Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**Пермский национальный исследовательский**

**политехнический университет**

Кафедра «Информационные технологии и автоматизированные системы»

**Структурное и объектно-ориентированное программирование на С++.**

**Лабораторный практикум.**

**Часть III.**

**Представление графических объектов и проектирование программ на С++.**

Авторы: Бартов Е.И.,   
Ерохин Н.В., Торопицын М.С.

**Пермь, 2018**

Оглавление

[Оглавление 3](#_Toc523773307)

[Лабораторная работа № 1.](#_Toc523773308) [Бинарные деревья 4](#_Toc523773309)

[1.1. Цель работы: 4](#_Toc523773311)

[1.2. Краткие теоретические сведения 4](#_Toc523773312)

[1.3. Постановка задачи 5](#_Toc523773313)

[1.4. Варианты заданий 6](#_Toc523773314)

[1.5. Методические указания 6](#_Toc523773340)

[1.6. Содержание отчета 7](#_Toc523773341)

[Лабораторная работа № 2.](#_Toc523773342) [Введение в теорию графов. Алгоритмы Дейкстры и Флойда 8](#_Toc523773343)

[2.1. Цель работы: 8](#_Toc523773345)

[2.2. Краткие теоретические сведения 8](#_Toc523773346)

[2.3. Варианты заданий 25](#_Toc523773347)

[2.4. Содержание отчета 33](#_Toc523773369)

[Лабораторная работа № 3.](#_Toc523773370) [Динамическое программирование. Задача коммивояжера 34](#_Toc523773371)

[3.1. Цель работы: 34](#_Toc523773373)

[3.2. Краткие теоретические сведения 34](#_Toc523773374)

[3.3. Постановка задачи 34](#_Toc523773375)

[3.4. Варианты заданий 36](#_Toc523773376)

[3.5. Методические указания 38](#_Toc523773377)

[3.6. Содержание отчета 38](#_Toc523773378)

[Лабораторная работа № 4.](#_Toc523773379) [STL – стандартная библиотека шаблонов в С++ 39](#_Toc523773380)

[4.1. Цель работы: 39](#_Toc523773382)

[4.2. Краткие теоретические сведения 39](#_Toc523773383)

[4.3. Постановка задачи 43](#_Toc523773384)

[4.4. Варианты заданий 44](#_Toc523773385)

[4.5. Методические указания 44](#_Toc523773386)

[4.6. Содержание отчета 45](#_Toc523773387)

## Лабораторная работа № 1

## Бинарные деревья



### Цель работы:

Получить практические навыки работы с бинарными деревьями.

### Краткие теоретические сведения

Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка.

Описать такую структуру можно следующим образом:

struct point

{

int data;//информационное поле

point \*left;//адрес левого поддерева

point \*right;//адрес правого поддерева

};

Начальный узел называется корнем дерева. Узел, не имеющий поддеревьев, называется листом. Исходящие узлы называются предками, входящие — потомками. Высота дерева определяется количеством уровней, на которых располагаются его узлы.

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется деревом поиска. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму количества узлов.

В идеально сбалансированном дереве количество узлов справа и слева отличается не более чем на единицу.

Линейный список можно представить как вырожденное бинарное дерево, в котором каждый узел имеет не более одной ссылки. Для списка среднее время поиска равно половине длины списка.

Деревья и списки являются рекурсивными структурами, т. к. каждое поддерево также является деревом. Таким образом, дерево можно определить как рекурсивную структуру, в которой каждый элемент является:

* либо пустой структурой;
* либо элементом, с которым связано конечное число поддеревьев.

Действия с рекурсивными структурами удобнее всего описываются с помощью рекурсивных алгоритмов.

**Обход дерева**

Для того, чтобы выполнить определенную операцию над всеми узлами дерева, все узлы надо обойти. Такая задача называется обходом дерева. При обходе узлы должны посещаться в определенном порядке. Существуют три принципа упорядочивания. Рассмотрим дерево, представленное на рисунке 1.1:

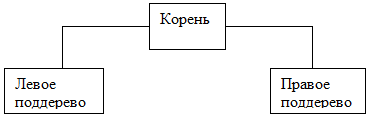


Рисунок 1.1 – Бинарное дерево

На этом дереве можно определить три метода упорядочивания:

Слева направо: Левое поддерево – Корень – Правое поддерево;

Сверху вниз: Корень – Левое поддерево – Правое поддерево;

Снизу вверх: Левое поддерево – Правое поддерево – Корень.

Эти три метода можно сформулировать в виде рекурсивных алгоритмов.

void Run(point\*p)

//обход слева направо

{

if(p)

{

<обработка p->data>

Run(p->left);//переход к левому поддереву

Run(p->right);//переход к правому поддереву

}

}

Если в качестве операции обработки узла поставить операцию вывода информационного поля, то мы получим функцию для печати дерева.

### Постановка задачи

1. Сформировать идеально сбалансированное бинарное дерево, тип информационного поля указан в варианте.

2. Распечатать полученное дерево.

3. Выполнить обработку дерева в соответствии с заданием, вывести полученный результат.

4. Преобразовать идеально сбалансированное дерево в дерево поиска.

5. Распечатать полученное дерево.

### Варианты заданий

### 1 . Тип информационного поля char. Найти количество элементов с заданным ключом.

### 2. Тип информационного поля int. Найти максимальный элемент в дереве.

### 3 . Тип информационного поля char\*. Найти количество листьев в дереве.

### 4 . Тип информационного поля double. Найти минимальный элемент в дереве.

### 5 . Тип информационного поля char. Найти высоту дерева.

### 6 . Тип информационного поля int. Найти среднее арифметическое элементов дерева.

### 7 . Тип информационного поля char\*. Найти количество элементов дерева, начинающихся с заданного символа.

### 8. Тип информационного поля char. Найти количество элементов с заданным ключом.

### 9 . Тип информационного поля double. Найти максимальный элемент в дереве.

### 10. Тип информационного поля int. Найти количество листьев в дереве.

### 11. Тип информационного поля double. Найти минимальный элемент в дереве.

### 12. Тип информационного поля char. Найти высоту дерева.

### 13. Тип информационного поля int. Найти среднее арифметическое элементов дерева.

### 14. Тип информационного поля char. Найти количество элементов с заданным ключом.

### 15. Тип информационного поля char\*. Найти количество элементов дерева, начинающихся с заданного символа

### 16. Тип информационного поля int. Найти максимальный элемент в дереве.

### 17. Тип информационного поля double. Найти количество листьев в дереве.

### 18. Тип информационного поля int. Найти минимальный элемент в дереве.

### 19. Тип информационного поля char. Найти высоту дерева.

### 20. Тип информационного поля double. Найти среднее арифметическое элементов дерева.

### 21. Тип информационного поля char\*. Найти количество элементов дерева, начинающихся с заданного символа.

### 22. Тип информационного поля char. Найти количество элементов с заданным ключом.

### 23. Тип информационного поля char\*. Найти количество листьев в дереве.

### 24. Тип информационного поля double. Найти максимальный элемент в дереве.

### 25. Тип информационного поля double. Найти минимальный элемент в дереве.

### Методические указания

1. Описания структур для формирования деревьев, а также функции для их обработки сохранить в библиотечном файле с расширением .h (например, point.h). Функцию main() сохранить в файле с расширением .cpp. Библиотечный файл подключить с помощью директивы #include “имя\_файла.h”.

2. Для выделения памяти под информационные поля типа char\* использовать операцию new, для удаления из памяти – операцию delete.

3. Для формирования элементов дерева написать отдельные функции.

4. Для формирования дерева, удаления добавления элементов, поиска заданных элементов написать отдельные функции.

5. В функции main() должны быть размещены только описания переменных и обращения к соответствующим функциям.

6. Если в дереве отсутствуют элементы, соответствующие критерию поиска (например, при удалении элемента с номером k, k больше, чем количество элементов в списке), должно быть выведено сообщение о том, что требуемые элементы не найдены.

7. Интерфейс реализовать с помощью текстового меню.

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Блок-схема

4. Текст программы

5. Тесты

## Лабораторная работа № 2

## Введение в теорию графов. Алгоритмы Дейкстры и Флойда



### Цель работы:

Получить практические навыки работы с графами.

### Краткие теоретические сведения

Реализовать обход графа, а также алгоритм Дейкстры или алгоритм Флойда. Вершина, с которой начать выполнение, указана в варианте.

**Обходы графов: в глубину и в ширину**

Обход графа – процедура однократного посещения всех вершин. Далее описаны 2 алгоритма обхода графа: в глубину и в ширину. Сначала будет разобран обход графа в глубину, затем обход графа в ширину.

**Обход графа в глубину**

По завершении обхода все вершины окажутся пройдёнными - обработанными. Если при обходе встречается вершина, которая уже была пройдена, то повторной обработки делать не нужно.

Рассмотрим следующий граф:

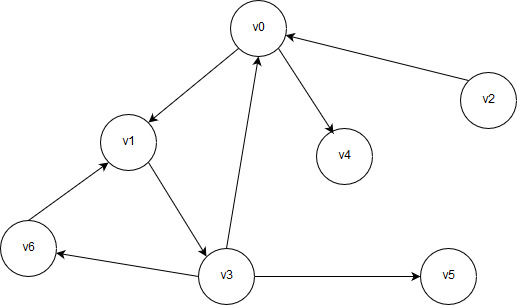


Рисунок 2.1 – Алгоритм обхода графа в глубину

Обход начинается с вершины v0. После обработки вершины v0 выполняется переход по одному из ребер к одной из соседних вершин. В данном примере существуют две возможности: перейти к вершине v1 или к вершине v4. Перейти от вершины v0 к вершине v2 или v3 невозможно, поскольку ребра, содержащие их, направлены в противоположную сторону. Итак, алгоритм обхода стоит перед выбором: перейти к вершине v1 или к вершине v4. Не имеет значения, как именно сделать этот выбор, - допустим, что выбрана вершина v1. После ее обработки рисунок будет выглядеть так:

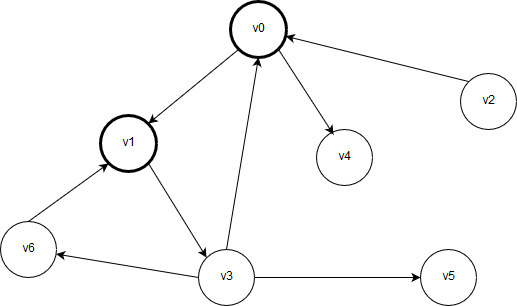


Рисунок 2.2 – Обработаны вершины v0 и v1

Затем алгоритм совершает переход к вершине, соседствующей с вершиной v1. Если соседняя вершина уже обработана, то алгоритм продвигается дальше, пропуская уже обработанные соседние вершины. В данном примере рассматривается только одна соседняя вершина v3 и обрабатывается. В данный момент обработанными являются три вершины, как показано ниже.

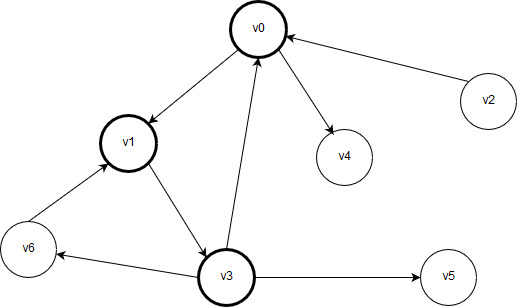


Рисунок 2.3 – Обработаны вершины v0, v1 и v3

Одной из соседних вершин по отношению к вершине v3 является вершина v0, однако она уже обработана. Таким образом, чтобы предотвратить зацикливание, переходить от вершины v3 в вершину v0 нельзя. Поэтому можно перейти к другому соседу: v5 или v6. Перейдем в вершину v5. После ее обработки на графе будут цветом отмечены вершины v0 и v1 как посещённые:

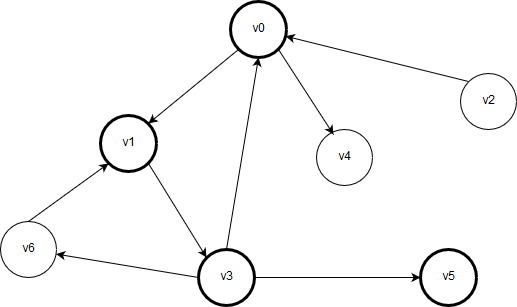
****

Рисунок 2.4 – Обработаны вершины v0, v1, v3 и v5

Поскольку у вершины v5 нет соседних вершин, дальнейший поиск в глубину невозможен. Далее, алгоритм возвращается назад и проверяет, нет ли у предыдущей вершины v3 других необработанных соседних вершин. У вершины v3 есть ещё необработанная соседняя вершина v6. Выполняется переход в вершину v6 и обработка вершины v6. После обработки вершины v6 рисунок имеет следующий вид:

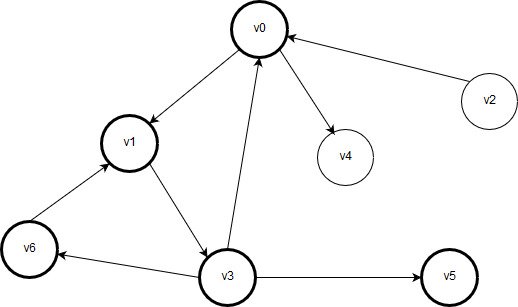
****

Рисунок 2.5 – Вершина v6 обработана после возврата к вершине v3

У вершины v6 есть соседняя вершина – v1, однако она уже обработана. Таким образом, у вершины v6 нет обработанных соседних вершин, поэтому обход возвращается назад – в вершину v3, чтобы проверить, остались ли у нее другие необработанные соседи. У вершины v3 больше нет необработанных соседей. Значит, алгоритм возвращается к предыдущей вершине v1, чтобы проверить, есть ли у нее непомеченные соседние вершины. У вершины v1 нет необработанных соседних вершин, поэтому выполняется возврат в вершину v0 для проверки, есть ли у вершины v0 необработанные соседи. Такой сосед есть – это вершина v4, поэтому алгоритм переходит в нее. После перехода граф выглядит так:

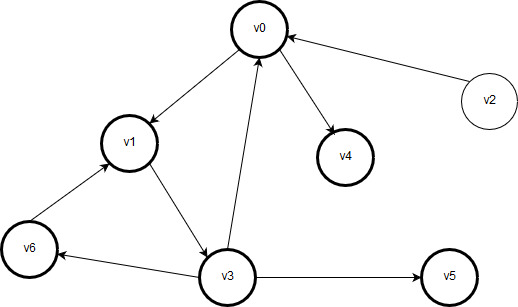
****

Рисунок 2.6 – Вершина v4 обработана

У вершины v4 соседних вершин нет, поэтому происходит возврат в вершину v0. У вершины v0 больше нет необработанных соседних вершин. Поскольку вершина v0 была отправной точкой нашего обхода, дальнейшее перемещение по графу невозможно (из вершины v0 в вершину v2 попасть невозможно). Обход графа завершен.

Во время обхода графа в глубину обрабатываются лишь те вершины, которые достижимы из начальной вершины графа. Существует еще одно важное свойство обхода графа в глубину: из начальной вершины осуществляется переход в соседнюю вершину, оттуда – в соседнюю к соседней и так далее – при этом обход графа осуществляется настолько глубоко, насколько это возможно, и только потом выполняется возврат в исходную вершину. Таким образом, алгоритм обхода графа в глубину работает рекурсивно.

**Обход в ширину**

Этот алгоритм использует очередь для отслеживания необработанных соседних вершин. Этот алгоритм обхода графа применяется для поиска компонент связности в графе, а также, как и алгоритм обхода в глубину, применяется при прохождении лабиринтов. Поиск начинается с начальной вершины, которая обрабатывается, маркируется и помещается в очередь. Возьмем для примера граф, рассмотренный на рисунке 2.1.

Предположим, что начальной является вершина v0. Тогда вершина является первой вершиной, подлежащей обработке, маркировке цветом и сохранению в очереди.

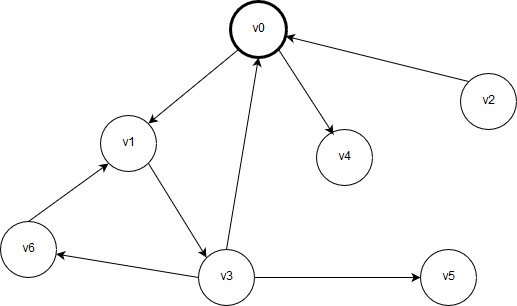




Рисунок 2.7 – Вершина v0 промаркирована и занесена в очередь

На данный момент в голове очереди находится начальная вершина v0.

После того, как начальная вершина обработана, промаркирована и помещена в очередь, начинается выполнение основной части алгоритма обхода графа в ширину. Основой алгоритма является циклический процесс, в котором обработанная вершина удаляется из очереди, а в очередь помещаются соседствующие с обработанной вершины. Таким образом, тело цикла состоит из двух основных шагов:

- Удалить вершину v из головы очереди

- Для каждой непомеченной вершины u, соседней по отношению к вершине v, обработать вершину u, маркировать ее и поместить в очередь (вершина u может иметь соседние необработанные вершины).

Эти действия выполняются до тех пор, пока очередь не станет пустой.

Рассмотрим работу алгоритма на примере, когда вершина v0 находится в голове очереди (рисунок 2.7). Вершина v0 удаляется из очереди и отмечается, что у нее остались необработанными две соседние вершины – v1 и v4. Каждая из них обрабатывается, маркируется и помещается в очередь. Допустим, сначала в очередь поставлена вершина v1, а затем вершина v4 (очередность может быть любой, вершина v4 может быть поставлена первой, а вершина v1 за ней – алгоритм будет работать корректно в любом случае). После того, как вершины v1 и v4 были поставлены в очередь, граф выглядит следующим образом:

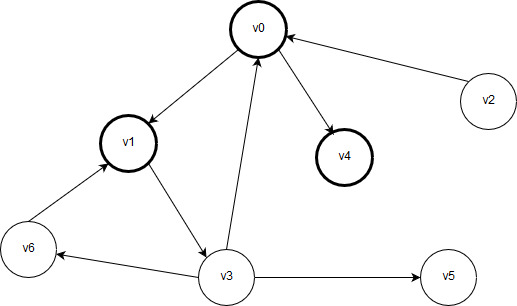




Рисунок 2.8 – Промаркированы вершины v0, v1, v4, в очереди находятся вершины v1 и v4

Поскольку очередь не пуста, два шага выполняются снова: первый элемент (вершина v1) удаляется, обрабатывается, все необработанные соседние вершины маркируются и помещаются в очередь. Единственным неотмеченным соседом вершины v1 является вершина v3, поэтому после обработки, маркировки и постановки в очередь вершины v3 граф выглядит так:

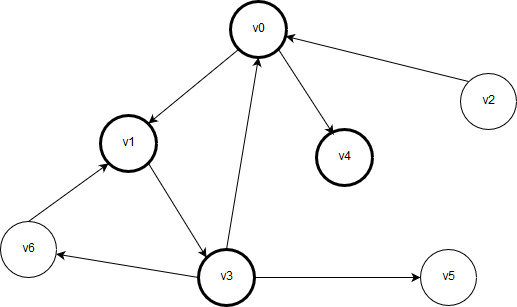




Рисунок 2.9 – Промаркированы вершины v0, v1, v4, v3, в очереди находятся вершины v3 и v4

Далее из головы очереди удаляется вершина v4. Поскольку у нее нет соседних вершин, никакие вершины больше не обрабатываются и не становятся в очередь, при этом граф выглядит так:

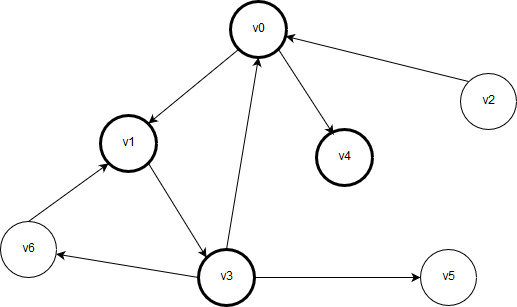




Рисунок 2.10 - Промаркированы вершины v0, v1, v4, v3, в очереди находится только вершина v3

Следующей из очереди удаляется вершина v3. У нее есть две непомеченные соседние вершины (вершины v5 и v6), которые обрабатываются, маркируются и помещаются в очередь. Граф выглядит следующим образом:

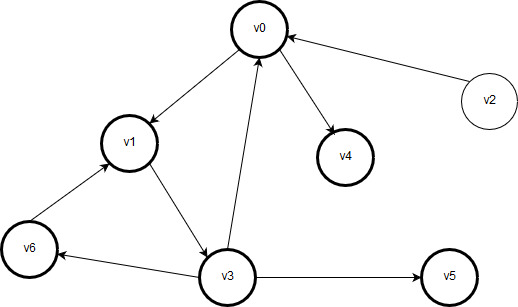




Рисунок 2.11 – Промаркированы вершины v0, v1, v4, v3, v5, v6, в очереди находятся вершины v5 и v6

Вершина v0 не обрабатывается повторно, так как она уже помечена.

Результат обхода графа в глубину и в ширину одинаков – отличается порядок обработки. Вершины v0, v1, v3, v4, v5 и v6 обработаны, поскольку все они достижимы из начальной точки (вершины 0). Вершина v2 не обрабатывается, поскольку не существует пути из вершины v0 в вершину v2. При обходе графа в ширину вершины обрабатываются в следующем порядке: сначала вершина v0, потом v1 и v4, затем обрабатываются их соседи (вершина v3) и так далее. При обходе графа в глубину сначала обрабатываются вершины v0, v1, v3 и v5.

**Алгоритм Дейкстры**

Рассмотрим следующий граф:

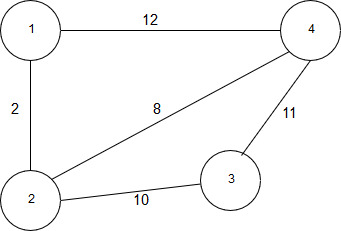


Рисунок 2.12 - Поиск кратчайшего пути по алгоритму Дейкстры

Пусть требуется найти кратчайшие расстояния от вершины 1 до всех остальных вершин графа. Длина пути между вершинами, называемая весом, обозначена над ребрами графа.

Алгоритм предполагает, что все вершины графа должны быть размечены, метка означает длину кратчайшего пути в эту вершину из заданной вершины 1. Следовательно, метка заданной вершины 1 устанавливается равной 0. Метки остальных вершин устанавливаются равными недостижимо большому числу, что отражает то, что расстояния от вершины 1 до других вершин пока не известны. В начале алгоритма все вершины графа помечаются как необработанные.

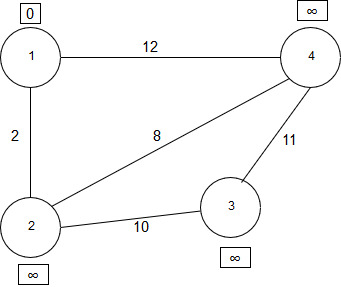


Рисунок 2.13 - Вершинам графа поставлены соответствующие метки

Первый шаг алгоритма:

Минимальную метку, равную нулю, имеет вершина 1. Её соседями являются вершины 2 и 4. Алгоритм обходит соседей вершины по очереди. Первый сосед вершины 1 – вершина 2, так как длина пути до неё минимальна. Длина пути в неё через вершину 1  
равна сумме кратчайшего расстояния до вершины 1 (значению её метки) и длины ребра, идущего из вершины 1 в вершину 2, то есть

0 + 2 = 2.

Из двух меток – текущей, равной бесконечности, и рассчитанной, равной 2, выбирается наименьшая. Поэтому новая метка 2-ой вершины будет равна 2, что показано на рисунке ниже.

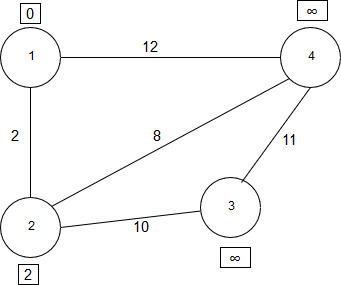


Рисунок 2.14 – Метка у вершины 2 изменена

Аналогично находятся длины путей для всех других соседей - вершины 4. От вершины 1 до вершины 4 кратчайшим будет расстояние в 12 единиц:

0 + 12 = 12.

Все соседи вершины 1 проверены. На данный момент все вершины, в которые можно попасть из вершины 1 напрямую, проверены. Вершина 1 отмечается как обработанная. Для графа, изображенного на рисунке ниже, таким расстоянием будет путь, равный двум единицам.

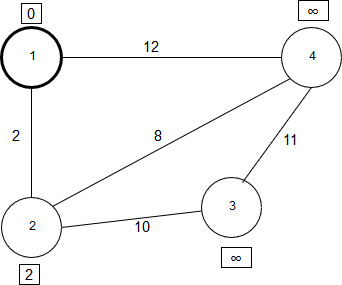


Рисунок 21.15 – Вершина 1 обработана

Второй шаг алгоритма:

Снова находится «ближайшая» из необработанных вершин. Это вершина 2 с меткой 2.

Согласно алгоритму, снова предпринимается попытка уменьшить метки соседей выбранной вершины, используя для перехода в них 2-ю вершину в качестве промежуточной. Соседями вершины 2 являются вершины 1 и 3. Вершина 1 уже обработана. Следующий сосед вершины 2 – вершина 4, так как до нее расстояние от вершины 2 будет меньше, чем до вершины 3. Длина пути от вершины 1 до вершины 4 составит 10 единиц:

2+8=10.

Полученное расстояние меньше 12, поэтому у вершины 4 ставится метка 10. Далее рассматривается вершина 3. Расстояние до нее от вершины 1 через вершину 2 составит 12 единиц:

2+10=12.

Полученное значение меньше бесконечности, поэтому метка вершины 3 меняется с бесконечности на 12. Вершина 2 отмечается как посещенная, что показано на рисунке ниже.

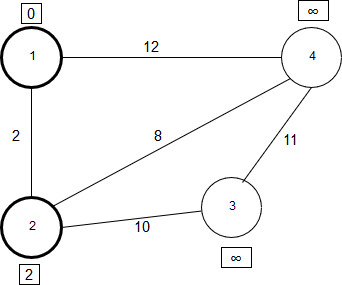


Рисунок 2.16 - Вершина 2 обработана

Третий шаг:

Обрабатывается вершина 4, после ее обработки ничего не меняется, поскольку если идти в вершину 3 через вершину 4 (предварительно посетив вершину 2, так как кратчайший путь в вершину 4 лежит через вершину 2), то расстояние составит

10+11=21 (ед).

Полученное расстояние больше, чем текущая метка вершины 3, поэтому метка вершины 3 меняться не будет. Граф на этом шаге выглядит так:

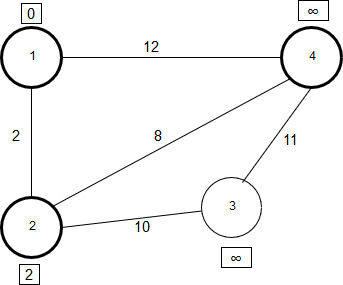


Рисунок 2.17 - Вершина 4 обработана

На четвертом шаге алгоритма обрабатывается вершина 3. После её обработки также ни одна метка не изменится, поскольку все остальные вершины уже обработаны, следовательно кратчайшие расстояния до них пересмотру не подлежат. На третьем шаге было выяснено, что от вершины 4 до вершины 3 путь более выгодным не является, поэтому метка вершины 3 осталась прежней. На текущем (и последнем) шаге алгоритма вершина 3 отмечается как посещенная. Граф выглядит так:

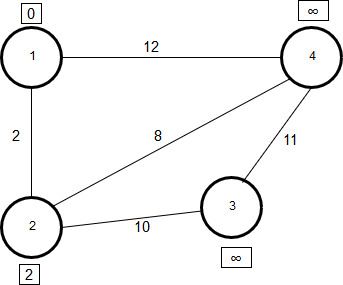


Рисунок 2.16 - Вершина 3 обработана

Таким образом, кратчайший путь от вершины 1 до вершины 3 проходит через вершины 1-2-3 и имеет минимальное расстояние, равное 12 единицам.

**Алгоритм Флойда**

Данный алгоритм ищет кратчайшие пути от каждой из вершин графа до всех остальных.

Рассмотрим граф:

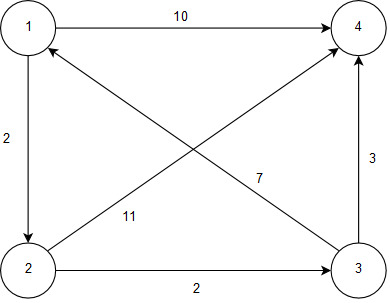


Рисунок 2.17 – Алгоритм Флойда

Метод предполагает создание двух матриц: матрицы кратчайших расстояний – первой матрицы и матрицы для построения пути – второй матрицы.

В начале алгоритма первая матрица изначально совпадает с матрицей смежности:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 10000 | 10 |
| 2 | 10000 | 0 | 2 | 11 |
| 3 | 7 | 10000 | 0 | 3 |
| 4 | 10000 | 10000 | 10000 | 0 |

Рисунок 2.18 – Первая матрица в начальном состоянии

В первой матрице значения 10000 взяты условно, на самом деле они отражают очень большое число, которое значительно больше веса любого из ребер данного графа. В дальнейшем при описании алгоритма вместо 10000 будет употребляться понятие «бесконечность». Расстояние из вершины до самой себя равно 0, в остальных полях матрицы все значения обозначают расстояния, которые необходимо преодолеть от одной вершины к другой, причем расстояния эти самые короткие. Во время работы алгоритма некоторые из значений, хранящихся в первой матрице, будут меняться, уменьшаясь.

Вторая матрица предназначена для построения кратчайшего пути:

Вторая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 0 | 4 |
| 2 | 0 | 0 | 3 | 4 |
| 3 | 1 | 0 | 0 | 4 |
| 4 | 0 | 0 | 0 | 0 |

Рисунок 2.19 – Вторая матрица в начальном состоянии

. Поясним, какие значения там хранятся: во второй матрице хранятся номера вершин, в которые надо пройти, чтобы попасть в желаемую. То есть, поле [1,2] означает: чтобы попасть из вершины 1 в вершину 2, необходимо пройти через вершину, которая хранится в поле [1,2], а именно через вершину 2. Так как путь из любой вершины до самой себя равен 0, то на главной диагонали матрицы стоят нули. Также нули стоят там, куда невозможно прийти на данный момент (так как из вершины 1 в вершину 3 попасть невозможно, то в поле матрицы [1,3] стоит 0).

Алгоритм Флойда работает следующим образом:

Пусть начальной вершиной является вершина 1. Алгоритм определяет, в какую вершину можно попасть из начальной вершины, используя только одну промежуточную вершину. Можно попасть, например, в вершину 3 через вершину 2; а также в вершину 4 через вершину 2. Кроме того, можно остаться в вершине 1.

Итак, определено, что, используя одну промежуточную вершину, можно попасть в вершины 3 и 4.

Рассчитаем в условных единицах затраты на преодоление каждого пути: чтобы попасть в вершину 3 из вершины 1 через вершину 2 потребуется 4 условных единицы. Чтобы попасть в вершину 4 через вершину 2, потребуется 13 условных единиц. Алгоритм сравнивает полученные значения с теми значениями, которые на данный момент находятся в первой матрице:

чтобы пройти из вершины 1 в вершину 3 потребуется бесконечно много условных единиц, то есть на данный момент добраться из вершины 1 в вершину 3 невозможно. В то же время найден путь, в котором придется затратить только 4 условных единицы для того, чтобы попасть из вершины 1 в вершину 3. Затраты в 4 условные единицы на преодоление пути меньше, чем бесконечное значение затрат, поэтому в первой матрице в поле [1,3] записывается вместо бесконечно большого значения значение 4. Затем длина пути из вершины 1 в вершину 4 через вершину 2 с затратами 13 условных единиц сравнивается со значением, хранящимся в поле [1,4] первой матрицы. Затраты в 13 условных единиц больше, чем в 10 условных единиц, то есть рассмотренный путь не является более выгодным. Следовательно, во второй матрице в поле [1,3] записывается 2, поскольку чтобы попасть из вершины 1 в вершину 3, нужно сначала пройти в вершину 2.

На этом этапе матрицы выглядят так:

Первая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 4 | 10 |
| 2 | 10000 | 0 | 2 | 11 |
| 3 | 7 | 10000 | 0 | 3 |
| 4 | 10000 | 10000 | 10000 | 0 |

Рисунок 2.20 – От вершины 1 найдено более короткое расстояние в вершину 3

Вторая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 2 | 4 |
| 2 | 0 | 0 | 3 | 4 |
| 3 | 1 | 0 | 0 | 4 |
| 4 | 0 | 0 | 0 | 0 |

Рисунок 2.21 – Во вторую матрицу записана в качестве промежуточной вершины вершина 2 (путь из вершины 1 в вершину 3)

Далее, алгоритм ищет, в какие вершины можно попасть из вершины 1, используя две промежуточные вершины. Такой является вершина 4 (путь через вершины 2 и 3). Затраты на преодоление этого пути составляют

2+2+3=7.

Больше таким способом попасть никуда нельзя.

Затраты в 7 условных единиц меньше, чем 10 условных единиц, поэтому в первую матрицу в поле [1,4] записывается значение 7. Во вторую матрицу в поле [1,4] записывается значение 2, поскольку, чтобы по кратчайшему пути попасть из вершины 1 в вершину 4, нужно сначала пройти вершину 2.

На этом этапе матрицы выглядят так:

Первая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 4 | 7 |
| 2 | 10000 | 0 | 2 | 11 |
| 3 | 7 | 10000 | 0 | 3 |
| 4 | 10000 | 10000 | 10000 | 0 |

Рисунок 2.22 – В первую матрицу занесено значение 7 – затраты на преодоление пути из вершины 1 в вершину 4

Вторая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 2 | 2 |
| 2 | 0 | 0 | 3 | 4 |
| 3 | 1 | 0 | 0 | 4 |
| 4 | 0 | 0 | 0 | 0 |

Рисунок 2.23 – Во вторую матрицу в поле [1,4] занесено значение 2, поскольку для перехода из вершины 1 в вершину 4 в качестве промежуточной использована вершина 2

Далее алгоритм ищет, в какие вершины можно попасть из вершины 1, используя три промежуточных вершины. Далее алгоритм прекратит поиски, поскольку используя четыре промежуточных вершины алгоритм придет в исходную вершину, то есть в вершину 1, что не имеет смысла. Используя три промежуточных вершины попасть ни в одну из вершин невозможно. Поиски для вершины 1 прекращены.

Затем алгоритм ищет кратчайшие пути от вершины 2 до всех остальных. Так же, как и для вершины 1, алгоритм проверяет, куда можно попасть из вершины 2, используя только одну промежуточную вершину. Такими являются вершины 1, куда можно попасть через вершину 3 и вершина 4, в которую также можно попасть через вершину 3.

Рассчитаем в условных единицах затраты на преодоление каждого пути: чтобы попасть из вершины 2 в вершину 1 через вершину 3, потребуется

2+7=9 условных единиц.

Чтобы попасть из вершины 2 в вершину 4 через вершину 3 потребуется

2+3=5 условных единиц.

Алгоритм сравнивает полученные значения со значениями, которые на данный момент находятся в первой матрице:

чтобы пройти из вершины 2 в вершину 1 потребуется бесконечно много условных единиц, то есть на данный момент добраться из вершины 2 в вершину 1 невозможно. В то же время найден путь, в котором придется затратить только 9 условных единиц для того, чтобы попасть из вершины 2 в вершину 1. Затраты в 9 условных единиц на преодоление пути меньше, чем бесконечное значение затрат, поэтому в первой матрице в поле [2,1] записывается вместо бесконечно большого значения значение 9. Затем длина пути из вершины 2 в вершину 4 через вершину 3 с затратами 5 условных единиц сравнивается со значением, хранящимся в поле [2,4] первой матрицы. Затраты в 5 условных единиц меньше, чем в 11 условных единиц, поэтому в поле [2,4] первой матрицы записывается вместо значения 11 значение 5. Во второй матрице поле [2,1] записывается 3, поскольку чтобы попасть из вершины 2 в вершину 1, нужно сначала пройти в вершину 3, в ячейку [2,4] также записывается значение 3, поскольку для того, чтобы попасть из вершины 2 в вершину 4, нужно сначала пройти вершину 3.

На этом этапе матрицы выглядят так:

Первая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 4 | 7 |
| 2 | 9 | 0 | 2 | 5 |
| 3 | 7 | 10000 | 0 | 3 |
| 4 | 10000 | 10000 | 10000 | 0 |

Рисунок 2.24 – В первую матрицу занесены значения 9 и 5 – затраты на преодоление путей от вершины 2 до вершин 1 и 4

Вторая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 2 | 2 |
| 2 | 3 | 0 | 3 | 3 |
| 3 | 1 | 0 | 0 | 4 |
| 4 | 0 | 0 | 0 | 0 |

Рисунок 2.25 – Во вторую матрицу в поля [2,1] и [2,4] занесено значение 3, поскольку для перехода из вершины 2 в вершины 1 и 4 в качестве промежуточной использована вершина 3

Затем алгоритм определяет, в какие вершины можно попасть из вершины 2, используя только две промежуточные вершины. Такой вершиной является вершина 2, однако осуществлять переход из вершины 2 в вершину 2 не имеет смысла, поэтому алгоритм прекращает поиск вершин, в которые можно попасть через две промежуточных вершины.

Далее алгоритм ищет вершины, в которые можно попасть из вершины 2 через три промежуточных вершины. Таких вершин нет, поэтому алгоритм прекращает поиск, так как через четыре промежуточных вершины в любом случае будет осуществлен переход в исходную вершину.

На следующем этапе работы алгоритм ищет кратчайшие пути от вершины 3 до всех остальных. Метод ищет вершины, в которые можно попасть из вершины 3 через одну промежуточную вершину. Такими вершинами являются вершины 2 и 4, в которые переход осуществляется через вершину 1. Рассчитаем затраты на преодоление каждого пути в условных единицах: чтобы попасть из вершины 3 в вершину 2 через вершину 1, потребуется

7+2=9 условных единиц.

Чтобы попасть в вершину 4 из вершины 3 через вершину 1, потребуется

7+10=17 условных единиц.

Алгоритм сравнивает полученные значения со значениями из первой матрицы:

чтобы пройти из вершины 3 в вершину 2 потребуется бесконечно много условных единиц, то есть на данный момент добраться из вершины 3 в вершину 2 невозможно. В то же время найден путь, в котором придется затратить только 9 условных единиц для того, чтобы попасть из вершины 3 в вершину 2. Затраты в 9 условных единиц на преодоление пути меньше, чем бесконечное значение затрат, поэтому в первой матрице в поле [3,2] записывается вместо бесконечно большого значения значение 9. Затем длина пути из вершины 3 в вершину 4 через вершину 1 с затратами 17 условных единиц сравнивается со значением, хранящимся в поле [3,4] первой матрицы. Затраты в 17 условных единиц больше, чем в 11 условных единиц, то есть рассмотренный путь не является более выгодным. Во второй матрице в поле [3,2] записывается 1, поскольку чтобы попасть из вершины 3 в вершину 2, нужно сначала пройти в вершину 1 в ячейку [3,4] ничего не записывается.

На этом этапе матрицы выглядят так:

Первая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 4 | 7 |
| 2 | 9 | 0 | 2 | 5 |
| 3 | 7 | 9 | 0 | 3 |
| 4 | 10000 | 10000 | 10000 | 0 |

Рисунок 2.26 – В первую матрицу занесено значение 9 – затраты на преодоление пути от вершины 3 до вершины 2

Вторая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 2 | 2 |
| 2 | 3 | 0 | 3 | 3 |
| 3 | 1 | 1 | 0 | 4 |
| 4 | 0 | 0 | 0 | 0 |

Рисунок 2.27 - Во вторую матрицу в поле [3,2] занесено значение 1, поскольку для перехода из вершины 3 в вершину 2 в качестве промежуточной использована вершина 1

Далее алгоритм ищет, в какие вершины можно попасть из вершины 3 через две промежуточных вершины. Такими вершинами являются вершины 3 и 4. В вершину 3 переходить не имеет смысла. В вершину можно попасть через вершины 1 и 2. Рассчитаем затраты в условных единицах на преодоление этого пути: чтобы попасть из вершины 3 в вершину 4 через две промежуточных вершины, потребуется

7+2+11=20 условных единиц.

Алгоритм сравнивает полученное значение со значением, которое на данный момент находится в первой матрице:

Чтобы пройти от вершины 3 до вершины 4 напрямую, потребуется затратить 3 условных единицы, поэтому путь через вершины 1 и 2, на который необходимо затратить 20 условных единиц, не является более выгодным, поэтому ни в первую, ни во вторую матрицу ничего не записывается.

Затем алгоритм ищет вершины, в которые можно попасть из вершины 3 через три промежуточных вершины. Таких вершин нет, поэтому алгоритм прекращает поиск, так как через четыре промежуточных вершины в любом случае будет осуществлен переход в исходную вершину.

На следующем этапе работы алгоритм определяет кратчайшие пути от вершины 4 до всех остальных. Алгоритм ищет, в какие вершины можно попасть из вершины 4 через одну промежуточную вершину. Таких вершин нет, поэтому алгоритм ищет вершины, в которые можно попасть из вершины 4 через две промежуточных вершины. Таких вершин также не существует. Алгоритм переходит к поиску вершин, в которые можно попасть из вершины 4 через три промежуточных вершины. Таких вершин не существует, поэтому алгоритм прекращает работу, поскольку через четыре промежуточных вершины переходить куда-либо бессмысленно – будет осуществлен возврат в исходную вершину.

Больше вершин нет, поэтому окончательный вид матриц выглядит так:

Первая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 4 | 7 |
| 2 | 9 | 0 | 2 | 5 |
| 3 | 7 | 9 | 0 | 3 |
| 4 | 10000 | 10000 | 10000 | 0 |

Рисунок 2.28 – Вид первой матрицы (матрицы кратчайших расстояний) после работы алгоритма

Вторая матрица:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № вершины | 1 | 2 | 3 | 4 |
| 1 | 0 | 2 | 2 | 2 |
| 2 | 3 | 0 | 3 | 3 |
| 3 | 1 | 1 | 0 | 4 |
| 4 | 0 | 0 | 0 | 0 |

Рисунок 2.29 – Вид второй матрицы (матрицы промежуточных вершин) после работы алгоритма

Покажем, как записывается кратчайший путь с помощью второй матрицы. Пусть требуется записать путь из вершины 1 в вершину 4. Сначала просматривается поле [1,4], где находится номер вершины 2, значит нужно идти в вершину 2. Далее, проверяется поле [2,4], куда надо идти из вершины 2, чтобы попасть в вершину 4. В этом поле записана вершина 3, значит, из вершины 2 идём в вершину 3. Далее проверяется поле [3,4], там находится номер вершины 4, значит, осуществляется переход в вершину 4. Далее рассматривается поле [4,4], и алгоритм прекращает выстраивание пути. Таким образом, путь с минимальными затратами из вершины 1 в вершину 4 проходит через следующие вершины:

1-2-3-4.

Результаты работы программы:

Кратчайший путь из вершины 1 в вершину 2: 1-2;

Кратчайший путь из вершины 1 в вершину 3: 1-2-3;

КП из вершины 1 в 4: 1-2-3-4;

КП из вершины 2 в 1: 2-3-1;

КП из вершины 2 в 3: 2-3;

КП из вершины 2 в 4: 2-3-4;

КП из вершины 3 в 1: 3-1;

КП из вершины 3 в 2: 3-1-2;

КП из вершины 3 в 4: 3-4.

### Варианты заданий

|  |  |
| --- | --- |
| №  варианта | Задание |
| 1 | Реализовать граф, а также алгоритм обхода графа в глубину (выполнение с вершины 1).  D:\Практика\Лабы\Рисунки графов\F1.jpg |
|  | |

|  |  |
| --- | --- |
| 2 | Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 6.  D:\Практика\Лабы\Рисунки графов\D2.jpg |
| 3 | Реализовать граф, а также алгоритм обхода графа в ширину (выполнение с вершины 4).  D:\Практика\Лабы\Рисунки графов\F3.jpg |
| 4 | Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 2.  D:\Практика\Лабы\Рисунки графов\D4.jpg |

|  |  |
| --- | --- |
| 5 | Реализовать граф, а также алгоритм Флойда, построив все необходимые матрицы.  Выполнение начать с вершины 1.  D:\Практика\Лабы\Рисунки графов\F5.jpg |
| 6 | Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 6.  D:\Практика\Лабы\Рисунки графов\D6.jpg |
| 7 | Реализовать граф, а также алгоритм обхода графа в глубину (выполнение с вершины 1).  D:\Практика\Лабы\Рисунки графов\F7.jpg |

|  |  |
| --- | --- |
| 8 | Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия. Выполнение начать с вершины 4.  D:\Практика\Лабы\Рисунки графов\D8.jpg |
| 9 | Реализовать граф, а также алгоритм обхода графа в ширину (выполнение с вершины 3).  D:\Практика\Лабы\Рисунки графов\F9.jpg |
| 10 | Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 6.  D:\Практика\Лабы\Рисунки графов\D10.jpg |

|  |  |
| --- | --- |
| 11 | Реализовать граф, а также алгоритм Флойда, построив все необходимые матрицы.  Выполнение начать с вершины 7.  F:\Projects\Глава по графам для поляковой\новые рисунки\Dijkstra11.jpg |
| 12 | Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 6.  F:\Projects\Глава по графам для поляковой\новые рисунки\Dijkstraa(1).jpg |
| 13 | Реализовать граф, а также алгоритм Флойда, построив все необходимые матрицы.  Выполнение начать с вершины 1.  D:\Практика\Лабы\Рисунки графов\F13.jpg |

|  |  |
| --- | --- |
| 14 | Реализовать граф, а также алгоритм Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 3.  D:\Практика\Лабы\Рисунки графов\D14.jpg |
| 15 | Реализовать граф, а также Флойда, построив все необходимые матрицы.  Выполнение начать с вершины 1.  D:\Практика\Лабы\Рисунки графов\F15.jpg |
| 16 | Реализовать граф, а также Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 2.  D:\Практика\Лабы\Рисунки графов\D16.jpg |

|  |  |
| --- | --- |
| 17 | Реализовать граф, а также Флойда, построив все необходимые матрицы.  Выполнение начать с вершины 3.  D:\Практика\Лабы\Рисунки графов\F17.jpg |
| 18 | Реализовать граф, а также Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 3.  D:\Практика\Лабы\Рисунки графов\D18.jpg |
| 19 | Реализовать граф, а также Флойда, построив все необходимые матрицы.  Выполнение начать с вершины 5.  D:\Практика\Лабы\Рисунки графов\F19.jpg |

### 

|  |  |
| --- | --- |
| 20 | Реализовать граф, а также Флойда, построив все необходимые матрицы.  Выполнение начать с вершины 3.  D:\Практика\Лабы\Рисунки графов\F17.jpg |
| 21 | Реализовать граф, а также Дейкстры, выполнив все необходимые действия.  Выполнение начать с вершины 3.  D:\Практика\Лабы\Рисунки графов\D18.jpg |
| 22 | Реализовать граф, а также Флойда, построив все необходимые матрицы.  Выполнение начать с вершины 5.  D:\Практика\Лабы\Рисунки графов\F19.jpg |

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Блок-схемы основных функций

4. Текст программы

5. Тесты

## Лабораторная работа № 3

## Динамическое программирование. Задача коммивояжера



### Цель работы:

Изучить алгоритм решения задачи коммивояжера и научиться использовать его в графе.

### Краткие теоретические сведения

**Практическое применение задачи коммивояжера**

Применение задачи коммивояжера на практике довольно обширно, ее можно использовать для поиска кратчайшего пути на гастролях эстрадной группы или обеспечения наименьшего времени выполнения производственного цикла и других решения других оптимизационных задач.

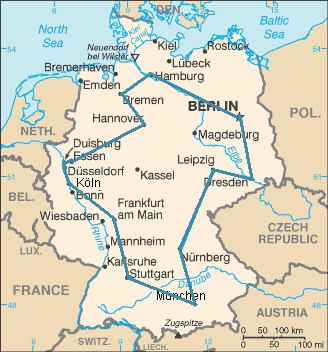


Рисунок 3.1 – Оптимальный маршрут коммивояжёра через 15 крупнейших городов Германии. Указанный маршрут является самым коротким из всех возможных 43589145600 вариантов

### Постановка задачи

Имеется N городов. Выезжая из исходного города А1, коммивояжер должен побывать во всех городах по одному разу и вернуться в город А1. Задача заключается в определении последовательности объезда городов, при которой коммивояжеру требуется минимизировать некоторый критерий эффективности: стоимость проезда, время пути, суммарное расстояние.

Для расчета затрат существует матрица условий, содержащая затраты на переход из каждого города в каждый, при этом считается, что можно перейти из любого города в любой, кроме того же самого. Целью решения является нахождения маршрута, удовлетворяющего всем условиям и при этом имеющего минимальную сумму затрат.

**Анализ задачи коммивояжера**

Для решения задачи коммивояжера составляется матрица условий, которая содержит расстояния между городами. Считается, что можно перейти из любого города в любой, кроме того же самого. Так как пути из города А в город А по условию нет, обозначим его «нн».

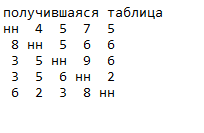


Рисунок 3.2 – Матрица условий

После построения изначальной матрицы условий проводим редукцию строк и столбцов.

Редукция строк: находим минимальное расстояние в строке, вычитаем его из всех расстояний в строке, кроме «нн».

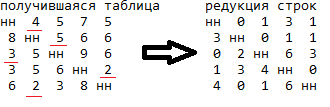


Рисунок 3.3 – Редукция строк (минимальные элементы подчеркнуты)

Редукция столбцов: находим минимальное расстояние в столбце, вычитаем его из всех расстояний в столбце, кроме «нн».

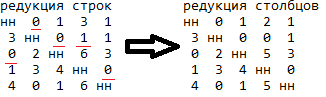


Рисунок 3.4 – Редукция столбцов (мин. элементы подчеркнуты)

Оценка нулей: каждый 0 в таблице оцениваем: считаем сумму минимального элемента в строке и в столбце и запоминаем результат.

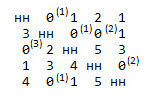


Рисунок 3.5 – Оценка нулей

Чистка карты: выбираем 0 с наибольшей оценкой, если таких несколько, выбираем любой, вычеркиваем строку и столбец, в котором находится этот 0 (вычеркивание — замена элементов строки и столбца на «нн»).

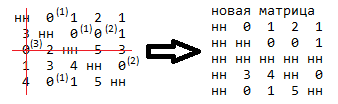


Рисунок 3.6 – Чистка карты

Координаты вычеркнутого столбца и строки запоминаем, назовем их ребрами. Повторяем редукцию столбцов, строк и чистку карты, пока последний переход не станет очевидным (сведем матрицу к размерам 2х2).

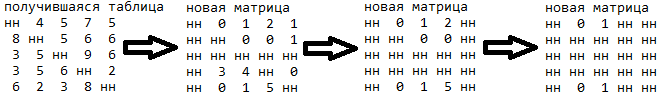


Рисунок 3.7 – Сведение исходной матрицы к матрице 2х2

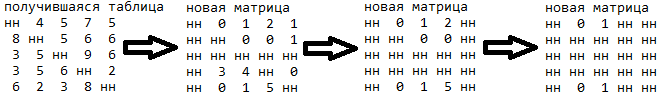


Рисунок 3.8 – Сведение исходной матрицы к матрице 2х2

Полученные ребра выстраиваем в последовательность и получаем наименьший путь, из исходной матрицы берем расстояния, двигаясь по проложенному пути, получаем оптимальное расстояние всего пути.

Результат решения задачи коммивояжера:

3->1->2->4->5->3

весь путь:

18

### Варианты заданий

|  |  |
| --- | --- |
| № матрицы | Матрица условий |
| 1 | |  |  |  |  | | --- | --- | --- | --- | | нн | 21 | 27 | 41 | | 20 | нн | 2 | 56 | | 14 | 38 | нн | 1 | | 5 | 24 | 46 | нн | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | |  |  |  |  | | --- | --- | --- | --- | | нн | 5 | 3 | 3 | | 5 | нн | 9 | 7 | | 3 | 9 | нн | 4 | | 3 | 7 | 4 | нн | |
| 3 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | нн | 25 | 40 | 31 | 27 | | 5 | нн | 17 | 30 | 25 | | 19 | 15 | нн | 6 | 1 | | 9 | 50 | 24 | нн | 6 | | 22 | 8 | 7 | 10 | нн | |
| 4 | |  |  |  |  | | --- | --- | --- | --- | | нн | 20 | 42 | 35 | | 20 | нн | 30 | 34 | | 42 | 30 | нн | 12 | | 35 | 34 | 12 | нн | |
| 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | нн | 47 | 22 | 46 | 29 | | 34 | нн | 25 | 34 | 19 | | 18 | 18 | нн | 33 | 7 | | 38 | 27 | 24 | нн | 38 | | 21 | 14 | 6 | 27 | нн | |
| 6 | |  |  |  |  | | --- | --- | --- | --- | | нн | 5 | 11 | 9 | | 10 | нн | 8 | 7 | | 7 | 14 | нн | 8 | | 12 | 6 | 15 | нн | |
| 7 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | нн | 97 | 60 | 73 | 17 | 52 | | 97 | нн | 41 | 52 | 90 | 30 | | 60 | 41 | нн | 21 | 35 | 41 | | 73 | 52 | 21 | нн | 95 | 46 | | 17 | 90 | 35 | 95 | нн | 81 | | 52 | 30 | 41 | 46 | 81 | нн | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | нн | 27 | 43 | 16 | 30 | 16 | | 7 | нн | 16 | 1 | 30 | 25 | | 20 | 13 | нн | 35 | 5 | 0 | | 21 | 16 | 25 | нн | 18 | 18 | | 12 | 46 | 27 | 48 | нн | 5 | | 23 | 5 | 5 | 9 | 5 | нн | |
| 9 | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | нн | 6 | 29 | 10 | 59 | 34 | 13 | | 6 | нн | 23 | 4 | 53 | 28 | 7 | | 23 | 21 | нн | 20 | 31 | 6 | 17 | | 8 | 3 | 20 | нн | 50 | 25 | 4 | | 43 | 40 | 22 | 38 | нн | 26 | 47 | | 26 | 24 | 4 | 22 | 18 | нн | 22 | | 10 | 5 | 17 | 4 | 35 | 19 | нн | |

### Методические указания

1. Написать программу, решающую задачу коммивояжера, указанную в варианте.

2. Для решения задачи коммивояжера использовать метод ветвей и границ.

3. Поэтапно продемонстрировать изменение исходной матрицы.

4. Исходная матрица может храниться в текстовом файле, либо объявлена в двумерном массиве в глобальной области.

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Блок-схемы основных функций

4. Текст программы

5. Тесты

## Лабораторная работа № 4

## STL – стандартная библиотека шаблонов в С++



### Цель работы:

Получить практические навыки при работе со стандартной библиотекой шаблонов.

### Краткие теоретические сведения

STL-библиотека – набор алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в С++. Библиотеки дают возможность сделать код намного короче, за счет обобщенного программирования.

**Контейнерные классы**

*Контейнер* – это объект, который хранит в себе другие объекты одного типа. Хранимые объекты могут быть объектами в смысле объектно-ориентированного программирования либо значениями встроенных типов. Данные, сохранённые в контейнере, *принадлежат ему*. Это означает, что, когда время существования контейнера истекает, то же самое происходит с сохранёнными в нём данными.

Библиотека STL (Standard Template Library) предоставляет целый набор шаблонных контейнерных классов. Они обладают некой базовой концепцией, которая обязует их иметь конкретные поля и методы. Например, все контейнеры имеют метод size(), который возвращает количество элементов в контейнере.

**Последовательности**

Базовую концепцию можно уточнять, добавляя требования. *Последовательность* – это важное уточнение, поскольку несколько типов контейнеров STL – vector, stack, queue, list, deque, forwart\_list, priority\_queue – являются последовательностями. Уточнение, заключается в том, что переход от элемента к элементу последовательности должен быть, по меньшей мере, однонаправленным, что гарантирует размещение элементов в определённом порядке, который не меняется от одного цикла итераций к другому.

Поскольку элементы в последовательности размещены в определённом порядке, становятся возможными такие операции, как вставка значений в определённую позицию и удаление определенного диапазона элементов. Для осуществления этих операций все последовательности обладают соответствующими методами:

* insert() – метод вставки элементов.
* erase() – метод удаления элементов.
* clear() – метод удаляет все элементы контейнера.

Рассмотрим шесть типов контейнеров-последовательностей более подробно.

**1. vector:**

*vector* – это представление динамического массива в виде класса, где поддерживается автоматическое управление памятью, которое позволяет динамически менять размер объекта vector, увеличивая и уменьшая его при добавлении или удалении элементов. Он предоставляет произвольный доступ к элементам с помощью операции индексации [] и метода at().

**2. deque:**

Класс шаблона *deque* представляет собой двустороннюю очередь – тип, кратко называемый «*дека»*. В том виде, в каком он реализован в STL, он напоминает контейнер vector, где поддерживается произвольный доступ к элементам. Основное различие между ними состоит в том, что вставка и удаление элементов из начала объекта deque – операция, выполняемая за постоянное время, в то время как для объекта vector эти операции линейны во времени.

**3. list:**

Класс шаблона *list* представляет собой двусвязный список. Каждый его элемент, за исключением первого и последнего, связан как с предшествующим элементом, так и последующим элементами, откуда следует, что по такому списку можно проходить в обоих направлениях. Различие между list и vector заключается в том, что list обеспечивает вставку и удаление за постоянное время в любой позиции списка. Также класс шаблона list имеет функции-члены, которые позволяют осуществить: слияние списков, сортировку списка, сворачивание повторяющихся элементов и другие.

**4. forward\_list:**

В С++11 появился новый класс контейнера forward\_list. Этот класс реализует односвязный список. В таком списке каждый элемент связан только со следующим элементом, но не с предыдущим.

**5. queue:**

Класс шаблона *queue* представляет собой простую очередь. Он не только не позволяет произвольный доступ к элементам очереди, но даже не разрешает выполнять итерацию по её элементам. Взамен queue ограничивается базовыми операциями, определяющими очередь.

**6. stack:**

Класс шаблона *stack* предоставляет типичный интерфейс стека. Он также, как и очередь, не разрешает произвольный доступ к элементам стека и не позволяет выполнять итерацию по своим элементам. Вместо этого *stack* ограничивается базовыми операциями, определяющими stack.

**Ассоциативные контейнеры**

*Ассоциативный контейнер* – ещё одно расширение концепции контейнеров. Ассоциативный контейнер связывает значение с ключом, который служит для отыскания значения. Например, ключом может быть номер квартиры в доме, а значением – массив строк с фамилиями людей, проживающих в данной квартире.

Преимущество ассоциативных контейнеров состоит в том, что они предоставляют быстрый доступ к своим элементам, то есть обеспечивают быстрый поиск данных по ключу. Подобно последовательности, ассоциативный контейнер позволяет вставлять элементы, однако нельзя указать определённое местоположение для вставляемых элементов. Это связано с тем, что ассоциативный контейнер обладает конкретным алгоритмом для определения места размещения данных, позволяя быстро извлекать их.

Библиотека STL предлагает четыре типа ассоциативных контейнеров set, multiset, map и multimap. Первые два типа объявлены в заголовочном файле set (set.h и multiset.h), а вторые два типа объявлены в заголовочном файле map (map.h и multimap.h).

Простейшим контейнером является *set*. Тип его значения совпадает с типом ключа, а ключи уникальны, то есть в наборе хранится не более одного экземпляра каждого значения ключа. Действительно, для set значение элемента является также его ключом. Тип *multiset* аналогичен *set*, за исключением того, что он может содержать более одного значения с одним и тем же ключом.

В контейнере типа *map* тип значения отличается от типа ключа, причём ключи уникальны и на каждый ключ приходится только одно значение. Тип *multimap* подобен *map*, за исключением того, что один ключ может быть связан с несколькими значениями.

**Контейнер vector**

Контейнер вектор представляет собой динамический массив с доступом к элементам по индексу.

Для использования контейнера вектор требуется подключить библиотеку <vector>, добавление элементов в конец будет осуществлять функция *push\_back (значение элемента)*, инициализируем вектор таким образом:

vector<тип данных вектора> название;

**Итераторы**

*Итератор* – вспомогательный объект, обеспечивающий доступ к элементам контейнера. Действие над итераторами: инкремент ++, декремент --, увеличение +.

Понимание работы итераторов – одно из главных условий работы с библиотекой STL. Подобно тому, как шаблоны обеспечивают независимость алгоритмов от типа хранимых данных, итераторы обеспечивают независимость от типа используемых контейнеров.

Работа *итераторов* аналогично тому, как работают указатели. Итератор может быть объектом, для которого определены операции над указателями, такие как разыменовывание и инкремент. Итераторы необходимы для того, чтобы предоставлять однотипный интерфейс для множества классов-контейнеров, включая те, для которых обычные указатели не работают. Например, чтобы пройти по списку, не получится использовать обычный указатель, так как элементы списка не обязательно находятся в ячейках памяти последовательно. В этом случае создаётся класс «итератор», который, за счёт своей внутренней реализации, позволит последовательно перемещаться по списку, и будет иметь интерфейс, аналогичный указателю.

Для подключения итератора добавим библиотеку <iterator>. Создадим итератор на вектор таким образом:

Vector <int> a; //создаем вектор   
vector <int>::iterator название итератора = a.begin();

//итератор указывает на начало вектора

Обращаться к элементам вектора будем, используя разыменование \*it.

#include <iostream>   
#include <vector> //подключаем библиотеку вектора  
#include <iterator> //подключаем библиотеку итератора  
using namespace std;   
void main() {  
 vector <int> a; //создаем вектор  
 for (int i = 0; i< 10; i++)   
 a.push\_back(i); //заполняем вектор значениями от 0 до 9  
 vector<int>::iterator it = a.begin();   
 //создаем итератор и указываем на начало вектора  
 while (it != a.end()) { /\*пока итератор не указывает на конец вектора **/** cout <<\*it <<" "; //обращаемся к значению, разыменовывая итератор  
 it++; //переходим на следующий элемент  
 }

}



Рисунок 4.1 – Выводим вектор через итератор

Для перемещения итератора на несколько значений нужно воспользоваться функцией advance (итератор, количество элементов).

void main() {   
 vector <int> a; //создаем вектор  
 for (int i = 0; i< 10; i++)   
 a.push\_back(i);  // заполняем вектор значениями от 0 до 9  
 vector<int>::iterator it = a.begin(); //создаем итератор  
 advance(it, 5); //перемещаем итератор на 5 элементов  
 cout « [\*it](https://vk.com/it) « endl; //выводим новое значение итератора  
}

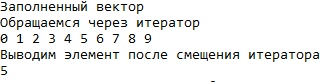


Рисунок 4.2 – Смещение итератора.

Последующее изучение итератора будет происходить по ходу изучения новых контейнеров.

**Предопределенные итераторы**

Библиотека STL предоставляет ряд заранее определённых итераторов.

Рассмотрим основные из них.

**Потоковые итераторы**

Шаблонные классы *ostream\_iterator* и *istream\_iterator* являются адаптерами – классами, которые преобразуют какой-то другой интерфейс в интерфейс, используемый STL. Они позволяют вставлять в поток и читать из потока данные с помощью итераторов, используя операции разыменовывания и инкремент. Итераторы этого вида можно создать, включив заголовочный файл iterator и сделав следующее объявление:

#include <iterator>

// Объявление итератора out\_iter типа ostream\_iterator

ostream\_iterator<int, char> out\_iter(cout, “ “);

// Использования итератора out\_iter для вывода на экран

\*out\_iter = 15; // аналогично << 15 << “ “;

Первым шаблонным аргументом является тип элементов для итератора – int.

Первый аргумент конструктора является объектом потока ostream, второй (необязательный) существует только у класса *ostream\_iterator*. Второй аргумент конструктора — это разделитель, который является С строкой и будет вставлен после каждой операции.

**Обратные итераторы**

Класс шаблона reverse\_iterator предоставляет итератор, при инкрементировании которого, вызывается его декремент. Он позволяет пройти по контейнеру в обратном порядке. Для этого у некоторых контейнеров существуют специальные методы rbegin() и rend(), которые возвращают объект типа reverse\_iterator. Для точного понимания, предположим, что в предыдущем примере мы бы использовали вместо begin() и end() методы rbegin() и rend(), тогда программа вместо 1|2|3|4|5|6| вывела бы 6|5|4|3|2|1|.

**Ассоциативный контейнер map**

Контейнеры map и vector довольно похожи, единственное отличие в том, что в map можно поместить два значения. Например, если требуется написать словарь, лучше, чем map, альтернативы не найти.

Для использования map требуется подключить библиотеку <map>. Map объявляется таким образом:

map <тип данных ключа, тип данных элемента> название {  
{ключ, значение}  
}

Для добавления элементов в контейнер будем пользоваться конструкцией вида:

map <тип 1, тип 2> название;  
название.insert(pair<тип 1, тип 2>(ключ, значение));

Функция insert вставляет элементы в контейнер.

Запись вида pair <тип 1, тип 2> (ключ, значение), где типу 1 соответствует ключ, а типу 2 значение, при этом тип 1 и тип 2 должны совпадать с типами контейнера map.

Удалять записи из контейнера будем с помощью функции *erase* (итератор на удаляемый элемент). Чтобы удалить элемент по ключу, необходимо воспользоваться функцией find (значение ключа). Этот метод возвращает итератор на элемент с таким же ключом.

### Постановка задачи

Написать и отладить три программы. Первая программа демонстрирует использование контейнерных классов для хранения встроенных типов данных.

Вторая программа демонстрирует использование контейнерных классов для хранения пользовательских типов данных.

Третья программа демонстрирует использование алгоритмов STL.

**В программе №1** реализовать следующие задачи:

1. Создать объект-контейнер в соответствии с вариантом задания и заполнить его данными, тип которых определяется вариантом задания.

2. Просмотреть контейнер.

3. Изменить контейнер, удалив из него одни элементы и заменив другие.

4. Просмотреть контейнер, используя для доступа к его элементам итераторы.

5. Создать второй контейнер этого же класса и заполнить его данными того же типа, что и первый контейнер.

6. Изменить первый контейнер, удалив из него n элементов после заданного и добавив затем в него все элементы из второго контейнера.

7. Просмотреть первый и второй контейнеры.

**В программе № 2** выполнить то же самое, но для данных пользовательского типа.

**В программе № 3** реализовать следующие задачи:

1. Создать контейнер, содержащий объекты пользовательского типа. Тип контейнера выбирается в соответствии с вариантом задания.

2. Отсортировать его по убыванию элементов.

3. Просмотреть контейнер.

4. Используя подходящий алгоритм STL, найти в контейнере элемент, удовлетворяющий заданному условию, указанному в варианте.

5. Используя подходящий алгоритм STL переместить элементы, удовлетворяющие заданному условию в другой (предварительно пустой) контейнер. Тип второго контейнера определяется вариантом задания.

6. Просмотреть второй контейнер.

7. Используя подходящий алгоритм STL отсортировать первый и второй контейнеры по возрастанию элементов.

8. Просмотреть их.

9. Используя подходящий алгоритм STL получить третий контейнер путем слияния первых двух.

10. Просмотреть третий контейнер.

11. Используя подходящий алгоритм STL подсчитать, сколько элементов, удовлетворяющих заданному условию, содержит третий контейнер.

12. Используя подходящий алгоритм STL определить, есть ли в третьем контейнере элемент, удовлетворяющий заданному условию.

### Варианты заданий

|  |  |  |  |
| --- | --- | --- | --- |
| №  п/п | Первый  контейнер | Второй  контейнер | Встроенный  тип данных |
| 1 | vector | list | int |
| 2 | list | deque | long |
| 3 | deque | stack | float |
| 4 | stack | queue | double |
| 5 | queue | vector | char |
| 6 | vector | stack | string |
| 7 | map | list | long |
| 8 | multimap | deque | float |
| 9 | set | stack | int |
| 10 | multiset | queue | char |
| 11 | vector | map | double |
| 12 | list | set | int |
| 13 | deque | multiset | long |
| 14 | stack | vector | float |
| 15 | queue | map | int |
| 16 | priority\_queue | stack | char |
| 17 | map | queue | char |
| 18 | multimap | list | int |
| 19 | set | map | char |
| 20 | multiset | vector | int |

### Методические указания

1. Все программы размещаются в одной рабочей области(workspace).

2. Рабочая область должна содержать три проекта (по числу программ).

3. При создании контейнеров в программе № 2 объекты загружать из потока.

4. Для вставки и удаления элементов контейнера в программе № 2 использовать соответствующие операции, определенные в классе контейнера.

5. Для замены, удаления и копирования элементов использовать алгоритмы copy, copy\_if, replace, replace\_if, replace\_copy\_if, remove, remove\_if, remove\_copy\_if.

6. Для поиска элемента в коллекции можно использовать алгоритм find\_if, либо for\_each, либо binary\_search, если контейнер отсортирован.

7. Для сравнения элементов при сортировке по возрастанию используется операция <, которая должна быть перегружена в пользовательском классе. Для сортировки по убыванию следует написать функцию comp и использовать вторую версию алгоритма sort.

8. Условия поиска и замены элементов выбираются самостоятельно и для них пишутся объекты-функции либо используются стандартные объекты-функции и функциональные адаптеры.

9. Для ввода-вывода объектов пользовательского класса следует перегрузить операции “>>” и “<<”.

10. Некоторые алгоритмы могут не поддерживать используемые в вашей программе контейнеры. Например, алгоритм sort не поддерживает контейнеры, которые не имеют итераторов произвольного доступа. В этом случае следует написать свой алгоритм. Например, для стека алгоритм сортировки может выполняться следующим образом: переписать стек в вектор, отсортировать вектор, переписать вектор в стек.

11. При перемещении элементов ассоциативного контейнера в не ассоциативный контейнер перемещаются только данные (ключи не перемещаются). И, наоборот, при перемещении элементов не ассоциативного контейнера в ассоциативный должен быть сформирован ключ.

12. Там, где можно следует использовать итераторные адаптеры.

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Блок-схемы основных функций

4. Текст программы

5. Тесты