использовать динамические массивы. Память под них выделяется с помощью операции new или функции malloc в динамической области памяти во время выполнения программы. Адрес начала массива хранится в переменной, называемой указателем. Например:

```
int n = 10;
int *mass1 = new int[n];
```

Во второй строке описан указатель на целую величину, которому присваивается адрес начала непрерывной области динамической памяти, выделенной с помощью операции new. Выделяется столько памяти, сколько необходимо для хранения n величин типа int. Величина n может быть переменной. Инициализировать динамический массив нельзя.

Обращение к элементу динамического массива осуществляется так же, как и к элементу обычного. Если динамический массив в какой-то момент работы программы перестает быть нужным и мы собираемся впоследствии использовать эту память повторно, необходимо освободить ее с помощью операции delete[], например: **`delete [] a;`** (размерность массива при этом не указывается).

```
delete[] mass1;
```

При необходимости создания многомерных динамических массивов сначала необходимо с помощью операции new выделить память под n указателей (вектор, элемент которого - указатель), при этом все указатели располагаются в памяти последовательно друг за другом. После этого необходимо в цикле каждому указателю присвоить адрес выделенной области памяти размером, равным второй границе массива

```
mass2=new int*[row];

// mass2 - указатель на массив указателей на одномерные массивы

for(i=0;i<row;i++)
```

```
mass2[i]=new int[col]; // каждый элемент массива указывает на одномерный

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

for (j=0;j<col;j++)
```

Освобождение памяти от двумерного динамического массива:

```
for(i=0;i<row;i++)    //удаление всех одномерных

delete[] mass2[i];    // массивов

delete[] mass2;       // удаление массива указателей на одномерные массивы
```

Классы. Программирование линейных алгоритмов с использованием функций инициализации set() и вывода результатов print(). Теоретические сведения

Основное отличие C++ от C состоит в том, что в C++ имеются классы. С точки зрения языка C классы в C++ - это структуры, в которых вместе с данными определяются функции. Это и есть инкапсуляция в терминах ООП.

Класс (class) - это тип, определяемый пользователем, включающий в себя данные и функции, называемые методами или функциями-членами класса.

Данные класса - это то, что класс знает.

Функции-члены (методы) класса - это то, что класс делает.

Таким образом, определение типа задаваемого пользователем (class) содержит спецификацию данных, требующихся для представления объекта этого типа, и набор операций (функций) для работы с подобными объектами.

Объявление класса

Приведем пример объявления класса

```
class my_Fun

{

// компоненты-данные

double x,y;

// компоненты-функции

public:

// функция инициализации
```

```
void set(char *c,double X)

{

x=X;

y=sin(x);

}

// функция вывода результатов

void print(void)

{

cout << point<<y << endl;

}

};
```

Обычно описания классов включают в заголовочные файлы (*.H), а реализацию функций-членов классов - в файлы *.CPP.

Для каждого объекта класса устанавливается область видимости либо явно – указанием уровня доступа одним из ключевых слов public, private, protected с двоеточием, либо неявно – по умолчанию. Указание области видимости относится ко всем последующим объектам класса, пока не встретится указание другой области видимости. Область видимости public разрешает доступ к объектам класса из любой части программы, в которой известен этот объект (общедоступный). Область видимости private разрешает доступ к объектам класса только из методов этого класса. Объекты с такой областью видимости называют частными. Область видимости protected определяется для защищенных объектов, она имеет смысл только в иерархической системе классов и разрешает доступ к объектам этой области из методов производных классов. В теле класса ключевое слово области видимости может использоваться неоднократно. Область видимости для объектов типа «класс» по умолчанию private.

Способы объявления и инициализации объектов и доступ к методам класса:

1. Прямой вызов

```
my_Fun Fun1; //объявление объекта1,но не инициализация

Fun1.set("Function1 = ",1.0); // инициализация данных

Fun1.print();                 // прямой вызов

cout << "Input enter1..." << endl<<endl;
```

2. Косвенный вызов

```
my_Fun *p1 = &Fun1;  // воспользовались объектом 1

                     // новая инициализация
```

```
p1->set("Function1 = ",1.0); // косвенный вызов

p1->print();                 // косвенный вызов

cout << "Input enter1..." << endl<<endl;
```

3. Динамическое выделение памяти

```
my_Fun *p1 = new my_Fun;

p1->set("Function1 = ",1.0);  // косвенный вызов

p1->print();                  // косвенный вызов

cout << "Input enter1..." << endl<<endl;

// удаляется динамически выделенный объект

delete p1;
```

Классы. Программирование линейных алгоритмов с использованием конструктора, деструктора, friend - функции инициализации set() и функции вывода результатов print(). Теоретические сведения

Конструктор класса

Конструктор – это метод класса, имя которого совпадает с именем класса. Конструктор вызывается автоматически после выделения памяти для переменной и обеспечивает инициализацию компонент – данных. Конструктор не имеет никакого типа (даже типа void) и не возвращает никакого значения в результате своей работы. Конструктор нельзя вызывать как обычную компонентную функцию в программе. Для класса может быть объявлено несколько конструкторов, различающихся числом и типами параметров. При этом даже если для объектного типа не определено ни одного конструктора, компилятор создает для него конструктор по умолчанию, не использующий параметров, а также конструктор копирования, необходимый в том случае, если переменная объектного типа передается в конструктор как аргумент. В этом случае создаваемый объект будет точной копией аргумента конструктора.

```
class my_Fun

{

// компоненты-данные

double x;

unsigned size;

public:

// объявление конструктора 1 (с параметрами)
```

```
my_Fun (double X=0);

// объявление конструктора 2 (без параметров)

my_Fun(void);

// объявление и описание деструктора

~my_Fun ()

{

cout<<"Destroyed object... "<<endl;

}

// описание конструктора 1

my_Fun::my_Fun (double X)

{

cout<<"Constructor1...."<<endl;

x=X;

}

// описание конструктора 2

my_Fun::my_Fun (void)

{

cout<<"Constructor2..."<<endl;

x=5.0;

}

}
```

Деструктор класса

Еще одним специальным методом класса является деструктор. Деструктор вызывается перед освобождением памяти, занимаемой объектной переменной, и предназначен для выполнения дополнительных действий, связанных с уничтожением объектной переменной, например, для освобождения динамической памяти, закрытия, уничтожения файлов и т.п. Деструктор всегда имеет то же имя, что и имя класса, но перед именем записывается знак ~ (тильда). Деструктор не имеет параметров и подобно конструктору не возвращает никакого значения. Таким образом, деструктор не может быть перегружен и должен существовать в классе в единственном экземпляре. Деструктор вызывается автоматически при уничтожении объекта. Таким образом, для статически определенных объектов деструктор вызывается, когда заканчивается блок программы, в котором

определен объект (блок в данном случае – составной оператор или тело функции). Для объектов, память для которых выделена динамически, деструктор вызывается при уничтожении объекта операцией delete.

Дружественная функция (friend)

В языке C++ одна и та же функция не может быть компонентом двух разных классов. Чтобы предоставить функции возможность выполнения действий над различными классами можно определить обычную функцию языка C++ и предоставить ей право доступа к элементам класса типа private, protected. Для этого нужно в описании класса поместить заголовок функции, перед которым поставить ключевое слово friend. Дружественная функция не является методом класса, не зависит от позиции в классе и спецификаторов прав доступа. Friend – функции получают доступ к членам класса через указатель, передаваемый им явно. Можно сделать все функции класса Y друзьями класса X в одном объявлении.

Класс «Динамическая строка» и перегрузка операций. Теоретические сведения

Для представления символьной (текстовой) информации можно использовать символы, символьные переменные и символьные константы.

Символьная константа представляется последовательностью символов, заключенной в кавычки: “Начало строки \n”. В C++ нет отдельного типа для строк. Массив символов - это и есть строка. Количества элементов в таком массиве на один элемент больше, чем изображение строки, т. к. в конец строки добавлен ‘\0’ (нулевой байт).

Присвоить значение массиву символов с помощью обычного оператора присваивания нельзя. Поместить строку в массив можно либо при вводе, либо с помощью инициализации:

```
char s[] = "ABCDEF";
```

Для работы со строками существует специальная библиотека string.h. Примеры функций для работы со строками из библиотеки string.h в таблице 2:

Таблица 2 - Функции работы со строками

Функция	**Прототип и краткое описание функции**
strcmp	int strcmp(const char *str1, const char *str2); Сравнивает строки str1 и str2. Если str1< str2, то результат отрицательный, если str1 = str2, то результат равен 0, если str1> str2, то результат положительный.
strcpy	char* strcpy(char*s1, const char *s2); Копирует байты из строки s1 в строку s2

strdup	char *strdup (const char *str); Выделяет память и перености в нее копию строки str.
strlen	unsigned strlen (const char *str); Вычисляет длину строки str.
strncat	char *strncat(char *s1, const char *s2, int kol); Приписывает kol символов строки s1 к строке s2.
strncpy	char *strncpy(char *s1, const char *s2, int kol); Копирует kol символов строки s1 в строку s2.
strnset	char *strnset(char *str, int c, int kol); Заменяет первые kol символов строки s1 символом c.

Строки, при передаче в функцию, в качестве фактических параметров могут быть определены либо как одномерные массивы типа char[], либо как указатели типа char*. В отличие от обычных массивов в этом случае нет необходимости явно указывать длину строки.

Функции преобразования строки S в число:

целое: int *atoi*(S); длинное целое: long *atol*(S); действительное: double atof(S); при ошибке возвращает значение 0.

Функции преобразования числа V в строку S:

целое: *itoa*(int V,char S,int kod); длинное целое: *ltoa*(long V,char S,int kod); 2<=kod<=36, для отрицательных чисел kod=10.

Перегрузка операций

Для перегрузки операции для класса в C++ используется следующий синтаксис:

```
<Тип>  operator <операция>(<входные параметры>)
{
<операторы>;
}
```

где < Тип > - тип, возвращаемый функцией;

operator - ключевое слово;

< операция > - перегружаемая операция.

В языке C++ имеются следующие ограничения на перегрузку операций:

- C++ не различает префиксную и постфиксную формы ++ и - -;
- переопределяемая операция должна присутствовать в языке (например, нельзя определить операцию с символом #);

- нельзя переопределить операторы, заданные следующими символами . * :: ? ;
- переопределённые операции сохраняют свой изначальный приоритет.

Наличие в классе конструктора String:: String(String&) и операторов присваивания позволяет защитить объекты класса от побитового копирования.

Пример: Ввести с клавиатуры строку символов. Признак окончания ввода строки - нажатие клавиши "Ввод". Программа должна определить длину введенной строки L и если L<10 – возвратить строку, которая не содержит заглавных латинских букв.

```
#include <iostream.h>

#define SIZE 255 //длина строки по умолчанию

#include <string.h>

#include <stdlib.h>

#include <stdio.h>

#include <istream.h>

class X{

char *str;

char *str_return;

public:

X(); //конструктор по-умолчанию

X(char*); //конструктор, которому можно передавать параметр

~X(); //деструктор

char* Run(); //метод, выполняющий поставленную задачу.

void Set(char*);

friend void print(X&); //функция-друг печати

friend ostream& operator<<(ostream&,X&); //перегрузка оператора вывода

friend istream& operator>>(istream&,X&); //перегрузка оператора ввода

friend char* Run(X&); //функция-друг, выполняющий поставленную задачу.

};

X::X(){

str=new char[SIZE];

str[0]='\0';
```

```
str_return=new char[SIZE];

str_return[0]='\0';

};

X::X(char *s){

str=new char[SIZE];

strcpy(str,s);

str_return=new char[SIZE];

str_return[0]='\0';

};

X::~X(){

delete[] str;

cout<<"...destructor has been called"<<endl;

};

void X::Set(char* s){

for (unsigned int i=0;i<strlen(s);i++)

str[i]=s[i];

str[i]='\0';

};

char* X::Run(){ /*метод, решающий конкретную задачу, в данном случае -
выделение из строки подстроки, не содержащей заглавных латинских букв, если
длина исходной строки меньше 10*/

int j=0;

if (strlen(str)<10) {

for (unsigned int i=0;i<strlen(str);i++)

if ( ((int)str[i]<65) || ((int)str[i]>90) ) {

str_return[j]=str[i]; j++;

};

str_return[j]='\0';

}

else strcpy(str_return,str);
```

```
return str_return;
};
char* Run(X &obj){return obj.Run();};
void print(X &obj){cout<<obj.str<<" "<<obj.str_return<<endl;};
ostream& operator<<(ostream &stream,X &ob) {
stream << ob.str ;
return stream;
};
istream &operator>>(istream &stream,X &ob){
stream >> ob.str;
return stream;
};
void main (void){
char s[265];

cout<<"Type anything and press \"Enter\":"<<endl;
cin.getline(s,256); //считываем полностью всю строку
X str(s); //доступ к методам класса непосредственно через переменную,
//начальное значение устанавливаем через конструктор
cout<<"You have type:"<<endl;
print(str);
cout<<"Output string:"<<endl;
cout<<Run(str)<<endl;
cout<<"Type anything and press \"Enter\":"<<endl;
cin.getline(s,256);
X *pstr; //доступ к методам класса через указатель
pstr=new X();
pstr->Set(s);
```

```
cout<<"You have type:"<<endl;

print(*pstr);

cout<<"Output string:"<<endl;

cout<<Run(*pstr)<<endl;

delete pstr;

};
```

Наследование классов, механизм виртуальных функций. Теоретические сведения

Наследование - механизм создания производного класса из базового. Т.е., к существующему классу можно что-либо добавить, или изменять его каким-либо образом для создания нового (производного) класса. Это мощный механизм для повторного использования кода. Наследование позволяет создавать иерархию связанных типов, совместно использующих код и интерфейс. Модификатор прав доступа используется для изменения доступа к наследуемым объектам в соответствии с правилами, указанными в таблице 1.

Таблица 1 – Доступ в классах при наследовании

Доступ в базовом классе	**Модификатор прав доступа**	**Доступ в производном классе**
private	private	не доступны
private	public	не доступны
protected	private	private
protected	public	protected
public	private	private
public	public	public

Ограничение на наследование

При определении производного класса не наследуются из базового:

1. конструкторы;
2. деструкторы;
3. операторы new, определенные пользователем;
4. операторы присвоения, определенные пользователем;
5. отношения дружественности.

Использование косвенной адресации с установкой указателей на базовый класс. Механизм косвенной адресации рассмотрим на примере:

```
class B

{

public:

int x;

B() {             // Конструктор по умолчанию

x = 4; }

};

class D : public B {      // Производный класс

public:

int y;

D()

{             // Конструктор по умолчанию

y = 5; }

};

void main(void)  {

D  d;// Конструктор класса D создает объект d

B *p; // Указатель установлен на базовый касс

p = &d;// Указатель p инициализируется адресом d

// косвенное обращение к объектам базового и производного классов

// «считываем их текущее состояние в переменные

int i = p -> x;  // Базовый класс виден напрямую

int j = ( ( D* ) p )p -> y;// Прямое преобразование указателя на D

// через переменные печатаем их текущее состояние

cout << " x_i= " << i << endl;

cout << " y_j= " << j << endl;

getch();
```

```
}
```

Виртуальная функция и механизм позднего связывания

Виртуальная функция объявляется в базовом или впроизводном классе и, затем, переопределяется в наследуемых классах. Совокупность классов (подклассов), в которых определяется и переопределяется виртуальная функция, называется полиморфическим кластером, ассоциированным с некоторой виртуальной функцией. В пределах полиморфического кластера сообщение связывается с конкретной виртуальной функцией-методом во время выполнения программы.

Обычную функцию-метод можно переопределить в наследуемых классах. Однако без атрибута virtual такая функция-метод будет связана с сообщением на этапе компиляции. Атрибут virtual гарантирует позднее связывание в пределах полиморфического кластера.

Часто возникает необходимость передачи сообщений объектам, принадлежащим разным классам в иерархии. В этом случае требуется реализация механизма позднего связывания. Чтобы добиться позднего связывания для объекта, его нужно объявить как указатель или ссылку на объект соответствующего класса. Для открытых производных классов указатели и ссылки на объекты этих классов совместимыми с указателями и ссылками на объекты базового класса (т.е. к объекту производного класса можно обращаться, как будто это объект базового класса). Выбранная функция-метод зависит от класса, на объект которого указывается, но не от типа указателя.

C++ поддерживает virtual функции-методы, которые объявлены в основном классе и переопределены в порожденном классе. Иерархия классов, определенная общим наследованием, создает связанный набор типов пользователя, на которые можно ссылаться с помощью указателя базового класса. При обращении к виртуальной функции через этот указатель в C++ выбирается соответствующее функциональное определение во время выполнения. Объект, на который указывается, должен содержать в себе информацию о типе, поскольку различия между ними может быть сделано динамически. Это особенность типична для ООП кода. Каждый объект “знает” как на него должны воздействовать. Эта форма полиморфизма называется чистым полиморфизмом.

В C++ функции-методы класса с различным числом и типом параметров есть действительно различные функции, даже если они имеют одно и тоже имя. Виртуальные функции позволяют переопределять в управляемом классе функции, введенные в базовом классе, даже если число и тип аргументов то же самое. Для виртуальных функций нельзя переопределять тип функции. Если две функции с одинаковым именем будут иметь различные аргументы, C++ будет считать их различными и проигнорирует механизм виртуальных функций. Виртуальная функция обязательно метод класса.

Программирование шаблона классов. Теоретические сведения

Достаточно часто встречаются классы, объекты которых должны содержать элементы данных произвольного типа (в том смысле, что их тип определяется отдельно для каждого конкретного объекта). В качестве примера можно привести любую структуру данных (массив указателей, массив, список, дерево). Для этого в C++ предлагаются средства, позволяющие определить некоторое множество идентичных классов с

параметризованным типом внутренних элементов. Они представляют собой особого вида заготовку класса, в которой в виде параметра задан тип (класс) входящих в него внутренних элементов данных. При создании конкретного объекта необходимо дополнительно указать и конкретный тип внутренних элементов в качестве фактического параметра. Создание объекта сопровождается созданием соответствующего конкретного класса для типа, заданного в виде параметра. Принятый в C++ способ определения множества классов с параметризованным внутренним типом данных (иначе, макроопределение) называется шаблоном (template).

Синтаксис шаблона рассмотрим на примере шаблона класса векторов, содержащих динамический массив указателей на переменные заданного типа.

```
// <class T> - параметр шаблона - класс "T", внутренний тип данных

// vector -  имя группы шаблонных классов

template <class T> class vector

{

int  tsize;         // Общее количество элементов

int  csize;         // Текущее количество элементов

T    **obj;        // Массив указателей на парам. объекты типа "T"

public:

T *operator[](int); // оператор [int] возвращает указатель на

// параметризованный объект  класса "T"

void insert(T*);   // включение указателя на объект типа "T"

int  index(T*);

};
```

Данный шаблон может использоваться для порождения объектов-векторов, каждый из которых хранит объекты определенного типа. Имя класса при этом составляется из имени шаблона "vector" и имени типа данных (класса), который подставляется вместо параметра "T":

```
vector<int>   a;

vector<double>     b;

extern class  time;

vector<time>      c;
```

Заметим, что транслятором при определении каждого вектора с новым типом объектов генерируется описание нового класса по заданному шаблону (естественно, неявно в процессе трансляции). Например, для типа int транслятор получит:

```
class vector<int>

{

int  tsize;

int  csize;

int  **obj;

public:

int *operator[](int);

void insert(int*);

int  index(int*);

};
```

Далее следует очевидное утверждение, что функции- методы шаблона также должны быть параметризированы, то есть генерироваться для каждого нового типа данных. Действительно, это так: функции-методы шаблона классов в свою очередь также являются шаблонными функциями с тем же самым параметром. То же самое касается переопределяемых операторов:

```
// параметр шаблона - класс "T", внутренний тип данных

// имя функции-элемента или оператора - параметризировано

//

template <class T>  T* vector<T>::operator[](int n)

{

if (n >=tsize) return(NULL);

return (obj[n]);

}

template <class T> int vector<T>::index(T *pobj)

{

int  n;

for (n=0; n<tsize; n++)
```

```
if (pobj == obj[n]) return(n);

return(-1);

}
```

Заметим, что транслятором при определении каждого вектора с новым типом объектов генерируется набор методов- функций по заданным шаблонам (естественно, неявно в процессе трансляции). При этом сами шаблонные функции должны размещаться в том же заголовочном файле, где размещается определение шаблона самого класса. Для типа int сгенерированные транслятором функции-методы будут выглядеть так:

```
int* vector<int>::operator[](int n)

{

if (n >=tsize) return(NULL);

return (obj[n]);

}

int vector<int>::index(int *pobj)

{

int  n;

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

if (pobj == obj[n]) return(n);

return(-1);

}
```

Множественное наследование с использованием абстрактных базовых классов, файлового ввода-вывода с применением потоков C++, функций обработки исключительных ситуаций. Теоретические сведения

Абстрактные классы

Если базовый класс используется только для порождения производных классов, то виртуальные функции в базовом классе могут быть "пустыми", поскольку никогда не будут вызваны для объекта базового класса. Базовый класс в котором есть хотя бы одна такая функция, называется *абстрактным*. Виртуальные функции в определении класса обозначаются следующим образом:

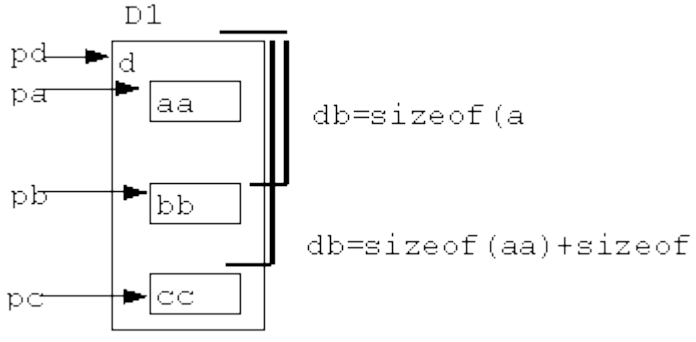
```
class base

{

public:

virtual print()=0;

virtual get() =0;

};
```

Определять тела этих функций не требуется.

Множественное наследование

Множественным наследованием называется процесс создания производного класса из двух и более базовых. В этом случае производный класс наследует данные и функции всех своих базовых предшественников. Существенным для реализации множественного наследования является то, что адреса объектов второго и последующих базовых классов не совпадают с адресом объекта производного класса. Этот факт должен учитываться транслятором при преобразовании указателя на производный класс в указатель на базовый и наоборот:

```
class d : public a,public b, public c { };

d       D1;

pd =    &D1;        // #define db sizeof(a)

pa =    pd;         // #define dc sizeof(a)+sizeof(b)

pb =    pd;         // pb = (char*)pd + db

pc =    pd;         // pc = (char*)pd + dc
```



Такое действие выполняется компилятором как явно при преобразовании в программе типов указателей, так и неявно, когда в объекте производного класса наследуется функция из второго и последующих базовых классов. Для вышеуказанного примера при определении в классе bb функции f() и ее наследовании в классе "d" вызов D1.f() будет реализован следующим образом:

```
this = &D1;                  // Указатель на объект производного класса

this = (char*)this + db      // Смещение к объекту базового класса

b::f(this);                  // Вызов функции в базовом классе
```

Механизм виртуальных функций при множественном наследовании имеет свои особенности. Во-первых, на каждый базовый класс в производном классе создается свой массив виртуальных функций (в нашем случае -для aa в d, для bb в d и для cc в d). Во-вторых, если функция базового класса переопределена в производном, то при ее вызове требуется преобразовать указатель на объект базового класса в указатель на объект производного. Для этого транслятор включает соответствующий код, корректирующий значение this в виде "заплаты", передающей управление командой перехода к переопределяемой функции, либо создает отдельные таблицы смещений.

Файловые потоки. Классы файловых потоков:

ifstream - файл ввода, производный от istream

ofstream - файл вывода, производный от ostream

fstream - файл ввода-вывода, производный от iostream

Флаги режимов работы с файлом:

```
enum ios::open_mode

{

in = 0x01,          // Открыть файл только для чтения

out =    0x02,          // Открыть файл только для записи

ate =    0x04,          // При открытии позиционироваться в конец файла

app =    0x08,          // Открыть существующий для дополнения

trunc =  0x10,     // Создание нового файла взамен существующего

nocreate=0x20,      // Не создавать новый файл при его отсутствии

noreplace=0x40,     // Не создавать новый файл, если он существует

binary=  0x80 // Двоичный файл ("прозрачный" ввод-вывод без

// преобразования символов конца строки)
```

```
};
```

Конструкторы объектов (для классов ifstream,ofstream,fstream) и функции открытия/закрытия файлов:

```
ifstream();         // Без открытия файлов

ifstream(           // С открытием файла в заданном

char *name,   // режиме imode

int imode=ios::in,

int prot=filebuf::openprot);

ifstream(int fd);  // С присоединенем файла с дескрип-

// тором fd

ifstream(     // То же, с явно заданным буфером

int fd,

char *buf, int sz);

void ifstream::open(

char *name,   // Открытие файла в заданном режиме

int imode=ios::in,

int prot=filebuf::openprot);

void close(); // Закрыть файл

void setbuf(

char *p,int sz);// Установить буфер потока

int  fd();          // Дескриптор открытого в потоке файла

int  is_rtl_open();     // 1 - файл открыт в потоке
```

Унаследованные переопределения операторов позволяют проверять наличие ошибок в потоках в виде:

```
fstream  ss;

if (ss) ...   или  if (!ss) ...
```

Обработка исключительных ситуаций

Средства обработки ошибочных ситуаций позволяют передать обработку исключений из кода, в котором возникло исключение, некоторому другому программному блоку, который выполнит в данном случае некоторые определенные действия. Таким образом, основная идея данного механизма состоит в том, что функция проекта, которая обнаружила непредвиденную ошибочную ситуацию, которую она не знает, как решить, генерирует сообщение об этом (бросок исключения). А система вызывает по этому сообщению программный модуль, который перехватит исключение и отреагирует на возникшее нештатное событие. Такой программный модуль называют «обработчик» или перехватчик исключительных ситуаций. И в случае возникновения исключения в его обработчик передаётся произвольное количество информации с контролем ее типа. Эта информация и является характеристикой возникшей нештатной ситуации.

Обработка исключений в C++ это обработка с завершением. Это означает, что исключается невозможность возобновления выполнения программы в точке возникновения исключения.

Для обеспечения работы такого механизма были введены следующие ключевые слова:

try - проба испытания;

catch - перехватить (обработать);

throw - бросать.

Кратко рассмотрим их назначение.

try - открывает блок кода, в котором может произойти ошибка; это обычный составной оператор:

```
try
{
код
};
```

Код содержит набор операций и операторов, который и будет контролироваться на возникновение ошибки. В него могут входить вызовы функции пользователя, которые компилятор так же возьмет на контроль. Среди данного набора операторов и операций обязательно указывают операцию броска исключения: **throw**.

Операция броска **throw** имеет следующий формат:

```
throw  выражение;
```

где - «выражение» определяет тип информации, которая и описывает исключение (например, конкретные типы данных).

catch - сам обработчик исключения, который перехватывает информацию:

```
catch ( тип параметр)

{

код

}
```

Через параметр обработчику передаются данные определенного типа, описывающие обрабатываемое исключение.

Код определяет те действия, которые надо выполнить при возникновении данной конкретной ситуации. В С++ используют несколько форм обработчиков. Такой обработчик получил название параметризованный специализированный перехватчик

Перехватчик должен следовать сразу же после блока контроля, т.е. между обработчиком и блоком контроля не должно быть ни одного оператора. При этом в одном блоке контроля можно вызывать исключения разных типов для разных ситуаций, поэтому обработчиков может быть несколько.

В этом случае их необходимо расположить сразу же за контролирующим блоком последовательно друг за другом.

Кроме того, запрещены переходы, как извне в обработчик, так и между обработчиками.

Можно воспользоваться универсальным или абсолютным обработчиком:

```
catch ( . . . )

{

код

}
```

где (...) - означают способность данного перехватчика обрабатывать информацию любого типа. Такой обработчик располагают последним в пакете специализированных обработчиков. Тогда, если исключение не будет перехвачено специализированными обработчиками, то буде выполнен последний - универсальный.

В случае не возникновения исключения, набор обработчиков будет обойден, т.е. проигнорирован.

Если же исключение было брошено, при возникновении критической ситуации, то будет вызван конкретный перехватчик при совпадении его параметра с выражением в операторе броска, т.е. управление будет передано найденному обработчику. После выполнения кода вызванного обработчика, управление передается оператору, который расположенный за последним перехватчиком, или проект корректно завершает работу.

Существенное отличие вызова конкретного обработчика от вызова обычной функции заключается в следующем: при возникновении исключения и передаче управления определенному обработчику, система осуществляет вызов всех деструкторов для всех объектов классов, которые были созданы с момента начала контроля и до возникновения исключительной ситуации с целью их уничтожения.

Блоки **try**, как составные блоки могут быть вложены:

```
try {
...
try
{
...
}
...
}
```

тогда, в случае возникновения исключения в некотором текущем блоке, поиск обработчика последовательно продолжается в блоках, предшествующих уровней вложенности с продолжением вызова деструкторов.

Основы теории графов

Ориентированные графы

Ориентированным графом G=(V,E) называется структурная составляющая из множества вершин E и множества дуг V, допускается что эти множества могут быть пустыми. Вершина – узел, а дуги – ребро. V=Ф, E=Ф.

Вершины *v и w н*азываются *концевыми* вершинами (или просто *концами*) ребра *e*{*v,w*}. Ребро, в свою очередь, *соединяет* эти вершины. Две концевые вершины одного и того же ребра называются *соседними*.

Пример:

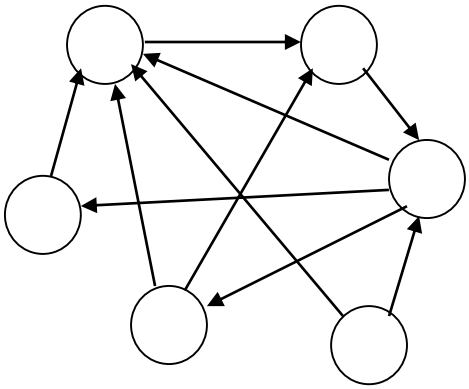


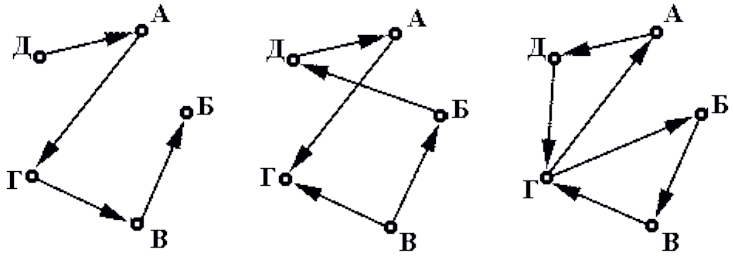
Рис 1.

С помощью графовых моделей в компьютере (математике) может быть представлена структура (практически любая) объекта предметной области, которая состоит из элементов (узлов) и их взаимодействий (направленных взаимодействий) между объектами или отношение между ними.
Пример: блок-схема.

Путем, в ориентированном графе от вершины V_1 к вершине V_n называется последовательность ориентированных ребер V_1V_2, V_2V_3, ..., $V_{n-1}V_n$, в которой конец каждого предыдущего ребра совпадает с началом следующего и каждое ребро встречается в этой последовательности только один раз.

Каждому узлу и/или дуге может быть сопоставлено некоторое значение, тогда он называется взвешенным (дуги), помеченным (узлы).
Путем в не взвешенном графе называется количество дуг, образующих путь, а во взвешенном – сумма весов (длин) дуг, образующих этот путь.
Путь называется простым, если все узлы его образующие различны, кроме первого и последнего.

На ниже приведенном рисунке показаны примеры путей в ориентированном графе. Причем, первые два пути— простые — ни одна из вершин не содержится в нем более одного раза. Третий путь не является простым, т. к. через вершину Г путь «проходил» дважды.



Представление ориентированных графов

Рассмотрим представление ориентированных графов на примере следующего орграфа:

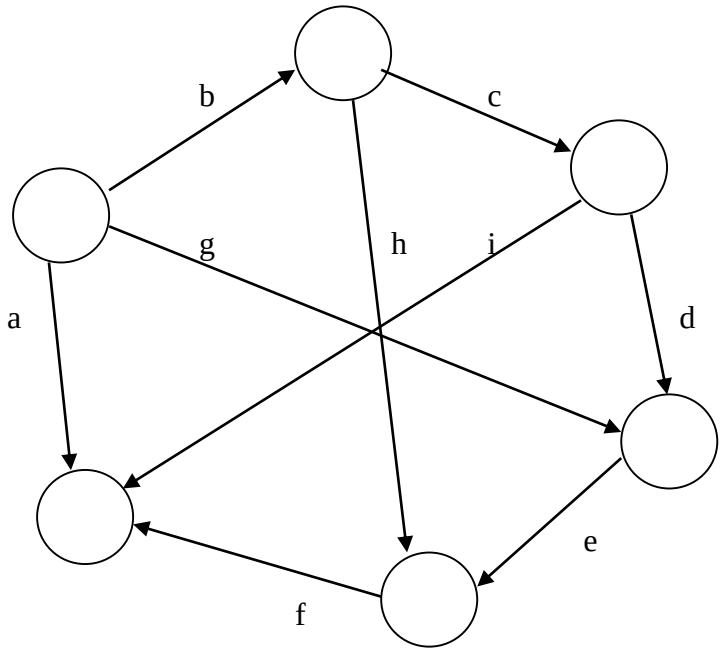


Рис 2.

Матрица смежности — таблица, где как столбцы, так и строки соответствуют вершинам графа. В каждой ячейке этой матрицы записывается число, определяющее наличие связи от вершины-строки к вершине-столбцу (либо наоборот).Недостатком является требования к памяти — очевидно, квадрат количества вершин.

a	0	0	0	0	0	1
b	1	0	0	0	0	0
c	1	0	0	0	0	1
d	1	0	0	0	1	0
e	1	1	1	0	0	0
f	0	0	0	0	1	0
	a	b	c	d	e	f

Матрица инцидентности - каждая строка соответствует определенной вершине графа, а столбцы соответствуют связям графа. В ячейку на пересечении *i*-ой строки с *j*-м столбцом матрицы записывается 1 в случае если связь *j* «выходит» из вершины *i*, −1 если связь «входит» в вершину, любое число отличное от 0,1,-1 если связь является петлей, и 0 во всех остальных случаях.

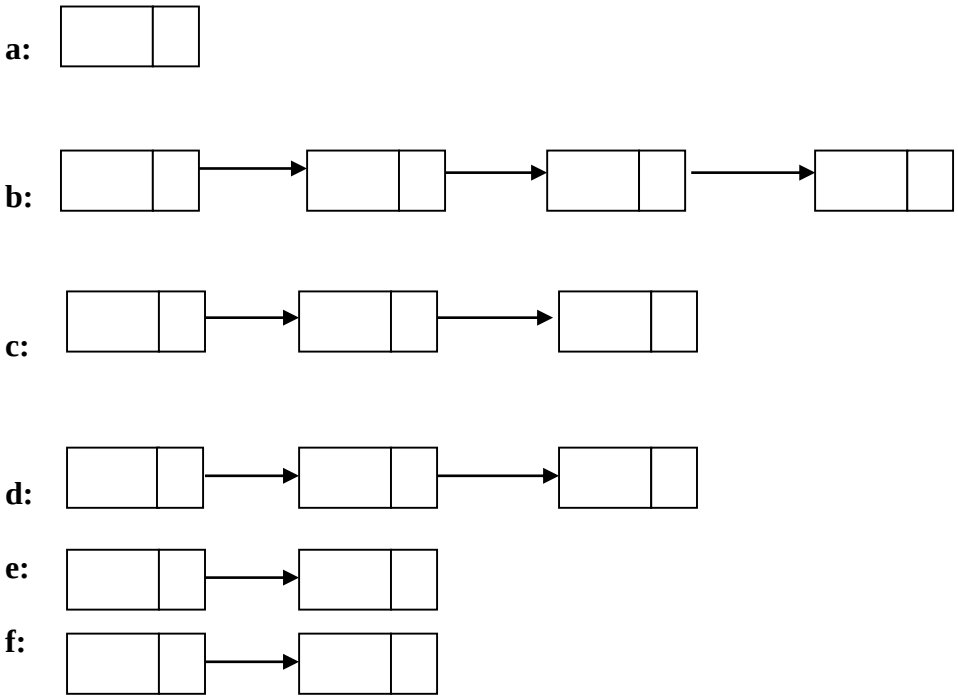
Данный способ является самым емким (размер пропорционален $|E||V|$) и неудобным для хранения, но облегчает нахождение циклов в графе

a	1	1	0	0	0	0	1	0	0
b	-1	0	0	0	0	-1	0	0	-1
c	0	0	0	0	-1	1	0	-1	0
d	0	0	0	-1	1	0	-1	0	0
e	0	0	-1	1	0	0	0	0	1
f	0	-1	1	0	0	0	0	1	0
	a	b	c	d	e	f	g	h	i

Список рёбер — это тип представления графа в памяти, подразумевающий, что каждое ребро представляется двумя числами — номерами вершин этого ребра. Список рёбер более удобен для реализации различных алгоритмов на графах по сравнению с матрицей смежности.

{a,b} {a,f} {a,d} {c,b} {d,c} {e,d} {e,b} {f,c} {f,e};

Список смежностей – содержит для каждой вершины из множества вершин V список смежных ей вершин. Каждый элемент списка смежностей является записью, содержащей данную вершину и указатель на следующую запись в списке (для последней записи в списке этот указатель – пустой). Входы в списки смежностей для каждой вершины графа хранятся в таблице (массиве)



Нахождение кратчайших путей между парами вершин

Алгори́тм Дейкстры — алгоритм на графах, изобретенный Э. Дейкстрой. Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании и технологиях, например, его использует протокол OSPF для устранения кольцевых маршрутов.

Каждой вершине из V сопоставим метку — минимальное известное расстояние от этой вершины до a. Алгоритм работает пошагово — на каждом шаге он «посещает» одну

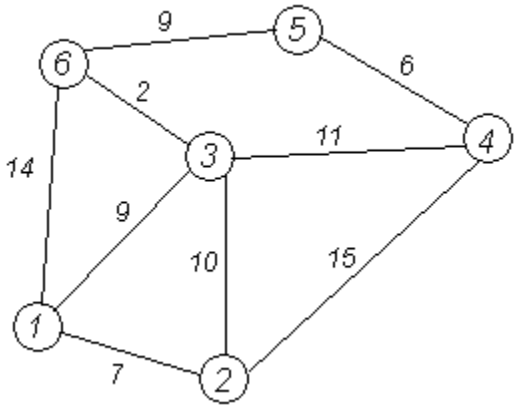
вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

Инициализация. Метка самой вершины *a* полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от *a* до других вершин пока неизвестны. Все вершины графа помечаются как непосещенные.

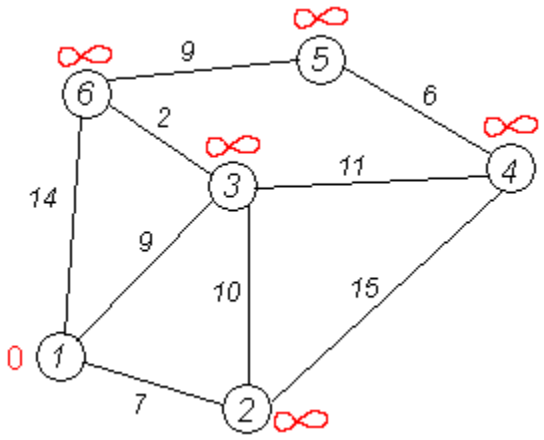
Шаг алгоритма. Если все вершины посещены, алгоритм завершается. В противном случае из еще не посещенных вершин выбирается вершина *u*, имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых *u* является предпоследним пунктом. Вершины, соединенные с вершиной *u* ребрами, назовем соседями этой вершины. Для каждого соседа рассмотрим новую длину пути, равную сумме текущей метки *u* и длины ребра, соединяющего *u* с этим соседом. Если полученная длина меньше метки соседа, заменим метку этой длиной. Рассмотрев всех соседей, пометим вершину *u* как посещенную и повторим шаг.

Пример

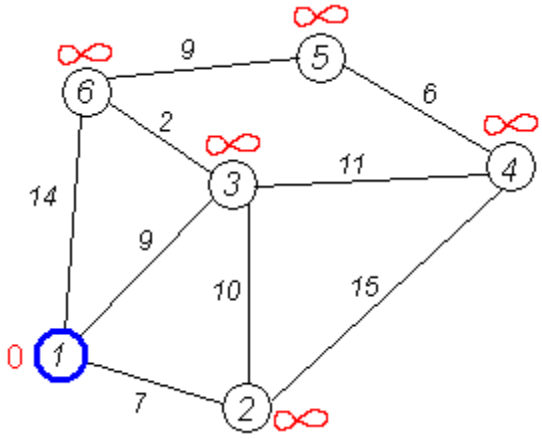
Рассмотрим выполнение алгоритма на примере графа, показанного на рисунке. Пусть требуется найти расстояния от 1-й вершины до всех остальных.



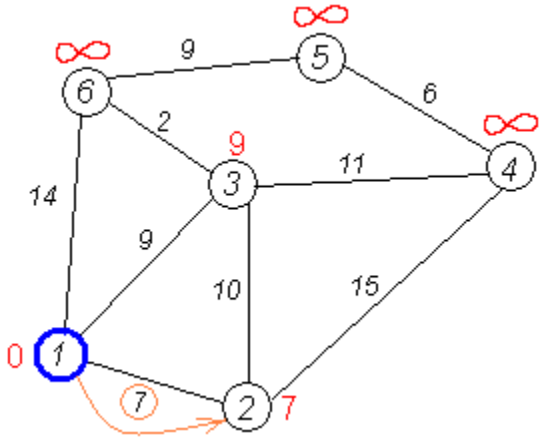
Кружками обозначены вершины, линиями — пути между ними (ребра графа). В кружках обозначены номера вершин, над ребрами обозначена их «цена» — длина пути. Рядом с каждой вершиной красным обозначена метка — длина кратчайшего пути в эту вершину из вершины 1.



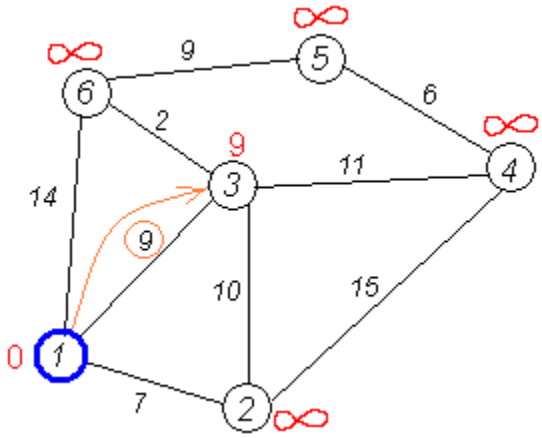
Первый шаг. Рассмотрим шаг алгоритма Дейкстры для нашего примера. Минимальную метку имеет вершина 1. Ее соседями являются вершины 2, 3 и 6.

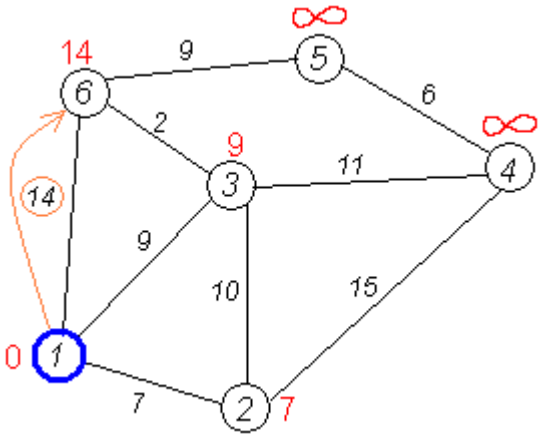


Первый по очереди сосед вершины 1 — вершина 2, потому что длина пути до нее минимальна. Длина пути в нее через вершину 1 равна кратчайшему расстоянию до вершины 1 + длина ребра, идущего из 1 в 2, то есть 0 + 7 = 7. Это меньше текущей метки вершины 2, поэтому новая метка 2-й вершины равна 7.



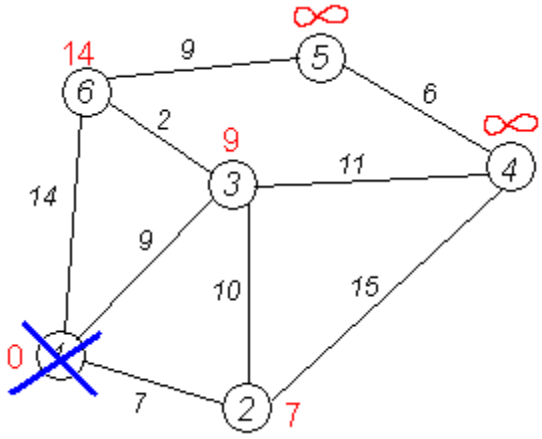
Аналогичную операцию проделываем с двумя другими соседями 1-й вершины — 3-й и 6-й.



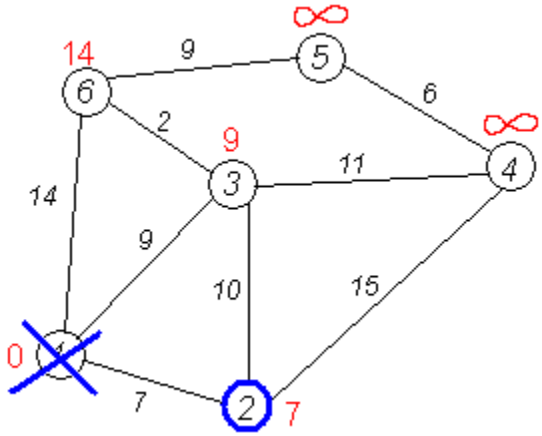


Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и обсуждению не подлежит (то, что это действительно так,

впервые доказал Дейкстра). Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.



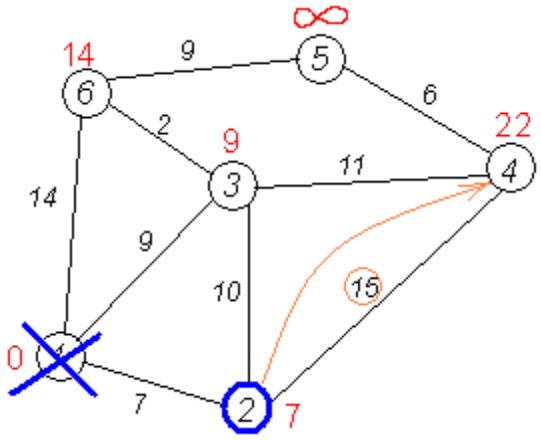
Второй шаг'. Шаг алгоритма повторяется. Снова находим «ближайшую» из непосещенных вершин. Это вершина 2 с меткой 7.



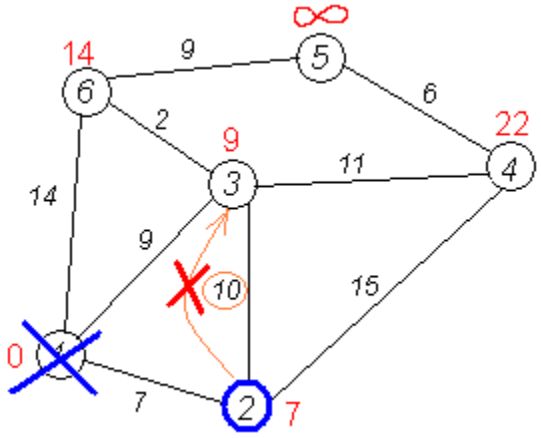
Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю. Соседями вершины 2 являются 1, 3, 4.

Первый (по порядку) сосед вершины 2 — вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем.

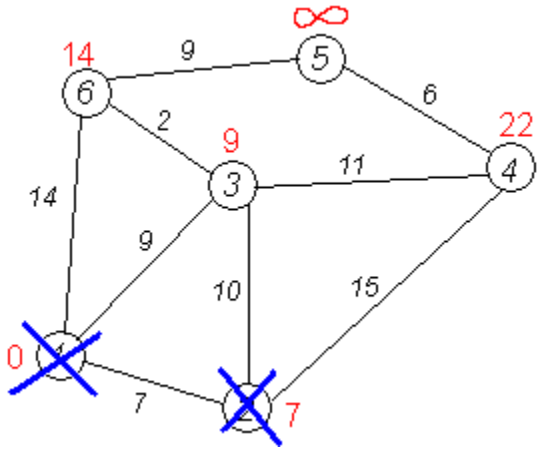
Следующий сосед вершины 2 — вершина 4. Если идти в неё через 2-ю, то длина такого пути будет = кратчайшее расстояние до 2 + расстояние между вершинами 2 и 4 = 7 + 15 = 22. Поскольку 22<∞, устанавливаем метку вершины 4 равной 22.



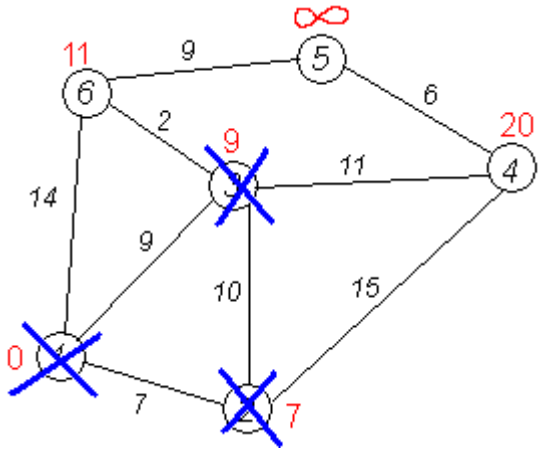
Ещё один сосед вершины 2 — вершина 3. Если идти в неё через 2, то длина такого пути будет = 7 + 10 = 17. Но текущая метка третьей вершины равна 9<17, поэтому метка не меняется.



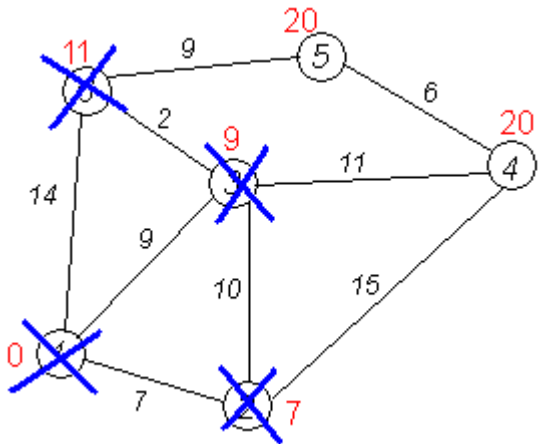
Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем ее как посещенную.

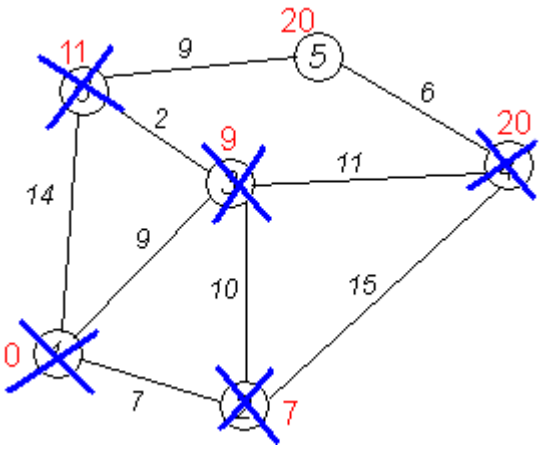


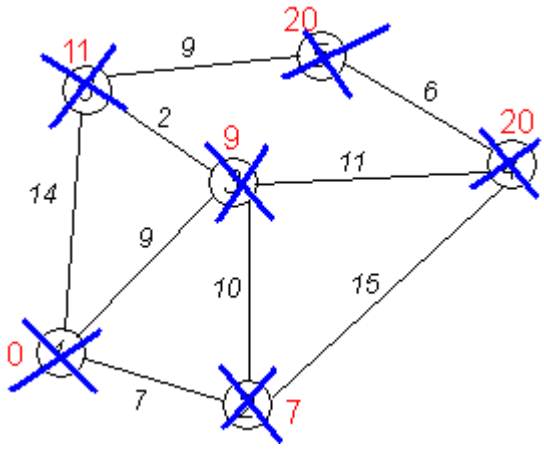
Третий шаг. Повторяем шаг алгоритма, выбрав вершину 3. После ее «обработки» получим такие результаты:



Дальнейшие шаги. Повторяем шаг алгоритма для оставшихся вершин (Это будут по порядку 6, 4 и 5).







Завершение выполнения алгоритма. Алгоритм заканчивает работу, когда вычеркнуты все вершины. Результат его работы виден на последнем рисунке: кратчайший путь от вершины 1 до 2-й составляет 7, до 3-й — 9, до 4-й — 20, до 5-й — 20, до 6-й — 11.

Литинг:

```
program Project1;
   {$APPTYPE CONSOLE}
   uses  SysUtils;
   const MAX_SIZE = 7;
   type Matrix = record
   Len:array[1..MAX_SIZE, 1..MAX_SIZE] of integer; //матрица
```

```
  Lab:array[1..MAX_SIZE] of integer;  //длина пути
  Vis:array[1..MAX_SIZE, 1..MAX_SIZE] of boolean;  //посещенные
  end;
  var MatrixSize ,i ,j, _i:integer;
  MM:Matrix;
  begin
 { TODO -oUser -cConsole Main : Insert code here }
  writeln('enter matrix size');
  readln(MatrixSize);
  writeln('enter matrix, "999" is unavaliable len');
  for i:=1 to MatrixSize do
       for j:=1 to MatrixSize do
               readln(MM.Len[i,j]);
  writeln('Your matrix:');
     for i:=1 to MatrixSize do begin
       for j:=1 to MatrixSize do
               write(MM.Len[i,j], '   ');
       writeln;
       end;
  readln;
  //init
  for i:=1 to MatrixSize do
  begin
  for j:=1 to MatrixSize do
  MM.Vis[i,j]:=false;
  MM.Lab[i]:=999;
  end;
  MM.Lab[1]:=0;
  //выполнение  алгоритма
  for i:=1 to MatrixSize do begin
  for j:=1 to MatrixSize do begin
  if MM.Vis[i,j]=false then
  if ((MM.Len[i,j]<>999) and (MM.Len[i,j]<>0)) then
  begin
  if MM.Len[i,j]+MM.Lab[i]<MM.Lab[j] then
  begin
    MM.Lab[j]:= MM.Len[i,j]+MM.Lab[i];
    MM.Vis[i,j]:=true;
    for _i:=1 to MatrixSize do
       write(MM.Lab[_i], '   ');
       writeln;
  end;
  end else  MM.Vis[i,j]:=true;
  end;
  end;
  writeln('//----------');
  for i:=1 to MatrixSize do
  write(MM.Lab[i], '   ');
  readln;
  end.
```

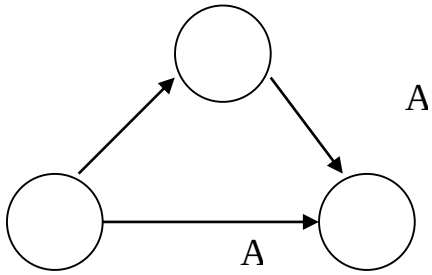
Алгоритм Дейкстры можно адаптировать для решения задач нахождения кратчайшего расстояния между всеми парами вершин в орграфе. Для этого достаточно последовательно каждую вершину выбрать в качестве источника, и для неё пересчитать алгоритмом Дейкстра пути.

Однако существует другой способ получения получения кратч. расстояний.

Метод Флойда или метод Флойда-Уоршелла (Warshell).

Пусть вершины пронумерованы от 1 до N.
И определена матрица C от 1 до N
Будем считать, что отсутствующая дуга обозначается +бесконечность. (т.е. как бы пусть есть, но он крайне невыгоден). На каждом шаге пересчитывается матрица A[i,j] A(NxN) , где элемент a[i,j] обозначает расстояние между i-м и j-м узлом. Чтобы знать, для какого шага пересчета матрицы A в ней хранится текущее значение, то значение матрицы A на k-м шаге обозначается через A_k[i,j]

На каждой k-й итерации k=1..n в алгоритме Флойда происходит попытка добавления в маршрут между узлом i и j пути через k-й узел , тогда узел K в маршрут между i и j узлом , если выполняется
условия A[j,k] + A[k,j] < A[i,j], и при этом запоминается новое меньшее расстояние.



Таким образом,значение ячейки A[i,j] на K-ом шаге представляет собой длину кратчайшего расстоянию между i j вершиной пути, построенного из узлов с номерами, не превосходящими K, построенного из узлов с номерами, не превосходящими K.
Каждой дуге поставлена неотрицательная стоимость.
Используется матрица C, C[i,j]>0;

инициализация массива

```
for k:=1 to n do
   for j:=1 to n do
      for i:=1 to n do
          if A[j,k] != A[k,j] < A[i,j] then
             begin A[i,j]:=A[i,k] + A[K,j];
              A [i,j] = C[i,j]
                P[i,j] = 0;


          end;
```

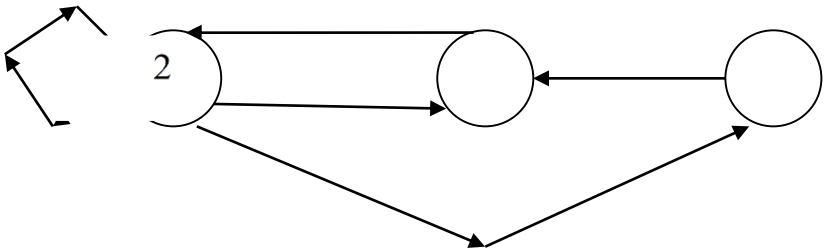
Псевдокод:

На каждом шаге алгоритм генерирует двухмерную матрицу `W`, `W[i][j]`=d_{ij}^{n}. Матрица `W` содержит длины кратчайших путей между всеми вершинами графа. Перед работой алгоритма матрица `W` заполняется длинами рёбер графа.

```
for k = 1 to n
  for i = 1 to n
    for j = 1 to n
```

```
W[i][j] = min(W[i][j], W[i][k] + W[k][j])
```

Пример:



A_0= 0 8 5
3 0 ∞
∞ 2 0

K=1

A[1,1] + A[1,1] < A[1,1] - нет
I=1 *A[1,1]* + A[1,2] < A[1,2] - нет
A[1,1] + A[1,3] < A[1,3] - нет

A[2.1] + *A[1,1]* < A[2,1] - нет
I=2 A[2,1] + A[1,2] < A[2,2] - нет
A[2,1] + A[1,3] < A[2,3] – да → A[2,3] = 8

I=3 A[3,1] + *A[1,1]* < A[3,1] – нет

Заметим, что на K-ом шаге могут измениться только ячейки A[i,j], которые имеют такие кроме i = k или j=k или j=i

K=3:

0	**8**	**5**
3	**0**	∞
∞	**2**	**0**

Вывод на печать кратчайших путей

Вывод на печать кратчайших путей

P[i,j] = K
Для реконструкции пути из i в j, длина которого есть A[i,j] введем массив P[i,j], значение элементов которого есть тот узел, через который надо пройти, чтобы попасть из i в j, а точнее до которого надо рекурсивно реконструировать путь.

Если P[i,j] = 0, то это значит что в пути задействована дуга из i в j
P[i.j] = K

Чтобы реконструировать путь, необходимо:

```
Procedure roconst(i,j) begin
If  P[i,j] <> 0 then begin
    Reconst(i,P[i,j]);
      Writeln(P[i,j]);
    Reconst(P[i,j],j);
End;

Вызов
 For i:=1 to N do
   For j:=1 to N do
     Reconst(i,j);
```

Построение транзитивного замыкания ориентированного графа, алгоритм Warshell

Алгоритм Warshell построения транзитивного замыкания ориентированного графа

В ряде задач необходимо определить, существует ли какой-либо путь из i в j, т.е. решить задач о достижимости j-го узла из узла i. Для этого строится матрица транзитивного замыкания, алгоритм построения которой предложил Warshell до Флойда.

Пусть дана матрица смежности графа. Каждый элемент либо 0 либо 1, true-false, тогда после K итераций (K..N) элемент матрицы A будет содержать единицу (true) если путь из i в j существует по следщ формуле

```
 For k:=1 to N do
    For i:=1 to N do
       For j:=1 to N do
          A[i,j] = A [i,j]  or  A[i,k]   and  A[k,j];
            1          2                      3
```

Эту формулу можно трактовать следующим образом: путь из i в j будет существовать (1) , если он уже существует к данному моменту (2) или (3) он может быть построен через узел K

Код программы для вычисления транзитивного замыкания

```
Procedure Warshall (var A:array[1..n][1..n] of boolean; C:array[1..n][1..n] of boolean);
Var i,j,k: integer;
Begin
        For i:=1 to n do
                For j:=1 to n do
                        A[i,j]:=C[i,j];
        For k:=1 to n do
                For i:=1 to n do
                        For j:=1 to n do
                                If  A[i,j]=false then A[i,j]:=A[i,k] and A[k,j];
```

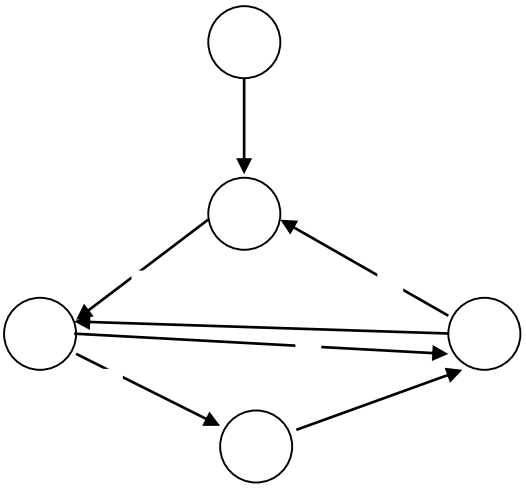
End;

Нахождение центра орграфа

Пусть необходимо найти центр орграфа.

Для определения понятия центральной вершины орграфа вводится понятие эксцентриситет (или максимальное удаление вершины v) определяется как максимум на множестве wэV из минимальных длин под вершины w к v.
Эксцентриситет (Ecc, Ex).
Тогда центром орграфа называет вершина с минимальным эксцентриситетом.
Центром орграфа является вершина, для которой максимальное расстояние (длина пути) до других вершин минимально.

	a	B	c	d	e
A	∞	1	∞	∞	∞
B	∞	∞	2	∞	∞
C	∞	∞	∞	2	4
D	∞	1	3	∞	∞
E	∞	∞	∞	5	∞



	a	B	c	d	e
A	∞	1	3	5	∞
B	∞	0	2	4	6
C	∞	3	0	2	4
D	∞	1	3	0	7
E	∞	6	8	5	0
max	∞	6	8	5	7

A[i,j] = i → j;

определение эксцентриситета каждой вершины
Ecc(a) = ∞
Ecc(b) = 6
Ecc(e) = 7

Обход ориентированных графов

При решении многих задач, касающихся графов, возникает необходимость обеспечить систематический обход вершин графа, что обеспечит посещение каждого узла для максимальной эффективности .

Различают два способа обхода:

1. поиск в ширину
2. поиск в глубину

1. Поиск в ширину.

1. Добавляет начальную вершину в очередь и помечает её как использованную.
2. Извлекает следующую вершину из очереди и добавляет в очередь смежные ей неиспользованные вершины, помечая их как использованные.
3. Если очередь непуста, переходит к пункту 2.

На каждой следующей итерации своей работы он расползается вширь по ребрам от вершин, до которых он дошёл к данной итерации. И так расползается он до тех пор, пока не побывает в каждой вершине текущей компоненты связности. Если мы представим, что рёбра есть трубы с одинаковой шириной, то подобное "расползание" очень напоминает распространение воды по трубам если начать эту самую воду равномерно заливать в начальную вершину.

Чаще всего поиск в ширину используется для нахождения кратчайшего пути от одной вершины до другой (почему он, собственно, и включён в данную главу), поэтому приведённая его реализация решает именно эту задачу. Путём несложных модификаций его можно изменить для решения других указанных задач.

Разобьём добавляемые в очередь вершины на несколько групп. Начальная вершина пусть находится в нулевой группе. Все вершины, добавленные при рассмотрении рёбер, исходящих из неё пусть образуют группу с номером 1. Все вершины, добавленные при рассмотрении рёбер, исходящих из вершин с номером группы **L** пусть образуют группу с номером **L + 1**. Несложно заметить, что в этом случае номер группы, в которой оказалась та или иная вершина, равен длине кратчайшего пути от начальной вершины до неё.

Писать поиск в ширину, как и большинство других алгоритмов, лучше для графа, заданного списком рёбер. В этом случае алгоритм более мобилен (это важно при модификациях) и даёт оптимальное время работы. Для его реализации помимо списка рёбер (**R**) и указателей списка (**G**) нам понадобится очередь **F: array[1..MaxN] of Integer** и массив меток **Use: array[1..MaxN] of Integer**. Заметим, что в данном алгоритме нам не важен вес ребра, поэтому из типа **TEdge** можно (и даже нужно для экономии памяти) исключить запись **C**. Ниже приведён фрагмент программы, выполняющий пункты 1-3 схемы действия алгоитма (здесь **I: Integer**, **X** - номер стартовой вершины, **First, Last** - указатели очереди).

```
1  For I := 1 to N do Use[I] := - 1;
2  Use[X] := 0;
3  First := 1;
4  Last := 1;
5  F[1] := X;
6  While First <= Last do Begin
```

```
 7     For I := G[F[First]] to G[F[First] + 1] - 1 do
 8       If Use[R[I].B] < 0 Then Begin
 9         Inc(Last);
10         F[Last] := R[I].B;
11         Use[R[I].B] := Use[R[I].A] + 1;
12       End;
13     Inc(First);
14   End;
```

В приведённом примере в элементе **Use[I]** хранится номер группы вершины **I**. Изначально номер группы всех вершин кроме стартовой равен -1 (строка 1), это значит, что они ещё не были использованы, а номер группы стартовой равен нулю (строка 2), т.к. она по определению лежит в нулевой группе.

Рассмотрим теперь подробнее работу приведённого фрагмента программы. Инициализация очереди, т.е. добавление в неё начальной вершины с номером **X** происходит в строках 3-5. В строке 5 мы помечаем эту вершину как использованную. В строке 6 запускается условный цикл, работающий пока очередь не пуста. В строке 7 просматриваются все рёбра, исходящие из первой вершины очереди. В строке 8 проверяется, использована ли вершина, в которую ведёт выбранное ребро, и если нет, то она добавляется в очередь: в строке 9 сдвигается указатель конца очереди; в строке 10 помещается в последний элемент очереди вершина, в которую ведёт выбранное ребро; в строке 11 номеру группы новой вершины присваивается значение, на единицу большее номера группы первой вершины очереди, т.к. новая вершины была добавлена при использовании ребра, исходящего из первой вершины очереди. В строке 13 сдвигается указатель начала очереди.

2. Поиск в глубину. Пусть задан граф $G = (V,E)$, где V — множество вершин графа, E — множество ребер графа. Предположим, что в начальный момент времени все вершины графа окрашены в *белый* цвет. Выполним следующие действия:

1. Из множества всех *белых* вершин выберем любую вершину, обозначим её v_1.
2. Выполняем для нее процедуру `DFS(`v_1`)`.
3. Перекрашиваем ее в *черный* цвет.
4. Повторяем шаги 1-3 до тех пор, пока множество *белых* вершин не пусто.

Процедура DFS (параметр — вершина $u \in V$)

1. Перекрашиваем вершину *u* в *серый* цвет.
2. Для всякой вершины *w*, смежной с вершиной u, выполняем следующие два шага:
    1. Если вершина *w* окрашена в белый цвет, выполняем процедуру `DFS(w)`.
    2. Окрашиваем *w* в *черный* цвет.

Иными словами:

При решении многих задач, касающихся ориентированных графов, необходим эффективный метод систематического обхода вершин и дуг орграфов. Таким методом

является метод поиск в глубину. Метод поиска в глубину является основой многих эффективных алгоритмов работы с графами. Предположим, что есть ориентированный граф G, в котором первоначально все вершины помечены меткой "$unvisited$". Поиск в глубину начинается с выбора начальной вершины v орграфа G, для этой вершины метка "$unvisited$" меняется на метку "$visited$". Затем для каждой вершины, смежной с вершиной v и не посещаемой раньше, рекурсивно применяется поиск в глубину. Когда все вершины, которых можно достичь из вершины v, будут рассмотрены, поиск заканчивается. Если некоторые вершины остались не посещенными, то выбирается одна из них и алгоритм повторяется. Этот процесс продолжается до тех пор, пока не будут обойдены все вершины орграфа G.

Метод получил свое название - поиск в глубину, поскольку поиск не посещенных вершин идет в направлении вглубь до тех пор, пока это возможно. Например, пусть x является последней посещенной нами вершиной. Выбираем очередную дугу (x,y)(ребро), выходящую из вершины x. Возможна следующая альтернатива: вершина y помечена меткой "$unvisited$"; вершина y помечена меткой "$visited$". Если вершина y уже посещалась, то отыскивается другая вершина, смежная с вершиной x; иначе вершина y метится меткой "$visited$" и поиск начинается заново от вершины y. Пройдя все пути, которые начинаются в вершине y, возвращаемся в вершину x, то есть в ту вершину, из которой впервые была достигнута вершина y. Затем процесс повторяется, то есть продолжается выбор нерассмотренных дуг, исходящих из вершины x, и так до тех пор, пока не будут исчерпаны все эти дуги.

Листинг:

```
Procedure dfs(v:вершина);
Var     w: вершина;
        i:integer;
Begin
        Mark[v]:=’vizited’;
        For i:=w to L[v] do
                If mark[i]=’unvizited’ then dfs[v];
End;
```

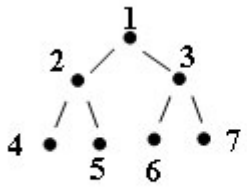
Или можно воспользоваться несколько другим подходом:

Описание функции:

```
while (Имеется хотя бы одна непосещенная вершина)
{
 Пусть p - первая из непосещенных вершин.
 Посетить вершину p и поместить ее в пустой стек S;
 while ( Стек S непуст )
 {
  Пусть p - вершина, находящаяся на верхушке стека S;
  if (У вершины p есть непосещенные смежные вершины)
    { Пусть q - первая непосещенная вершина из вершин, смежных
      вершине p. Пройти по ребру (p,q), посетить вершину q и
      поместить ее в стек S }
  else Удалить вершину p из стека S
```

```
  }
}
```

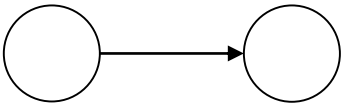
Пример работы алгоритма:



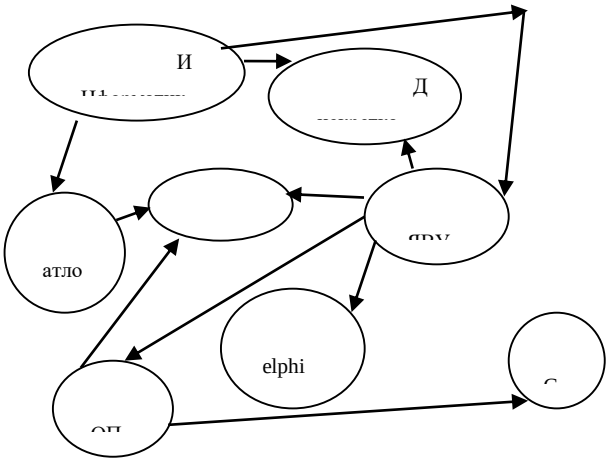

```
(
(1 .(2 3))
(2 .(4 5))
(3 .(6 7))
(4 .( ) )
(5 .( ) )
(6 .( ) )
(7 .( ) )
)
```

Топологическая сортировка

Пусть граф ациклицен и ориентирован, причем дуга из V в W означает что узел V должен быть посещен раньше узла W, т.е. V предшествует W, например выполнение некоторых работ при их планировании.



топологической сортировкой называется – процесс линейного упорядочивания вершин ациклического графа таким образом, что если существует дуга из V в W, то в упорядоченном списке вершин дуга V должна предшествовать W



Информатика, Матлогика,
Посещая узел, стираем исходящие дуги. Затем берем также те, в которые нет входящих узлов.

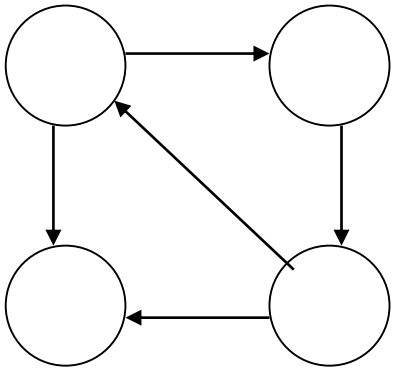
1. ищется узел, у которого отсутствуют исходящие дуги (поскольку граф ацикличен, то такой узел обязан быть). Данный узел помечаем как посещенный или удаляем из орграфа, а также удаляем все входящие в него дуги. И помещаем этот узел в стек.
2. повторяем шаг первый, пока есть непосещенный узлы
3. очередность выборки элементов из стека соответствует результату топологической сортировки.

Сильная связанность

Сильная связность

Сильносвязанной компонентой орграфа называют максимальное множество вершин, в котором существует пути из любой вершины в любую другую вершину этого множества.
Метод поиска в глубину можно использовать для нахождения сильносвязанных компонент орграфа.
Пусть G=(V,E) – ориентированный граф Множество вершин V разбивается на классы эквивалентности V_i так, что вершины v и w будут эквиваленты тогда, и только тогда, когда существуют пути v → w и w → v.
Пусть E_i множество дуг, начало и концы которых принадлежат множество V_i, тогда $G_i=(V_i,E_i)$ называется сильносвязанной компонентой графа G.
Граф, состоящий из одной сильносвязанной компоненты – называется сильносвязанным графом. Заметим, что ССК G_i может состоять и из одной вершины.
Дуги, начало и концы которых принадлежат разным классам эквивалентности образуют редуцированный орграф, узлами которого являются классы эквивалентности, т.е. G_i
Вершинами приведенного, или редуцированного графа являются ССК, а дуга в редуцированном графе существуют тогда и только тогда, когда существует хотя бы одна какая либо дуга графа G, ведущая от какого либо узла одной сильносвязанной компоненты к какому-нибудь узлу другой ССК.
Редуцированный граф всегда является ациклическим (в противном случае он был бы одним классом эквивалентности).

Например

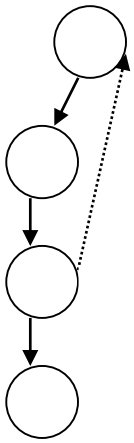


ССК здесь: d и abc

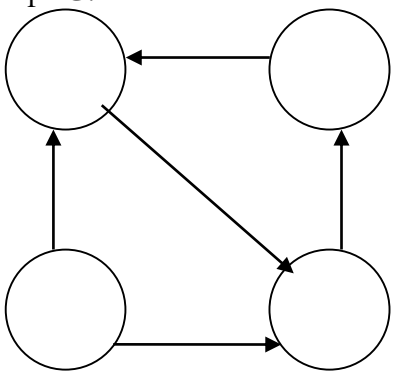
Алгоритмы нахождения сильносвязанной компоненты

Алгоритм нахождения ССК.

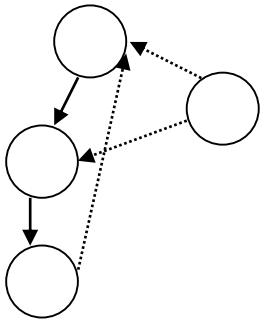
1. Выполняем поиск в глубину на графе G, при этом нумеруем каждую посещенную вершину после обхода всех вершин, которые доступны из текущей, т.е.



конструируется новый граф G’ путем обращения (изменения напр. Дуг на противоположные) Всех дуг графа G.



2. выполняется обход в глубину на графе G’, начиная с вершины с наибольшим номером



3. каждое дерево в глубинном лесу третьего шага соответствует отдельной ССК графа G (точнее, узлы, образующие дерево, соответств. ССК)

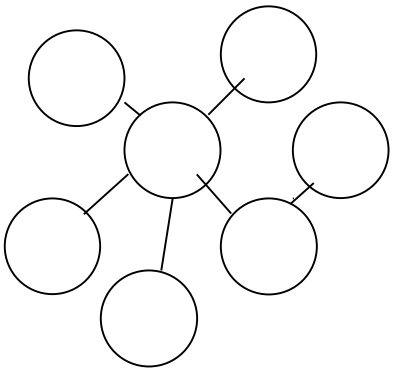
Неориентированные графы

Неориентированные графы
Неориентированный граф G=(V,E) состоит из множества узлов V и множество дуг (v,w) принадлежащих E, где v,w- неупорядоченная пара вершин. (v,w)=(w,v)
Определение: аналогично орграфу. Дополнение: если для вершин v_1 и v_n существует путь v_1 v_2 v_n – то такие вершины называются связанными. Граф называется связным, если любая пара вершин – связная.

Пусть дан граф G=(V,E) с множеством вершин V и ребер E. Граф G’ = (V’,E’) называется подграфом графа G – если множество вершин V’ есть подмножество вершин V, а множество E’ образуется множеством {(v,w):v э V’ и w э V}, при этом E’ также

подмножество E. Если множество E’ состоит из всех ребер v→w множества E, таких что обе вершины v и w принадлежат V’ , то в этом случае G’ называется индуцированным подграфом графа G. ССК графа G называется максимальный связный индуцированный подграф графа G.

Циклом на неорграфе называется путь (простой) длины не менее 3 от какой либо вершины к самой себе. Граф называется циклическим, если он содержит хотя бы один цикл. Связный ациклицеский граф, представляющий собой дерево без корня называется свободным деревом



Если узлов n, то дуг n-1.
Свободные деревья имеют два свойства.

1. Свободное дерево с n узлами имеет ровно n-1 дугу
2. Если в свободное дерево добавить хотя бы одну дугу, то образуется цикл.

Представление неориентированного графа

Представление неориентированных графов.
Для представления неорграфов используются те же самые методы, что для орграфов (матрицы смежности, список смежности, матрица инцинденции и список дуг, в которых каждая неориент. Дуга представляется парой ориентированных)

Остовные деревья минимальной стоимости (ОДМС)

Остовным деревом графа G=(V,E), в котором каждое ребро v,w помечено стоимостью c(v,w) называется свободное дерево, содержащее все вершины V графа G. Стоимостью остовного дерева называется сумма стоимостей всех ребер, входящих в остовное дерево.

Типичное применение остовных деревьев минимальной стоимости – задача оптимизации прокладки коммуникаций (например, компьютерных сетей.)
Таким образом, задача построения остовного дерева заключается в нахождения такого подмножества ребер исходного графа, чтобы их суммарная стоимость была минимальна при соблюдении требования связности графа и отсутствия цикла.
Существует два основных алгоритма построения ОДМС: алгоритм Прима и алгоритм Крускала

Алгоритм Прима, Алгоритм Крускала

Алгоритм Прима.
В этом алгоритме строится множество U (вершин), из которого вырастает остовное дерево
Сначала множество U пустое, и в начальный момент времени в него помещается произвольная вершина – т.к. рано или поздно в ОДМС войти все вершины.
На каждом шаге выбирается такое ребро с наименьшей ценой, чтобы один инциндентный ей узел принадлежал множеству U, а другой €V\U ; после чего второй конец выбранного ребра также помещается в U. Этот процесс повторяется до тех пор, пока V \ U = \

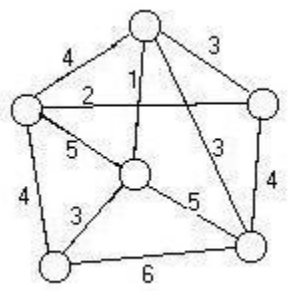
Пример:

1. U { 4 }
2. min { (4,1)=5;(4,3)=5;(4,6) } = 4,6
3. U={ 4 } + { 6 }
4. U ≠ V
5. min { (4,1)=5; (4,3)=5; (6,3)=4; (6,5)=6 }
6. U={4,6,3}
7. U ≠ V
8. min { (4,1); (6,5); (3,5); (3,2) }
9. U = { 4,6,3,1}
10. U ≠ V
11. min { (6,5),(3,5),(3,2),(1,2)}
12. U = {4,6,3,1,2}

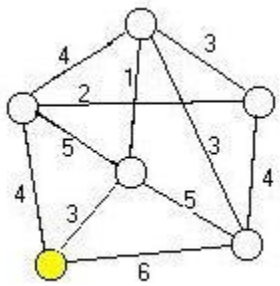
Листинг

```
procedure алгоритмПрима(V,E,w,r)
begin
    Vt := r;
    d[r] := 0;
    for всех вершин v из (V\Vt) do
        if дуга (r,v) существует
              d[v] := w(r,v);
           else
              d[v] := бесконечность;
    while (Vt != V) do
    begin
        найти вершину u такую, что
        d[u] = min{d[v],v из (V\Vt)};
        Vt := Vt + {u};
        for всех v из (V\Vt) do
              d[v] := min{d[v],w(u,v)};
    endwhile;
end;
```

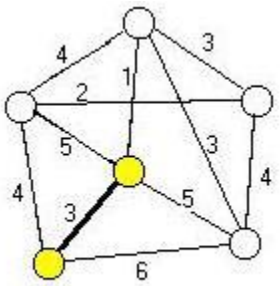
Приведем пример работы алгоритма на простом примере :



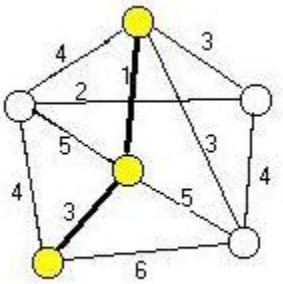
Начало работы алгоритма.
Остовное дерево пусто.



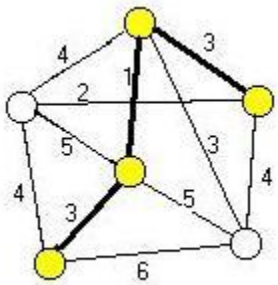
Выбрана произвольная вершина-
корень будущего дерева



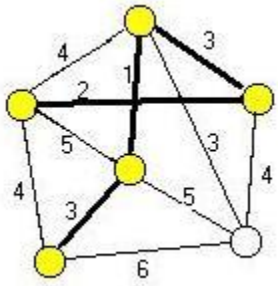
Множество расстояний от дерева до незанятых вершин: {3,4,6}.
Минимальное расстояние - 3. Добавлена новая вершина.



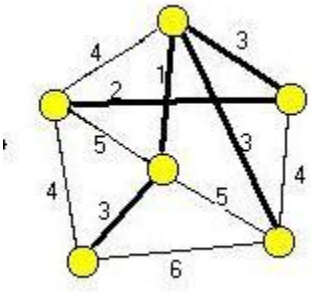
Множество расстояний от дерева до незанятых вершин:
{4,6,5,1,5}.
Минимальное расстояние - 1. Добавлена новая вершина.



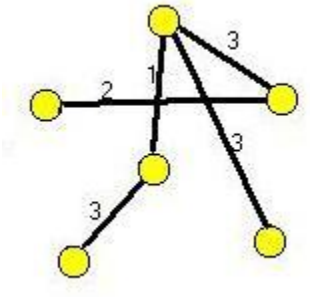
Множество расстояний от дерева до незанятых вершин:
{4,6,5,5,4,3,3}.
Минимальное расстояние - 3. Добавлена новая вершина.



Множество расстояний от дерева до незанятых вершин:
{4,6,5,5,4,3,2,4}.
Минимальное расстояние - 2. Добавлена новая вершина.



Множество расстояний от дерева до незанятых вершин:
{4,6,5,5,4,4,3}.
Минимальное расстояние - 3. Добавлена новая вершина.



Остовное дерево минимального веса построено.
Его вес - 12.

Алгоритм Крускала

В отличие от Прима в алгоритме Крускала граф, соответствующий ОДМС уже имеет все вершины. И на каждом шаге из всего еще не используемого множества дуг выбирается дуга с наименьшей стоимостью если оно не образует цикл, то она добавляется в множество дуг ОДМС, причем, если изначально множество дуг упорядочено по возрастанию цены, то такой алгоритм будет иметь трудоемкость равную количеству дуг графа G, помноженное на трудоемкость проверки ацикличности графа.

Список дуг:
(1,3)-1
(4,6)-2
(2,5)-3
(3,6)-4
(1,4)-5
(3,4)-5
(2,3)-5
(1,2),(3,5),(5,6)-6
0.
1. пока не конец спика дуг
проверка на ацикличность графа ОДМС при включении в него очередной дуги

Листинг:

```
procedure Craskal;
 // Алгоритм Краскала
 var i,j,q,m,t: integer;
 h: TEdges;
 Link: array of integer;// № связности для вершинθ
 begin
 ClearVisit; Stack_Init(Stack);
 N:=Length(Node); m:=0;θ
 for i:=0 to N-1 do // формируем массив всех ребер
```

```
 with Node[i] do begin
 LL:=Length(Edge);
 for j:=0 to LL-1 do begin
 Inc(m); SetLength(Edges,m);
 Edges[m-1].n1:=i; Edges[m-1].n2:=Edge[j].NumNode;
 Edges[m-1].A:=Edge[j].A;
 end;
 end;
 for i:=0 to m-2 do // Сортируем ребра графа по возрастанию весов
 for j:=m-1 downto i+1 do
 if Edges[j].A
 h:=Edges[j]; Edges[j]:=Edges[j-1]; Edges[j-1]:=h;
 end;
 SetLength(Link,N); // Вначале все ребра в разных компонентахθ
 // связности
 for i:=0 to N-1 do Link[i]:=i;
 i:=0; q:=N-1;
 while (iθ<=m-1) and (q<>0) do begin
 // если вершины в разных компонентах связности
 if Link[Edges[i].n1]θ<>Link[Edges[i].n2] then begin
 t:=Edges[i].n2; Push(Stack,i); // поместить в стек № ребра
 for j:=0 to N-1 do
 if Link[j]=t thenθ
 Link[j]:=Link[Edges[i].n1];// в один компонент связности
 q:=q-1;
 end;
 Inc(i);
 end;
 SetColorEdge; // закраска ребер
 SetLength(Edges,0);
 end;
```

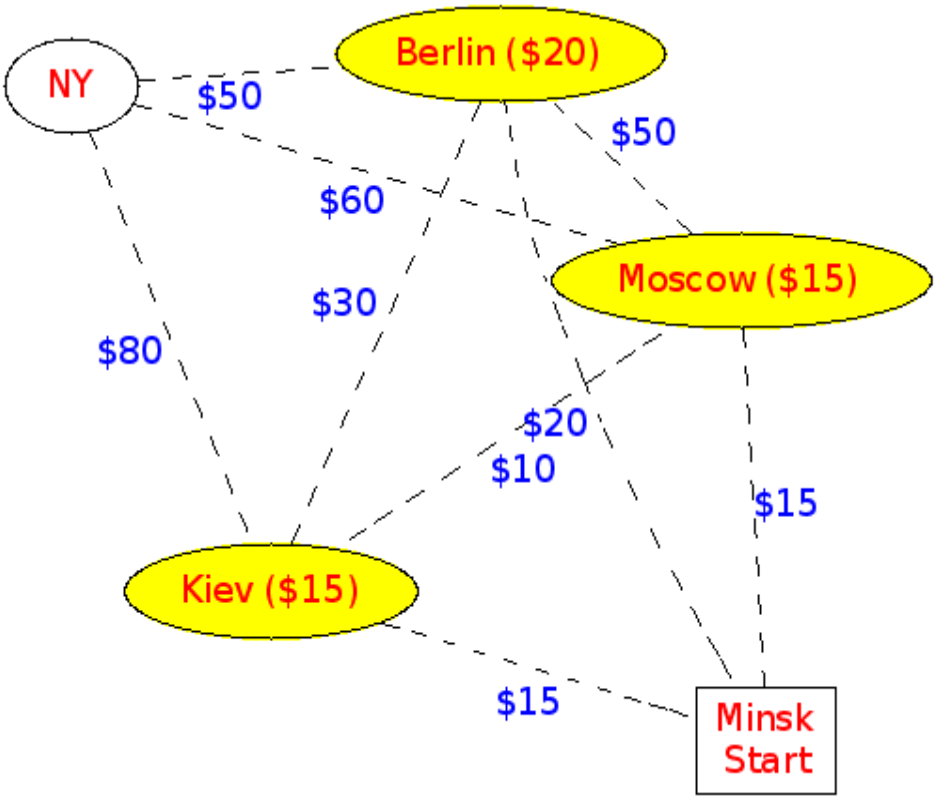
Процедура SetColorEdge извлекает номера ребер из стека и перекрашивает ребра структуры Node[i] (листинг 14.39)

Закраска ребер минимального остова

```
 procedure SetColorEdge; // закраска ребер
 var i,j,t: integer;
 begin
 while not Stack_IsEmpty(Stack) do begin
 Stack_Pop(Stack,t); // взять из стека № ребра
 for i:=0 to N-1 do
 with Node[i] do begin
 LL:=Length(Edge);
 for j:=0 to LL-1 do
 if ((i=Edges[t].n1) and (Edge[j].NumNode=Edges[t].n2)) or
 ((i=Edges[t].n2) and (Edge[j].NumNode=Edges[t].n1)) then
 SetEdgeBlack(i,j); // закрасить ребро
 end;
 end;
 end;
```

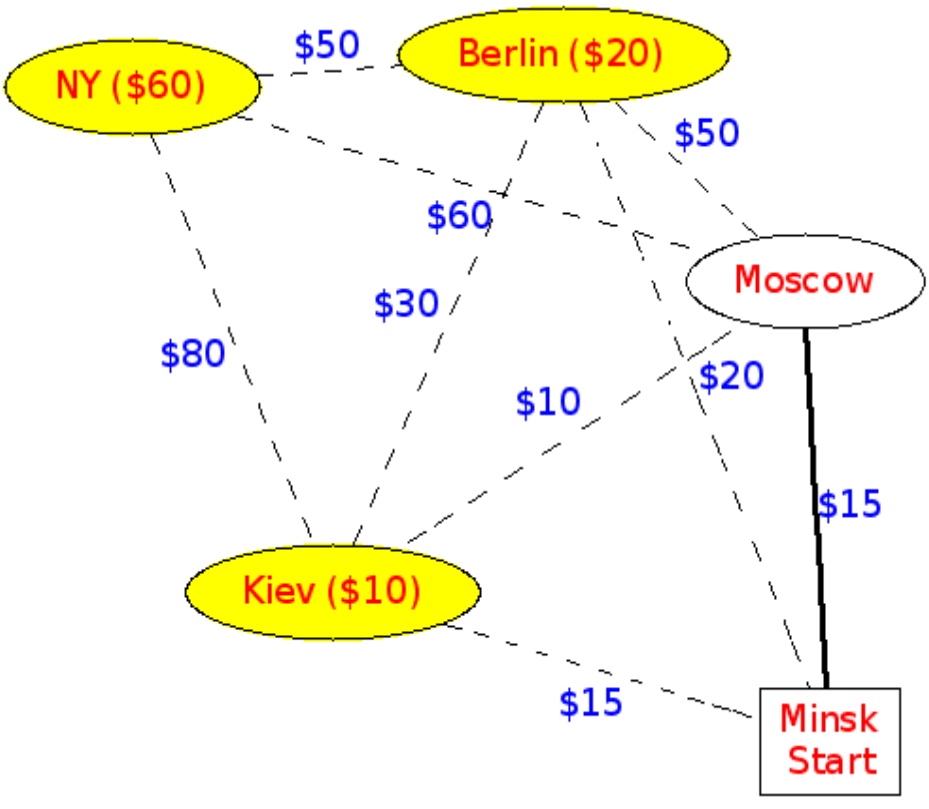
Трассировка алгоритма

Итерация 1



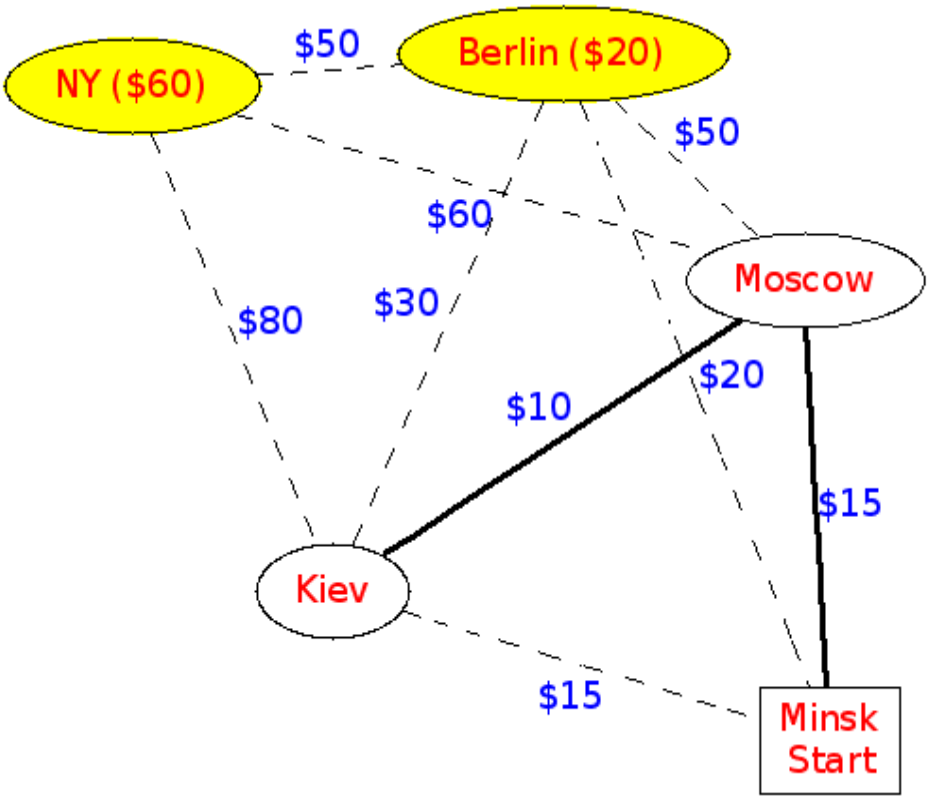
[svg]

Итерация 2



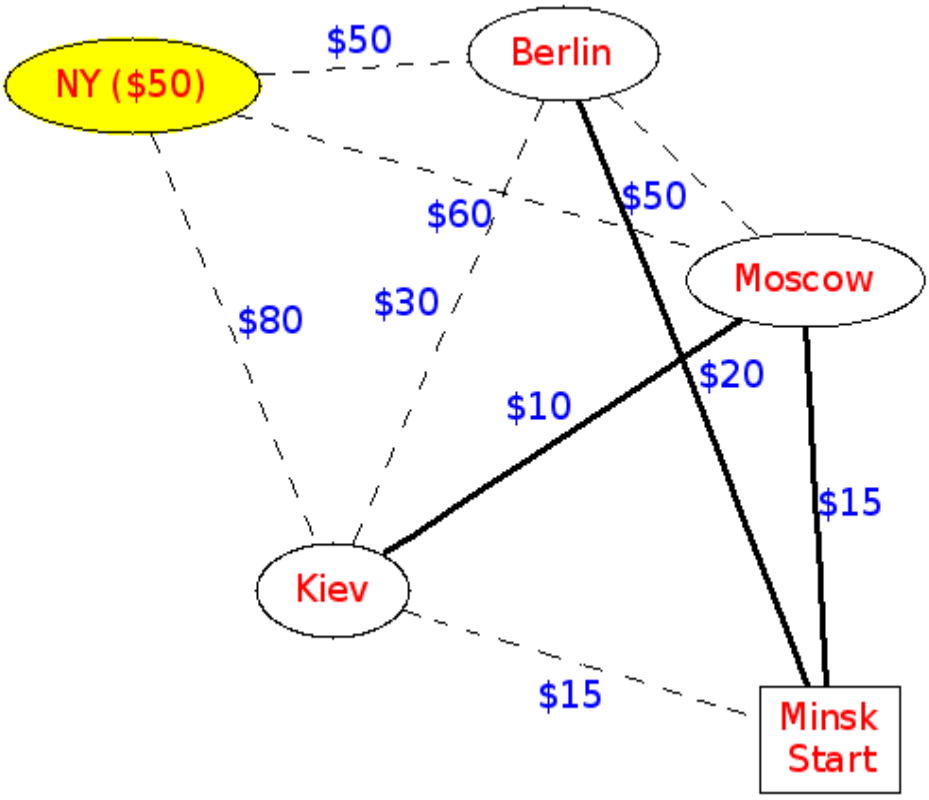
[svg]

Итерация 3



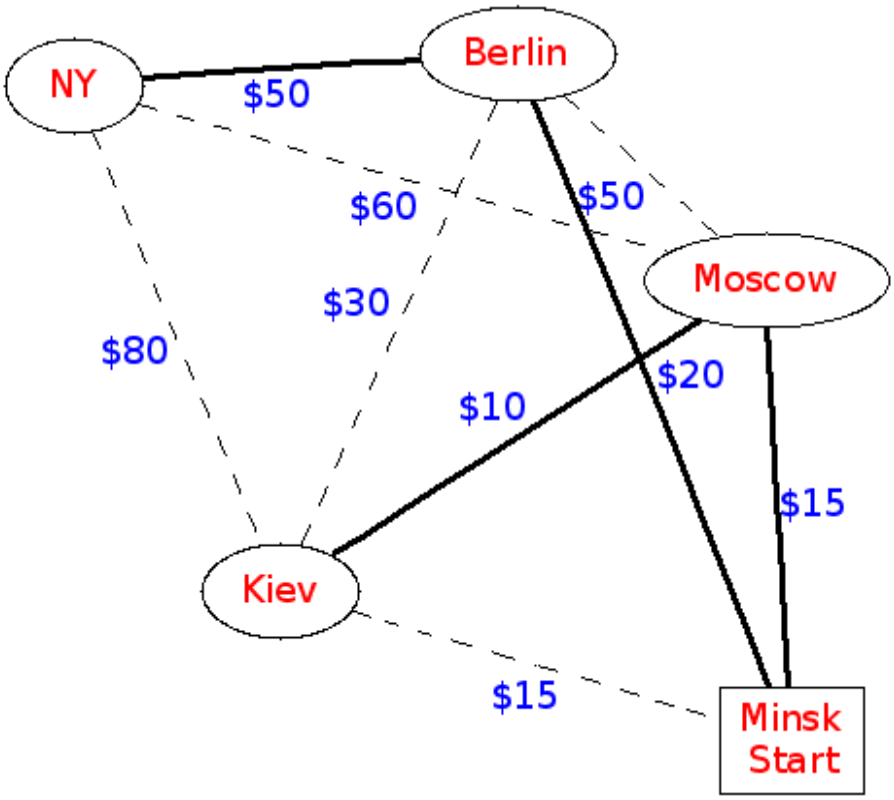
[svg]

Итерация 4



[svg]

Итерация 5



[s

Обход неориентированных графов, поиск в ширину, поиск в глубину

Обход неориентированных графов.
Различаютдва типа дуг (вместо 4-х в орграфе) при обходе в глубину: ребра дерева ведут от посещенного узла к непосещенному и обратные дуги, к-рые обобщают поперечные дуги , обратные дуги и прямые дуги орграфа т.к. в неориентированном графе направление дуг не учитывается.
Ребро дерева в глубинном остовном лесу не орграфа образуется путем непосредственного вызова (v,w) – dfs(w) после вызова dfs(v)
Обратная дуга (v,w) образуется, если между вызовами dfs от (w) и dfs(v) был произведен вызов dfs для любого другог узла

Алгоритм использует очередь, в которую помещаются узлы «вторичного распространения волны», а точнее узлы, которые должны быть рассмотрены на очередном шаге .
Поиск в ширину.или обход по принципу волны.
bfs() – алгоритм обхода в ширину.

Листинг:

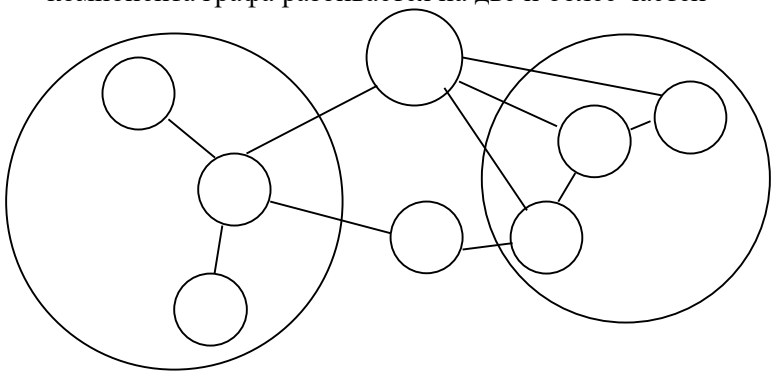
```
Procedure bfs(v);
Var     q:queue;
        x,y:вершина;
begin
        mark [v]:=visited;
        Enqueue(v, Q)
        While not empty(Q) do
                Begin
                        X:=front(Q);
                        Dequeue(Q);
                Для каждой вершины y смежной с вершиной x do
                        If mark[y]:=unvisited then begin
                                Mark[y]:=visited;
                                Enqueue(y,Q);
                                Insert((x,y), T);
                        End;
                End;
End;
```

1. в цикле перебирая все еще не посещенные узлы выполняем все остальные шаги, помещаем очередной узел в очередь.
2.на каждом шаге вынимаем очередной узел из очереди: для данного узла перебираем все инцидентные дуги для поиска еще не посещенной вершины
3. для каждой найденной ранее не посещенной вершины помечаем её и помещаем в очередь
4. в шаге 2-3 выполняем до тех пор, пока очередь не пуста
5. Очередность помещения в очередь и образует очередность обхода

Поскольку узлы обрабатываются в порядке возрастания удаленности от начальной точки обхода вне зависимости от весов дуг. Его можно использовать для определения кратчайшего расстояния от источника до любого узла невзешенного неориентированного графа

Точки сочленения

Точкой сочленения такая вершина v при удалении которой и всех инциндентных ей ребер связная компонента графа разбивается на две и более частей



Связный граф, не имеющий точек сочленения называется двусвязным

Метод нахождения точек сочленения часто применяется при решении задачи о k-связности. Граф называется k-связным, если удаление любых k-1 вершин не ведет к расчленению графа.

Утверждение Если между любой парой вершин v и w существуе не менее k различных путей таких, что за исключением вершин v и w никакая другая другая вершина графа не входит в разные пути, то такой граф является k-связным.

Таким образом, двусвязный граф, не имеющий точек сочленения, имеет связность k=2.

Схема нахождения точки сочленения

Схема нахождения точек сочления.

1. На основе поиска в глубину определяем глубинное число для каждого узла.
2. Для каждой вершины вычисляем число low[v] равное минимуму глубинных чисел

а) глубинных чисел

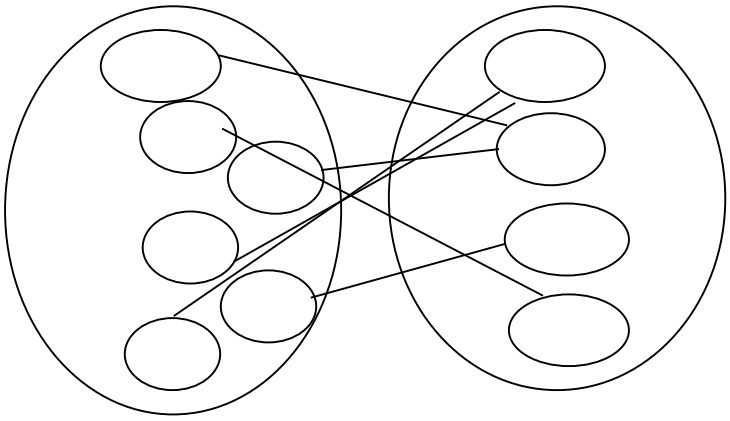
б) глубинного значения узлов Z, для которых существуе обратная дуга (v,z)

в) из множества значений low[x] потомков узла v.

Точками сочленения
Являются корень остовного дерева, имеющий двух или более сыновей.
Вершина v, отличная корня, будет точкой сочленения тогда, и только тогда, когда имеет такого сына w, что low[w] ≥ dfnumber[v].

Поиск покрытий и паросочетаний

Поиск покрытий и паросочетаний.
Паросочетанием графа некоторое множество его дуг такое, что каждая вершина графа инциндентна не более чем одной дуге этого множества.
Покрытием называется некоторое множество дуг в графе такое, что каждая его вершина инциндента по крайней мере одной дуге этого множества
Два множества вершин одного графа. Пусть v1 – дисциплины а v2 – преподаватели
Обеспечить одновременное прочтение максимально возможного числа дисциплин
Таким образом паросочетание может ассоциироваться с распараллеливанием, а покрытие - в некоторой.



любое подмножество паросочетаний является паросочетанием, а любое множество дуг включающее в качестве подмножества покрытие графа является покрытием этого графа.
Ы

Различают четыре типа задач по парасочетаниям и покрытиям.

1. Задача о паросочетании максимальной мощности.
2. Задача о паросочетании с максимальным весом
3. Задача о покрытии минимальной мощности
4. Задача о покрытии с минимальным весом

Задача о паросочетании с максимальным весом и покрытии с минимальным весом рассматриваются дуги с положительными ценами. В противном случае вес каждой дуги на максимальное по модулю. Решение задачи о паросочетании максимальной мощности дает решение задачи о покрытии минимальной мощности. Т.е. если паросочетание максимальной мощности M* - паросочетание известно, то с его помощью можно найти покрытие минимальной мощности. C* - покрытие мин. Мощности.
От паросочетания к покрытию.
Пусть M – произв паросочет. Выделим

Теорема. Если M – паросочетание максимальной мощности, то тогда C’ в алгоритме 1. покрытие минимальной мощности, и наоборот, если в алгоритме 2 C-покрытие миимальной мощности, то M’ – паросочетание максимальной мощности.
|X|=|C’|+|M|
|X|=|M’|+|C|
на заданной матрице смежности получить покрытие минимальной мощности с максимальным весом. Получить новую матрицу и нарисовать полученный и исходный графы.
Оценить трудоемкость.

Список литературы

1. Род Хаггарти. Дискретная математика для программистов: 2-е издание, исправленное. М.:Техносфера 2018 — 400 с.
2. Новиков Ф. Дискретная математика: Учебник для вузов. 3-е изд. Стандарт третьего поколения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Учебник для вузов»).
3. Ерусалимский Я. М. Дискретная математика. Теория, задачи, приложения. – 2018. — 476 с.
4. Тишин В. В. Дискретная математика в примерах и задачах. — 2-с идд.. испр. — СПб.: БХВ-Петербург, 2016. — 336 с.: (Учебная литература для вузов)
5. Авдошин С. М., Набсбин А. А. Дискретная математика. Модулярная алгебра, криптография, кодирование. — М.: ДМК Пресс, 2017. - 352 с.: ил.
6. Дискретная математика: прикладные задачи и сложность алгоритмов : учебник и практикум для академического бакалавриата / А. Е. Андреев, А. А. Болотов, К. В. Коляда, А. Б. Фролов. — 2-е изд., испр. и доп. — М. : Издательство Юрайт, 2018. — 317 с.
7. Задачи но дискретной математике / С. В. Борзунов, С. Д. Кургалин. — СПб.: БХВ-Петербург, 2016. — 528 с.: ил. — (Учебная литература для вузов)

8. Судоплатов С. В. Дискретная математика : учебник и практикум для академического бакалавриата / С. В. Судоплатов, Е. В. Овчинникова. — 5-е изд., испр. и доп. — М. : Издательство Юрайт, 2018. — 279 с.
9. Гисин, В. Б. Дискретная математика : учебник и практикум для академического бакалавриата / В. Б. Гисин. — М. : Издательство Юрайт, 2018. — 383 с.
10. Брайан У. Керниган, Роб Пайк, Практика программирования М.:Вильямс 2017. 288с.
11. Роберт Мартин. Чистая архитектура. Искусство разработки программного обеспечения СПБ.: Питер 2018 352с
12. Майкл Ховард, Дэвид Лебланк, Джон Виега, Как написать безопасный код на C++, Java, Perl, PHP, ASP.NET М.:ДМК Пресс 2017 288с.
13. Сергей Никифоров, Прикладное программирование М.:Лань 2018 124с.
14. Э Фримен, Э Фримен, К Сьерра, Б Бейтс. Паттерны проектирования //СПб.: Питер. – 2018. 656 с.
15. И.Г. Гниденко, Ф.Ф. Павлов, Д.Ю. Федоров Технологии и методы программирования. М.: Юрайт 2017 235с.
16. Адитья Бхаргава, Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. М.:Издательство «Питер» 2017. 288 с.
17. Дино Эспозито, Андреа Сальтарелло, Microsoft .NET. Архитектура корпоративных приложений М.:Вильямс 2017 432с.