

Министерство науки и высшего образования
Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Пермский национальный исследовательский
политехнический университет»

О.А. Полякова, О.Л. Викентьева

**ТЕХНОЛОГИИ РАЗРАБОТКИ
ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ
ПРОГРАММ НА ЯЗЫКЕ С++**

В трех частях

ЧАСТЬ III. ПРЕДСТАВЛЕНИЕ ГРАФИЧЕСКИХ ОБЪЕКТОВ
И ПРОЕКТИРОВАНИЕ ПРОГРАММ
НА АЛГОРИТМИЧЕСКОМ ЯЗЫКЕ С++

*Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия*

Издательство
Пермского национального исследовательского
политехнического университета
2021

ББК 32.973-018я7
УДК 681.3.06 (075)
П12

Рецензенты:

д-р экон. наук, проф. *P.A. Файзрахманов*
(Пермский национальный исследовательский
политехнический университет);
д-р пед. наук, проф. *Е.Г. Плотникова*
(Пермский филиал Национального исследовательского
университета «Высшая школа экономики»)

Полякова, О.А.

П12 Технологии разработки объектно-ориентированных программ
на языке C++ : учеб. пособие : в 3 ч. / О.А. Полякова,
О.Л. Викентьева. – Пермь : Изд-во Перм. нац. исслед. политехн.
ун-та, 2019–2021.

ISBN 978-5-398-02186-8

Ч. III. Представление графических объектов и проектирование
программ на алгоритмическом языке C++. – 203 с.

ISBN 978-5-398-02499-9

Рассмотрены формы представления графических объектов и вопросы применения основных принципов объектно-ориентированного программирования в сложных программных системах на языке высокого уровня C++, которые демонстрируются на примерах. Подготовлено на основе многолетнего опыта работы авторов и используется для ведения лекционных и практических занятий в Пермском национальном исследовательском политехническом университете, Пермском филиале Национального исследовательского университета «Высшая школа экономики» и в других вузах Российской Федерации. Является продолжением пособий, выпущенных в 2019 г.

Предназначено для студентов направлений:

09.03.01 – «Информатика и вычислительная техника»;

09.03.04 – «Программная инженерия»;

10.05.03 – «Обеспечение информационной безопасности распределенных информационных систем»;

15.03.04 – «Автоматизация технологических процессов и устройств»;

27.03.04 – «Управление и информатика в технических системах»;

38.03.05 – «Бизнес – Информатика»;

15.03.06 – «Мехатроника и робототехника».

ББК 32.973-018я7
УДК 681.3.06 (075)

ISBN 978-5-398-02499-9 (ч. III)

ISBN 978-5-398-02186-8

© ПНИПУ, 2021

ОГЛАВЛЕНИЕ

Глава 22. Бинарные деревья	5
22.1. Понятие дерева	5
22.2. Бинарные деревья	8
22.3. Реализация бинарного дерева в C++.....	12
22.4. Бинарные деревья поиска	26
22.5. Программа для работы с бинарным деревом поиска	27
22.6. Идеально сбалансированное дерево	36
22.7. Представление бинарного дерева через массив и сортировка.....	38
22.8. Вертикальная печать дерева	43
Глава 23. Введение в теорию графов	64
23.1. Виды графов. Терминология теории графов	64
23.2. Представление графов. Матрица смежности.....	68
23.3. Проектирование графов на алгоритмическом языке C++	72
23.4. Алгоритмы обхода графов.....	91
23.5. Поиск кратчайших путей в графах	106
Глава 24. Динамическое программирование.	
Решение задачи коммивояжера	143
24.1. Практическое применение задачи коммивояжера	143
24.2. Постановка задачи коммивояжера.....	143
24.3. Анализ задачи коммивояжера	144
24.4. Решение задачи коммивояжера.....	145
Глава 25. STL-библиотеки в C++	152
25.1. Контейнерные классы	152
25.2. Контейнер vector.....	155
25.3. Итераторы	156
25.4. Предопределенные итераторы	158
25.5. Ассоциативный контейнер map	160

Глава 26. Windows Forms в C++.....	164
26.1. Разработка проекта.....	164
26.2. Проектирование формы, описание свойств	166
26.3. Разработка калькулятора для чисел римской системы счисления	167
Глава 27. Вертикальное представление дерева.	
Использование графической библиотеки OpenGL.....	171
27.1. Графическая библиотека OpenGL.....	171
27.2. Представление бинарного дерева в OpenGL	178
Список литературы.....	201

ГЛАВА 22. БИНАРНЫЕ ДЕРЕВЬЯ

22.1. Понятие дерева

Определения

Дерево – это структура данных, представляющая собой совокупность элементов и отношений, образующих иерархическую структуру этих элементов.

Дерево – одна из наиболее широко распространенных структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов.

Дерево – граф иерархической структуры. Между любыми двумя его вершинами существует единственный путь. Дерево не содержит циклов и петель, является связным графом, не содержащим циклы (см. гл. 23).

Сравнение дерева со списком

Массивы и связанные списки определяют коллекции объектов, доступ к которым осуществляется последовательно, т.е. эти объекты находятся в определенном порядке. Такие структуры данных называются линейными списками, поскольку имеют уникальные первый и последний элементы и у каждого внутреннего элемента есть только один потомок (рис. 22.1).

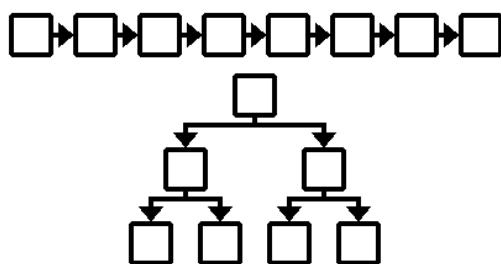


Рис. 22.1. Список и дерево (визуальное сравнение)

Однако во многих приложениях обнаруживается нелинейный порядок объектов, где элементы могут иметь нескольких потомков. Такое упорядочение называют *иерархическим*, поскольку это

название происходит от церковного распределения власти – от епископа к пасторам, дьяконам и т.д. Одной из нелинейных структур является дерево. При этом список – это частный случай дерева, т.е. это дерево, где у каждого узла, кроме нижнего, есть только один потомок.

Некоторые термины

Прежде чем анализировать деревья, следует познакомиться с некоторыми терминами, связанными с ними. Рассмотрим дерево, показанное на рис. 22.2.

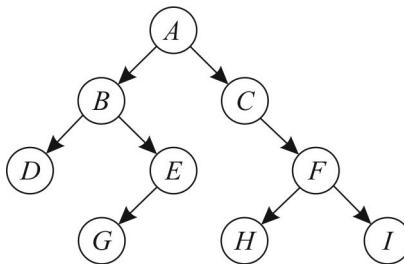


Рис. 22.2. Пример дерева

Все одиночные элементы – узлы дерева:

- *A* – самый верхний узел дерева, *корень*;
- *B, C, D, E, F* – *внутренние узлы* дерева;
- *G, H, I* – самые нижние узлы дерева, *терминальные узлы*, или *листья*.

Более сложные структуры:

- *A–B, A–C, B–D, B–E, C–F, E–G, F–H, F–I* – *ветви* дерева;
- дерево с верхним узлом *B* – *поддерево* дерева с верхним узлом *A*.

Характеристики узлов:

- *C* – узел, являющийся *потомком* узла *A* и *предком* узла *F*;
- узел *A* находится на *уровне* 1, так как это корень дерева, узлы *B* и *C* – на уровне 2. Таким образом, если элемент *X* находится на уровне *i*, то его потомок *Y* находится на уровне *i + 1*;
- число непосредственных потомков внутреннего узла – его *степень*. Например, степень узла *B* равна 2;

- число ветвей, которые нужно пройти от корня до узла, – длина пути. Так, длина пути к узлу I равна 3.

Характеристики дерева:

- максимальный уровень элементов дерева называется *высотой* дерева (в данном случае высота дерева равна 4);
- максимальная степень всех узлов дерева называется *степенью дерева* (в данном случае степень дерева равна 2);
- сумма длин путей всех его узлов называется *длиной внутреннего пути* дерева (в данном случае длина внутреннего пути равна 17).

Свойства и особенности дерева

Каждое дерево обладает следующими свойствами:

- существует узел, в который не входит ни одной ветви (корень);
- в каждую вершину, кроме корня, входит одна ветвь.

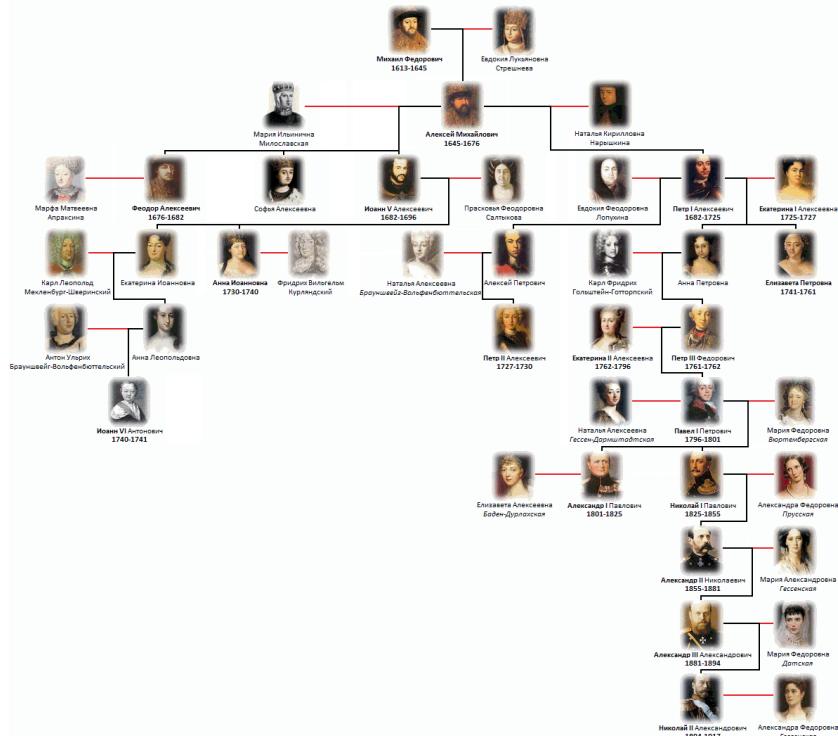


Рис. 22.3. Генеалогическое дерево

Деревья являются рекурсивными структурами, так как 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.

Примеры использования деревьев

Деревья особенно часто используют на практике при изображении различных иерархий. Например, популярны генеалогические деревья (рис. 22.3).

Кроме того, деревья описывают управляющий аппарат компаний, во главе которой стоит президент, а далее идут начальники отделов и менеджеры (рис. 22.4).

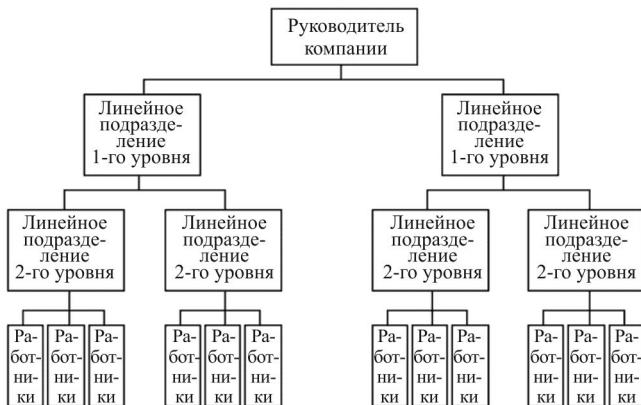


Рис. 22.4. Дерево управления компанией

22.2. Бинарные деревья

Упорядоченные и бинарные деревья

До этого мы рассматривали понятие дерева как нечто общее, глобальное. Хотя деревья общего вида достаточно важны, мы сосредоточимся на ограниченном классе деревьев, где каждый узел

имеет не более двух потомков. Но прежде стоит уделить внимание следующему понятию.

Упорядоченное дерево – это дерево, у которого ветви каждого узла упорядочены.

Другими словами, упорядоченным деревом мы будем называть такое дерево, в котором важен порядок следования его поддеревьев.

На рис. 22.5 представлены два различных упорядоченных дерева. Два потомка одного узла вместе с их поддеревьями поменялись местами, и получилось совершенно другое дерево. Если не считать дерево упорядоченным и поменять местами некоторых потомков, то ничего не изменится, дерево будет тем же.



Рис. 22.5. Примеры упорядочивания деревьев

Бинарное дерево – это частный случай упорядоченного дерева, т.е. в таком дереве тоже важен порядок следования поддеревьев. Причем бинарное дерево – это такое упорядоченное дерево, степень которого равна 2, т.е. каждый узел связан не более чем с двумя поддеревьями, которые называются *левыми* и *правыми поддеревьями*.

В бинарном дереве очень важно знать, какой из потомков узла находится слева, а какой справа, поэтому нам понадобилось понятие упорядоченного дерева.

Классификация бинарных деревьев

Бинарные деревья имеют унифицированную структуру, допускающую разнообразные алгоритмы прохождения и эффективный доступ к элементам.

По степени узлов бинарные деревья делятся:

- на *строгие*: узлы дерева имеют степень либо 0 (листья), либо 2 (внутренние узлы) (рис. 22.6);
- на *нестрогие*: узлы дерева имеют степень либо 0 (листья), либо 1–2 (внутренние узлы) (см. рис. 22.6),

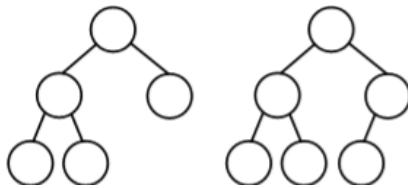


Рис. 22.6. Строгое и нестрогое деревья

- а также *полные*: все узлы дерева имеют степень 2, кроме листьев (рис. 22.7);
- *неполные*: некоторые узлы, не являющиеся листьями, имеют степень меньше 2 (см. рис. 22.7).

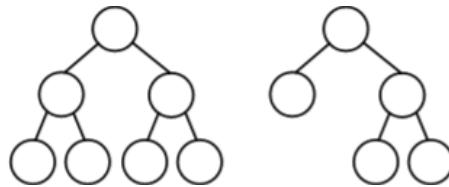


Рис. 22.7. Полное и неполное деревья

Виды бинарных деревьев

На рис. 22.8 показаны возможные комбинации видов бинарных деревьев (строгое/нестрогое и полное/неполное). Полного нестрогого дерева не существует.

	Строгое	Нестрогое
Полное		Не существует
Неполное		

Рис. 22.8. Комбинации видов бинарных деревьев

Плотность дерева

В общем случае у бинарного дерева на k -м уровне может быть до 2^{k-1} узлов (если считать, что корень – это не первый, а нулевой уровень, то будет до 2^k узлов).

Число узлов, приходящееся на уровень, является показателем плотности дерева. Интуитивно плотность есть мера величины дерева (число узлов) по отношению к высоте дерева.

Деревья с большой плотностью очень важны в качестве структур данных, так как они содержат более короткие пути от корня. Плотное дерево позволяет хранить большие коллекции данных и осуществлять эффективный доступ к элементам.

Примеры использования бинарных деревьев

Пример бинарного дерева из реальной жизни – это генеалогическое дерево с отцом и матерью человека, которые, как бы это странно ни выглядело, в бинарном дереве будут являться потомками этого человека, хотя на самом деле они его предки (рис. 22.9).

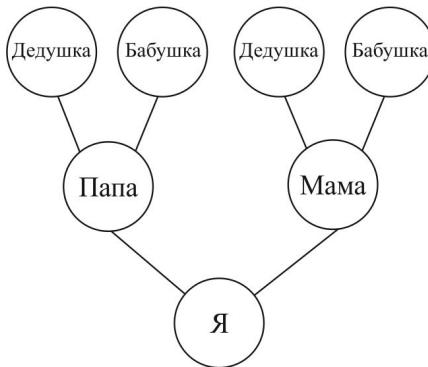


Рис. 22.9. Генеалогическое дерево
(пример бинарного дерева)

Арифметическое выражение с двухместными операциями, где каждая операция представляет собой узел с операндами в качестве поддеревьев, тоже можно показать с помощью дерева.

Например, выражение $A + B \cdot C - D/(A + B)$ можно представить в виде такого бинарного дерева на рис. 22.10.

Действия выполняются от листьев к корню (снизу вверх)

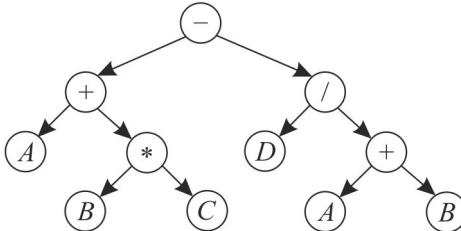


Рис. 22.10. Дерево выражения $A + B \cdot C - D/(A + B)$

22.3. Реализация бинарного дерева в C++

Заголовочный файл *tree.h* – класс *Tree*

Подготовим основные поля (рис. 22.11), конструктор, деструктор и прототипы функций в классе Tree будущего заголовочного файла. Но подробно рассмотрим каждую из еще не реализованных функций позже.

```
#pragma once
#include "stdafx.h"
#include <iostream>
using namespace std;
template <class T>
class Tree {
private:
    T data; // Данные типа Т
    Tree* left; // Указатель на узел слева
    Tree* right; // Указатель на узел справа
    Tree* parent; // Указатель на предка
public:
    Tree(T); // Конструктор
    ~Tree(); // Деструктор
    void insertLeft(T); // Вставить узел слева
    void insertRight(T); // Вставить узел справа
        // Добавить поддерево слева
    void addLeftTree(Tree<T>* tree) { left = tree; }
        // Добавить поддерево справа
    void addRightTree(Tree<T>* tree) { right= tree; }
    Tree<T>* ejectLeft(); // Извлечь поддерево слева
    Tree<T>* ejectRight(); // Извлечь поддерево справа
```

```

void deleteLeft(); // Удалить поддерево слева
void deleteRight(); // Удалить поддерево справа
void setData(Tdt) {data=dt;} // Установить данные для узла
                           /* Пройти дерево с печатью элементов
                           в прямом порядке (сверху вниз) */
static void preOrder(Tree<T>*);
                           /* Пройти дерево с печатью элементов в
                           симметричном порядке (слева направо) */
static void inOrder(Tree<T>*);
                           /* Пройти дерево с печатью элементов
                           в обратном порядке (снизу вверх) */
static void postOrder(Tree<T>*);
Tree<T>* copyTree(); // Скопировать дерево
T getData() { if (this != NULL) return data; else return
NULL; }
                           // Получить данные с узла
void printTree(int); // Печать дерева горизонтально
void printVTree(int); // Печать дерева вертикально
                           /* Прохождение по дереву и запись в файл */
void Tree<T>::obh(Tree<T>* node);
void printVTree2(); // Печать дерева вертикально (2)
int getAmountOfNodes(); // Получить количество элементов
дерева
                           /* Получить высоту дерева
                           (считает с текущего узла по направлению к листьям) */

int getHeight();
                           /* Проходить по каждому уровню дерева
                           начиная с корня и вывести элементы */
void levelScan();
void deleteTree() { delete this; } // Удалить дерево
                           /* Если дерево неполное, сделать его полным
                           (недостающие узлы приобретут данные NULL) */
Tree<T>* replaceNULLforEmpty();
Tree<T>* getLeft(); // Получить левый узел
Tree<T>* getRight(); // Получить правый узел
Tree<T>* getParent(); // Получить родителя
                           /* Найти элемент и добавить к нему слева узел */
void findElement_insertLeft(Tree<T>*, T, T);

```

```

/* Найти элемент и добавить к нему справа узел */
void findElement_insertRight(Tree<T>*, T, T);
        /* Построить идеально сбалансированное
           дерево по данному количеству элементов */
static Tree<T>* balancedTree(int n);
};

template <class T>
int getLevel(Tree<T>* tree);
#include "tree.cpp"

```

Каждый узел бинарного дерева имеет четыре поля:

- 1) данные;
- 2) указатель на предка;
- 3) указатель на левого потомка;
- 4) указатель на правого потомка.

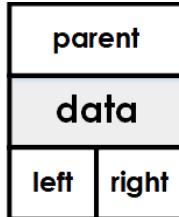


Рис. 22.11. Объект «узел» и его поля:
данные и три указателя

Конструктор узла. Деструктор поддерева

Напишем конструктор узла (он создает узел) и деструктор поддерева (он удаляет не просто узел, а еще и его потомков).

В конструкторе каждому из полей будущего узла присваиваются определенные значения. Поле data принимает значение по аргументу конструктора, а остальные поля: left, right, parent – приобретают значения NULL. Пока что они ни на что не ссылаются, т.е. данный узел не имеет предка и левого и правого потомков.

В классе есть функция удаления дерева, которая удаляет корневой элемент, тем самым вызывая деструктор. А деструктор должен удалить левого и правого потомков. Для потомков тоже работает свой деструктор, который рекурсивно, как цепная реакция, будет удалять в дальнейшем всех их потомков вплоть до полного удаления дерева:

```

template<class T>
Tree<T>::Tree(T dt) { // Конструктор узла
    data = dt; // Присвоим данные по аргументу
    left = NULL; // Обнулим все указатели
}

```

```

    right = NULL;
    parent = NULL;
}
template<class T>
Tree<T>::~Tree() { // Деструктор дерева
    delete this->left;
    delete this->right;
}

```

Вставка узла слева

```

template <class T>
void Tree<T>::insertLeft(T dt) {
    Tree<T> node = new Tree(dt);
    if (this->left != NULL)
        /* Сделать так, чтобы его предком был новый узел */
        this->left->parent = node;
        /* Теперь у нового узла левый
           потомок – это левый потомок вызывающего узла */
        node->left = this->left;
        /* А вызывающий узел теперь имеет
           левого потомка – это новый узел */
        this->left = node;
        /* Указываем, что у нового узла теперь
           предком является текущий узел */
    node->parent = this;
}

```

На рис. 22.12 схематично показано добавление узла слева посредством изменения значения полей вызывающего (элемент *A*), добавляемого (элемент *D*) узлов и того узла, который был левым потомком вызывающего узла изначально (элемент *B*).

Нулевой пункт показывает первоначальное положение узлов: узел *A* – корневой для узлов *B* и *C*, являющихся левым и правым потомками соответственно.

На первом шаге выделяется память для узла *D*, не являющегося ничьим предком и ничьим потомком.

Если необходимо добавить его слева от узла *A* (он же вызывающий), нужно узнать, есть ли сейчас слева от узла *A* какой-либо

узел. Если есть, то сделаем так, чтобы у этого узла предком стал добавляемый узел. Так, на рис. 22.12 на втором шаге родителем узла B становится узел D .

На третьем шаге закрепляем информацию о том, что левый потомок добавляемого узла D – это узел B , путем смены значения $left$ узла D на соответственное.

Далее, нужно, чтобы D стал левым потомком узла A . Для этого $left$ узла A теперь будет ссылаться на узел D (шаг 4).

И наконец, делаем родителем узла D узел A (шаг 5).

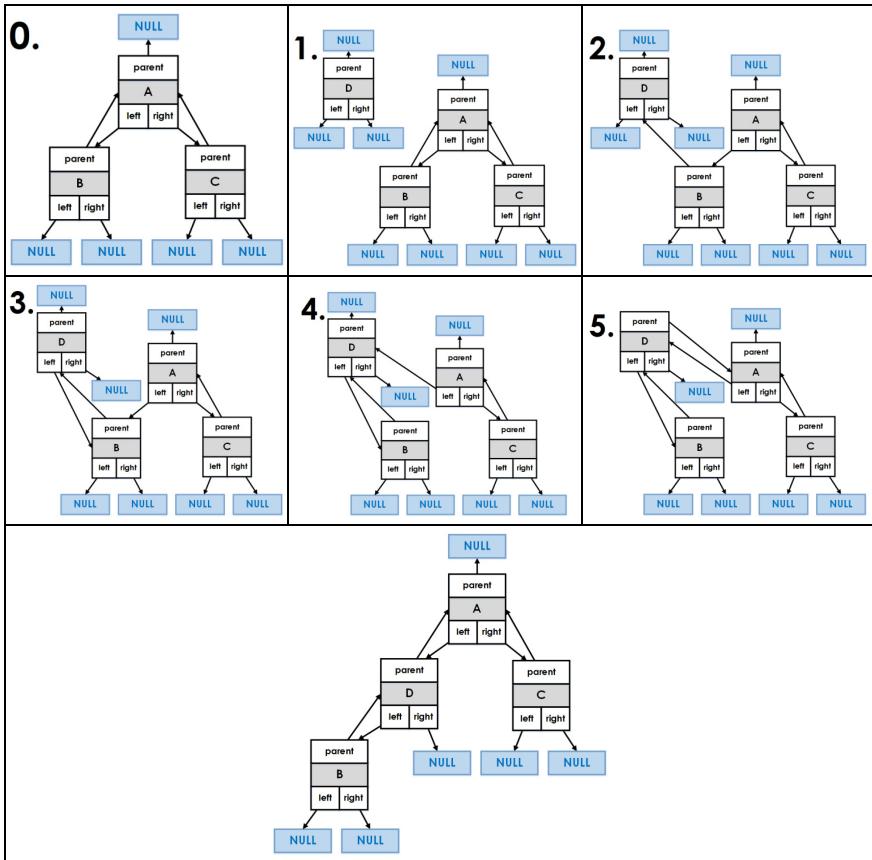


Рис. 22.12. Поэтапная схема добавления узла слева

Вставка узла справа

Аналогичным образом работает функция для добавления узла справа (рис. 22.13):

```
template<class T>
void Tree<T>::insertRight(T dt) {
    Tree<T> node = new Tree(dt);
    if (this->right != NULL)
        this->right->parent = node;
    node->right = this->right;
    this->right = node;
    node->parent = this;
}
```

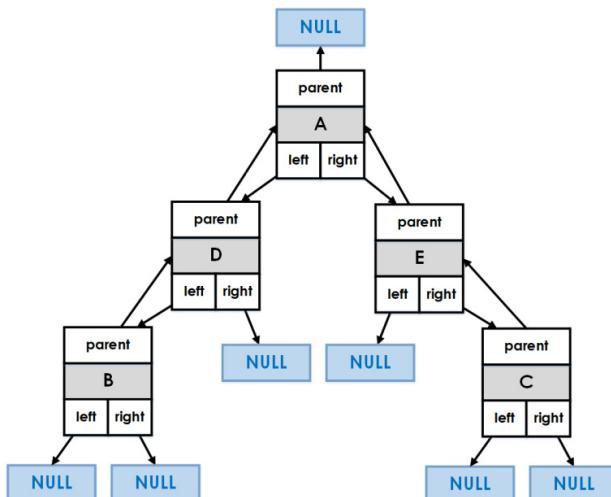


Рис. 22.13. Добавление узла справа

Ввод в файле main.cpp

Добавление узлов дерева с помощью двух описанных до этого функций осуществляется довольно просто. Чтобы добавить новый узел к определенному узлу, воспользуемся функциями, реализованными еще в файле класса, – `getLeft()` и `getRight()`, возвращающими узлы слева и справа соответственно.

Дерево на рис. 22.14 соответствует коду ниже. Сначала создается узел `A` (корень), затем добавляется слева узел `B`, затем получа-

ем узел слева от корня и к нему добавляем узел D . Остальные узлы дерева добавляются аналогично:

```
Tree<int>* tree = newTree<int>('A');
tree->insertLeft('B');
tree->getLeft()->insertLeft('D');
tree->getLeft()->getLeft()->insertLeft('G');
tree->getLeft()->insertRight('E');
tree->getLeft()->getLeft()->insertRight('H');
tree->insertRight('C');
tree->getRight()->insertRight('F');
tree->getRight()->getRight()->insertLeft('I');
tree->getRight()->getRight()->insertRight('J');
```

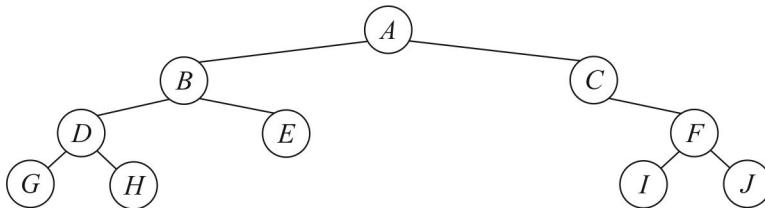


Рис. 22.14. Вывод структуры дерева

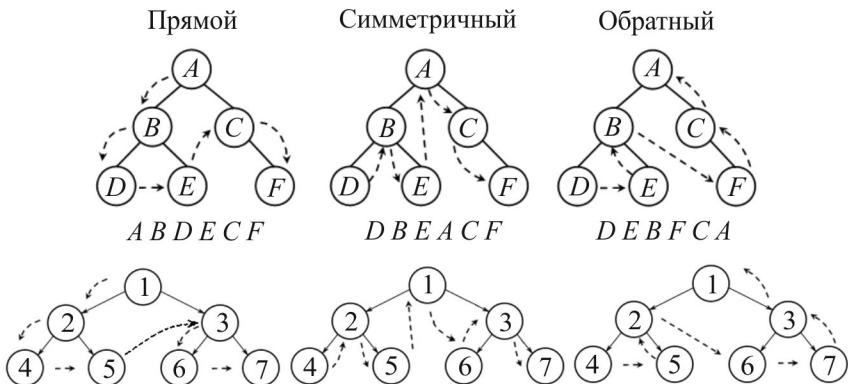
Обход дерева

Обход дерева – это посещение всех узлов дерева в определенном порядке, причем каждый узел посещается только один раз.

При обходе все вершины дерева должны посещаться в определенном порядке. Существует несколько способов обхода всех вершин дерева. Можно выделить три наиболее часто используемых способа обхода дерева (рис. 22.15):

- прямой;
- симметричный;
- обратный.

Существует множество задач, которые можно выполнять на деревьях. Одна из них весьма распространенная – выполнение заданной операции с каждый элементом дерева. Чтобы отредактировать какие-либо или даже все элементы дерева, нужно совершить обход дерева.



Прямой обход (сверху вниз)

Совершим прямой обход по дереву подобно и с комментариями по рис. 22.16.

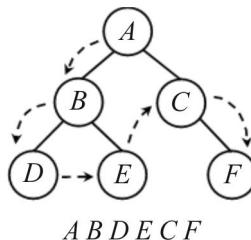


Рис. 22.16. Прямой обход дерева

- Сначала посещается корень – A .

Далее осуществляется переход вниз, при этом необходимо придерживаться левой стороны дерева:

- Посещается верхний узел левого поддерева – B .
- Выполняется проход по левому потомку этого элемента – D .

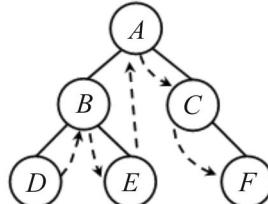
Он оказывается листом, т.е. ниже двигаться нет возможности, поэтому:

- Выполняется переход к правому потомку верхнего узла этого же поддерева – E .

Левое поддерево узла A полностью пройдено, начинается обход правого поддерева:

- Через узел C .
- Достигнут нижний узел – лист F .

Симметричный обход (слева направо)



$D \ B \ E \ A \ C \ F$

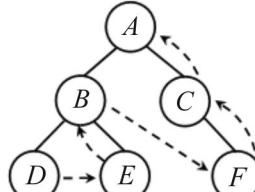
Рис. 22.17. Симметричный обход

• Начало с самого нижнего левого листа – D (рис. 22.17). Чтобы продвигаться направо, нужно пройти через предка этого листа:

- Совершается проход наверх к предку – B . У предка есть еще потомок, поэтому нужно посетить и его:
- Посещается узел E . Узел B и все его потомки пройдены, поэтому идет движение вверх:

- Посещается корень – A . У него тоже есть потомок:
- Узел C .
- У которого есть потомок – лист F .

Обратный обход (снизу вверх)



$D \ E \ B \ F \ C \ A$

Рис. 22.18. Обратный обход дерева

Обратный обход (рис. 22.18) отличается от предыдущего симметричного тем, что относительно определенного узла к другому потомку одного предка не нужно сначала проходить через предка, а надо сразу идти к его потомку. Затем следуем к проигнорированному предку. Например:

- Вновь начало с самого нижнего левого листа – D . Чтобы продвигаться направо, в симметричном обходе в предыдущем случае нужно было пройти через предка этого листа.

- В обратном обходе этого не требуется.
- Так что необходимо перейти к узлу E .
- Направляемся к узлу – предку этих потомков – B .

Левое поддерево пройдено. Обход через корень игнорируется, так что начало будет с самого низа его правого поддерева:

- Лист F .
- Выше – C .
- Конец на корне A .

Функции прямого, симметричного, обратного обходов дерева

Эти три метода прохождения дерева можно легко реализовать в виде рекурсий.

В каждой функции имеется аргумент – текущий узел. Если этот узел не NULL, т.е. существует, то выведем его, а также спустимся к его левому либо правому потомку.

Главную роль в каждой из функций прохождения играет порядок выполнения описанных до этого операций с узлом, т.е. очень важно и уникально для каждой функции прохождения дерева то, в какой последовательности эти действия будут располагаться: вывод, переход к левому или правому поддереву:

```
// ПРЯМОЙ ОБХОД (СВЕРХУ ВНИЗ)
template<class T>
void Tree<T>::reorder(Tree<T> *node) {
    if (node != NULL) {
        cout << node->getData() << " ";
        reorder(node->left);
        reorder(node->right);
    }
}
// СИММЕТРИЧНЫЙ ОБХОД (СЛЕВА НАПРАВО)
```

```

template<class T>
void Tree<T>::inOrder(Tree<T> *node) {
    if (node != NULL) {
        inOrder(node->left);
        cout <<node->getData() <<““;
        inOrder(node->right);
    }
}
// ОБРАТНЫЙ ОБХОД (СНИЗУ ВВЕРХ)
template<class T>
void Tree<T>::postOrder(Tree<T> *node) {
    if (node != NULL) {
        postOrder(node->left);
        postOrder(node->right);
        cout<<node->getData() <<““;
    }
}

```

Извлечение левого/правого поддерева и его удаление

Чтобы извлечь, например, левое поддерево, нужно нарушить связь между вызывающим узлом и его левым потомком. В итоге вызывающий узел лишится левого под дерева, а функция вернет корень этого извлеченного под дерева (рис. 22.19).

Чтобы удалить поддерево слева, нужно сначала его извлечь (для благополучного изменения связей), а затем удалить его корень.

Аналогичные действия производятся с извлечением и удалением правого под дерева:

```

template<class T>
Tree<T>* Tree<T>::ejectLeft() { // Извлечение
    if (this->left != NULL) {
        Tree<T>* temp = this->left;
        this->left = NULL;
        return temp;
    }
    return NULL;
}
template<class T>
void Tree<T>::deleteLeft() { // Удаление
    Tree<T>* temp = this->ejectLeft();

```

```

    delete temp;
}
template<class T>
Tree<T>* Tree<T>::ejectRight() {
    if (this->right != NULL) {
        Tree<T>* temp = this->right;
        this->right = NULL;
        return temp;
    }
    return NULL;
}
template<class T>
void Tree<T>::deleteRight() {
    Tree<T>* temp = this->ejectRight();
    delete temp;
}

```

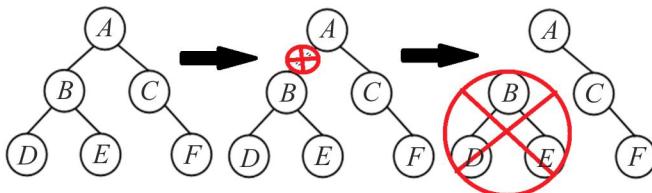


Рис. 22.19. Извлечение и удаление поддерева

Печать дерева, развернутого на 90° против часовой стрелки и отраженного по вертикали

Данный вид печати дерева использует симметричный метод прохождения дерева.

Для того чтобы напечатать дерево горизонтально, понадобится рекурсивная функция, изменяющимся аргументом которой является некоторое число, отвечающее за то, сколько пробелов будет выведено перед данными, содержимым текущего узла. Причем чем глубже совершается проход по дереву, соответственно, чем больше уровеней, тем больше пробелов будет напечатано для данного узла.

Схематично симметричный обход по дереву (рис. 22.20) показан на рис. 22.21, а его печать – на рис. 22.22:

```

void Tree<T>::printTree(int level) {
    if (this != NULL) {
        this->left->printTree(level+1);
        for (int i = 1; i < level; i++) cout << " ";
        cout << this->getData() << endl;
        this->right->printTree(level + 1);
    }
}

```

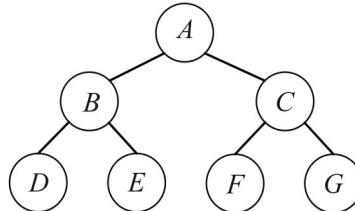


Рис. 22.20. Дерево для теста горизонтальной печати

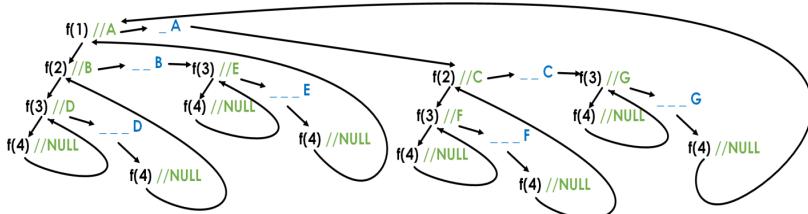


Рис. 22.21. Схема прохождения по дереву с печатью узлов

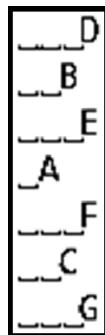


Рис. 22.22. Горизонтальная печать дерева (вывод):
нижние подчеркивания обозначены символом пробела

Стоит заметить, что при выводе левое поддерево всегда будет печататься выше правого. Кроме того, если все узлы хранят в себе максимум один символ в качестве данных, печать будет в классическом виде.

Вертикальная печать будет рассмотрена отдельно позже.

Копирование дерева

Функция копирования принимает корень исходного дерева и создает его дубликат (рис. 22.23).

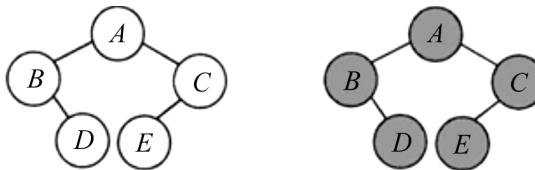


Рис. 22.23. Копирование дерева: оригинал и дубликат

Она использует для посещения узлов обратный метод прохождения, гарантирующий, что спустимся по дереву на максимальную глубину, прежде чем начнем операцию посещения, которая создает узел для нового дерева (рис. 22.24).

Таким образом, функция копирования строит новое дерево снизу вверх:

```
Tree<T>* Tree<T>::copyTree() {
    Tree<T>* tree = new Tree<T> (this->data);
    if (this->parent != NULL)
        tree->parent = this->parent;
    if (this->left != NULL)
        tree->left = this->left->copyTree();
    if (this->right != NULL)
        tree->right = this->right->copyTree();
    return tree;
}
```

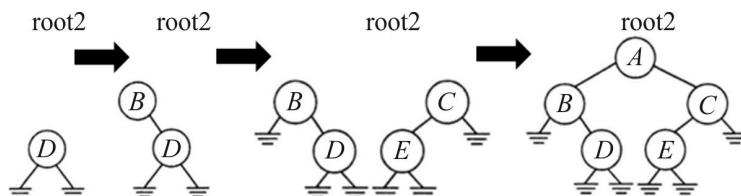


Рис. 22.24. Схема копирования дерева поэтапно

22.4. Бинарные деревья поиска

Определение и свойства

Бинарные деревья часто используются для представления множества данных, элементы которого находятся по уникальному ключу.

Бинарное дерево поиска – это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева, левое и правое, являются бинарными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X .
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X .

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения «меньше».

Два бинарных дерева поиска представлены на рис. 22.25 и 22.26.

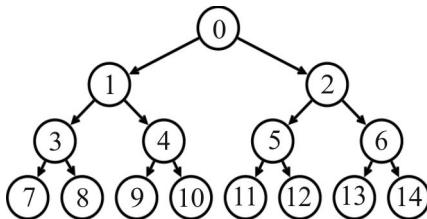


Рис. 22.25. Первый пример бинарного дерева поиска

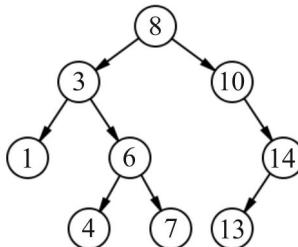


Рис. 22.26. Второй пример бинарного дерева поиска

22.5. Программа для работы с бинарным деревом поиска

Заголовочный файл tree.h – класс SearchTree

Можно не рассматривать функции класса SearchTree, которые по принципу работы повторяют или достаточно похожи на ранее разобранные функции класса Tree.

В связи с тем, что ранее многое уже было рассказано, для простоты создадим новый класс и объявим прототипы только части функций, аналогичных функциям класса Tree, плюс некоторые новые, с которыми нам предстоит познакомиться в дальнейшем:

```
template <class S>
class SearchTree {
public:
    S data; // Данные типа Т
    SearchTree* left; // Указатель на узел слева
    SearchTree* right; // Указатель на узел справа
    SearchTree* parent; // Указатель на предка
    SearchTree(S); // Конструктор
    ~SearchTree(); // Деструктор
    void deleteSearchTree() { delete this; } // Удалить дерево
    void printSearchTree(int); // Горизонтальная печать дерева
    void inOrder(SearchTree<S*>*); // Симметричный обход дерева
    void setData(Sdt) {data = dt;} // Установить данные для узла
    SearchTree<S*>* next(); // Найти следующий элемент
    SearchTree<S*>* prev(); // Найти предыдущий элемент
    void insertNode(S); // Вставить узел
    void deleteNode(S); // Удалить узел
    SearchTree<S*>* findElement(S); // Найти элемент
    SearchTree<S*>* findMax(); // Найти максимум
    SearchTree<S*>* findMin(); // Найти минимум
};
```

Поиск элемента

Для поиска элемента в бинарном дереве поиска можно воспользоваться следующей функцией, которая принимает в качестве параметра искомый ключ. Для каждого узла функция сравнивает значение его ключа с искомым ключом. Если ключи одинаковы, то функция возвращает текущий узел, в противном случае функция вызывается рекурсивно для левого или правого поддеревьев (рис. 22.27):

```
template <class S>
SearchTree<S>* SearchTree<S>::findElement(S dt) {
    if ((this == NULL) || (dt == this->data))
        return this;
    if (dt < this->data) return this->left->findElement(dt);
    else return this->right->findElement(dt);
}
```

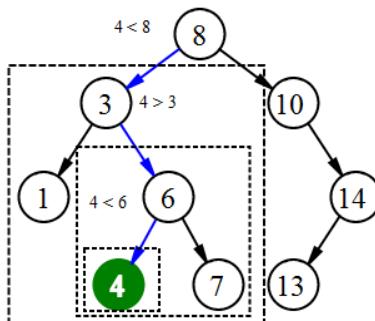


Рис. 22.27. Схема нахождения элемента 4
для данного дерева

Поиск минимального/максимального элемента по ключу

Минимальный/максимальный элемент – самый левый/правый элемент поддерева с данным корнем.

Чтобы найти минимальный элемент в бинарном дереве поиска, необходимо просто следовать указателям `left` от корня дерева, пока не встретится значение `NULL`. Если у вершины есть левое поддерево, то по свойству бинарного дерева поиска в нем хранятся все элементы с меньшим ключом. Если его нет, значит, эта вершина и есть минимальная.

Аналогично находится и максимальный элемент. Для этого нужно следовать правым указателям:

```
template <class S>
SearchTree<S>* SearchTree<S>::findMin() {
    if (this->left == NULL) return this;
    return this->left->findMin();
}
template <class S>
SearchTree<S>* SearchTree<S>::findMax() {
    if (this->right == NULL) return this;
    return this->right->findMax();
}
```

Нахождение предыдущего/следующего элемента (по ключу)

Если у узла есть правое поддерево, то следующий за ним элемент будет минимальным элементом в этом поддереве. Если у него нет правого под дерева, то нужно следовать вверх, пока не встретим узел, который является левым дочерним узлом своего родителя. Поиск предыдущего выполняется аналогично. Если у узла есть левое поддерево, то следующий за ним элемент будет максимальным элементом в этом поддереве. Если у него нет левого поддерева, то нужно следовать вверх, пока не встретим узел, который является правым дочерним узлом своего родителя.

В листинге ниже первая функция ищет следующий элемент, а вторая – предыдущий:

```
template <class S>
SearchTree<S>* SearchTree<S>::next() {
    SearchTree* tree = this;
    if (tree->right != NULL)
        return tree->right->findMin();
    SearchTree<S>* t = tree->parent;
    while ((t!=NULL) && (tree==t->right)) {
        tree = t;
        t = t->parent;
    }
}
```

```

    return t;
}
template <class S>
SearchTree<S>* SearchTree<S>::prev() {
    SearchTree* tree = this;
    if (tree->left != NULL)
        return tree->left->findMax();
    SearchTree<S>* t = tree->parent;
    while ((t != NULL) && (tree == t->left)) {
        tree = t;
        t = t->parent;
    }
    return t;
}

```

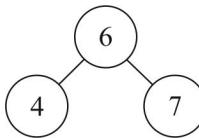


Рис. 22.28. Пример родителя с двумя потомками

Следующий для 4 – это 6, следующий для 6 – это 7, следующий для 7 – это NULL (не существует). Предыдущий для 4 – это NULL (не существует), предыдущий для 6 – это 4, следующий для 7 – 6 (рис. 22.28).

Вставка узла

Как уже было сказано ранее, бинарное дерево поиска – это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше или равно значению родителя для каждого узла дерева. Иными словами, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла порядок дерева сохраняется.

На основе этого создадим функцию добавления узлов (рис. 22.29):

```

template <class S>
void SearchTree<S>::insertNode(S dt) {
    SearchTree<S>* tree = this;
    while (tree != NULL) {
        if (dt >= tree->data) {
            if (tree->right != NULL) {
                tree = tree->right;
            } else {
                SearchTree<S>* t = new SearchTree<S>(dt);
                t->parent = tree;
                tree->right = t;
                break;
            }
        }
        else if (dt < tree->data) {
            if (tree->left != NULL) {
                tree = tree->left;
            } else {
                SearchTree<S>* t = new SearchTree<S>(dt);
                t->parent = tree;
                tree->left = t;
                break;
            }
        }
    }
}

```

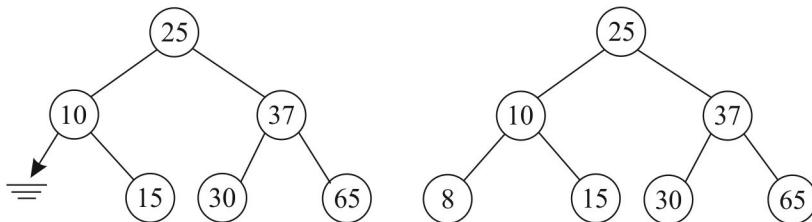


Рис. 22.29. Пример вставки узла до и после узла 8

Код демонстрирует, как с помощью вставок узла можно составить и вывести горизонтально бинарное дерево поиска (рис. 22.30, 22.31):

```

SearchTree<int>* tree = new SearchTree<int>(2);
tree->insertNode(1);
tree->insertNode(3);
tree->insertNode(6);
tree->insertNode(4);
tree->insertNode(7);
tree->insertNode(1);
tree->printSearchTree(1);

```

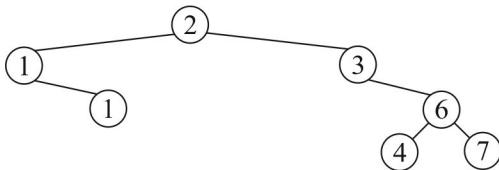


Рис. 22.30. Пример бинарного дерева
для вывода на консоль

1
1
2
3
4
6
7

Рис. 22.31. Пример вывода
бинарного дерева
горизонтально

Удаление узла

При удалении разрывается связь с удаляемым узлом, а все узлы поддерева удаляемого элемента заново вставляются в исходное дерево.

Для удаления узла из бинарного дерева поиска нужно рассмотреть три возможные ситуации:

- у узла нет потомков;
- у узла один потомок;
- у узла два потомка.

1. У узла нет потомков

Если у узла нет потомков, то у его предка нужно просто заменить указатель на NULL:

```

template<class S>
voidSearchTree<S>::deleteNode(S dt) {
    SearchTree<S>* e = this->findElement(dt);
    SearchTree<S>* p = e->parent; //предок удаляемого элемента

```

```

// Первый случай: удаляемый элемент не имеет потомков
if ((e->left == NULL) && (e->right == NULL)) {
    if (p->left == e) p->left = NULL;
    if (p->right == e) p->right = NULL;
}

```

На рис. 22.32 показано удаление узла, содержащего в себе число 13, не имеющего потомков, которое заключается в удалении из памяти данного узла и присваивании соответствующему (left/right) полю родительского узла 14 значения NULL.

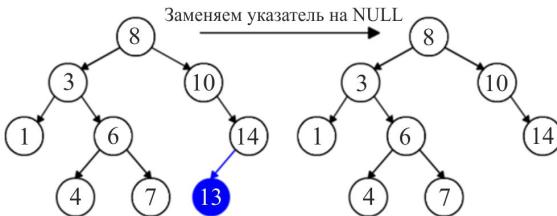


Рис. 22.32. Удаление узла, содержащего число 13

2. У узла один потомок

Если у узла есть только один потомок, то нужно создать новую связь между вызывающим узлом и потомком удаляемого узла:

```

// Второй случай: удаляемый элемент имеет одного потомка
else if ((e->left == NULL) || (e->right == NULL)) {
    if (e->left == NULL) {
        if (p->left == e) p->left = e->right;
        else p->right = e->right;
        e->right->parent = p;
    }
    перенос части кода

```

```

else {
    if (p->left == e) p->left = e->left;
    else p->right = e->left;
    e->left->parent = p;
}
}

```

Нужно учесть, что левый потомок удаляемого элемента (если работаем именно с ним) должен стать левым потомком предка удаляемого элемента, если удаляемый элемент – левый потомок своего предка. В противном случае, если удаляемый элемент находится справа от своего предка, левый потомок удаляемого элемента будет правым потомком предка удаляемого. И учтем, что левый потомок удаляемого элемента приобретет нового предка – предка удаляемого элемента.

Если же работаем с правым потомком удаляемого элемента, то аналогично ситуации, рассмотренной до этого для левого потомка, он должен стать левым потомком предка удаляемого элемента, если удаляемый элемент – левый потомок своего предка. И иначе: он будет правым потомком предка удаляемого. Изменим родителя правому потомку удаляемого элемента – теперь это предок удаляемого элемента.

Не имеет значения, находится этот потомок удаляемого элемента слева или справа. Но в зависимости от того, каким потомком (левым или правым) является удаляемый элемент, будет поставлен потомок этого удаляемого элемента относительно предка удаляемого элемента.

На рис. 22.33 удаляется узел, содержащий число 14, который имеет левого потомка – узел с числом 13. После удаления узла с числом 14 узел с 13 стал правым потомком узла с числом 10, бывшего предка узла с числом 14.

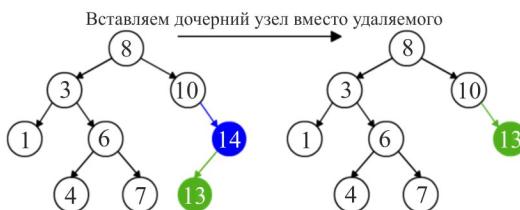


Рис. 22.33. Удаление узла, содержащего число 14

3. У узла два потомка

Если у узла два потомка, то нужно найти следующий за ним элемент (у этого элемента не будет левого потомка), его правого

потомка нужно поместить на место найденного элемента, а удаляемый узел заменить найденным узлом. Таким образом, свойство бинарного дерева поиска не будет нарушено:

```
// Третий случай: удаляемый элемент имеет двух потомков
else {
    SearchTree<S>* s = e->next(); // Следующий элемент за удаляемым
    e->data = s->data;
    if (s->parent->left == s) {
        s->parent->left = s->right;
        if (s->right != NULL)
            s->right->parent = s->parent;
    } else {
        s->parent->right = s->right;
        if (s->right != NULL)
            s->right->parent = s->parent;
    }
}
```

Рассмотрим, как работает алгоритм, представленный кодом листинга выше, на данном примере (рис. 22.34).

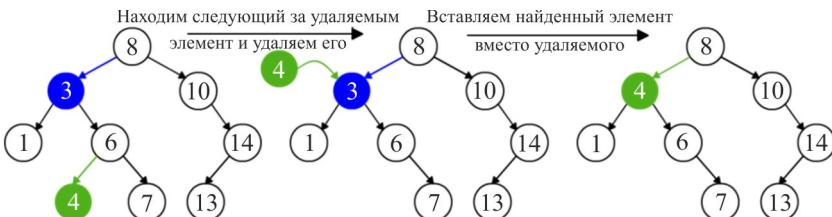


Рис. 22.34. Удаление узла, содержащего число 3

Во второй строке кода создается копия соединяющего элемента, являющегося следующим после удаляемого. На рисунке для удаляемого узла, содержащего число 3, это узел, содержащий число 4, как минимальный элемент правого поддерева корня с числом 3.

В третьей строке данные удаляемого элемента меняются на данные следующего. Происходит замещение элементов. Так, на рисунке значение 3 меняется на значение 4.

В четвертой строке выясняется, является ли следующий узел левым потомком своего предка. В данном примере следующий узел, содержащий число 4, действительно был левым потомком узла, содержащего число 6.

В таком случае в пятой строке левым потомком родительского элемента 6 станет правый потомок следующего узла, содержащего 4. Иными словами, что было справа от узла с числом 4, будет слева от узла с числом 6. Слева от узла, содержащего число 4, никогда ничего не будет, потому что он является следующим после узла с числом 3, поэтому минимальным в своем поддереве.

В шестой строке проверяется, есть ли у следующего узла, содержащего 4, что-либо справа, какое-либо поддерево.

В седьмой строке, если бы это условие выполнялось, корень данного поддерева стал бы иметь нового предка – в данном случае узел с числом 6.

Последующие строки выполняются аналогично для ситуации, когда следующий элемент является правым потомком своего предка.

22.6. Идеально сбалансированное дерево

Определение и свойства

Идеально сбалансированное бинарное дерево – такое дерево, у которого для каждого его узла количество узлов в левом и правом поддеревьях различается не более чем на 1 (рис. 22.35). Такое дерево будет иметь минимальную высоту при данном числе узлов (не более чем $\log_2 n$, где n – число узлов).

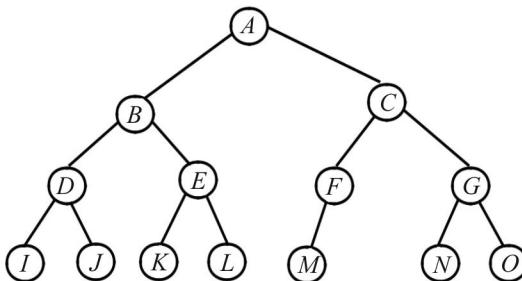


Рис. 22.35. Идеально сбалансированное бинарное дерево

Для достижения минимума высоты нужно располагать по максимуму узлы на всех уровнях, кроме самого нижнего. Это можно сделать очень просто, если располагать узлы поровну слева и справа от каждого узла.

Поиск по идеально сбалансированному бинарному дереву поиска потребует не более $\log_2 n$ сравнений.

Построение идеально сбалансированного дерева

Попробуем сформулировать это правило с помощью рекурсии: (n – данное число узлов):

- Взять один узел в качестве корня.
- Построить левое поддерево с $n_{\text{left}} = n/2$ (деление по модулю) узлами тем же способом.
- Построить правое поддерево с $n_{\text{right}} = n - 1 - n_{\text{left}}$ узлами тем же способом.

Перенос части кода вниз:

```
template <class T>
Tree<T>* Tree<T>::balancedTree(int n) {
    if (n == 0) return NULL;
    cout << "data=";
    T dt;
    cin >> dt;
    tree = new Tree<T>(dt);
    tree->addLeftTree(balancedTree(n / 2));
    tree->addRightTree(balancedTree(n - n / 2 - 1));
    return tree;
}
```

Обратите внимание, что узлы включаются в дерево при обратном ходе в порядке, обратном их формированию, т.е. снизу вверх (от листьев к корню).

Код ниже осуществляет ввод количества узлов, создание идеально сбалансированного дерева по данному значению и вывод дерева на экран (рис. 22.36):

```
cout << "n=";
int n;
cin >> n;
tree = Tree<int>::balancedTree(n);
tree->printVTree(1);
```

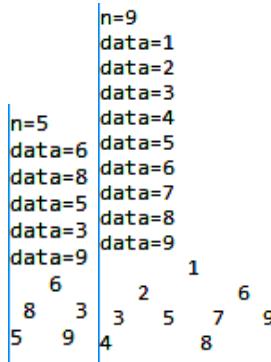


Рис. 22.36. Результат работы программы
(построение идеально сбалансированного дерева)

22.7. Представление бинарного дерева через массив и сортировка

Бинарные деревья, представляемые массивами

До этого для построения дерева мы непосредственно использовали функции, работающие с добавлением узлов в дереве.

Деревья можно представить в виде массива. Рассмотрим рис. 22.37. Массив A с элементами $\{5, 1, 3, 9, 6, 2, 4, 7, 0, 8\}$ есть последовательный список, элементы которого могут представлять узлы бинарного дерева с корнем $A[0]$; потомками первого уровня $A[1]$ и $A[2]$; потомками второго уровня $A[3], A[4], A[5]$ и $A[6]$ и т.д. Корневой узел имеет индекс 0, а всем остальным узлам индексы назначаются в порядке, определяемом поперечным (уровень за уровнем) методом прохождения.

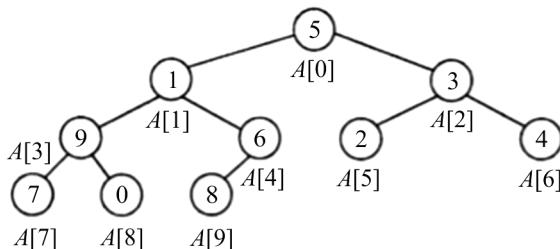


Рис. 22.37. Массив и соответствующее ему дерево

Недостатки представления бинарного дерева в виде массива

Несмотря на то, что массивы обеспечивают естественное представление деревьев, возникает проблема, связанная с отсутствующими узлами, которым должны соответствовать неиспользуемые элементы массива. В данном примере выше массив имеет четыре неиспользуемых элемента, т.е. треть занимаемого деревом пространства (рис. 22.38).

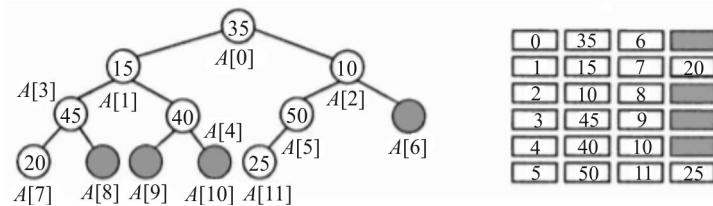


Рис. 22.38. Бинарное дерево и эквивалент в виде массива

Вырожденное дерево (рис. 22.39), у которого есть только правые поддеревья, имеет более неэффективное представление в виде массива.

Вырожденное дерево – особая форма дерева, у которого есть единственный лист E (см. рис. 22.39) и каждый внутренний узел и корень имеет только одного потомка. Вырожденное дерево эквивалентно связанному списку.

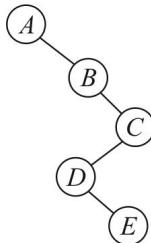


Рис. 22.39. Пример вырожденного дерева

Преимущества представления бинарного дерева в виде массива

Преимущества представляемых массивами деревьев проявляются тогда, когда требуется прямой доступ к узлам. Индексы, идентифицирующие потомков и родителя данного узла, вычисляются просто.

В табл. 22.1 для каждого узла $A[i]$ в массиве из N элементов индекс его сыновей вычисляется по следующим формулам:

- индекс левого потомка = $2 \cdot i + 1$ (не определен при $2 \cdot i + 1 >= N$);
- индекс правого потомка = $2 \cdot i + 2$ (не определен при $2 \cdot i + 2 >= N$).

Поднимаясь от потомков к родителю, можно заметить, что родителем узлов $A[3]$ и $A[4]$ является $A[1]$, родителем $A[5]$ и $A[6]$ – $A[2]$ и т.д. Общая формула для вычисления родителя для узла $A[i]$ следующая:

- Индекс родителя $(i - 1)/2$ (не определен при $i = 0$).

Таблица 22.1

Связь узлов дерева с их потомками

Уровень	Родитель	Значение	Левый сын	Правый сын
0	0	$A[0] = 5$	1	2
1	1	$A[1] = 1$	2	4
	2	$A[2] = 3$	5	6
2	3	$A[3] = 9$	7	8
	4	$A[4] = 6$	9	10 = NULL
	5	$A[5] = 2$	11 = NULL	12 = NULL
	6	$A[6] = 4$	13 = NULL	14 = NULL
3	7	$A[7] = 7$	–	–
	8	$A[8] = 0$	–	–
	9	$A[9] = 8$	–	–

Ввод-вывод дерева через массив

Создадим рекурсивные функции, комбинирующие поиск элемента с данными dt1 и добавление узла слева либо справа в зависимости от необходимости с данными dt2. Они используют симметричный обход по дереву:

```
template <class T>
void Tree<T>::findElement_insertLeft(Tree<T>* node, Tdt1,
Tdt2) {
    if (node != NULL) {
        findElement_insertLeft(node->getLeft(), dt1, dt2);
        if (node->getLeft() == NULL) {
            if (dt1 < node->getValue())
                node->setLeft(new Tree<T>(dt1));
            else
                node->setLeft(new Tree<T>(dt2));
        }
        findElement_insertLeft(node->getRight(), dt1, dt2);
    }
}
```

```

    if (dt1 == node->getData()) node->insertLeft(dt2);
    findElement_insertLeft(node->getRight(), dt1, dt2);
}
}

template <class T>
void Tree<T>::findElement_insertRight(Tree<T>* node, Tdt1,
Tdt2) {
    if (node != NULL) {
        findElement_insertRight(node->getLeft(), dt1, dt2);
        if (dt1 == node->getData()) node->insertRight(dt2);
        findElement_insertRight(node->getRight(), dt1, dt2);
    }
}

```

В коде ниже будет производиться составление дерева по данному массиву с использованием определенных до этого формул нахождения левого и правого потомков:

```

vector<int> arr = {19, 33, 13, 17, 3, 31, 35, 18, 15};
Tree<int>* tree = new Tree<int>(arr.at(0));
for (int i = 0; i < arr.size(); i++) {
    int left = 2 * i + 1;
    int right = left + 1;
    if (left < arr.size()) {
        tree->findElement_insertLeft(tree, arr.at(i),
arr.at(left));
    }
    if (right < arr.size()) {
        tree->findElement_insertRight(tree, arr.at(i),
arr.at(right));
    }
}
tree->printVTree(2);

```

На рис. 22.40 показано, каким будет это дерево.

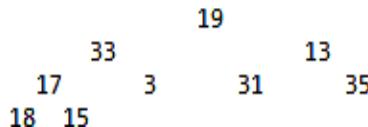


Рис. 22.40. Результат работы программы
(ввод-вывод дерева через массив)

Сортировка с помощью бинарного дерева поиска

Сортировка с помощью двоичного дерева – универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска по ключам массива (списка), с последующей сборкой результирующего массива путем обхода узлов построенного дерева в необходимом порядке следования ключей.

Алгоритм в целом следующий:

1. Построение двоичного дерева.
2. Сборка результирующего массива путем обхода узлов в необходимом порядке следования ключей.

Построим бинарное дерево поиска по массиву, составленному из чисел, расположенных в случайном порядке (рис. 22.41).

19 33 13 17 3 31 35 18 15

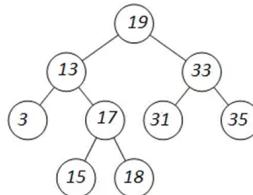


Рис. 22.41. Бинарное дерево поиска,
построенное из массива

Обойдем дерево с помощью симметричного обхода. При этом получим отсортированную последовательность чисел (рис. 22.42).

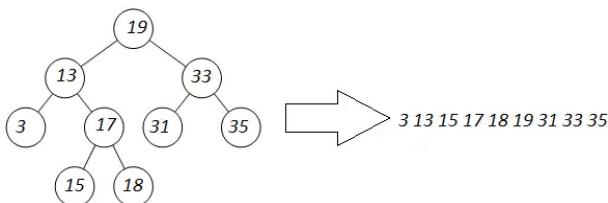


Рис. 22.42. Симметричный обход дерева

В коде ниже задается массив, и по нему в дальнейшем создается бинарное дерево поиска, симметричный обход и вывод которого предоставит нам отсортированную последовательность чисел [16]:

```
vector<int> arr = {19, 33, 13, 17, 3, 31, 35, 18, 15};  
SearchTree<int>* tree = newSearchTree<int>(arr.at(0));  
for (int i = 1; i < arr.size(); i++) {  
    tree->insertNode(arr.at(i));  
}  
tree->inOrder(tree);
```

22.8. Вертикальная печать дерева

Горизонтальная печать дерева очень проста и удобна. С помощью нее можно выводить достаточно большие деревья, несмотря на большие значения высоты дерева и количество листов.

Правда, к сожалению, вариант печати, показанный ранее, демонстрирует дерево не совсем верно. Мало того, что дерево развернули на 90° , так ему еще и поменяли местами левое и правое поддеревья, в результате чего выводится дерево, отраженное относительно оригинала.

У вертикальной печати вывод дерева такой, каким все его обычно представляют. Корень сверху, а остальные узлы вплоть до листьев располагаются ниже в своем порядке. Функция, печатающая дерево вертикально, довольно сложная, к тому же прибегает к еще не разобранным функциям:

- `voidTree<T>::printVTree(int k)` печатает дерево вертикально (k – коэффициент расширения печатаемого дерева, о котором подробнее будет рассказано позже).

Кроме функции печати, будет разобрано еще несколько полезных функций, которые в будущем нам могут пригодиться:

- `Tree<T>* Tree<T>::replaceNULLforEmpty()` превращает неполное дерево в полное, добавляя недостающие узлы с данными `NULL` (высота дерева не меняется!);
- `intTree<T>::getHeight()` возвращает высоту дерева (начинает отсчет с данного узла в качестве корня до листьев);
- `intTree<T>::getAmountOfNodes()` возвращает количество узлов в дереве (`NULL` не в счет).

Другие вспомогательные функции (без прототипов в классе):

- `Tree<T>* replace_help(Tree<T>* node, int h)` – рекурсивная функция, используемая функцией под пунктом 1 из предыдущего перечня;
 - `int getLevel(Tree<T>* tree)` возвращает уровень текущего узла (начинает отсчет с данного узла в качестве внутреннего узла / листа до корня);
 - `int getPos(int index, int width, int curLevel, int maxLevel)` вычисливает количество пробелов для данного узла, которые нужно поставить перед ним при выводе.

Кроме того, для ознакомления с `printVTree(int k)`, чтобы лучше понять, рассмотрим следующую функцию:

- `voidTree<T>::levelScan()` выводит элементы уровень за уровнем.

Более того, в некотором месте основной функции будут найдены координаты элементов: строка, в которой нужно распечатать элемент (его уровень), и количество пробелов, если считать с самого начала строки (но не позиции предыдущего элемента). На будущее это может пригодиться при визуализации дерева, так как этим способом можно узнать координаты элементов – *x* (число пробелов) и *y* (строка).

Поперечное прохождение

```
#include<vector>
. . .
template<class T>
void Tree<T>::levelScan() {
    vector<Tree<T>*> V;
    Tree<T> *p = this;
    V.push_back(p);
    for (int i = 0; i < this->getAmountOfNodes(); i++) {
        if (V.at(i)->left != NULL)
            V.push_back(V.at(i)->left);
        if (V.at(i)->right != NULL)
            V.push_back(V.at(i)->right);
    }
    for (int i = 0; i < V.size(); i++)
        cout << V.at(i)->getData() << " ";
        cout << endl;
}
```

Данная функция вертикальной печати требует нового алгоритма прохождения, который сканирует дерево уровень за уровнем начиная с корня на уровне 1 (0). Этот метод, называемый *поперечным прохождением* или *прохождением уровней*, не спускается рекурсивно вдоль деревьев, а просматривает дерево поперек, посещая все узлы на одном уровне, и затем переходит на уровень ниже.

Таким образом, в отличие от рекурсивного спуска, здесь более предпочтителен итерационный алгоритм, использующий список элементов.

Для каждого узла в список помещается всякий непустой левый и правый указатель на потомка этого узла. Это гарантия того, что одноуровневые узлы следующего уровня будут посещаться в нужном порядке.

Составим функцию, которая поможет нам понять в дальнейшем функцию вертикальной печати.

Алгоритм функции, использующей поперечное прохождение, следующий:

- Помещаем в начало списка корень.

Пока счетчик не достигнет текущего числа узлов в дереве:

- Добавляем узел в список.
- Используем этот узел для идентификации его потомков на следующем уровне при последующих итерациях.

Выведем все элементы списка (рис. 22.43) [16].

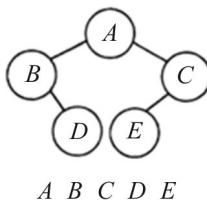


Рис. 22.43. Вывод всех элементов списка

Параметры для печати

Дерево будет печататься в зависимости от двух аргументов, влияющих на его вывод. Первый аргумент – понятно, само дерево

(точнее, это указатель на узел дерева, который будет вызывать функцию класса). Последний – сочетание ширины данных и растяжения дерева.

Ширина данных – это максимум символов в элементе при выводе, т.е. то, насколько длинным может быть элемент. Раастяжение дерева – это пропорциональное увеличение расстояния между элементами при печати. Оба параметра представляют собой целые числа.

На рис. 22.44 показано, как будет меняться печать одного и того же дерева в зависимости от двух параметров. Это некие коэффициенты, влияющие как на протяженность печатаемого дерева, так и на позиции элементов (здесь белыми числами показано количество пустого пространства (пробелов) перед каждым элементом). Но для реализации функции они будут объединены в один, так как координаты элементов, какой бы максимальной ширины их данные ни были, эти два параметра меняют одинаково.

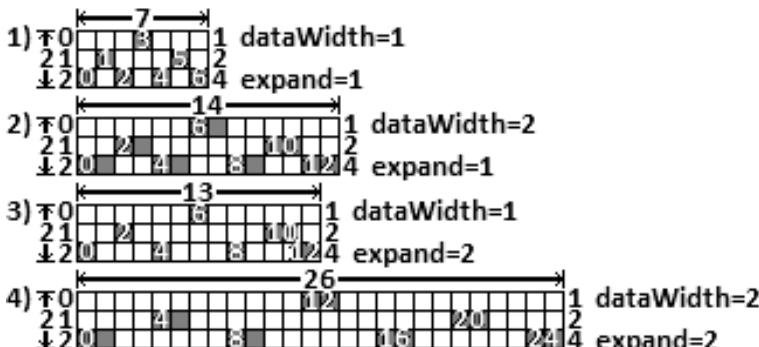


Рис. 22.44. Изменение печати дерева

Столбик слева от матрицы дерева – это уровни (условимся, что отсчет идет с нуля). Столбик справа – максимальное количество элементов для определенного уровня. Числа, заключенные в стрелках, показывают высоту или ширину дерева.

Функция вычисления высоты дерева

Может понадобиться рассчитать высоту дерева, поэтому составим функцию вычисления высоты дерева:

```

template <class T>
int Tree<T>::getHeight() {
    int h1 = 0, h2 = 0, hadd = 0;
    if (this == NULL) return 0;
    if (this->left != NULL) h1 = this->left->getHeight();
    if (this->right != NULL) h2 = this->right->getHeight();
    if (h1 >= h2) return h1 + 1;
    else return h2 + 1;
}

```

Функция работает рекурсивно, проходя по левому и правому поддеревьям. Отдельно считаются значения высоты для каждого поддерева, и между ними выбирается максимальная. Не забываем прибавить единицу, так как текущее значение переменной – это самый нижний уровень, если учитывать, что нумерация идет с нуля.

Функция вычисления количества узлов дерева

Пригодится знать количество узлов дерева, особенно когда требуется сделать неполное дерево полным, заполнив недостающие узлы данными со значением NULL:

```

template<class T>
intTree<T>::getAmountOfNodes() {
    if (this == NULL) return 0;
    if ((this->left == NULL)&&(this->right == NULL)) return
1;
    int l = 0;
    int r = 0;
    if (this->left != NULL) l=this->left->getAmountOfNodes();
    if (this->right != NULL) r=this->right-
>getAmountOfNodes();
    return (l + r + 1);
}

```

Функция работает рекурсивно, проходя по левому и правому поддеревьям. Ее алгоритм прохождения дерева аналогичен рассмотренной до этого функции определения высоты, поэтому особого внимания не требует.

Функция преобразования дерева из неполного в полное

Чтобы дерево было полным, нужно, чтобы все внутренние узлы и не самые нижние узлы дерева имели двух потомков.

Для проверки на наличие/отсутствие двух потомков пригодится знать высоту дерева и уровень текущего узла. В связи с тем, что функция определения уровня работает для любого узла корректно, а функция определения высоты – только для корня, высоту дерева нужно посчитать заранее в главной функции, потому что проход по узлам рекурсивный, а пересчитывать несколько раз высоту нет необходимости (данные о высоте на каждом узле разные).

Исходя из этого, создана вспомогательная функция, хранящая значение высоты:

```
template <class T>
Tree<T>* replace_help(Tree<T>* node, int h);
template <class T>
Tree<T>* Tree<T>::replaceNULLforEmpty() {
    Tree<T>* node = this->copyTree();
    int h = node->getHeight();
    node=replace_help(node,h);
    return node;
}
template <class T>
Tree<T>* replace_help(Tree<T>* node, int h) {
    int curLevel = getLevel(node);
    if ((node->getLeft() == NULL) && (curLevel != h - 1)) {
        node->insertLeft(NULL);
    }
    if ((node->getRight() == NULL) && (curLevel != h - 1)) {
        node->insertRight(NULL);
    }
    if (node->getLeft()!=NULL) node-
>addLeftTree(replace_help(node->getLeft(),h));
    if (node->getRight()!=NULL) node-
>addRightTree(replace_help(node->getRight(),h));
    return node;
}
```

Структура элемента и функция определения позиции

Конечная функция вертикальной печати будет использовать два списка и поперечное прохождение узлов дерева. В одном списке будут находиться узлы, а во втором – позиции этих узлов для печати, а точнее, структуры, в которых есть два поля: номер столб-

ца и строки. Когда узел добавляется в первый список, во второй заносится соответствующая ему информация о печати.

Столбец элемента – это его позиция в строке при печати; «значение столбца – 1» – это количество пробелов перед ним, а строка элемента – это уровень элемента при печати. Везде идет нумерация с нуля.

Функция возвращения позиции обозначена специфичной функцией нахождения количества пробелов перед элементом, а следовательно, и его позиции, по четырем аргументам:

- номер (индекс) элемента на уровне;
- необходимая ширина дерева (но не конечная);
- текущий уровень элемента;
- максимальный уровень («высота дерева – 1»).

Функция преобразует четыре параметра на входе в один на выходе. Она работает с прямой, формулу которой она предварительно составляет по тем четырем параметрам. Прямая $y = kx + m$, где y – это то, что будет на выходе, а x – это первый аргумент (индекс элемента); k и m – это специальный коэффициент и отступ по оси Oy , которые будут получены с помощью несложных вычислений.

Но чтобы прийти к такому упрощенному варианту формулы прямой, необходимо сначала поработать с таким ее видом (формула прямой, проходящей через две точки):

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1}.$$

И если немного преобразовать формулу, она приобретет следующий вид:

$$y = \frac{y_2 - y_1}{x_2 - x_1} x - \frac{y_2 - y_1}{x_2 - x_1} x_1 + y_1,$$

$$\text{где } k = \frac{y_2 - y_1}{x_2 - x_1}, \quad m = \frac{y_2 - y_1}{x_2 - x_1} x_1 + y_1 = -kx_1 + y_1.$$

Опишем, как получить x_1, x_2, y_1, y_2 . x – это индекс элемента на данном уровне (представим уровень как массив), а y – это его пози-

ция (количество пробелов). Если вспомнить рис. 22.44 дерева в виде таблицы, где расстояние между элементами минимизировано так, чтобы можно было их различать (по пустому пространству видно, что это два элемента, а не один), то можно заметить следующие закономерности.

Первый элемент уровня, обозначенный индексом 0, всегда находится на позиции, вычисляемой по формуле

$$y_1 = \text{width} \div 2^{\text{curLevel}+1}, \text{ при том, что } x_1 = 0,$$

где *целочисленное деление обозначено div для наглядности*;

width – это минимальная ширина дерева; *curLevel* – текущий уровень, на котором находится элемент. Учтем, что *width* и *curLevel* – взаимосвязанные параметры (минимальная ширина дерева для печати предполагает определенную высоту дерева, так же как и высота требует определенной ширины), т.е. нужно следить, как меняются их значения, знать, не противоречат ли они друг другу. Например, при ширине 7 не может быть уровня 3, как и при уровне 3 не может быть ширины 7.

Теперь известны x_1 и y_1 . Осталось найти x_2 и y_2 , и здесь тоже наблюдается закономерность.

Во-первых, условимся, что x_2 – это всегда индекс самого последнего на уровне элемента, т.е. он равен «максимальному числу элементов на уровне – 1». Во-вторых, его позиция всегда будет вычисляться по формуле

$$y_2 = \text{width} - 2^{\text{maxLevel} - \text{curLevel}}, \text{ при том, что } x_2 = 2^{\text{curLevel}} - 1,$$

где *maxLevel* и *curLevel* – максимальный уровень дерева (уровень низшего листа) и текущий уровень элемента.

Исходя из предыдущих вычислений, организована такая функция определения позиции элемента.

Учтем, что индекс первого элемента и индекс последнего элемента на одном уровне могут совпадать. Речь идет о корне.

Ввиду этого для него вернем y_1 сразу, не вычисляя формулу прямой. Иначе по формуле дальше произойдет деление на 0.

Данная функция очень полезна при высчитывании координат элементов. Эти x и y очень пригодятся, когда понадобится нарисовать дерево по его элементам, а не просто вывести в консоли.

Создадим заголовочный файл для объявления прототипов вспомогательных функций:

```
#ifndef TREETOOLS_H
#include "tree.h"
#define TREETOOLS_H
int getPos(int index, int width, int curLevel, int
maxLevel);
#endif
```

И описание его функций:

```
#include "stdafx.h"
#include "treetools.h"
#include <math.h>
int getPos(int index, int width, int curLevel, int
maxLevel)
{
    int x1 = 0;
    int x2 = pow(2, curLevel) - 1;
    int y1 = width / pow(2, curLevel + 1);
    int y2 = width - pow(2, maxLevel - curLevel);
    if (x1 == x2) return y1;
    // y=x(y2-y1)/(x2-x1)-x1(y2-y1)/(x2-x1)+y1
    // y=kx+m: k=(y2-y1)/(x2-x1), m=-x1(y2-y1)/(x2-x1)+y1-
    -x1*k+y1
    double k = (y2 - y1) / (x2 - x1);
    double m = -x1 * k + y1;
    int y = (int)(k*index + m);
    return y;
}
```

Организация основной функции

Все необходимые функции разобраны, осталось организовать основную функцию вертикальной печати дерева.

Составим план действий с деревом и его элементами:

1. Объявим структуру с координатами, посчитаем высоту дерева, максимальное количество листов на нижнем уровне (предположим, что дерево полное), минимальную ширину дерева для печати,

инициализируем переменную, представляющую собой значение позиции строки вывода уровня элементов, с индексом «ноль», позицию первого элемента (корня в этой строке), счетчик элементов для цикла, создадим объект структуры, а также объекты двух списков, в первом из которых будут собраны элементы, а во втором – их структуры с координатами при выводе:

```
struct pos {
    int col; // Столбец (x)
    int str; // Стока (y)
};
template <class T>
void Tree<T>::printVTree(int k) {
    int height = this->getHeight();
        /* Максимальное число листов на
           нижнем уровне (нумерация с нуля) */
    int maxLeafs = pow(2, height - 1);
        /* Минимальная ширина дерева для печати
           (не конечная, но необходимая) */
    int width = 2 * maxLeafs - 1;
    int curLevel = 0; // Номер строки (на выводе)
    int index = 0;
        // Номер элемента в строке (нумерация с нуля)
        /* Позиция корня (число пробелов перед ним) */
    int factSpaces = getPos(index, width, curLevel, height - 1);
    pos node;
    vector<Tree<T>*> V;
    vector<pos> Vi;
```

Далее необходимо позаботиться о том, чтобы при выводе элементов не было ничего упущено. А для этого нужно заполнить дерево пустыми узлами так, чтобы не поменялась высота дерева, но все узлы, кроме нижних, приобрели одного-двух потомков в зависимости от того, сколько отсутствует.

Пустые узлы при выводе будут представляться некоторым числом пробелов. По стандарту каждый пустой элемент – это один пробел, но с учетом, что есть коэффициент k расширения дерева, пробелов станет в k раз больше.

Итак:

2. Создадим буферное дерево, являющееся копией вызывающего, сделаем его полным в случае его неполноты. Добавим первый узел в список элементов и его координаты в список структур.

В дальнейшем будем работать именно с этим буферным деревом, а значит, и выводить его, а не оригинал:

```
Tree<T>* t = this->copyTree();
t=t->replaceNULLforEmpty();
Tree<T>* p = t;
V.push_back(p);
node.col = factSpaces;
node.str = curLevel;
Vi.push_back(node);
```

3. Теперь совершаем обход по дереву так же, как это происходило в функции сканирования элементов по уровням.

Нужно запомнить их координаты во время прохождения, поэтому необходимо следить за изменениями y , отвечающего за положение на оси Oy . Для этого была создана переменная $index$, обозначающая, какой по счету элемент на данном уровне дерева.

Добравшись до последнего узла какого-либо уровня, нужно перейти на следующую строчку, увеличив на 1 $curLevel$ и, естественно, обнулив $index$. Это произойдет только в том случае, если $2^{curLevel}$ окажется меньше $curLevel+1$.

Таким образом, следим за переходом на новую строчку.

Далее, два аналогичных друг другу условия добавления элементов. Если слева или справа от данного узла в дереве будут какие-либо элементы, мы добавим их в список V , а их координаты (x мы посчитаем с помощью функции $getPos()$) мы запишем в список Vi . Параллельно с этим увеличивается $index$:

```
for (int i = 0; i < t->getAmountOfNodes(); i++) {
    if (pow(2, curLevel) <= index + 1) {
        index = 0;
        curLevel++;
    }
}
```

```

if (V.at(i)->left != NULL) {
    V.push_back(V.at(i)->left);
    factSpaces = getPos(index, width, curLevel, height - 1);
    node.col = factSpaces;
    node.str = curLevel;
    Vi.push_back(node);
    index++;
}
if (V.at(i)->right != NULL) {
    V.push_back(V.at(i)->right);
    factSpaces = getPos(index, width, curLevel, height - 1);
    node.col = factSpaces;
    node.str = curLevel;
    Vi.push_back(node);
    index++;
}
}

```

4. Визуализируем представление дерева. Если бы было нужно нарисовать дерево по его координатам, то это с легкостью можно было бы осуществить, воспользовавшись известными `col` и `str` структуры `node`.

Но если нужно вывести дерево в консоли, необходимо `col` поменять на количество пробелов от предыдущего элемента (если такой имеется) текущего уровня до данного. Для этого вычтем из `col` данного элемента `col` предыдущего (через обращение к соответствующим структурам), если они находятся в одной строке.

После редактирования позиций можно выводить элементы в цикле со счетчиком.

Создадим `flag`, который будет следить за изменениями `curLevel`, и в случае, если `curLevel` станет больше, чем был до этого, `flag` приравняется к нему и спровоцирует переход на следующую строчку.

Затем самое простое – это высчитывание количества пробелов (теперь можно использовать коэффициент расширения дерева k) и вывод:

```

        /* Редактируем позиции в строчках
        (теперь они обозначают количество пробелов
        перед данным символом начиная с предыдущего символа):
        до этого эти значения представляли
        собой координаты (как x) */
for (int i = V.size() - 1; i >= 0; i--) {
    if (i != 0) {
        if (Vi.at(i - 1).str == Vi.at(i).str) Vi.at(i).col =
Vi.at(i).col-Vi.at(i - 1).col-1;
    }
}
int flag = 0; // Следит за тем, что у меняется
for (int i = 0; i < V.size(); i++) {
    node = Vi.at(i);
    curLevel = node.str;
        /* Переход на новую строчку будет, когда у1
        станет меньше у (слежка за изменением у) */
    if (flag < curLevel) {
        flag = curLevel;
        cout << endl;
    }
    factSpaces = node.col;
    int realSpaces = k*factSpaces;
    for (int j = 0; j < realSpaces; j++) cout << " ";
        if (V.at(i)->getData()!=NULL) cout << V.at(i)-
>getData();
        else for (int j=0; j<k; j++) cout << " ";
    }
    cout << endl;
}

```

Выведем дерево в вертикальной печати и представим код листинга ниже (рис. 22.45).

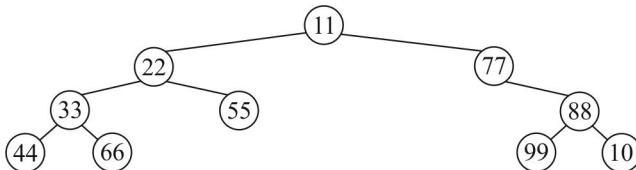


Рис. 22.45. Бинарное дерево для вывода на консоль

```

Tree<int>* tree = new Tree<int>(11);
tree->insertLeft(22);
tree->getLeft()->insertLeft(33);
tree->getLeft()->getLeft()->insertLeft(44);
tree->getLeft()->insertRight(55);
tree->getLeft()->getLeft()->insertRight(66);
tree->insertRight(77);
tree->getRight()->insertRight(88);
tree->getRight()->getRight()->insertLeft(99);
tree->getRight()->getRight()->insertRight(10);
cout << "Горизонтальная печать" << endl;
tree->printTree(1);
cout << endl;
cout << "Вертикальная печать" << endl;
tree->printVTree(2);

```

На рис. 22.46 приведена горизонтальная и вертикальная печать одного и того же дерева.

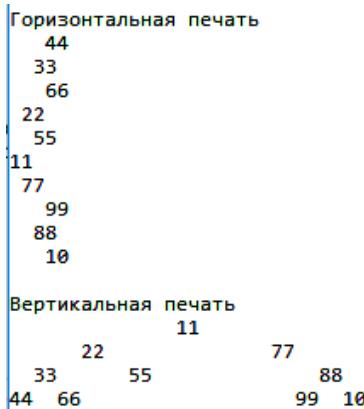


Рис. 22.46. Горизонтальная и вертикальная печать одного и того же дерева

Другой способ вертикальной печати дерева

Вертикальную печать дерева можно осуществить и другим способом, без использования формул.

Необходимо совершить поперечное прохождение по дереву, про которое было рассказано в подразд. 22.8. Однако, в отличие от

предыдущего варианта, в этом в качестве структуры данных для хранения узлов будет предоставлена очередь, а не список.

Все, что поэлементно будет входить в очередь, запишется в файл. Назовем его print.txt. Откроем поток ввода в него. Вычислим количество элементов дерева, а затем создадим очередь для этого количества. Для начала помещаем в очередь указатель на ссылку корня. Затем выводим в файл данные корня, удаляем его из очереди, а его потомков (левое и правое поддеревья) заносим в очередь. Эти действия производятся в цикле, пока очередь не окажется пустой. Для обхода дерева в сопровождении с записью в файл реализована функция obh():

```
template <class T>
/* Построчный обход дерева и запись элементов в файл */
void Tree<T>::obh(Tree<T>* node) {
    ofstream f("print.txt");
    /* Кол-во элементов в дереве */
    int amount = node->getAmountOfNodes();
    queue<Tree<T>*> q; // Очередь указателей
    q.push(node);
    // Для начала поместим в очередь корень
    while (!q.empty()) {
        Tree<T>* temp = q.front();
        q.pop();
        f << temp->data << endl;
        /* Если есть левый наследник, то помещаем
         его в очередь для дальнейшей обработки */
        if (temp->left)
            q.push(temp->left);
        /* Если есть правый наследник, то помещаем
         его в очередь для дальнейшей обработки */
        if (temp->right)
            q.push(temp->right);
    }
    f.close();
}
```

Отвечать за альтернативно реализованную вертикальную печать дерева будет функция printVTree2(). Для простоты понимания она не будет принимать дополнительных аргументов, например ширину данных:

Подсчитываем количество строк amount в файле, чтобы сформировать массив, в котором будут храниться элементы дерева. Считываем из файла элементы в массив mas, используя функцию преобразования atoi() (функция atoi() преобразует строку string в целое значение типа int). Рассчитываем высоту дерева height. Создаем массив, в котором хранятся числа, обозначающие количество пробелов перед выводом первого числа в строке дерева (0-й индекс соответствует последней строке, а индекс со значением height-1 – первой строке). Если высота дерева равна 1 (соответственно, элемент всего один), то печатаем 0-й элемент массива mas, иначе в цикле от 0 до height-1 печатаем элементы дерева построчно (для первой строки берутся пробелы из массива пробелов в последней ячейке, затем для каждой строки по убыванию). Для печати последней строки создается массив last_str с количеством элементов, равным максимальному количеству элементов в последней строке дерева, и изначально заполняется 2 000 000 000. Затем имитируем алгоритм построения последней строки идеально сбалансированного дерева. Изначально элементы встают не на свои места, но потом в цикле от 0 до максимального количества элементов в последней строке дерева заполняем ячейки, где значение не 2 000 000 000, по очереди из массива mas, чтобы элементы встали на свои места. Печатаем последнюю строку, а где 2 000 000 000 – печатаем два пробела:

```
template <class T>
void Tree<T>::printVTree2() {
    obh(this);           // Обход дерева и заполнение файла
    ifstream f("print.txt");
    int amount = 0;        // Кол-во элементов в дереве
    while (!f.eof()) {
        char str[255];
        f.getline(str, 255);
        amount++;
    }
    f.close();
    amount--;            // Кол-во элементов в дереве
    ifstream f1("print.txt");
    int* mas = new int[amount];
    for (int i = 0; i < amount; i++) {
```

```

        // Считывание элементов в массив
    char str[255];
    f1.getline(str, 255);
    mas[i] = atoi(str);
}
f1.close();
int height = this->getHeight();
int count = 0; // Счетчик для вывода
int* spaces = new int[height];
// Массив нужных пробелов
spaces[0] = 0; // При глубине 1 будет 0 пробелов
for (int i = 1; i < height; i++)
    // Вычисление пробелов
    spaces[i] = spaces[i-1] * 2 + 1;
int amount_p = 0; // Кол-во напечатанных элементов
if (height == 1) {
    cout << mas[0] << endl;
} else {
    /* 1 - индекс для вывода пробелов |
       вывод всех строк, кроме последней */
    for (int i = 0, l = height - 1; i < height - 1; i++, l--) {
        for (int j = 0, k = 0; j < pow(2, i); j++, k++) {
            if (k == 0) // Вывод первого числа в строке
                /* Печатаем пробелы */
            for (int u = 0; u < spaces[l]; u++)
                cout << " ";
            else {
                for (int u = 0; u < spaces[l+1]; u++)
                    cout << " ";
                }
            cout << mas[count++];
            amount_p++;
        }
        cout << endl;
    }
    /* Будет храниться последняя строка */
    int* last_str = new int[pow(2, height- 1)];
    for (int i = 0; i < pow(2, height - 1); i++)
        last_str[i] = 2000000000;
// amount - это amount_p (кол-во оставшихся элементов)
    int sch1 = 0;

```

```

        // счетчик для последнего массива
        /* Второй счетчик для последнего массива */
        int sch2 = spaces[height - 2] + 1;
                    /* Строим массив так, как
                       встают элементы в дереве */
for (int i = amount_p; i < amount; i += 2) {
    if (i <= amount - 1)           //если еще есть элементы
    {
        last_str[sch1] = mas[i];
        sch1 += 2;
    }
    if (i + 1 <= amount - 1) {
        last_str[sch2] = mas[i + 1];
        sch2 += 2;
    }
}
if (sch1 >= pow(2, height - 1) || sch2 >= pow(2,
height - 1)) {           //Заполняем вторую половину массива
    sch1 = 1;
    sch2 = spaces[height- 2] + 2;
}
/*
Исправляем неправильное построение массива */
for (int i = 0; i < pow(2, height- 1); i++)
    if (last_str[i] != 2000000000)
        last_str[i] = mas[amount_p++];
            /* Печатаем последнюю строку */
for (int i = 0; i < pow(2, height - 1); i++)
    if (last_str[i] != 2000000000)
        cout << last_str[i] << " ";
    else cout << " ";
    delete[] last_str;
    cout << endl;
}
delete[] mas;
delete[] spaces;
}

```

Код смещен вниз

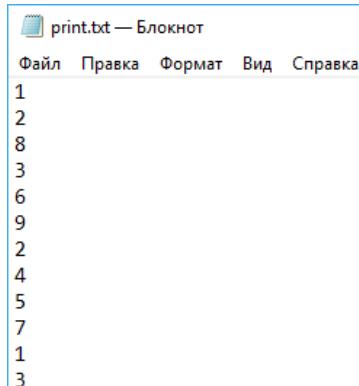
Напечатаем идеально сбалансированное дерево (рис. 22.47):

```
cout << "n=";
int n;
cin >> n;
Tree<int>* tree = Tree<int>::balancedTree(n);
tree->printVTree2();
```

```
n=12
data=1
data=2
data=3
data=4
data=5
data=6
data=7
data=8
data=9
data=1
data=2
data=3
Вертикальная печать
      1
     2   8
    3   6   9   2
   4 5 7   1   3
```

Рис. 22.47. Вертикальная печать идеально сбалансированного дерева

В файл print.txt записались следующие данные (рис. 22.48).



print.txt — Блокнот

Файл Правка Формат Вид Справка

```
1
2
8
3
6
9
2
4
5
7
1
3
```

Рис. 22.48. Данные файла print.txt

Лучше ли этот способ вертикальной печати, чем тот, что был представлен в предыдущих подразделах? Условимся называть его вторым, а предыдущий – первым.

Второй способ отлично справляется с печатью идеально сбалансированных деревьев, однако, когда будет нужно напечатать несбалансированное дерево, могут возникнуть дефекты.

Выведем дерево, представленное на рис. 22.49.

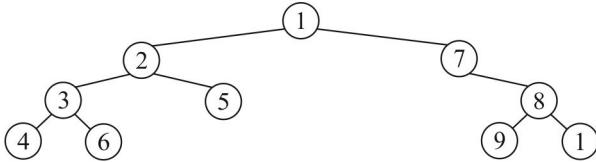


Рис. 22.49. Пример несбалансированного дерева

Его описывает код листинга ниже. Для того чтобы показать различия в результатах работы первого и второго способов вертикальной печати, они будут реализованы оба (рис. 22.50):

```
Tree<int>* tree = new Tree<int>(1);
tree->insertLeft(2);
tree->getLeft()->insertLeft(3);
tree->getLeft()->getLeft()->insertLeft(4);
tree->getLeft()->insertRight(5);
tree->getLeft()->getLeft()->insertRight(6);
tree->insertRight(7);
tree->getRight()->insertRight(8);
tree->getRight()->getRight()->insertLeft(9);
tree->getRight()->getRight()->insertRight(1);
cout << "Вертикальная печать (1)" << endl;
tree->printVTree(1);
cout << "Вертикальная печать (2)" << endl;
tree->printVTree2();
```

Дефекты происходят потому, что в файл элементы с данными, не равными NULL, отдельно взятого уровня записываются в том порядке, в каком они находятся изначально, если каждый уровень представить в виде массива (по принципу первого способа вертикальной печати). Но места, на которых элементов нет, не учитываются.

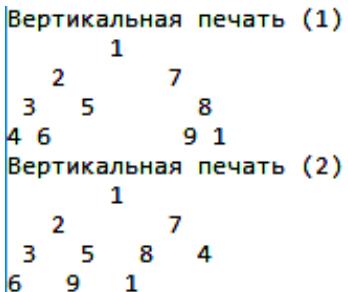


Рис. 22.50. Различия первого и второго способов вертикальной печати

Идеально сбалансированное дерево строится так, что только нижний уровень может иметь недостающие элементы, тогда печать будет корректной. Но если не только на нижнем уровне будет не хватать элементов, чтобы заполнить уровень до конца, произойдет смещение элементов: какой-нибудь элемент может оказаться «не на своей строке» при выводе, как можно видеть на рис. 22.49.

Проблема решается пополнением дерева, которое было позаимствовано функцией первого способа вертикальной печати. Этот метод реализован ранее представленной функцией `replaceNULLforEmpty()`.

Однако, если программист хочет работать только с идеально сбалансированными деревьями, можно обойтись и без пополнения.

Оба способа вертикальной печати (если второй дополнить функцией пополнения дерева) хорошо работают при печати любых деревьев. Но, если требуется напечатать идеально сбалансированное дерево, рациональнее прибегнуть к методам второго способа без пополнения.

Достоинство первого способа вертикальной печати в том, что он идентифицирует координаты элементов по формулам, что можно использовать уже не при выводе дерева в консоли, а в отображении дерева, как рисунка. Но всегда нужно пополнять дерево, прежде чем его вывести. Во втором способе, как уже говорилось, пополнение дерева не обязательно, если дерево идеально сбалансированно.

ГЛАВА 23. ВВЕДЕНИЕ В ТЕОРИЮ ГРАФОВ

23.1. Виды графов. Терминология теории графов

Графом называется множество узлов и связей между ними. Каждый узел называется *вершиной*, а каждая связь – *ребром*. Каждое ребро соединяет две вершины. Графы делятся на *ориентированные* и *неориентированные*. Ориентированный граф (рис. 23.1) – такой граф, в котором можно двигаться от вершины к вершине только в одном направлении. Например, можно уехать из Перми в Екатеринбург, а вот из Екатеринбурга в Пермь – нельзя. В неориентированных графах (рис. 23.2) можно двигаться в обе стороны, т.е. из Екатеринбурга можно будет вернуться в Пермь. Вершина графа называется *изолированной*, если она не имеет связей с другими вершинами.

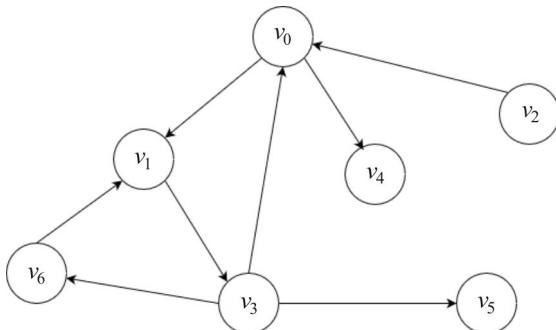


Рис. 23.1. Ориентированный граф

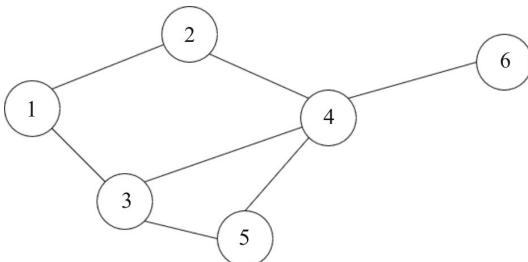


Рис. 23.2. Неориентированный граф

Граф, в котором присутствуют как ориентированные ребра, так и неориентированные, называется *смешанным* (рис. 23.3).

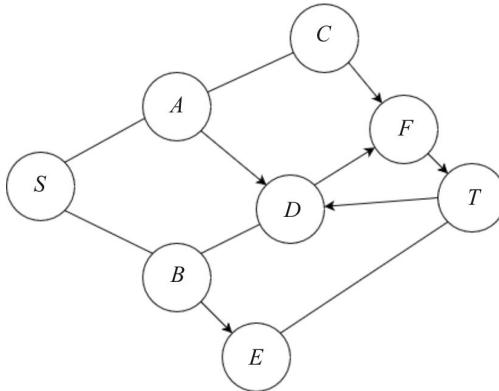


Рис. 23.3. Смешанный граф

Кроме того, графы делятся на *взвешенные* и *невзвешенные*. Граф, в котором каждому ребру в соответствие поставлено некоторое числовое значение – *вес*, называется взвешенным графом (например, длина дороги). Если никакого числового значения ребрам не поставлено, то граф называется невзвешенным. Чаще всего в названии графа указывают как его ориентированность или неориентированность, так и его взвешенность или невзвешенность (рис. 23.4).

Граф, в котором между любой парой вершин существует как минимум один путь, называется *связным*. Если в графе существует хотя бы одна вершина, не связанная с другими, он называется *несвязным* (рис. 23.5).

Граф, в котором число ребер близко к максимальному (когда каждая вершина графа связана с любой другой вершиной графа ребрами), называется *плотным графом* (рис. 23.6).

Граф с противоположным свойством, имеющий малое число ребер, называется *разреженным графом*.

Петлей называется ребро, которое соединяет вершины v_1 и v_2 , причем v_1 и v_2 совпадают. Иными словами, петля – это ребро, которое начинается и заканчивается в одной вершине (рис. 23.7).

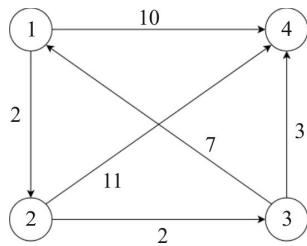


Рис. 23.4. Взвешенный связный
ориентированный граф

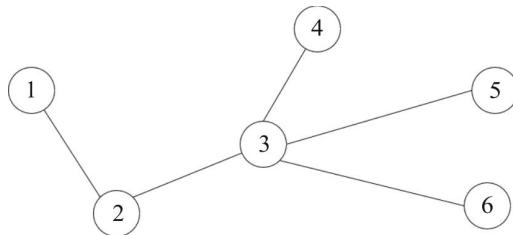


Рис. 23.5. Несвязный граф

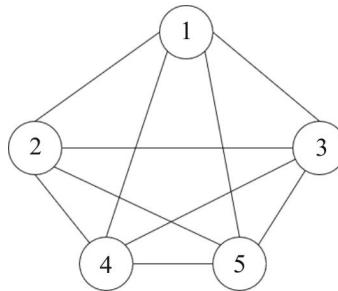


Рис. 23.6. Плотный граф

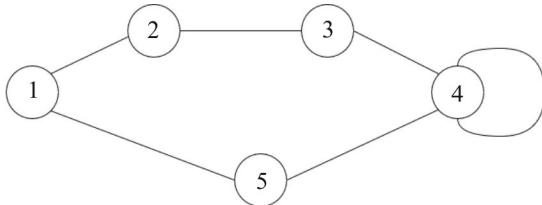


Рис. 23.7. Граф с петлей

Инцидентность – понятие, используемое только в отношении ребра и вершины. Если v_1, v_2 – вершины, а $e = (v_1, v_2)$ – соединяющее их ребро, тогда вершина v_1 и ребро e инцидентны, вершина v_2 и ребро e тоже инцидентны. Две вершины (или два ребра) инцидентными быть не могут. Например, на рис. 23.3 вершина E инцидентна ребру ET .

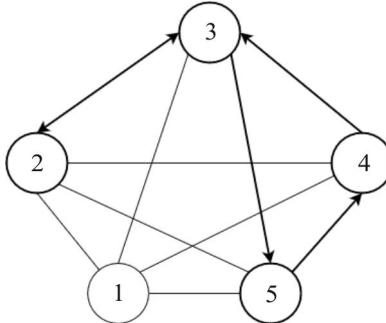


Рис. 23.8. Замкнутый маршрут
(вершины 2–3–5–4–3–2)

Смежность – понятие, используемое в отношении только двух ребер либо только двух вершин: два ребра, инцидентные одной вершине, называются смежными; две вершины, инцидентные одному ребру, также называются смежными. Например, на рис. 23.3 вершины A и B смежные, поскольку инциденты одному ребру.

Маршрутом называется такая чередующаяся последовательность вершин и ребер $\text{vertex}_0, \text{edge}_1, \text{vertex}_1, \text{edge}_2, \text{vertex}_2, \dots, \text{edge}_k, \text{vertex}_k$, $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$ {displaystyle $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$ } ..., //, в которой любые два соседних элемента инцидентны. Если $v_0 = v_k$ {displaystyle $v_0 = v_k$ } $\text{vertex}_0 = \text{vertex}_k$, то маршрут *замкнут* (рис. 23.8), иначе – *открыт* (рис. 23.9).

Цепь – маршрут, все ребра которого различны (рис. 23.10). Если все вершины такого маршрута различны, то цепь называется *простой*. В цепи $\text{vertex}_0, \text{edge}_1, \dots, \text{edge}_k, \text{vertex}_k$ вершины vertex_0 и vertex_k называются *концами* цепи.

Замкнутая цепь называется *циклом* (рис. 23.11).

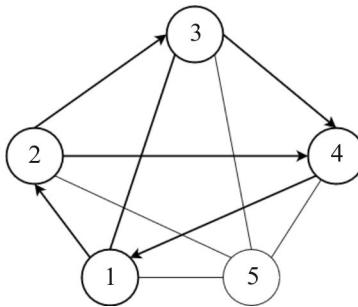


Рис. 23.9. Открытый маршрут
(вершины 2–4–1–2–3–4–1)

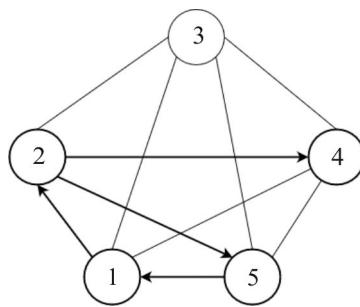


Рис. 23.10. Открытая цепь
(вершины 2–5–1–2–4)

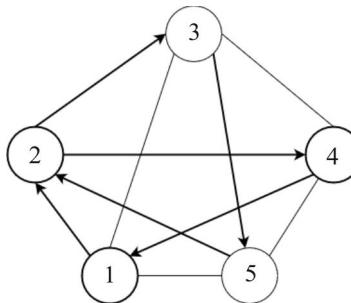


Рис. 23.11. Замкнутая цепь
(вершины 2–4–1–2–3–5–2)

23.2. Представление графов. Матрица смежности

Матрица смежности графа – это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1 для невзвешенного графа (рис. 23.12 и 23.13). Значения 1 и 0 отображают существование ребра между вершинами: если между вершинами 1 и 2 есть ребро, то в матрице на пересечении первой строки и второго столбца и второй строки и первого столбца будут стоять единицы. Если граф взвешенный (рис. 23.14), то элементами матрицы смежности будут числа, соответствующие весам ребер, соединяющих вершины, на пересечении которых в таблице стоит элемент.

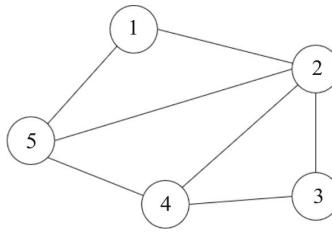


Рис. 23.12. Невзвешенный неориентированный граф

Матрица смежности для графа, изображенного на рис. 23.12, представлена следующим образом:

Номер вершины	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

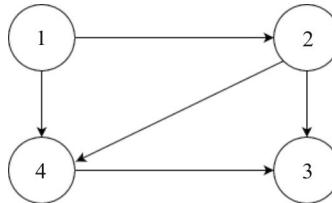


Рис. 23.13. Невзвешенный ориентированный граф

Матрица смежности для графа, изображенного на рис. 23.13, представлена следующим образом:

Номер вершины	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0

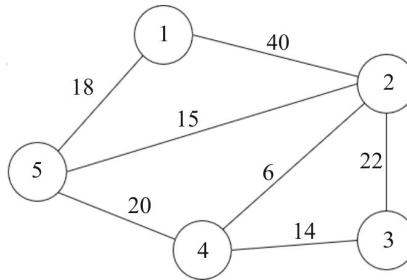


Рис. 23.14. Взвешенный неориентированный граф

Матрица смежности для графа, изображенного на рис. 23.14, представлена следующим образом:

Номер вершины	1	2	3	4	5
1	0	40	0	0	18
2	40	0	22	6	15
3	0	22	0	14	0
4	0	6	14	0	22
5	18	15	0	20	0

Граф может быть представлен с помощью *вектора смежности* (рис. 23.15). В векторе смежности для каждой вершины хранятся номера смежных с ней вершин.

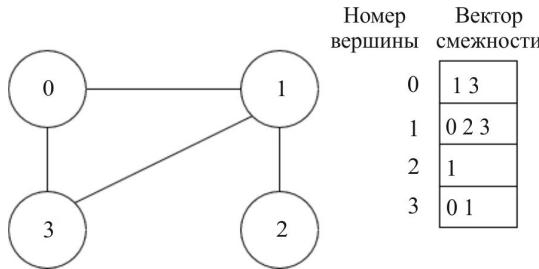


Рис. 23.15. Представление графа с помощью вектора смежности

Еще одной формой представления графа является *матрица инцидентности*, в которой указываются связи между инцидентными элементами графа (ребра и вершины). Столбцы матрицы соответст-

вуют ребрам, строки – вершинам. Ненулевое значение в ячейке матрицы указывает на связь между вершиной и ребром (их инцидентность).

Пусть граф имеет вид, представленный на рис. 23.16.

Матрица инцидентности, соответствующая графу, выглядит следующим образом:

Номер вершины	a	b	c	d	e
1	1	1	0	0	0
2	1	0	1	1	0
3	0	0	0	1	1
4	0	1	1	0	1

Приведем пример построения матрицы инцидентности для взвешенного ориентированного графа (рис. 23.17).

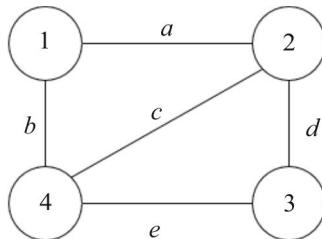


Рис. 23.16. Пример матрицы инцидентности для взвешенного неориентированного графа

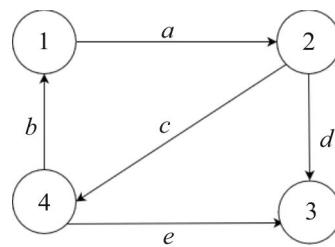


Рис. 23.17. Пример матрицы инцидентности для ориентированного графа

Матрица инцидентности, соответствующая графу, выглядит следующим образом:

Номер вершины	a	b	c	d	e
1	1	-1	0	0	0
2	-1	0	1	1	0
3	0	0	0	-1	-1
4	0	1	-1	0	1

Список смежности – один из способов представления графа в виде коллекции списков вершин. Каждой вершине графа соответ-

ствует список, состоящий из «соседей» этой вершины. Если граф взвешенный, то рядом с номером вершины-соседа также указывается длина ребра до этого соседа (рис. 23.18).

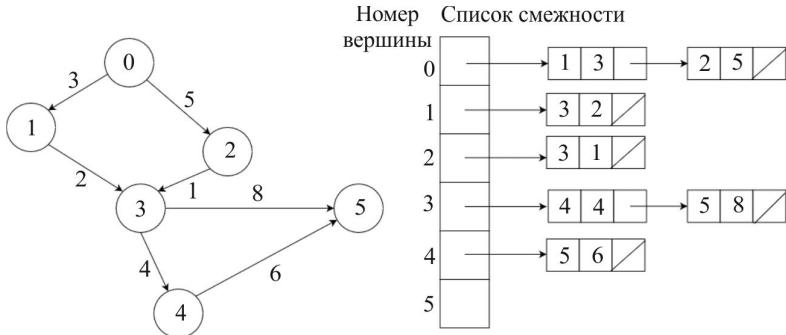


Рис. 23.18. Список смежности для ориентированного взвешенного графа

Если хватает памяти компьютера, то матрицу смежности реализовать и применить легче, чем списки смежности. В графе, представленном списком смежности, удобно искать вершины, смежные с данной, добавлять ребра и вершины, работать с разреженными графиками. Однако при таком представлении неудобно удалять элемент и выполнять поиск элемента.

23.3. Проектирование графов на алгоритмическом языке C++

В программном представлении графа работают с матрицей смежности, так как при ее использовании удобнее выполнять поиск элемента.

Создается класс Graph, он будет шаблонным, чтобы узел графа мог содержать данные разного типа (например, строки или числа). Перед описанием класса объявляется целочисленная константа – максимальный размер матрицы смежности (пусть она равна 20).

Граф будет представлен следующим образом: все вершины графа заносятся в вектор вершин (вектор – это контейнер из библиотеки STL (Standard Template Library [18]), где индекс каждой вершины соответствует ее индексу в матрице смежности. Напри-

мер, если в векторе вершин у вершины индекс [1], то в матрице смежности эта вершина располагается по индексу [1][1]. В приватном (private) поле класса Graph хранится вектор вершин vertList (от vertex list) и матрица смежности (размером 20×20). Для работы с графом в поле public описаны следующие методы:

Конструктор *Graph()*

В нем происходит заполнение матрицы смежности нулями.

Названия переменных составлены таким образом, что они отображают смысл того, для чего они предназначены. Здесь Graph – это имя класса, maxSize – максимальный размер матрицы смежности, adjMatrix (от adjacency matrix) – матрица смежности.

Код конструктора:

```
template<class T>
/* Объявление шаблона с формальным параметром класс T */
Graph<T>::Graph() {
    /* Конструктор, который инициализирует
       значения объектов класса Graph*/
    /* Перебор строк и столбцов матрицы
       смежности и заполнение ее нулями */
    for (int i = 0; i < maxSize; ++i) {
        for (int j = 0; j < maxSize; ++j) {
            this->adjMatrix[i][j] = 0;
        }
    }
}
```

Деструктор *~Graph()*

Деструктор удаляет объект класса Graph.

Код деструктора:

```
template<class T>
Graph<T>::~Graph() {
}
```

Метод *int GetVertPos(const T& vertex)*

Находит позицию вершины в векторе вершин. Сравнивает вершину, ссылку на которую получил, с каждой вершиной в векто-

ре. Как только будет найдено совпадение – возвращается номер этой вершины. Если ни одного совпадения не было, возвращается число, которое равно -1 .

Краткий словарь

GetVertPos(const T& vertex) – GetVertexPosition – получить позицию вершины типа T; vertList – это список вершин; IsEmpty – является ли пустым; IsFull – является ли полным. GetAmountVerts – получить количество вершин; GetAmountEdges – получить количество ребер; amount – количество; vertListSize – размер списка вершин.

Код метода int GetVertPos(const T& vertex):

```
template<class T>
int Graph<T>::GetVertPos(const T& vertex) {
    for (int i = 0; i < this->vertList.size(); ++i) {
        if (this->vertList[i] == vertex)
            return i;
    }
    return -1;
}
```

Mетод bool IsEmpty()

Проверяет, пуст ли граф, т.е. в нем нет ни одной вершины. Проверяется, пуст ли вектор вершин. Если да, то возвращает значение `true`, если нет, то `false`.

Код метода bool IsEmpty():

```
template<class T>
bool Graph<T>::IsEmpty() {
    if (this->vertList.size() != 0)
        return false;
    else
        return true;
}
```

Mетод bool IsFull()

Проверяет, достигло ли количество вершин в графе максимально возможного предела (для нашего примера это 20). Если размер вектора вершин равен 20, то метод вернет `true`, если меньше, то `false`. Если граф пуст, то метод вернет `false`.

Код метода bool IsFull():

```
template<class T>
bool Graph<T>::IsFull() {
    return (vertList.size() == maxSize);
}
```

Метод int GetAmountVerts()

Метод возвращает количество вершин в графе, т.е. размер вектора вершин. Если график пуст, то метод вернет 0.

Код метода int GetAmountVerts():

```
template<class T>
int Graph<T>::GetAmountVerts() {
    return this->vertList.size();
}
```

Метод int GetAmountEdges()

Метод возвращает количество ребер в графике. Для каждой из вершин графа в цикле проверяется: если в матрице смежности данные ячейки с индексами $[i][j]$ равны данным ячейки с индексами $[j][i]$ и это значение не ноль, то найден вес ребра, соединяющего две вершины. В конце, после цикла, он делится нацело на 2 (если график неориентированный). Деление на 2 нужно потому, что в неориентированном графике и в ячейке $[i][j]$, и в ячейке $[j][i]$ значение одно и то же. Это означает, что в цикле одно и то же значение будет посчитано 2 раза: сначала для ячейки $[i][j]$, затем для ячейки $[j][i]$. Следовательно, между каждыми двумя вершинами к концу работы цикла будет по 2 ребра, что неправильно, поскольку ребер изначально было по одному между каждыми двумя вершинами. Ввиду этого нужно итоговое значение счетчика поделить нацело на 2, чтобы результат был корректным. Если график ориентированный, то деление на 2 и проверка на равенство элемента $[i][j]$ элементу $[j][i]$ не нужны. Если график невзвешенный (до этого был описан алгоритм для взвешенных графов), то в условии проверяется равенство элемента матрицы смежности единице; если график ориентированный, то только элемента $[i][j]$; если график неориентированный, то элементов $[i][j]$ и $[j][i]$.

Код метода GetAmountEdges() для взвешенного ориентированного графа:

```
template<class T>
int Graph<T>::GetAmountEdges() {
    int amount = 0; // Обнуляем счетчик
    if (!this->IsEmpty()) {
        // Проверяем, что граф не пуст
        for (int i = 0, vertListSize = this-
>vertList.size(); i <
            vertListSize; ++i) {
            for (int j = 0; j < vertListSize; ++j)
{
            if (this->adjMatrix[i][j] == 1) // Находим ребра
                amount += 1; // Считаем количество ребер
            }
        }
        return amount; // Возвращаем количество ребер
    }
    else
        return 0; // Если граф пуст, возвращаем 0
}
```

Код метода GetAmountEdges() для взвешенного неориентированного графа:

```
template<class T>
int Graph<T>::GetAmountEdges() {
    int amount = 0; // Обнуляем счетчик
    if (!this->IsEmpty()) { // Проверяем, что граф не пуст
        for (int i = 0, vertListSize = this->vertList.size();
            i < vertListSize; ++i) {
            for (int j = 0; j < vertListSize; ++j)
{
            if (this->adjMatrix[i][j] ==
                this->adjMatrix[j][i] &&
                this->adjMatrix[i][j] != 0) // Находим ребра
                amount += 1; // Считаем количество ребер
            }
        }
        return (amount / 2);
    }
}
```

```

        /* Приводим счетчик к корректному
           результату и возвращаем его */
    }
else
    return 0;      // Если граф пуст, возвращаем 0
}

```

Код метода GetAmountEdges() для невзвешенного ориентированного графа:

```

template<class T>
int Graph<T>::GetAmountEdges() {
    int amount = 0; // Обнуляем счетчик
    if (!this->IsEmpty()) { // Проверяем, что граф не
    пуст
        for (int i = 0, vertListSize = this-
>vertList.size(); i < vertListSize; ++i) {
            for (int j = 0; j < vertListSize; ++j)
{
                if (this->adjMatrix[i][j] == 1)
// Находим ребра
                    amount += 1; // Считаем
    количество ребер
}
}
return amount; // Возвращаем количество ребер
}
else
    return 0; // Если граф пуст, возвращаем 0
}

```

Код метода GetAmountEdges() для невзвешенного неориентированного графа:

```

template<class T>
int Graph<T>::GetAmountEdges() {
    int amount = 0; // Обнуляем счетчик
if (!this->IsEmpty()) { // Проверяем, что граф не пуст
    for (int i = 0, vertListSize = this-
>vertList.size(); i <
    vertListSize; ++i) {

```

```

        for (int j = 0; j < vertListSize; ++j)
    {
        if (this->adjMatrix[i][j] == 1 && this-
            adjMatrix[j][i] == 1)          // Находим ребра
            amount += 1;                // Считаем количество ребер
        }
    }
    return (amount / 2);
    /* Приводим счетчик к корректному
       результату и возвращаем его */
}
else
    return 0;      // Если граф пуст, возвращаем 0
}

```

Метод int GetWeight(const T& vertex1, const T& vertex2)

Метод возвращает вес ребра, соединяющего вершины vertex1 и vertex2, ссылки на которые метод принимает. Сначала производится проверка, что граф не пуст, затем вычисляются позиции вершин vertex1 и vertex2 в графе и из матрицы смежности извлекается значение на пересечении этих вершин. В невзвешенных графах этот метод не используется.

Краткий словарь

GetWeight – получить вес; vertPos1 – позиция первой вершины; vertPos2 – позиция второй вершины; vertex1 – исходная вершина; vertex2 – конечная вершина (в которую идет ребро, вес которого ищет функция).

Код метода GetWeight(const T & vertex1, const T & vertex2):

```

template<class T>
int Graph<T>::GetWeight(const T & vertex1, const T &
vertex2) {
    if (!this->IsEmpty()) {
        int vertPos1 = GetVertPos(vertex1);
        int vertPos2 = GetVertPos(vertex2);
        return adjMatrix[vertPos1][vertPos2];
    }
    return 0;
}

```

Метод `std::vector<T> GetNbrs(const T& vertex)`

Рассмотрим граф, представленный на рис. 23.19.

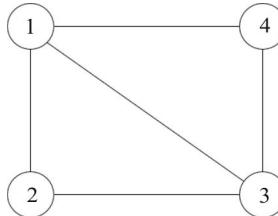


Рис. 23.19. Невзвешенный неориентированный
граф (метод GetNbrs)

Матрица смежности, соответствующая графу, выглядит следующим образом:

Номер вершины	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Опишем, как работает метод `GetNbrs`, на примере приведенного выше графа. Но перед описанием непосредственно работы алгоритма рассмотрим матрицу смежности. После ее анализа можно сделать следующие выводы:

1. Для вершины 1 соседями являются вершины 2, 3, 4.
2. Для вершины 2 соседями являются вершины 1 и 3.
3. Для вершины 3 соседями являются вершины 1, 2, 4.
4. Для вершины 4 соседями являются вершины 1 и 3.

Сначала метод получает вершину, соседей которой необходимо найти. Пусть требуется найти соседей вершины 1. Алгоритм работает следующим образом:

Создается список, состоящий из соседей (изначально он пуст), – `nbrsList`. В матрице смежности вычисляется позиция вершины, соседей которой необходимо найти. Вершина 1 имеет позицию `[1][1]` – пересечение первой строки и первого столбца. В специальную переменную запоминается значение 1.

Затем в ходе циклического процесса проверяется, что элемент с индексом строки, как у вершины 1, и индексом столбца i не равен 0 и что элемент с индексом строки i и индексом столбца, равным индексу строки у вершины 1, не равен 0. Если это так, то найден сосед, он заносится в список соседей. Например, проверяется элемент [1][2]: обнаруживается, что он не равен нулю, т.е. между вершиной 1 и вершиной 2 есть ребро. Далее проверяется, что находится в ячейке [2][1]. Там тоже находится не ноль, значит, от вершины 2 до вершины 1 есть ребро, значит, они соседи. Вершина 2 заносится в список соседей. Подобные действия выполняются для других вершин. Если бы граф был ориентированный, то в циклическом процессе для поиска соседей проверялось бы, что в ячейках с индексом вершины 1 и индексом столбца i находятся не нули, а вторая проверка отсутствовала бы.

Функция возвращает список соседей.

Таким образом, в списке соседей окажутся вершины 2, 3, 4. После анализа матрицы смежности было выяснено, что соседями вершины 1 являются вершины 2, 3, 4. Значит, функция работает корректно.

Краткий словарь

GetNbrs (GetNeighbors) – получить соседей; nbrsList (neighborsList) – список соседей (neighbors); pos – позиция.

Код метода GetNbrs(const T& vertex) для взвешенного и невзвешенного ориентированных графов:

```
template<class T>
std::vector<T> Graph<T>::GetNbrs(const T & vertex) {
    std::vector<T> nbrsList; // Создание списка соседей
    int pos = this->GetVertPos(vertex);
    /* Вычисление позиции vertex
       в матрице смежности */
    if (pos != (-1)) {
        /* Проверка, что vertex
           есть в матрице смежности */
        for (int i = 0, vertListSize = this-
>vertList.size(); i < vertListSize; ++i) {
            if (this->adjMatrix[pos][i] != 0) // Вычисление соседей
```

```

        nbrsList.push_back(this->vertList[i]);
                                // Запись соседей в вектор
    }
}
return nbrsList;           // Возврат списка соседей
}

```

Код метода GetNbrs(const T& vertex) для взвешенного и не-взвешенного неориентированных графов:

```

template<class T>
std::vector<T> Graph<T>::GetNbrs(const T & vertex) {
    std::vector<T> nbrsList; // Создание списка соседей
    int vertPos = this->GetVertPos(vertex);
                                /* Вычисление позиции
                                   vertex в матрице смежности */
    if (vertPos != (-1)) {
        // Проверка, что vertex есть в матрице смежности

        for (int i = 0, vertListSize = this->vertList.size(); i <
             vertListSize; ++i) {
            if (this->adjMatrix[vertPos][i] != 0 &&
                this->adjMatrix[i][vertPos] != 0)
                /* Вычисление соседей */

                nbrsList.push_back(this->vertList[i]);
                                /* Запись соседей в вектор */
        }
    }
    return nbrsList;           // Возврат списка соседей
}

```

Метод void InsertVertex(const T& vertex)

Метод служит для добавления вершины. Проверяет, что вектор вершин не полон, и добавляет туда вершину vertex, ссылку на которую он получил.

Краткий словарь

InsertVertex – вставить вершину.

Код метода void InsertVertex(const T& vertex):

```

template<class T>
void Graph<T>::InsertVertex(const T& vertex) {
    if (!this->IsFull()) {
        this->vertList.push_back(vertex);
    }
    else {
        cout << "Граф уже заполнен. Невозможно
добавить новую
вершину " << endl;
        return;
    }
}

```

Метод void InsertEdge(const T& vertex1, const T& vertex2)

Метод вставляет между вершинами vertex1 и vertex2 ребро весом weight (для взвешенного графа). Сначала проверяется, что vertex1 и vertex2 есть в векторе вершин, затем берутся их номера и проверяется, что между ними еще нет ребра: если нет, то в матрицу смежности вставляется значение weight (или 1 – для невзвешенного графа), если есть, то выводится об этом сообщение на экран. Если граф ориентированный, то вставляется значение в матрицу смежности по индексам [vertPos1][vertPos2]. Если граф неориентированный, то еще вставляется значение по индексам [vertPos2][vertPos1]. Если граф невзвешенный, то третий параметр weight не нужен.

Краткий словарь

InsertEdge – вставить ребро.

Код метода void InsertEdge(const T& vertex1, const T& vertex2) для взвешенного ориентированного графа:

```

template<class T>
void Graph<T>::InsertEdge(const T & vertex1, const T &
vertex2, int weight) {
    if (this->GetVertPos(vertex1) != (-1) && this-
>GetVertPos(vertex2) != (-1)) {
        int vertPos1 = GetVertPos(vertex1);
        int vertPos2 = GetVertPos(vertex2);
        if (this->adjMatrix[vertPos1][vertPos2] != 0)
{
            cout << "Ребро между этими вершинами
уже существует"
            << endl;
}
}

```

```

                return;
            }
        else {
            this->adjMatrix[vertPos1][vertPos2] =
weight;
        }
    else {
        cout << "Обеих вершин (или одной из них) нет
в графе "
        << endl;
        return;
    }
}

```

Код метода void InsertEdge(const T& vertex1, const T& vertex2) для взвешенного неориентированного графа:

```

template<class T>
void Graph<T>::InsertEdge(const T & vertex1, const T &
vertex2, int weight) {
    if (this->GetVertPos(vertex1) != (-1) && this->
->GetVertPos(vertex2) != (-1)) {
        int vertPos1 = GetVertPos(vertex1);
        int vertPos2 = GetVertPos(vertex2);
        if (this->adjMatrix[vertPos1][vertPos2] != 0
            && this->adjMatrix[vertPos2][vertPos1]
!= 0) {
            cout << "Ребро между вершинами уже
есть" << endl;
            return;
        }
        else {
            this->adjMatrix[vertPos1][vertPos2] =
weight;
            this->adjMatrix[vertPos2][vertPos1] =
weight;
        }
    }
    else {
        cout << "Обеих вершин (или одной из них) нет
в графе " << endl;
        return;
    }
}

```

Код метода void InsertEdge(const T& vertex1, const T& vertex2) для невзвешенного ориентированного графа:

```
template<class T>
void Graph<T>::InsertEdge(const T & vertex1, const T &
vertex2) {
    if (this->GetVertPos(vertex1) != (-1) && this->
->GetVertPos(vertex2) != (-1)) {
        int vertPos1 = GetVertPos(vertex1);
        int vertPos2 = GetVertPos(vertex2);
        if (this->adjMatrix[vertPos1][vertPos2] != 0)
{
            cout << "Ребро между этими вершинами
уже существует"
            << endl;
            return;
        }
        else {
            this->adjMatrix[vertPos1][vertPos2] =
1;
        }
    }
    else {
        cout << "Обеих вершин (или одной из них) нет
в графе "
        << endl;
        return;
    }
}
```

Код метода void InsertEdge(const T& vertex1, const T& vertex2) для невзвешенного неориентированного графа:

```
template<class T>
void Graph<T>::InsertEdge(const T & vertex1, const T &
vertex2) {
    if (this->GetVertPos(vertex1) != (-1) && this-
>GetVertPos(vertex2) != (-1)) {
        int vertPos1 = GetVertPos(vertex1);
        int vertPos2 = GetVertPos(vertex2);
        if (this->adjMatrix[vertPos1][vertPos2] != 0
            && this->adjMatrix[vertPos2][vertPos1]
!= 0) {
            cout << "Ребро между этими вершинами
уже существует"
            << endl;
```

```

                return;
            }
        else {
            this->adjMatrix[vertPos1][vertPos2] =
1;
            this->adjMatrix[vertPos2][vertPos1] =
1;
        }
    }
    else {
        cout << "Обеих вершин (или одной из них) нет
в графе "
        << endl;
        return;
    }
}

```

Метод void Print()

Этот метод используется для печати матрицы смежности графа. Самым первым печатается номер вершины, а затем значения из матрицы смежности для нее. Например, 3 0 1 0 1 будет означать, что вершина 3 не связана с вершинами 1 и 3, но связана с вершинами 2 и 4.

Код метода void Print():

```

template<class T>
void Graph<T>::Print() {
    if (!this->IsEmpty()) {
        cout << "Матрица смежности графа: " << endl;
        for (int i = 0, vertListSize = this-
>vertList.size(); i <
< vertListSize; ++i) {
            cout << this->vertList[i] << " ";
            for (int j = 0; j < vertListSize; ++j)
{
                cout << " " << this-
>adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
    else {
        cout << "Граф пуст " << endl;
    }
}

```

Файл main.cpp

Рассмотрим следующие графы (рис. 23.20, 23.21):

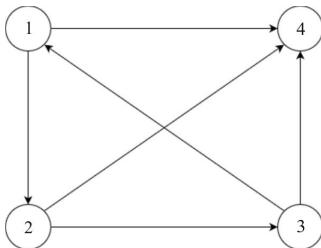


Рис. 23.20. Невзвешенный
ориентированный граф
(функция main())

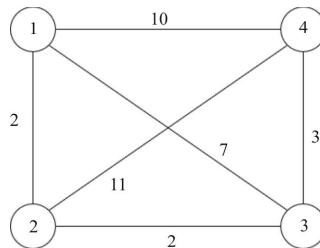


Рис. 23.21. Взвешенный
неориентированный граф
(функция main())

Здесь выполняется ввод вершин графа и вызываются функции.

Далее будут рассмотрены невзвешенный ориентированный (см. рис. 23.20) и взвешенный неориентированный (см. рис. 23.21) графы (так как в остальных случаях функции выглядят так же). Обе функции похожи, за исключением того, что для взвешенного графа еще требуется переменная веса.

В функции main() создается объект graph класса Graph типа int, а также объявляются переменные: amountVerts (количество вершин), amountEdges (количество ребер), vertex (вершина), sourceVertex, targetVertex (исходная и конечная вершины, sourceVertex – это вершина, из которой ребро будет идти в вершину targetVertex), для взвешенного графа еще edgeWeight (вес ребра).

Затем пользователь вводит нужное ему количество вершин и ребер. После этого в цикле пользователь вводит вершины.

Объявляется и инициализируется указатель на вершину (инициализируется адресом введенной вершины), затем у объекта graph вызывается метод InsertVertex, куда передается вершина.

После этого цикла идет ввод ребер. Ввод ребер осуществляется также в цикле, пользователь вводит стартовую вершину, конечную вершину и вес ребра (для взвешенного графа); после того как все данные введены, вызывается функция InsertEdge у объекта класса Graph с двумя или тремя параметрами (в зависимости от того, взве-

шенный граф или нет). После этого вызывается метод Print и выводится матрица смежности.

После этого программа завершает работу.

Краткий словарь

vertPtr (vertexPointer) – указатель на вершину; аналогично targetVertPtr (targetVertexPointer) – указатель на конечную (целевую) вершину; sourceVertPtr – указатель на исходную вершину.

Код файла main.cpp для невзвешенного ориентированного графа:

```
#include "Unweighted directed Graph.h"
#include <iostream>
#include <conio.h>
#include <locale>
using namespace std;
int main() {
    setlocale(LC_ALL, "Russian");
{
    Graph<int> graph; /* Создание графа,
    содержащего вершины с номерами целого типа */
    int amountVerts, amountEdges, vertex,
sourceVertex,
    targetVertex;
    // Создание необходимых для ввода графа переменных
    cout << "Введите количество вершин графа: ";
    cin >> amountVerts; cout << endl;
    // Ввод количества ребер графа в переменную amountVerts
    cout << "Введите количество ребер графа: ";
    cin >> amountEdges; cout << endl;
    // Ввод количества ребер графа в переменную amountEdges
    for (int i = 0; i < amountVerts; ++i) {
        cout << "Вершина: "; cin >> vertex; // Ввод вершины
        int* vertPtr = &vertex; /* Запоминаем
        адрес вершины с помощью указателя */
        graph.InsertVertex(*vertPtr);
    /* Передаем ссылку на вершину в функцию InsertVertex;
        происходит вставка вершины в вектор вершин */
        cout << endl;
    }
    for (int i = 0; i < amountEdges; ++i) {
        cout << "Исходная вершина: "; cin >>
sourceVertex;
```

```

        cout << endl;           // ввод исходной вершины
        int* sourceVertPtr = &sourceVertex;
        /* Запоминаем адрес исходной вершины */
        cout << "Конечная вершина: "; cin >>
targetVertex;
        cout << endl;
        /* Ввод вершины, до которой будет идти ребро от
           исходной вершины */
        /* Запоминаем адрес конечной вершины (до которой
           будет идти ребро от исходной вершины) */
        int* targetVertPtr = &targetVertex;
        /* Вставка ребра между исходной и конечной
           вершинами, в функцию передаются ссылки на исходную и
           конечную вершины */
        graph.InsertEdge(*sourceVertPtr, *targetVertPtr);
    }
    cout << endl;
    graph.Print();      // Печать матрицы смежности графа
}
_getch();
return 0;
}

```

Файл main.cpp для взвешенного неориентированного графа:

```

#include "Weighted undirected Graph.h"
#include <iostream>
#include <conio.h>
#include <locale>
using namespace std;

int main() {
    setlocale(LC_ALL, "Russian");
{
        Graph<int> graph;           /* Создание графа,
           содержащего вершины с номерами целого типа */
        int amountVerts, amountEdges, vertex, sourceVertex,
targetVertex, edgeWeight;
        /* Создание необходимых для ввода графа переменных
           (добавилась переменная edgeWeight) */
        cout << "Введите количество вершин графа: ";
        cin >> amountVerts; cout << endl;
// Ввод количества ребер графа в переменную amountVerts

```

```

        cout << "Введите количество ребер графа: ";
cin >> amountEdges; cout << endl;
        // Ввод количества ребер графа в amountEdges
        for (int i = 0; i < amountVerts; ++i) {
cout << "Вершина: "; cin >> vertex; // Ввод вершины
        int* vertPtr = &vertex;
/* Запоминаем адрес вершины с помощью указателя */
graph.InsertVertex(*vertPtr);
/* Передаем ссылку на вершину в функцию InsertVertex;
происходит вставка вершины в вектор вершин */
        cout << endl;
    }
    for (int i = 0; i < amountEdges; ++i) {
cout << "Исходная вершина: "; cin >> sourceVertex;
cout << endl; // ввод исходной вершины
int* sourceVertPtr = &sourceVertex;
/* Запоминаем адрес исходной вершины */
cout << "Конечная вершина: "; cin >> targetVertex;
cout << endl;
/* Ввод вершины, до которой будет идти ребро от
исходной вершины (ввод конечной вершины) */
int* targetVertPtr = &targetVertex;
// Запоминаем адрес конечной вершины
cout << "Вес ребра: "; cin >> edgeWeight;
cout << endl;
/* Ввод числового значения веса ребра в
переменную edgeWeight */
graph.InsertEdge(*sourceVertPtr, *targetVertPtr,
edgeWeight);
/* Вставка ребра весом edgeWeight между
исходной и конечной вершинами */
    }
    cout << endl;
    graph.Print(); // Печать матрицы смежности
}
_getch();
return 0;
}

```

Результат работы программы:

Введите количество вершин графа: 4

Введите количество ребер графа: 6

Вершина: 1

Вершина: 2
Вершина: 3
Вершина: 4
Исходная вершина: 1
Конечная вершина: 2
Исходная вершина: 1
Конечная вершина: 4
Исходная вершина: 2
Конечная вершина: 4
Исходная вершина: 2
Конечная вершина: 3
Исходная вершина: 3
Конечная вершина: 4
Исходная вершина: 3
Конечная вершина: 1
Матрица смежности графа: // Граф представлен на рис. 23.20
1 0 1 0 1
2 0 0 1 1
3 1 0 0 1
4 0 0 0 0

Еще один пример результата работы программы:

Введите количество вершин графа: 4
Введите количество ребер графа: 6
Вершина: 1
Вершина: 2
Вершина: 3
Вершина: 4
Исходная вершина: 1
Конечная вершина: 2
Вес ребра: 2
Исходная вершина: 1
Конечная вершина: 4
Вес ребра: 10
Исходная вершина: 2
Конечная вершина: 3
Вес ребра: 2
Исходная вершина: 3
Конечная вершина: 4
Вес ребра: 3
Исходная вершина: 3

Конечная вершина: 1

Вес ребра: 7

Исходная вершина: 2

Конечная вершина: 4

Вес ребра: 11

Матрица смежности графа: // Граф представлен на рис. 23.21

1 0 2 7 10

2 2 0 2 11

3 7 2 0 3

4 10 11 3 0

23.4. Алгоритмы обхода графов

Обход графа в глубину

Обход графа – процедура однократного посещения всех вершин. Алгоритм обхода графа в глубину начинает выполнение с одной из вершин графа – начальной вершины, фиксирует информацию о посещении этой вершины и, перемещаясь по ребру, посещает соседние вершины. Обход графа в глубину применяется в различных вычислениях на графах, например в алгоритме Диница для поиска максимального потока в транспортной сети. Кроме того, правило левой руки при прохождении лабиринта (идти, ведя левой рукой по стенке) также является обходом в глубину (рис. 23.22). По завершении обхода все вершины окажутся пройденными – обработанными. Если при обходе встречается вершина, которая уже была пройдена, то повторной обработки делать не нужно.

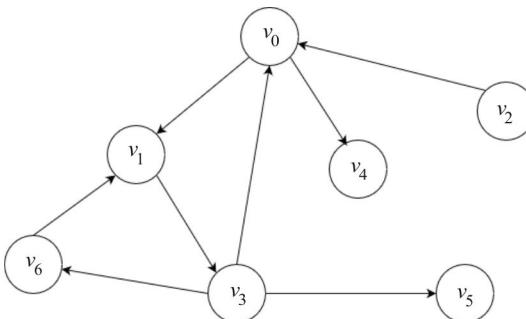


Рис. 23.22. Алгоритм обхода графа в глубину

Обход начинается с вершины v_0 . Неважно, какая именно обработка выполняется на каждой вершине, – может быть, выводится на печать метка вершины или выполняется более сложная процедура. После обработки вершины v_0 выполняется переход по одному из ребер к одной из соседних вершин. В данном примере существуют две возможности: перейти к вершине v_1 или к вершине v_4 . Перейти от вершины v_0 к вершине v_2 или v_3 невозможно, поскольку ребра, содержащие их, направлены в противоположную сторону. Итак, алгоритм обхода стоит перед выбором: перейти к вершине v_1 или к вершине v_4 . Не имеет значения, как именно сделать этот выбор, – допустим, что выбрана вершина v_1 . После ее обработки рисунок будет выглядеть следующим образом (рис. 23.23).

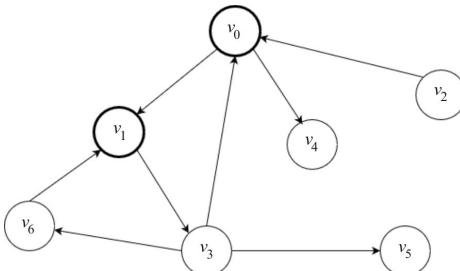


Рис. 23.23. Обработаны вершины v_0 и v_1

Затем алгоритм совершает переход к вершине, соседствующей с вершиной v_1 . Если соседняя вершина уже обработана, то алгоритм продвигается дальше, пропуская уже обработанные соседние вершины. В данном примере рассматривается только одна соседняя вершина v_3 и обрабатывается. В данный момент обработанными являются три вершины, как показано ниже (рис. 23.24).

Одной из соседних вершин по отношению к вершине v_3 является вершина v_0 , однако она уже обработана. Таким образом, чтобы предотвратить зацикливание, переходить от вершины v_3 в вершину v_0 нельзя. Исходя из этого, можно перейти к другому соседу: v_5 или v_6 . Переядем в вершину v_5 .

После ее обработки на графике будут цветом отмечены вершины v_0 и v_1 как посещенные (рис. 23.25).

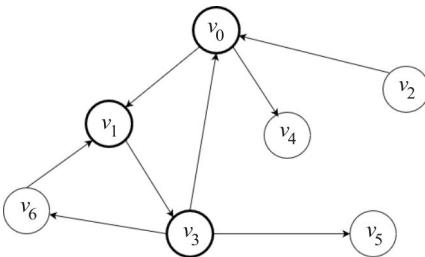


Рис. 23.24. Обработаны вершины v_0 , v_1 и v_3

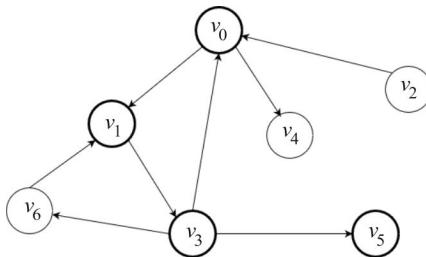


Рис. 23.25. Обработаны вершины v_0 , v_1 , v_3 и v_5

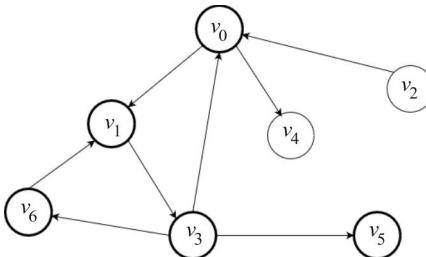


Рис. 23.26. Вершина v_6 обработана
после возврата к вершине v_3

Поскольку у вершины v_5 нет соседних вершин, дальнейший поиск в глубину невозможен. Далее алгоритм возвращается назад и проверяет, нет ли у предыдущей вершины v_3 других необработанных соседних вершин. У вершины v_3 есть еще необработанная соседняя вершина v_6 . Выполняются переход в вершину v_6 и обработка вершины v_6 . После обработки вершины v_6 рисунок имеет следующий вид (рис. 23.26).

У вершины v_6 есть соседняя вершина – v_1 , однако она уже обработана. Таким образом, у вершины v_6 нет обработанных соседних вершин, поэтому обход возвращается назад – в вершину v_3 , чтобы проверить, остались ли у нее другие необработанные соседи. У вершины v_3 больше нет необработанных соседей. Значит, алгоритм возвращается к предыдущей вершине v_1 , чтобы проверить, есть ли у нее непомеченные соседние вершины. У вершины v_1 нет необработанных соседних вершин, поэтому выполняется возврат в вершину v_0 для проверки, есть ли у вершины v_0 необработанные соседи. Такой сосед есть – это вершина v_4 , поэтому алгоритм переходит в нее. После перехода граф выглядит так, как показано на рис. 23.27.

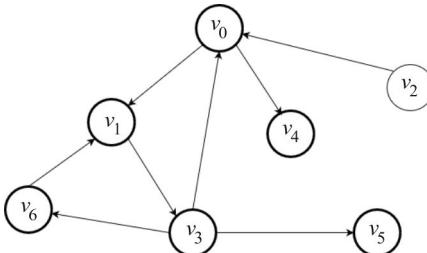


Рис. 23.27. Вершина v_4 обработана

У вершины v_4 соседних вершин нет, поэтому происходит возврат в вершину v_0 . У вершины v_0 больше нет необработанных соседних вершин. Поскольку вершина v_0 была отправной точкой нашего обхода, дальнейшее перемещение по графу невозможно (из вершины v_0 в вершину v_2 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 [8].

Рассмотрим программную реализацию обхода графа в глубину.

Краткий словарь

visitedVerts – посещенные вершины (vertices); startVertex – начальная вершина; neighbors – соседи; size – размер; DFS (DepthFirstSearch) – поиск в глубину.

В заголовочном файле появится массив посещенных вершин
bool *visitedVerts = new bool[20], о нем будет рассказано позже. Алгоритм обхода графа в глубину выполняет функция void DFS(T& startVertex, bool *visitedVerts).

Она принимает в качестве входных параметров ссылку на вершину, из которой нужно начать обход графа, и указатель на первый элемент массива из логических значений. В этом массиве индекс каждой вершины соответствует индексу вершины в векторе вершин vertList. Если в visitedVerts элемент по какому-либо индексу указывает на 1, это значит, что вершина по этому индексу уже посещена. Например, если *visitedVerts[0] = 1, то вершина vertList[0] уже обработана. С помощью массива visitedVerts помечаются посещенные вершины.

Как только выполнение программы переходит в функцию, происходит следующее:

- Выводится сообщение о том, что вершина startVertex посещена.
- В массиве visitedVerts по индексу вершины startVertex в векторе vertList устанавливается значение true (вершина посещена).
- Создается вектор вершин, соседних с startVertex, в него записываются соседи startVertex с помощью функции GetNbrs.
- Работает цикл с параметром. Он работает, пока значение переменной цикла меньше размера вектора соседей startVertex. В цикле осуществляется проверка, что вершина по индексу i (переменная цикла) в векторе соседей не посещена (т.е. что в массиве visitedVerts по индексу this->GetVertPos(neighbors[i]) стоит не значение true). Если это так, то функция DFS вызывается для соседа neighbors[i]. В функцию передается этот сосед и наш массив visitedVerts.

Затем алгоритм повторится уже для соседа. На экран выводится сообщение, что сосед посещен, в visitedVerts записывается значение true по нужному индексу, а затем для соседей переданного в функ-

цию соседа выполняются те же действия. Как только у кого-то не окажется соседей, пойдет возврат из рекурсии.

В функции main() после печати графа добавится несколько строк – ввод стартовой вершины и вызов функции DFS().

Код метода void DFS(T& startVertex, bool *visitedVerts):

```
template<class T>
void Graph<T>::DFS(T& startVertex, bool *visitedVerts) {
    // Выводим сообщение о том, что вершина посещена
    cout << "Вершина " << startVertex << " посещена" << endl;
    // Отмечаем в массиве посещенных вершин, что
    // вершина посещена
    visitedVerts[this->GetVertPos(startVertex)] = true;
    std::vector<T> neighbors = this-
>GetNbrs(startVertex);
    // Создаем вектор соседей
    for (int i = 0, size = this-
>GetNbrs(startVertex).size();
        i < size; ++i) {
        if (visitedVerts[this-
>GetVertPos(neighbors[i])] != true)
            /* В цикле проверяем, что соседи текущей вершины еще не
               посещены и вызываем функцию обхода графа в глубину */
            this->DFS(neighbors[i], visitedVerts);
    }
}
```

Код файла main.cpp:

```
#include "Unweighted directed Graph.h"
#include <iostream>
#include <conio.h>
#include <locale>
using namespace std;

int main() {
    setlocale(LC_ALL, "Russian");
{
    Graph<int> graph;
    /* Создание графа, содержащего вершины с
       номерами целого типа */
```

```

int amountVerts, amountEdges, vertex, sourceVertex,
targetVertex;
// Создание необходимых для ввода графа переменных
cout << "Введите количество вершин графа: ";
cin >> amountVerts; cout << endl;
// Ввод количества ребер графа в переменную amountVerts
cout << "Введите количество ребер графа: ";
cin >> amountEdges;
cout << endl;
/* Ввод количества ребер графа в
переменную amountEdges */
for (int i = 0; i < amountVerts; ++i) {
    cout << "Вершина: "; cin >> vertex; // Ввод вершины
    int* vertPtr = &vertex; /* Запоминаем адрес вершины
с помощью указателя */
    graph.InsertVertex(*vertPtr);
    /* Передаем ссылку на вершину в функцию
InsertVertex;
происходит вставка вершины в вектор вершин */
    cout << endl;
}
for (int i = 0; i < amountEdges; ++i) {
    cout << "Исходная вершина: ";
    cin >> sourceVertex;
    cout << endl; // Ввод исходной вершины
    int* sourceVertPtr = &sourceVertex;
    /* Запоминаем адрес исходной вершины */
    cout << "Конечная вершина: "; cin >> targetVertex;
    cout << endl;
    /* Ввод вершины, до которой будет идти ребро
от исходной вершины. Запоминаем адрес конечной вершины
(до которой будет идти ребро от исходной вершины) */
    int* targetVertPtr = &targetVertex;
    /* Вставка ребра между исходной и конечной
вершинами, в функцию передаются ссылки на исходную и
конечную вершины */
    graph.InsertEdge(*sourceVertPtr,
*targetVertPtr); }
cout << endl;
graph.Print(); // Печать матрицы смежности графа
}
cout << "Введите вершину, с которой начать обход: ";

```

```

    cin >> vertex;
    cout << endl; /* Ввод начальной вершины, с которой
начнется обход графа в глубину (в нашем случае это 0) */
    int* vertPtr = &vertex; /* Запоминаем адрес
введенной вершины */
    /* Вызываем функцию обхода графа в глубину,
в функцию передаются ссылка на введенную вершину
и вектор посещенных вершин */
    graph.DFS(*vertPtr, visitedVerts);
}

```

В качестве результата приведена матрица смежности и выведенные сообщения во время выполнения метода DFS, поскольку ввод данных был продемонстрирован ранее.

Матрица смежности графа: // график представлен на рис. 23.22

```

0 0 1 0 0 1 0 0
1 0 0 0 1 0 0 0
2 1 0 0 0 0 0 0
3 1 0 0 0 0 1 1
4 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0

```

Введите вершину, с которой начать обход: 0

Вершина 0 посещена
Вершина 1 посещена
Вершина 3 посещена
Вершина 5 посещена
Вершина 6 посещена
Вершина 4 посещена

Обход графа в ширину

Существует еще один алгоритм обхода графа – обход в ширину (англ. Breadth First Search). Этот алгоритм использует очередь для отслеживания необработанных соседних вершин. Этот алгоритм обхода графа применяется для поиска компонентов связности в графике, 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 график, рассмотренный на рис. 23.22.

Предположим, что начальной является вершина v_0 . Тогда вершина является первой вершиной, подлежащей обработке, маркировке цветом и сохранению в очереди (рис. 23.28).

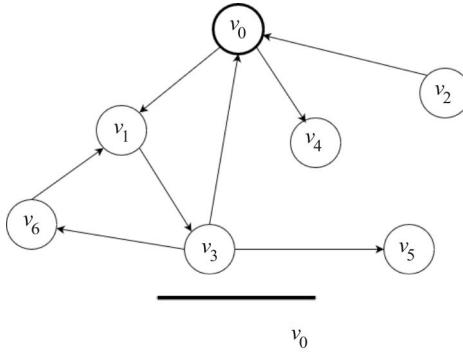


Рис. 23.28. Вершина v_0 промаркирована
и занесена в очередь

На данный момент в голове очереди находится начальная вершина v_0 .

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 вершиной. 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 v_0 находится в голове очереди. Вершина v_0 удаляется из очереди

и отмечается, что у нее остались необработанными две соседние вершины – v_1 и v_4 . Каждая из них обрабатывается, маркируется и помещается в очередь. Допустим, сначала в очередь поставлена вершина v_1 , а затем вершина v_4 (очередность может быть любой, вершина v_4 может быть поставлена первой, а вершина v_1 за ней – алгоритм будет работать корректно в любом случае). После того как вершины v_1 и v_4 были поставлены в очередь, граф выглядит следующим образом (рис. 23.29).

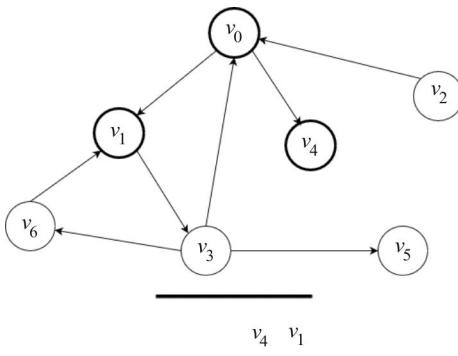


Рис. 23.29. Промаркированы вершины v_0 , v_1 , v_4 ,
в очереди находятся вершины v_1 и v_4

Поскольку очередь не пуста, два шага выполняются снова: первый элемент (вершина v_1) удаляется, обрабатывается, все необработанные соседние вершины маркируются и помещаются в очередь. Единственным неотмеченным соседом вершины v_1 является вершина v_3 , поэтому после обработки, маркировки и постановки в очередь вершины v_3 граф выглядит следующим образом (рис. 23.30).

Далее из головы очереди удаляется вершина v_4 . Поскольку у нее нет соседних вершин, никакие вершины больше не обрабатываются и не становятся в очередь, при этом граф выглядит следующим образом (рис. 23.31).

Следующей из очереди удаляется вершина v_3 . У нее есть две непомеченные соседние вершины (вершины v_5 и v_6), которые обрабатываются, маркируются и помещаются в очередь. Граф выглядит следующим образом (рис. 23.32).

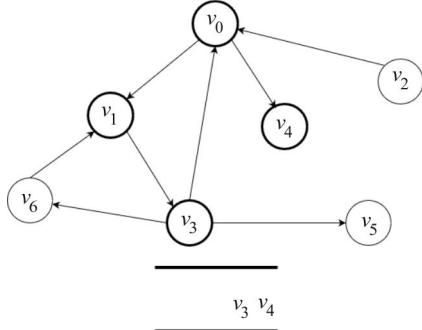


Рис. 23.30. Промаркированы вершины v_0, v_1, v_4 , v_3 , v_4 в очереди находятся вершины v_3 и v_4

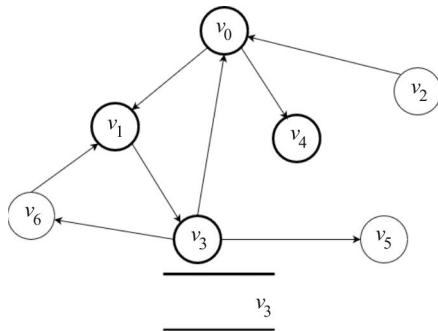


Рис. 23.31. Промаркированы вершины v_0, v_1, v_4 , v_3 , в очереди находится только вершина v_3

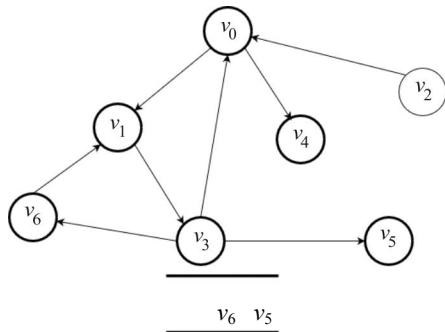


Рис. 23.32. Промаркированы вершины v_0, v_1, v_4, v_3 , v_5, v_6 , в очереди находятся вершины v_5 и v_6

Вершина v_0 не обрабатывается повторно, так как она уже помечена.

Результат обхода графа в глубину и в ширину одинаков – отличается порядок обработки. Вершины v_0, v_1, v_3, v_4, v_5 и v_6 обработаны, поскольку все они достижимы из начальной точки (вершины 0). Вершина v_2 не обрабатывается, поскольку не существует пути из вершины v_0 в вершину v_2 . При обходе графа в ширину вершины обрабатываются в следующем порядке: сначала вершина v_0 , потом v_1 и v_4 , затем обрабатываются их соседи (вершина v_3) и т.д. При обходе графа в глубину сначала обрабатываются вершины v_0, v_1, v_3 и v_5 [8].

Краткий словарь

VertsQueue (VerticesQueue) – очередь вершин.

Алгоритм обхода графа в ширину реализован в функции void BFS(T& startVertex, bool *visitedVerts). Рассмотрим ее работу.

Она, как и функция DFS, принимает в качестве входных параметров вершину, с которой надо начать обход графа, и массив значений булева типа для вершины (посещена или нет вершина по определенному индексу; индекс в массиве visitedVerts совпадает с номером вершины в векторе вершин). Кроме того, в описании класса появляется очередь std::queue<T> VertsQueue () .

Когда выполнение программы переходит внутрь функции BFS, выполняется проверка, является ли посещенной вершиной startVertex. Если ее еще не посещали, то она заносится в очередь, затем на экран выводится сообщение о том, что вершина обработана, и в массиве visitedVerts для нее устанавливается значение true. Если вершина обработана, ничего не происходит, управление передается следующим операторам программы: выделяются соседи вершины startVertex, из очереди удаляется голова, затем в цикле для всех соседей выполняются следующие действия.

- Выполняется проверка, посещен ли сосед.
- Если не посещен, то эта соседняя вершина помещается в очередь и на экран выводится сообщение о том, что вершина обработана.

После работы этого цикла работает следующий цикл:

- Пока очередь не пуста, циклически вызывается функция BFS, в которую в качестве начальной вершины передается головная вершина очереди.

- В функции main() после ввода всех вершин и ребер пользователь вводит начальную вершину и вызывается функция BFS.

Код функции void BFS(T& startVertex, bool *visitedVerts):

```
template<class T>
void Graph<T>::BFS(T & startVertex, bool *visitedVerts) {
    // Проверяем, была ли ранее посещена вершина startVertex
    if (visitedVerts[this->GetVertPos(startVertex)] == false)
    { this->VertsQueue.push(startVertex);
        /* Если startVertex еще не была посещена,
           то она заносится в очередь вершин VertsQueue */
        // Выводится сообщение, что startVertex теперь посещена
        cout << "Вершина " << startVertex << " обработана" <<
        endl;
        /* В массиве посещенных вершин отмечаем,
           что вершина по индексу, который на данный момент имеет
           startVertex, является посещенной */
        visitedVerts[this->GetVertPos(startVertex)] = true;
    }
    /* Создаем вектор соседей startVertex
    std::vector<T> neighbors = this->GetNbrs(startVertex);
    this->VertsQueue.pop();      // Удаляем голову очереди
        /* В цикле проверяем, посещен ли сосед вершины
        startVertex, если он еще не посещен, то он помещается в
        очередь вершин VertsQueue */
    for (int i = 0, size = neighbors.size(); i < size; ++i)
    {
        if (visitedVerts[this->GetVertPos(neighbors[i])] != true)
        {
            this->VertsQueue.push(neighbors[i]);
            // А также он помещается в массив посещенных вершин
            visitedVerts[this->GetVertPos(neighbors[i])] = true;
            cout << "Вершина " << neighbors[i] << " обработана"
            << endl;
            /* Затем на экран выводится сообщение,
               что соседняя вершина обработана */
        }
    }
    while (!this->VertsQueue.empty()) {
        /* В цикле вызывается функция обхода графа в ширину
           до тех пор, пока в очереди есть вершины */
        this->BFS(this->VertsQueue.front(), visitedVerts);
    }
}
```

Код файла main.cpp:

```
#include "Unweighted directed Graph.h"
#include <iostream>
#include <conio.h>
#include <locale>
using namespace std;
int main() {
    setlocale(LC_ALL, "Russian");
{
    Graph<int> graph;           /* Создание графа,
        содержащего вершины с номерами целого типа */
    int amountVerts, amountEdges, vertex, sourceVertex,
    targetVertex;
    // Создание необходимых для ввода графа переменных
    cout << "Введите количество вершин графа: ";
    cin >> amountVerts;
    cout << endl;
/* Ввод количества ребер графа в переменную amountVerts */
    cout << "Введите количество ребер графа: ";
    cin >> amountEdges;
    cout << endl;           /* Ввод количества ребер графа в
                                переменную amountEdges */
    for (int i = 0; i < amountVerts; ++i) {
        cout << "Вершина: "; cin >> vertex; // Ввод вершины
        int* vertPtr = &vertex;
        /* Запоминаем адрес вершины с
            помощью указателя */
        graph.InsertVertex(*vertPtr);
        /* Передаем ссылку на вершину в функцию
        InsertVertex; происходит вставка вершины в вектор
            вершин */
        cout << endl;
    }
    for (int i = 0; i < amountEdges; ++i) {
        cout << "Исходная вершина: "; cin >> sourceVertex;
        cout << endl;           // ввод исходной вершины
        int* sourceVertPtr = &sourceVertex;
        /* Запоминаем адрес исходной вершины */
        cout << "Конечная вершина: ";
        cin >> targetVertex;
        cout << endl;
```

```

        /* Ввод вершины, до которой будет идти
           ребро от исходной вершины */
        /* Запоминаем адрес конечной вершины
           (до которой будет идти ребро от исходной вершины) */
        int* targetVertPtr = &targetVertex;
        /* Вставка ребра между исходной и конечной
           вершинами, в функцию передаются ссылки на исходную и
           конечную вершины */
        graph.InsertEdge(*sourceVertPtr, *targetVertPtr);
    }
    cout << endl;
    graph.Print(); // Печать матрицы смежности графа
}
cout << "Введите вершину, с которой начать обход: ";
cin >> vertex;
cout << endl; /* Ввод начальной вершины, с которой
                  начнется обход графа в глубину (в нашем
                  случае это 0) */
int* vertPtr = &vertex; /* Запоминаем адрес
                           введенной вершины */
/* Вызываем функцию обхода графа в ширину,
   в функцию передаются ссылка на введенную
   вершину и вектор посещенных вершин */
graph.BFS(*vertPtr, visitedVerts);
}
_getch();
return 0;
}

```

Результат работы программы:

```

Матрица смежности графа: // Граф представлен на рис. 23.22
0 0 1 0 0 1 0 0
1 0 0 0 1 0 0 0
2 1 0 0 0 0 0 0
3 1 0 0 0 0 1 1
4 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0
Введите вершину, с которой начать обход: 0
Вершина 0 обработана
Вершина 1 обработана

```

Вершина 4 обработана
Вершина 3 обработана
Вершина 5 обработана
Вершина 6 обработана

23.5. Поиск кратчайших путей в графах

Одной из задач оптимизации является решение задачи нахождения кратчайшего пути. Особенно актуальными являются задачи выстраивания маршрутов грузоперевозок, что влечет за собой одновременное решение задач экономических затрат (топливо для автомобилей, зарплаты сотрудникам, затраты на восстановление запасных деталей и сервисное обслуживание). В этом подразделе будут рассмотрены два алгоритма поиска кратчайших путей: алгоритм Дейкстры, который рассчитывает кратчайшие расстояния от одной вершины графа до всех остальных, и алгоритм Флойда, в котором рассчитываются кратчайшие пути – от каждой из вершин до всех остальных.

Алгоритм Дейкстры

Краткий словарь

AllVisited – все посещены; FillLabels – заполнить метки; labelList – список меток; Dijkstra – Дейкстра; curSrc – текущая исходная вершина; flag – флагок; minLabel – минимальная метка; counter – счетчик; count – количество.

В 1959 году нидерландский ученый Эдсгер Дейкстра разработал алгоритм поиска минимального расстояния от заданного пункта вершины до всех остальных пунктов (вершин). Алгоритм Дейкстры работает только в графах без ребер с отрицательным весом (рис. 23.33).

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 неизвестны. В начале алгоритма все вершины графа помечаются как необработанные (рис. 23.34).

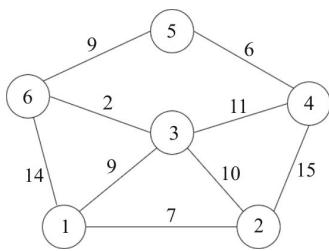


Рис. 23.33. Поиск кратчайшего пути по алгоритму Дейкстры

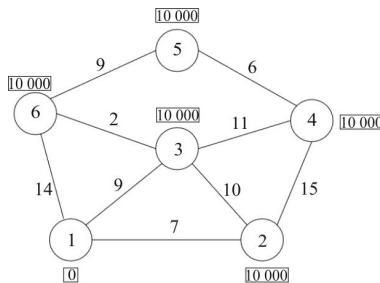


Рис. 23.34. Вершинам графа поставлены соответствующие метки

Первый шаг алгоритма:

Минимальную метку, равную нулю, имеет вершина 1. Ее соседями являются вершины 2, 3 и 6. Алгоритм обходит соседей вершины по очереди. Первый сосед вершины 1 – вершина 2, так как длина пути до нее минимальна. Длина пути в нее через вершину 1 равна сумме кратчайшего расстояния до вершины 1 (значению ее метки) и длины ребра, идущего из вершины 1 в вершину 2, т.е.

$$0 + 7 = 7.$$

Из двух меток – текущей, равной 10 000, и рассчитанной, равной 7, – выбирается наименьшая, поэтому новая метка 2-й вершины будет равна 7, что показано на рис. 23.35.

Аналогично находятся длины путей для всех других соседей – вершин 3 и 6. До вершины 3 от вершины 1 кратчайшим будет расстояние в 9 единиц:

$$0 + 9 = 9,$$

а до вершины 6 кратчайшим расстоянием будет путь в 14 единиц:

$$0 + 14 = 14.$$

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 рис. 23.35, таким расстоянием будет путь, равный 7 единицам.

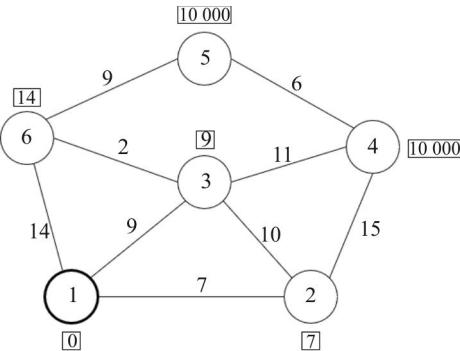


Рис. 23.35. Обработана вершина 1

Второй шаг алгоритма:

Снова находится «ближайшая» из необработанных вершин. Это вершина 2 с меткой 7. 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, 3 и 4. Вершина 1 уже обработана. Следующий сосед вершины 2 – вершина 3, так как имеет минимальную метку из вершин, отмеченных как необработанные. Длина пути от вершины 1 до вершины 3 составит 17 единиц (рис. 23.36):

$$7 + 10 = 17.$$

Но в вершину 3 из вершины 1 можно попасть напрямую, и длина пути будет составлять 9 единиц, что меньше расстояния, равного 17 единицам, поэтому метка вершины 3 остается равной 9. После произведенных действий график выглядит следующим образом:

Есть еще один сосед вершины 2 – вершина 4. Если двигаться к вершине 4 через вершину 2, то длина такого пути будет

$$7 + 15 = 22 \text{ (ед.)}.$$

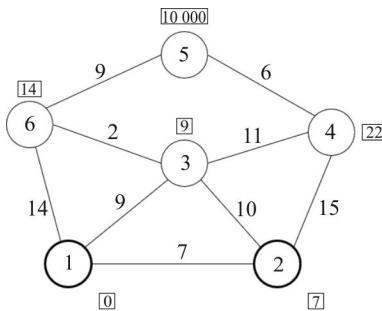


Рис. 23.36. Обработаны вершины 1 и 2

Поскольку 22 меньше, чем 10 000, метка вершины 4 устанавливается равной 22. Все соседи вершины 2 просмотрены, она помечается как обработанная.

Третий шаг:

Обрабатывается вершина 3, после ее обработки метка вершины меняется и составляет 20 единиц, а метка вершины 6 – 11 единиц. И граф выглядит следующим образом (рис. 23.37).

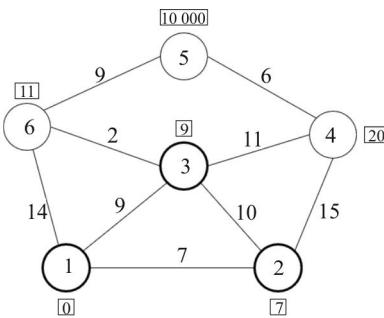


Рис. 23.37. Обработаны вершины 1, 2, 3

Четвертый шаг:

Из двух вершин – 4-й и 6-й – выбирается вершина с наименьшей меткой и отмечается как обработанная. На рис. 23.38 это вершина 6. Производится вычисление текущей метки вершины 5 по ребру между вершинами 6 и 5:

$$11 + 9 = 20.$$

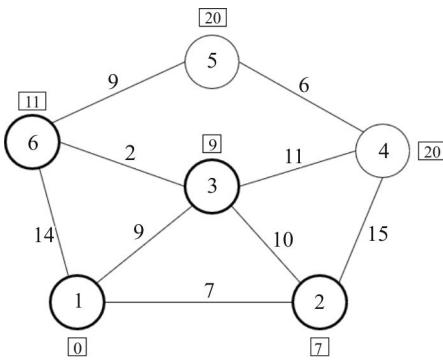


Рис. 23.38. Обработаны вершины 1, 2, 3, 6

Пятый шаг:

Обрабатывается вершина 4 и считается расстояние от нее до вершины 5:

$$20 + 6 = 26.$$

Поскольку расстояния от вершины 6 до вершины 5 меньше 26 единиц, метка вершины 5 остается неизменной (рис. 23.39).

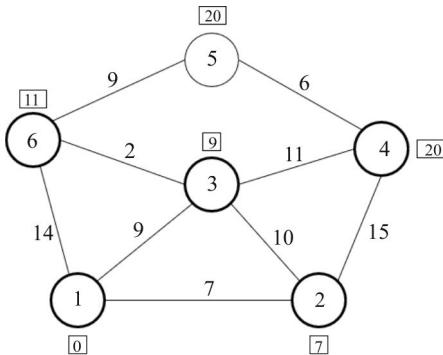


Рис. 23.39. Обработаны вершины 1, 2, 3, 6, 4

Шестой шаг:

Обработка вершины 5 (рис. 23.40).

Таким образом, кратчайший путь из вершины 1 в вершину 5 проходит через вершины 1–3–6–5 и имеет минимальное расстояние, равное 20 единицам.

Рассмотрим программную реализацию алгоритма Дейкстры. Алгоритм реализует функция `int Dijkstra(T& startVertex)`. Она возвращает кратчайшее расстояние от вершины `startVertex` до всех остальных вершин графа.

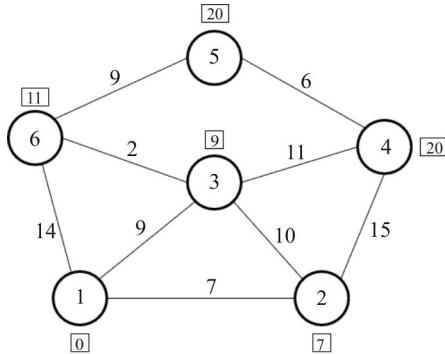


Рис. 23.40. Обработаны все вершины графа

Работа алгоритма:

В цикле проверяется, что в матрице смежности нет значений, меньших 0, поскольку алгоритм не работает для графов с ребрами отрицательного веса. Если такие есть, то функция возвращает значение, равное -1 . Если ребер с отрицательным весом нет, то программа продолжит работу.

Создается временная вершина абстрактного типа `T`, счетчик и метка, равная очень большому числу, пусть это будет $1\,000\,000$. В специально созданный вектор с именем `neighbors` записываются соседи начальной вершины. Вектор меток – `labelList` – с самого начала заполнен значениями $1\,000\,000$, кроме элемента, индекс которого соответствует индексу начальной вершины, он равен 0. Для его заполнения служит функция `FillLabels()`.

Затем проверяется, если в векторе меток метка по номеру вершины (индексу), соответствующему номеру вершины (индексу) соседа в векторе вершин `vertList`, больше, чем сумма метки текущей начальной вершины и ребра от этой вершины до соседа, то нужно в векторе меток по этому индексу поставить сумму метки начальной вершины и ребра от нее до соседа, что отражено на рис. 23.35.

После этого определяется наименьшая метка в векторе меток и проверяется, что рассмотрены все соседи. В массиве visitedVerts фиксируется, что текущая начальная вершина является обработанной (так как просмотрены все ее соседи). Затем выполняется поиск новой опорной вершины – такой, у которой из всех соседей метка наименьшая. Как только такая вершина найдена, у нее берутся соседи и записываются в вектор соседей.

Далее работает цикл до тех пор, пока не будут посещены все вершины. Во вложенном цикле проверяются соседи:

- Рассмотрена ли соседняя с текущей опорной вершиной вершина-сосед ранее.
- Если вершина не рассмотрена, то проверяется, если в векторе меток метка по индексу, соответствующему индексу соседа в векторе вершин vertList, больше, чем сумма метки текущей начальной вершины и ребра от этой вершины до соседа, то нужно в векторе меток по этому индексу поставить сумму метки начальной вершины и ребра от нее до соседа.
- Затем проверяется, больше ли текущая минимальная метка той метки, что расположена над текущим соседом. Если текущая метка больше, то минимальной метке присваивается значение метки над текущим рассматриваемым соседом. Если текущая метка меньше, то программа работает дальше.

Эти действия повторяются до тех пор, пока не будут обработаны соседи. Затем проверяется, что обработаны все соседи, и в массиве обработанных вершин для текущей опорной вершины устанавливается значение true. Затем снова определяется из соседних вершин вершина с наименьшей меткой, она становится текущей опорной вершиной, и у нее обрабатываются соседи.

Как только все вершины обработаны, это проверяет специальный метод bool AllVisited(bool *visitedVerts), который возвращает true, если в массиве visitedVerts у всех элементов значения true, в противном случае метод возвращает значение false.

На экран выводятся расстояния между начальной вершиной и всеми остальными.

Код метода AllVisited:

```
template<class T>
bool Graph<T>::AllVisited(bool * visitedVerts) {
    /* Устанавливаем в флагок начальное значение false,
       так как при объявлении переменные лучше
       инициализировать сразу */
    bool flag = true;
    /* В цикле проверяем, если в массиве посещенных
       вершин есть хотя бы одно значение false, то флагок
       переводится в значение false */
    for (int i = 0, size = this->vertList.size(); i < size;
        ++i) {
        if (visitedVerts[i] != true)
            flag = false;
        if (flag == false)
            return false;
        /* Если флагок равен false, то не все вершины
           посещены, функция возвращает значение false */
        else return true;
        /* Если флагок равен true, то все вершины
           посещены, функция возвращает true */
    }
}
```

Код метода FillLabels:

```
template<class T>
void Graph<T>::FillLabels(T& startVertex) {
    for (int i = 0, size = vertList.size(); i < size;
        ++i) {
        this->labelList.push_back(1000000);
        /* Заполнение списка меток значениями 1000000 */
    }
    int pos = this->GetVertPos(startVertex);
    /* По индексу вершины, от которой требуется найти
       кратчайшие расстояния до всех остальных,
       устанавливается метка 0 */

    this->labelList[pos] = 0;
}
```

Код метода Dijkstra:

```
template<class T>
int Graph<T>::Dijkstra(T & startVertex) {
```

```

        for (int i = 0, size = this->vertList.size(); i <
size; ++i) {
            for (int j = 0; j < size; ++j) {
                if (this->adjMatrix[i][j] < 0)
                    /* Проверяем, что граф не содержит ребер
                     отрицательного веса */
                return -1;
            }
        }
        /* Объявление опорной вершины абстрактного типа T,
         счетчика, переменной для определения
         минимальной метки */
    T curSrc; int counter = 0;
    int minLabel = 1000000;
    // Создаем вектор из соседей текущей начальной вершины
    std::vector<T> neighbors = this-
>GetNbrs(startVertex);
    /* Проверяем, если в векторе меток метка по индексу,
соответствующему индексу соседа в векторе вершин vertList,
больше, чем сумма метки текущей начальной вершины и ребра
от этой вершины до соседа, то нужно в векторе меток
по этому индексу поставить сумму метки начальной вершины
и ребра от нее до соседа */
    for (int i = 0; i < neighbors.size(); ++i) {
        if (this->labelList[this->GetVertPos(startVertex)] +
            this->GetWeight(startVertex, neighbors[i]) <
            this->labelList[this->GetVertPos(neighbors[i]))]
        {
            this->labelList[this->GetVertPos(neighbors[i])] =
            this->GetWeight(startVertex, neighbors[i]);
        }
        if (this->labelList[this->GetVertPos(neighbors[i])] 
            < minLabel)
            minLabel = this->labelList[this->
                ->GetVertPos(neighbors[i])];
            /* Определяем наименьшую метку у соседних
             опорной вершине вершин */
    }
    // Проверяем, что рассмотрели всех соседей
    for (int i = 0; i < neighbors.size(); ++i) {
        if (this->labelList[this-
>GetVertPos(neighbors[i])]
```

```

!= 1000000)
        counter += 1;
}
/* Фиксируем, что начальная вершина теперь считается
   посещенной (так как всех соседей посмотрели) */
if (counter == neighbors.size())
visitedVerts[this->GetVertPos(startVertex)] = true;
/* Ищем новую опорную вершину - такую, у которой из
   всех соседей метка наименьшая */
for (int i = 0; i < neighbors.size(); ++i) {
    if (this->labelList[this->GetVertPos(neighbors[i])] ==
minLabel)
        curSrc = neighbors[i];
}

/* Так как начальная вершина новая, то теперь мы
   будем рассматривать ее соседей */
neighbors = this->GetNbrs(curSrc);
// Работаем, пока все вершины не будут считаться
   посещенными
while (!AllVisited(visitedVerts)) {
    int count = 0;
    minLabel = 10000;
    for (int i = 0; i < neighbors.size(); ++i) {

        // Проверяем, была ли вершина-сосед
           уже рассмотрена ранее
        if (visitedVerts[this-
>GetVertPos(neighbors[i])] != true) {
            /* Проверяем, если в векторе меток метка по индексу,
               соответствующему индексу соседа в векторе вершин
               vertList, больше, чем сумма метки текущей начальной
               вершины и ребра от этой вершины до соседа, то нужно
               в векторе меток по этому индексу поставить сумму
               метки начальной вершины и ребра от нее до соседа */
            if (this->labelList[this->GetVertPos(curSrc)] +
+
GetWeight(curSrc, neighbors[i]) <
this->labelList[this->
->GetVertPos(neighbors[i])]) {
                this->labelList[this->
->GetVertPos(neighbors[i])] =

```

```

        (this-
>labelList[this->
                ->GetVertPos(curSrc)]
+ this->GetWeight(curSrc,
neighbors[i]));
}
if (this->labelList[this->
->GetVertPos(neighbors[i])] <
minLabel) {
    minLabel = this-
>labelList[this->
                ->GetVertPos(neighbors[i])];
}
}
count += 1; // Считаем соседей
}
/* Проверяем, что посетили всех соседей текущей опорной вершины: если да, то текущая опорная вершина отмечается как посещенная */
if (count == neighbors.size())
visitedVerts[this->GetVertPos(curSrc)] = true;
// Ищем среди соседей новую опорную вершину
for (int i = 0; i < neighbors.size(); ++i) {
if (this->labelList[this->
GetVertPos(neighbors[i])] == minLabel)
curSrc = neighbors[i];
}
// Заносим в вектор соседей соседей новой опорной вершины
neighbors = this->GetNbrs(curSrc);
}
// Вывод результатов на экран

for (int i = 0; i < this->GetVertPos(startVertex); ++i) {
cout << "Кратчайшее расстояние от вершины "
<< startVertex
                << " до вершины " << this->vertList[i]
<< " равно "
                << this->labelList[this->
GetVertPos(this->
                ->vertList[i])] << endl;
}

```

```

        for (int i = this->GetVertPos(startVertex) + 1; i <
this->
            ->vertList.size(); ++i) {
                cout << "Кратчайшее расстояние от вершины "
<< startVertex
                    << " до вершины " << this->vertList[i]
<< " равно "
                    << this->labelList[this-
>GetVertPos(this->
            ->vertList[i])] << endl;
    }
}

```

Файл main.cpp:

```

#include "Weighted undirected Graph.h"
#include <iostream>
#include <conio.h>
#include <locale>
using namespace std;
int main() {
    setlocale(LC_ALL, "Russian");
    {
        Graph<int> graph;
        int amountVerts, amountEdges, vertex,
sourceVertex,
        targetVertex, edgeWeight;
        cout << "Введите количество вершин графа: ";
        cin >> amountVerts; cout << endl;
        cout << "Введите количество ребер: ";
        cin >> amountEdges; cout << endl;

        for (int i = 0; i < amountVerts; ++i) {
            cout << "Вершина: "; cin >> vertex;
            int* VertPtr = &vertex;
            graph.InsertVertex(*VertPtr);
            cout << endl;
        }
        for (int i = 0; i < amountEdges; ++i) {
            cout << "Исходная вершина: ";
            cin >> sourceVertex; cout << endl;
            int* sourceVertPtr = &sourceVertex;
            cout << "Конечная вершина: ";

```

```

        cin >> targetVertex; cout << endl;
        int* targetVertPtr = &targetVertex;
        cout << "Вес ребра: ";
        cin >> edgeWeight; cout << endl;
graph.InsertEdge(*sourceVertPtr, *targetVertPtr,
edgeWeight);
}
cout << endl;
graph.Print();
cout << "Введите вершину, от которой хотите
найти расстояния до остальных: "; cin >> vertex; cout <<
endl;
/* Ввод вершины, от которой требуется найти
кратчайшие расстояния до всех остальных вершин
графа */
int* VertPtr = &vertex;
/* В указатель запоминается адрес начальной вершины */
graph.FillLabels(*VertPtr);
/* Устанавливаем начальные метки
вершинам графа с помощью функции FillLabels */
graph.Dijkstra(*VertPtr);
/* Ищем кратчайшие пути от вершины
vertex до всех остальных вершин
графа с помощью функции Dijkstra */
}
_getch();
return 0;
}

```

Результат работы программы:

Матрица смежности графа: // Граф представлен на рис. 23.34

1	0	7	9	0	0	14
2	7	0	10	15	0	0
3	9	10	0	11	0	2
4	0	15	11	0	6	0
5	0	0	0	6	0	9
6	14	0	2	0	9	0

Введите вершину, от которой хотите найти расстояния до остальных:

Кратчайшее расстояние от вершины 1 до вершины 2 равно 7
 Кратчайшее расстояние от вершины 1 до вершины 3 равно 9

Кратчайшее расстояние от вершины 1 до вершины 4 равно 20

Кратчайшее расстояние от вершины 1 до вершины 5 равно 20

Кратчайшее расстояние от вершины 1 до вершины 6 равно 11

Алгоритм Флойда

Краткий словарь

vertListSize (vertexListSize) – размер списка вершин; shortestPathMatrix – матрица кратчайших путей – минимальных затрат; secondMatrix – вторая матрица; PrintSPMatrix (PrintShortestPathsMatrix) – печатать матрицу кратчайших путей; Route – маршрут; Verts (Vertices) – вершины; weight – вес; isVertexExists – существует ли вершина; routes – маршруты; routesToWatch – маршруты к просмотру; currentRoute – текущий маршрут; routeToWatch – маршрут к просмотру; lastRouteVertex – последняя вершина маршрута; neighbor – сосед; endVertex – конечная вершина; stepVertex – промежуточная вершина; shortestRoutes – кратчайшие маршруты; minimWeight – минимальный вес; shortestRoute – кратчайший маршрут; minWeight – наименьший вес; PrintSP – печатать кратчайшие пути; cur – текущая вершина; col – текущий столбец.

Алгоритм (рис. 23.41) разработан Робертом Флойдом и Стивеном Уоршеллом и предназначен для нахождения кратчайшего пути от каждой вершины графа до всех остальных. Практическим применением данного алгоритма может быть, например, определение минимального временного интервала для доставки письма [13].

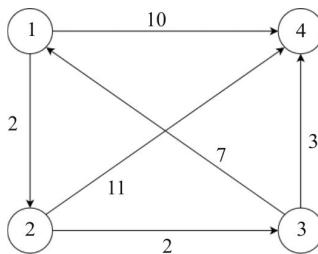


Рис. 23.41. Алгоритм Флойда

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.

Первая матрица в начальном состоянии:

Номер вершины	1	2	3	4
1	0	2	10 000	10
2	10 000	0	2	11
3	7	10 000	0	3
4	10 000	10 000	10 000	0

В первой матрице значения 10 000 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 10 000 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 в начальном состоянии:

Номер 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

Поясним, какие значения там хранятся: во второй матрице хранятся номера вершин, в которые надо пройти, чтобы попасть в желаемую. Иными словами, 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. Поскольку путь из любой вершины до самой себя равен 0, на главной диагонали матрицы стоят нули. Также нули стоят там, куда невозможно пройти 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, т.е. 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, т.е. 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 следующим образом.

Первая матрица (от вершины 1 найдено более короткое расстояние в вершину 3):

Номер вершины	1	2	3	4
1	0	2	4	10
2	10 000	0	2	11
3	7	10 000	0	3
4	10 000	10 000	10 000	0

Вторая матрица (во 2-ю матрицу записана в качестве промежуточной вершины вершина 2 (путь из вершины 1 в вершину 3)):

Номер 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

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 следующим образом.

Первая матрица (в 1-ю матрицу занесено значение 7 – затраты на преодоление пути из вершины 1 в вершину 4):

Номер вершины	1	2	3	4
1	0	2	4	7
2	10 000	0	2	11
3	7	10 000	0	3
4	10 000	10 000	10 000	0

Вторая матрица (во 2-ю матрицу в поле [1, 4] занесено значение 2, поскольку для перехода из вершины 1 в вершину 4 в качестве промежуточной использована вершина 2):

Номер 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

Далее алгоритм ищет, в какие вершины можно попасть из вершины 1, используя три промежуточные вершины. Далее алгоритм прекратит поиски, поскольку, используя 4 промежуточные вершины, алгоритм придет в исходную вершину, т.е. в вершину 1, что не имеет смысла. Используя три промежуточные 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

$$2 + 7 = 9 \text{ условных единиц.}$$

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

$$2 + 3 = 5 \text{ условных единиц.}$$

Алгоритм сравнивает полученные значения со значениями, которые на данный момент находятся в 1-й матрице:

Чтобы пройти из вершины 2 в вершину 1, требуется бесконечно много условных единиц, т.е. 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 значение 9 вместо бесконечно большого значения. 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] 1-й матрицы. Затраты в 5 условных единиц меньше, чем в 11 условных единиц, поэтому в поле [2, 4] 1-й матрицы вместо значения 11 записывается значение 5. Во 2-й матрице в 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 следующим образом.

Первая матрица (в 1-ю матрицу занесены значения 9 и 5 – затраты на преодоление путей от вершины 2 до вершин 1 и 4):

Номер вершины	1	2	3	4
1	0	2	4	7
2	9	0	2	5
3	7	10 000	0	3
4	10 000	10 000	10 000	0

Вторая матрица (во 2-ю 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):

Номер 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

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 промежуточные вершины.

Далее алгоритм ищет вершины, в которые можно попасть из вершины 2 через три промежуточные вершины. Таких вершин нет, поэтому алгоритм прекращает поиск, так как через четыре промежуточные 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

$$7 + 2 = 9 \text{ условных единиц.}$$

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

$$7 + 10 = 17 \text{ условных единиц.}$$

Алгоритм сравнивает полученные значения со значениями из 1-й матрицы:

Для того чтобы пройти из вершины 3 в вершину 2, потребуется бесконечно много условных единиц, т.е. 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 1-й матрице в поле [3, 2] вместо бесконечно большого значения записывается 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] 1-й матрицы. Затраты в 17 условных единиц больше, чем в 11 условных единиц, т.е. рассмотренный путь не является более выгодным. Во 2-й 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 следующим образом.

Первая матрица (в 1-ю матрицу занесено значение 9 – затраты на преодоление пути от вершины 3 до вершины 2):

Номер вершины	1	2	3	4
1	0	2	4	7
2	9	0	2	5
3	7	9	0	3
4	10 000	10 000	10 000	0

Вторая матрица (во 2-ю матрицу в поле [3, 2] занесено значение 1, поскольку для перехода из вершины 3 в вершину 2 в качестве промежуточной использована вершина 1):

Номер 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

Далее алгоритм ищет, в какие вершины можно попасть из вершины 3 через две промежуточные вершины. Такими вершинами являются вершины 3 и 4. В вершину 3ходить 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 промежуточные вершины, потребуется

$$7 + 2 + 11 = 20 \text{ условных единиц.}$$

Алгоритм сравнивает полученное значение со значением, которое на данный момент находится в 1-й 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 1-ю, ни во 2-ю матрицу ничего не записывается.

Затем алгоритм ищет вершины, в которые можно попасть из вершины 3 через три промежуточные вершины. Таких вершин нет, поэтому алгоритм прекращает поиск, так как через четыре промежуточные 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 промежуточные вершины. Таких вершин также не существует. Алгоритм переходит

к поиску вершин, в которые можно попасть из вершины 4 через три промежуточные вершины. Таких вершин не существует, поэтому алгоритм прекращает работу, поскольку через четыре промежуточные вершины переходить куда-либо бессмысленно – будет осуществлен возврат в исходную вершину.

Больше вершин нет, поэтому окончательный вид матриц выглядит следующим образом.

Первая матрица (вид 1-й матрицы, матрицы кратчайших расстояний, после работы алгоритма):

Номер вершины	1	2	3	4
1	0	2	4	7
2	9	0	2	5
3	7	9	0	3
4	10 000	10 000	10 000	0

Вторая матрица (вид 2-й матрицы, матрицы промежуточных вершин, после работы алгоритма):

Номер 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

Покажем, как записывается кратчайший путь с помощью 2-й 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 идем 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.

Программная реализация алгоритма Флойда

Приватными членами класса Graph теперь помимо вектора вершин и матрицы смежности являются еще 1-я и 2-я матрицы – shortestPathsMatrix и secondMatrix, обе размером таким же, как и матрица смежности. В конструктор, а также в функцию добавления внесены незначительные изменения. В конструкторе помимо матрицы смежности заполняются еще матрица кратчайших расстояний shortestPathsMatrix и 2-я матрица secondMatrix. Если индекс i равен индексу j , то и в матрицу смежности, и в матрицу минимальных затрат будет записан 0. В противном случае будет записано 10 000. Во 2-ю матрицу изначально записываются только нули.

Код конструктора:

```
template<class T>
Graph<T>::Graph() {
    for (int i = 0; i < maxSize; ++i) {
        for (int j = 0; j < maxSize; ++j) {
            if (i == j) {
                this->adjMatrix[i][j] = 0;
                this->shortestPathsMatrix[i][j] = 0;
            }
            else {
                this->adjMatrix[i][j] = 10000;
                this->shortestPathsMatrix[i][j] = 10000;
            }
        }
    /* Заполнение матрицы кратчайших расстояний
       условными бесконечностями */
    this->secondMatrix[i][j] = 0;
```

```

        /* Заполнение второй матрицы нулями */
    }
}
}

```

Изменениям также подверглась функция InsertEdge, в ней теперь для выявления уже существующего ребра проверяется, не равен ли элемент матрицы смежности на пересечении 1-й и 2-й вершин нулю или 10 000. Если равен, значит, ребро еще не рассмотрено. Помимо вставки веса в матрицу смежности также производится вставка вершины назначения во 2-ю матрицу.

Код функции InsertEdge:

```

template<class T>
void Graph<T>::InsertEdge(const T & vertex1, const T &
vertex2, int weight) {
    if (this->GetVertPos(vertex1) != (-1) && this->
->GetVertPos(vertex2) != (-1)) {
        int vertPos1 = GetVertPos(vertex1);
        /* Вычисляется позиция исходной вершины
           в векторе вершин */
        int vertPos2 = GetVertPos(vertex2);
        /* Вычисляется позиция конечной вершины в
           векторе вершин */
        /* Выполняется проверка, существует ли уже ребро
           между исходной и конечной вершинами, если да,
           то выводится сообщение на экран */
        if ((this->adjMatrix[vertPos1][vertPos2] != 10000)
&&
        (this->adjMatrix[vertPos1][vertPos2] != 0)) {
            std::cout << "Ребро между этими вершинами уже
существует" << std::endl;
            return;
        }
        /* Если ребра между исходной и конечной
           вершинами еще нет, то в матрицу смежности в
           соответствующую ячейку заносится значение веса
           ребра от исходной вершины до конечной;
           во вторую матрицу заносится конечная вершина,
           так как на данный момент для того, чтобы попасть
           в конечную вершину из исходной, надо пройти в

```

```

        конечную вершину */
    else {
        this->adjMatrix[vertPos1][vertPos2] = weight;
        this->secondMatrix[vertPos1][vertPos2] = vertex2;
    }
}
else {
    std::cout << "Обеих вершин (или одной из них) в
графе нет "
    << std::endl;
    /* Если исходной и конечной вершин в графе
       нет, то выведется сообщение на экран */
    return;
}
}

```

Функция печати матрицы смежности графа Print также подверглась изменениям. Во 2-м цикле проверяем, равен ли элемент 10 000; если да, то ставится пробел.

Код функции Print:

```

template<class T>
void Graph<T>::Print() {
    if (!this->IsEmpty()) {
std::cout << "Матрица смежности графа:: " << std::endl;
for (int i = 0, vertListSize = this->vertList.size();
     i < vertListSize; ++i) {
    std::cout << this->vertList[i] << " ";
    for (int j = 0; j < vertListSize; ++j)
{
    if (this->adjMatrix[i][j] == 10000)
        std::cout << " " << this->adjMatrix[i][j] << "";
        /* Если встречается значение 10000,
           то ставится один пробел */
    else
        std::cout << " " << this->adjMatrix[i][j] << " ";
        /* Если не 10000, то ставится два пробела */
    }
    std::cout << std::endl;
}
}

```

```

// Если в графе нет вершин, выводится сообщение на экран
else {
    std::cout << "Граф пуст " << std::endl;
}

```

Для печати матрицы кратчайших путей служит функция PrintSPMatrix, она повторяет алгоритм функции Print, только еще печатает 2-ю матрицу.

Код функции PrintSPMatrix:

```

template<class T>
void Graph<T>::PrintSPMatrix() {
    if (!this->IsEmpty()) {
        std::cout << "Матрица кратчайших путей графа: "
        << std::endl;
    for (int i = 0, vertListSize = this->vertList.size();
        i < vertListSize; ++i) {
            // Цикл выводит всю матрицу
            std::cout << this->vertList[i] << " ";
        // В цикле выводятся значения для определенной вершины
            for (int j = 0; j < vertListSize; ++j)
{
            /* Если в матрице кратчайших расстояний
               встретилось значение 10000,
               то ставится один пробел */
            if (this->shortestPathsMatrix[i][j] == 10000)
                std::cout << " " << this->shortestPathsMatrix[i][j] << "";
            else
                std::cout << " " << this->shortestPathsMatrix[i][j] << "
";
                    // Иначе два пробела
            }
            std::cout << std::endl;
        }
        std::cout << "Матрица следующих шагов: " <<
std::endl;
            // Цикл выводит всю матрицу
            for (int i = 0, size = this->vertList.size();
i < size; ++i)
{

```

```

        std::cout << this->vertList[i] << " ";
        // Вывод значений для определенной вершины
        for (int j = 0; j < size; ++j) {
            std::cout << " " << this->secondMatrix[i][j]
            << " ";
        }
        std::cout << std::endl;
    }
}
}

```

Для реализации алгоритма Флойда служит функция Floyd(). Прежде чем приступать к ее разработке, необходимо изучить следующие типы контейнеров библиотеки STL: std::pair, std::map, а также изучить итераторы. В функции описана структура Route (маршрут) с полями вектора вершин, входящих в этот маршрут, и весом маршрута. Кроме того, в структуре определена функция isVertexExists типа bool, которая принимает очередную вершину. Она проверяет, есть ли переданная вершина в маршруте. Делаet это она с помощью функции find, которой передается значение итератора начала вектора вершин, значение итератора его конца и вершина, которую надо искать. Если по выполнении текущее значение итератора не равно значению итератора конца, значит, вершина в маршруте есть (функция find вернет значение итератора на нужную вершину, как только найдет ее). Если же текущее значение равно значению итератора конца вектора вершин, значит, вершины нет. Далее для каждой вершины работает цикл с параметром, в нем создается вектор из маршрутов routes и очередь из маршрутов, которые необходимо просмотреть routesToWatch. Далее создается новый маршрут route, в него добавляется рассматриваемая вершина. В очередь из маршрутов, которые надо просмотреть, добавляется только что созданный маршрут.

Далее циклический процесс работает, пока очередь не закончится:

- Извлекается маршрут из очереди и присваивается маршруту currentRoute.
- В вектор из маршрутов добавляется извлеченный из очереди маршрут. Пока что это только возможный маршрут.

- Берется последняя вершина извлеченного маршрута (вершина, до которой он идет).
- Создается копия текущего рассматриваемого маршрута, в маршрут-копию добавляются соседи конченой вершины текущего рассматриваемого маршрута, если этих соседей нет в рассматриваемом маршруте.
- У нового маршрута увеличивается вес (к весу изначального маршрута-копии прибавляется еще вес ребра от конечной вершины до соседа).
- Новый маршрут добавляется в очередь из маршрутов, которые необходимо посмотреть. После цикла самый первый маршрут удаляется (с помощью функции `erase`), поскольку он содержит одну вершину – самого себя.

Рассмотрим на примере. Пусть рассматривается вершина 1 из графа на рис. 23.41 и ищутся возможные маршруты из вершины 1.

Начинаем с первого маршрута, т.е. с маршрута (1). Он добавляется в `routesToWatch`. Цикл работает до тех пор, пока есть маршруты на просмотр.

Происходит извлечение маршрута (1) и запись его в `routes` как возможного маршрута из вершины 1.

Затем берется вершина, до которой этот маршрут идет (это вершина 1), и берутся соседи (это вершины 2 и 4). Создаются новые маршруты (1, 2) и (1, 4), и они добавляются в очередь `routesToWatch`. Теперь очередь состоит из маршрутов (1, 2) и (1, 4).

Очередь на просмотр не пуста, цикл продолжается. Извлекается маршрут (1, 2). Он записывается в `routes` как возможный маршрут из вершины (1).

Теперь `routes = { (1), (1, 2) }`. Берется вершина, до которой этот маршрут идет (это вершина 2). Извлекаются соседи, это 3 и 4. Создаются новые маршруты – (1, 2, 3) и (1, 2, 4) и добавляются в очередь на просмотр. На данный момент очередь выглядит следующим образом: (1, 4), (1, 2, 3), (1, 2, 4).

Очередь на просмотр не пуста, цикл продолжается. Извлекается маршрут (1, 4). Он заносится в `routes` как возможный маршрут из вершины 1.

Теперь $\text{routes} = \{(1), (1, 2), (1, 4)\}$. Берется вершина, до которой этот маршрут идет (это вершина 4). Извлекаются соседи – их нет, поэтому новые маршруты не добавляются. Очередь на просмотр на данный момент содержит следующее: $(1, 2, 3), (1, 2, 4)$.

Очередь на просмотр не пуста, цикл продолжается. Извлекается маршрут $(1, 2, 3)$. Он записывается в routes как возможный маршрут из вершины 1.

Теперь $\text{routes} = \{(1), (1, 2), (1, 4), (1, 2, 3)\}$. Берется вершина, до которой этот маршрут идет, – 3. Берутся его соседи, это 1 и 4. Вершина 1 не рассматривается, потому что она участвует в маршруте. Создается новый маршрут до вершины 4 – $(1, 2, 3, 4)$ и записывается в очередь маршрутов на просмотр. Теперь она выглядит так: $(1, 2, 4), (1, 2, 3, 4)$.

Очередь не пуста, цикл продолжается. Извлекается маршрут $(1, 2, 4)$ и записывается в routes . Теперь $\text{routes} = \{(1), (1, 2), (1, 4), (1, 2, 3), (1, 2, 4)\}$. Берется вершина, до которой идет маршрут, это 4, соседей нет. Очередь на просмотр сейчас выглядит так: $(1, 2, 3, 4)$.

Очередь не пуста, работа цикла продолжается. Извлекается маршрут $(1, 2, 3, 4)$ и записывается в routes . Теперь $\text{routes} = \{(1), (1, 2), (1, 4), (1, 2, 3), (1, 2, 4), (1, 2, 3, 4)\}$. Находится точка, до которой идет маршрут, это 4, соседей нет. Очередь на просмотр пуста, цикл завершается. Из routes удаляется 1-й маршрут. Теперь $\text{routes} = \{(1, 2), (1, 4), (1, 2, 3), (1, 2, 4), (1, 2, 3, 4)\}$.

Перейдем далее к алгоритму. После работы цикла while создается массив ассоциаций: $\text{std::map}<\text{T}, \text{std::pair}<\text{T, int}>>$ shortestRoutes . В нем хранится пара «ключ – значение». Ключом является конечная вершина, а значением пара, содержащая вершину, в которую надо сделать шаг от начальной вершины, чтобы попасть в вершину-ключ, и вес этого маршрута.

Далее работает цикл с параметром, который берет из вектора routes маршруты, у каждого берет конечную и 2-ю вершины, затем проверяет, есть ли в массиве shortestRoutes информация по конечной вершине.

Если информации там нет, то создается пара: конечная вершина – (2-я вершина маршрута–вес маршрута) – пара, у которой 2-м элементом является еще одна пара. Если в shortestRoutes уже что-то

есть, то проверяется, наименьшего ли веса там хранится маршрут до конечной вершины. Если нет, то значение у ключа изменяется, а именно изменяются 2-я вершина маршрута и вес маршрута.

Например, рассмотрим работу этой части кода для маршрутов из вершины 1. Как было показано ранее, маршрутов из нее всего пять: 1–2, 1–4, 1–2–3, 1–2–4, 1–2–3–4. Сначала извлекается маршрут 1–2, у него берутся конечная и 2-я вершины (в данном случае они совпадают).

Затем определяется, есть ли в массиве самых коротких маршрутов маршрут до вершины 2: нет, такого маршрута нет. Значит, в массив добавляется вершина 2, промежуточная вершина 2 и вес маршрута (он равен 2).

Далее извлекается маршрут 1–4. Записываются его конечная и промежуточные вершины в соответствующие переменные (они совпадают). Выполняется то же самое, что и раньше. Маршрута до вершины 4 нет, значит, данные добавляются в shortestRoutes.

Затем то же самое выполняется для маршрута 1–2–3 (в нем промежуточной уже будет вершина 2, а конечной – вершина 3). Данные для вершины 3 добавлены в shortestRoutes.

Затем извлекается маршрут 1–2–4. Записываются его конечная и промежуточные вершины в переменные, затем проверяется, есть ли в массиве ассоциаций shortestRoutes информация по вершине 4. Информация есть, до вершины 4 есть маршрут весом 10, в котором надо сначала пройти вершину 4. Значит, сравнивается вес текущего маршрута 1–2–4 (он равен 11) и вес уже существующего (он равен 10). Маршрут 1–2–4 не является более выгодным, поэтому изменений не происходит.

Затем извлекается маршрут 1–2–3–4 весом 7. У него извлекаются последняя и 2-я вершины. Проверка на существование информации по вершине 4 дает истину, значит, сравнивается вес текущего маршрута с весом уже существующего. Вес текущего маршрута меньше, значит, он более выгоден, значит, в shortestRoutes для вершины 4 записывается пара (2, 7), т.е., чтобы попасть в вершину 4 быстрее всего, нужно сначала пойти в вершину 2, и вес маршрута составит 7. Все маршруты рассмотрены. В shortestRoutes теперь данные для всех вершин, в которые надо добраться из вершины 1.

После работы этого цикла заполняется 2-я матрица, у каждого из маршрутов извлекаются последняя и промежуточная вершины, а также вес маршрута до последней вершины. После этого данные заносятся в 1-ю и 2-ю матрицы, и так далее для каждой из вершин будут произведены поиски всех маршрутов, информация будет занесена в матрицы.

Код функции Floyd:

```
template<class T>
void Graph<T>::Floyd() {
    struct Route {
        std::vector<T> Verts; // Вершины маршрута
        int weight{ 0 }; // Вес маршрута
        /* Эта функция проверяет, есть ли в
           маршруте эта вершина, то есть проходит ли
           он через нее */
        bool isVertexExists(const T vertex) const {
            return std::find(Verts.cbegin(),
                Verts.cend(), vertex)
                != Verts.cend();
        }
    };
    for (int i = 0, size = this->vertList.size(); i <
size; ++i) {
        // Вектор всевозможных маршрутов из текущей вершины
        std::vector<Route> routes;
        {
            std::queue<Route> routesToWatch;
            /* Очередь вершин, которые нужно просмотреть */
            {
                Route route; // Создание нового маршрута
                // Добавление в маршрут текущей вершины
                route.Verbs.push_back(this->vertList[i]);
                routesToWatch.push(route);
                // Добавление маршрута в очередь
            }
            // Цикл работает, пока очередь не пуста
            while (!routesToWatch.empty()) {
                /* Запоминание маршрута,
                   находящегося в голове очереди */
```

```

        Route currentRoute = routesToWatch.front();
                // Удаление маршрута из головы очереди
                routesToWatch.pop();
        // Добавление нового маршрута в вектор маршрутов
                routes.push_back(currentRoute);
                /* Создание и инициализация
                   последней вершины маршрута */
        const T lastRouteVertex =
                = currentRoute.Verts.back();
        /* Цикл работает для всех соседей последней
           вершины текущего рассматриваемого маршрута */

for (const T& neighbor : this->
        ->GetNbrs(lastRouteVertex)) {
        /* Если в текущем рассматриваемом
           маршруте нет вершины, соседней с последней
           вершиной маршрута, то создается новый
           маршрут (копия) */
if (!currentRoute.isVertexExists(neighbor)){
        Route routeToWatch = currentRoute;
                /* Затем в маршрут-копию помещается
                   сосед последней вершины currentRoute */
        routeToWatch.Verts.push_back(neighbor);
                /* У нового маршрута увеличивается вес
                   (так как в него только что добавили еще одну вершину) */
                routeToWatch.weight += this->
                        ->adjMatrix[this->
                                ->GetVertPos(lastRouteVertex)][this->
                                ->GetVertPos(neighbor)];
        // В очередь заносится этот маршрут-копия
                routesToWatch.push(routeToWatch);
                }
        }
    }

/* Удаление первого маршрута из вектора
   маршрутов, так как первый маршрут идет сам в себя
   (из вершины 1 в вершину 1 через промежуточные вершины) */
    routes.erase(routes.begin());
}

// Массив ассоциаций:
/* Первый T - endVertex означает,
   что, чтобы попасть в endVertex std::pair<T, int>,

```

надо сделать шаг Т (у нас вершины графа могут иметь в качестве идентификаторов разные типы данных, во всех примерах используется число целого типа.

Здесь имеется в виду, что нужно сделать шаг в вершину типа Т, по сути, следующую вершину), и вес этого маршрута будет int */

```

std::map<T, std::pair<T, int>>
shortestRoutes;
{
    // В цикле будут выявлены кратчайшие маршруты

    for (const Route& route : routes) {
        /* Создание и инициализация последней
           вершины маршрута */
        const T endVertex = route.Verts.back();
        /* Создание и инициализация промежуточной
           вершины, в которую надо делать шаг */
        const T stepVertex = route.Verts[1];
        /* Нужно посмотреть shortestRoutes:
           если там до endVertex находится более короткий путь,
               то его не трогать; если его там нет для
               endVertex, то добавить пару; если он есть,
               но длиннее, то его изменить */
        if (shortestRoutes.find(endVertex) ==
            == shortestRoutes.end()) {
            shortestRoutes.insert(std::make_pair(endVertex,
                std::make_pair(stepVertex, route.weight)));
        }
        else {
            const int minimWeight =
                shortestRoutes[endVertex].second;
            if (minimWeight > route.weight)
                shortestRoutes[endVertex] =
                = std::make_pair(stepVertex, route.weight);
        }
    }
    // Цикл заполнения матриц данными
    for (const std::pair<const T, std::pair<T, int>>
        &shortestRoute : shortestRoutes) {
        /* Извлечение конечной вершины текущего
           рассматриваемого кратчайшего пути */

```

```

const T endVertex = shortestRoute.first;
    // Извлечение промежуточной вершины
const T stepVertex = shortestRoute.second.first;
    // Извлечение веса кратчайшего маршрута
const int minWeight = shortestRoute.second.second;
this->shortestPathsMatrix[i][this->
    ->GetVertPos(endVertex)] = minWeight;
    // Заполнение первой матрицы

this->secondMatrix[i][this->GetVertPos(endVertex)]
    == stepVertex;           // Заполнение второй матрицы
}
}

```

Для выводения на экран исходного графа с найденным кратчайшим путем с минимальными затратами служит функция PrintSP(). Рассмотрим работу алгоритма функции.

Сначала объявляются и инициализируются переменные cur и col. В переменной cur будет храниться вершина, в которую осуществляется переход (в которую идем) на данный момент, а в переменной col будет храниться конечная вершина, в которую нужно попасть.

Затем работает цикл для каждой из вершин графа. В нем выводится сообщение, от какой вершины сейчас будут печататься кратчайшие пути.

Затем внутри работает еще один цикл: он ищет во 2-й матрице конечную вершину, в которую надо идти; она равна 2-му индексу элемента 2-й матрицы, не равному 0. Затем переменной cur присваивается это значение (конечной вершины). После этого выводится сообщение, до какой вершины сейчас будут выводиться кратчайшие пути (на экран выводится вершина из вектора вершин с индексом col).

После этого работает еще один цикл, в котором осуществляется перемещение по 2-й матрице. Условие цикла: cur не равен 0, т.е. должна существовать следующая вершина, чтобы была возможность продвигаться. В цикле на экран выводится вершина cur, затем проверяется, есть ли следующая вершина: если да, то в cur записывается ее значение; если нет, то в cur записывается значение 0, цикл

завершается. За ним завершается и его внешний цикл (первый вложенный). После этого печатаются пути уже для следующей вершины (вернее, от нее) до всех остальных. И выполняются те же действия.

Код функции PrintSP:

```
template<class T>
void Graph<T>::PrintSP() {
    int cur = 0, col = 0;
    for (int i = 0, size = this->vertList.size(); i < size; ++i) {
        std::cout << "Кратчайший путь от вершины " << this->vertList[i]; // Вывод исходной вершины на экран
        for (int j = 0; j < size; ++j) {
            // Проверка, что есть следующая(промежуточная) вершина
            if (this->secondMatrix[i][j] != 0) {
                col = j;
            }
            // Запоминаем конечную вершину (в которую идем)
            // Присвоение в cur промежуточной вершины
            cur = this->secondMatrix[i][j];
            // Вывод промежуточной вершины на экран
            std::cout << " к вершине " << this->vertList[j]
            << ": ";
            /* Вывод на экран исходной вершины
             * (так как вывод выглядит, например,
             * так: 1-2-3-4; сейчас мы вывели 1) */
            std::cout << this->vertList[i] << " ";
            /* Цикл, который идет по второй матрице;
             * в нем изменяется промежуточная вершина */
            while (cur != 0) {
                // Вывод текущей промежуточной вершины на экран
                std::cout << cur << " ";
                // Проверка, что есть следующая
                // промежуточная вершина
                if (this->secondMatrix[this->GetVertPos(cur)][col] != 0) {
                    // Если есть, то она присваивается в cur
                    cur = this->secondMatrix[this->GetVertPos(cur)][col];
                }
                // Если нет, то cur обнуляется, цикл завершится
                else cur = 0;
            }
        }
    }
}
```

```

        std::cout << std::endl;
    }
}
std::cout << std::endl;
}
}

```

Код файла main.cpp:

```

#include "Weighted directed Graph.h"
#include <iostream>
#include <conio.h>
using namespace std;
int main() {
{
    Graph<int> graph;
    int amountVerts, amountEdges, vertex,
sourceVertex,
    targetVertex, edgeWeight;
    cout << "Введите количество вершин графа: ";
    cin >> amountVerts; cout << endl;
    cout << "Введите количество ребер: ";
    cin >> amountEdges; cout << endl;
    for (int i = 0; i < amountVerts; ++i) {
        cout << "Вершина: "; cin >> vertex;
        int* VertPtr = &vertex;
        graph.InsertVertex(*VertPtr);
        cout << endl;
    }
    for (int i = 0; i < amountEdges; ++i) {
        cout << "Исходная вершина: ";
        cin >> sourceVertex; cout << endl;
        int* sourceVertPtr = &sourceVertex;
        cout << "Конечная вершина: ";
        cin >> targetVertex; cout << endl;
        int* targetVertPtr = &targetVertex;
        cout << "Вес ребра: ";
        cin >> edgeWeight; cout << endl;
        graph.InsertEdge(*sourceVertPtr,
                         *targetVertPtr, edgeWeight);
    }
    cout << endl;
}

```

```

        graph.Print(); // Печать матрицы смежности
        graph.Floyd();
    // Поиск кратчайших путей с помощью функции Floyd
    graph.PrintSPMatrix(); // Печать матрицы
    кратчайших путей
        graph.PrintSP(); // Печать кратчайших путей
    }
    getch();
    return 0;
}

```

Результат выполнения программы:

Матрица смежности графа: - // Граф представлен на
рис. 23.41

```

1 0 2 1000 10
2 1000 0 2 11
3 7 1000 0 3
4 1000 1000 1000 0

```

Матрица кратчайших путей графа:

```

1 0 2 4 7
2 9 0 2 5
3 7 9 0 3
4 1000 1000 1000 0

```

Матрица следующих шагов:

```

1 0 2 2 2
2 3 0 3 3
3 1 1 0 4
4 0 0 0 0

```

Кратчайший путь от вершины 1 к вершине 2: 1 2

к вершине 3: 1 2 3
к вершине 4: 1 2 3 4

Кратчайший путь от вершины 2 к вершине 1: 2 3 1

к вершине 3: 2 3
к вершине 4: 2 3 4

Кратчайший путь от вершины 3 к вершине 1: 3 1

к вершине 2: 3 1 2
к вершине 4: 3 4

Кратчайший путь от вершины 4

Поскольку вершина 4 ни с какой другой не связана, программа ничего не вывела.

ГЛАВА 24. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ. РЕШЕНИЕ ЗАДАЧИ КОММИВОЯЖЕРА

24.1. Практическое применение задачи коммивояжера

Применение задачи коммивояжера на практике довольно обширно: ее можно использовать для поиска кратчайшего пути при гастролях эстрадной группы или обеспечения наименьшего времени выполнения производственного цикла [3].

24.2. Постановка задачи коммивояжера

Существует N городов, соединенных путями по принципу «каждый с каждым». Все пути имеют вес, расстояние между городами. Задача коммивояжера состоит в том, чтобы обехать все города, побывав в каждом лишь один раз, при этом требуется, чтобы сумма расстояний между городами была минимальной (рис. 24.1).



Рис. 24.1. Это оптимальный маршрут коммивояжера через 15 крупнейших городов Германии. Указанный маршрут является самым коротким из всех возможных 43 589 145 600 вариантов

24.3. Анализ задачи коммивояжера

Для решения задачи коммивояжера составим матрицу условий, она будет содержать расстояния между городами. Считается, что можно перейти из любого города в любой другой, кроме того же самого. Поскольку пути из города А в город А по условию нет, обозначим его «нн»[3].

Матрица условий:

нн	4	5	7	5
8	нн	5	6	6
3	5	нн	9	6
3	5	6	нн	2
6	2	3	8	нн

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».

Редукция строк (минимальные элементы подчеркнуты):

Получившаяся таблица					Редукция строк				
нн	<u>4</u>	5	7	5	нн	0	1	3	1
8	нн	<u>5</u>	6	6	3	нн	0	1	1
3	5	нн	<u>9</u>	6	0	2	нн	6	3
3	5	6	нн	<u>2</u>	1	3	4	нн	0
6	2	3	8	нн	4	0	1	6	нн

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

Редукция столбцов (минимальные элементы подчеркнуты):

Редукция строк					Редукция столбцов				
нн	<u>0</u>	1	3	1	нн	0	1	2	1
3	нн	<u>0</u>	<u>1</u>	1	3	нн	0	0	1
0	2	нн	<u>6</u>	3	0	2	нн	5	3
1	3	4	нн	<u>0</u>	1	3	4	нн	0
4	0	1	6	нн	4	0	1	5	нн

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

Оценка нулей:

нн	$\theta^{(1)}$	1	2	1
3	нн	$\theta^{(1)}$	$\theta^{(2)}$	1
$\theta^{(3)}$	2	нн	5	3
1	3	4	нн	$\theta^{(2)}$
4	$\theta^{(1)}$	1	5	нн

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»).

Чистка карты:

Новая матрица
нн θ 1 2 1
нн нн θ 0 1
нн нн нн нн нн
нн 3 4 нн 0
нн 0 1 5 нн

нн	$\theta^{(1)}$	1	2	1
3	нн	$\theta^{(1)}$	$\theta^{(2)}$	1
$\theta^{(3)}$	2	нн	5	3
1	3	4	нн	$\theta^{(2)}$
4	$\theta^{(1)}$	1	5	нн

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 2×2).

Сведение исходной матрицы к матрице 2×2 :

Получившаяся таблица	Новая матрица	Новая матрица	Новая матрица
нн 4 5 7 5	нн 0 1 2 1	нн 0 1 2 нн	нн 0 1 нн нн
8 нн 5 6 6	нн нн 0 0 1	нн нн 0 0 нн	нн нн нн нн нн
3 5 нн 9 6	нн нн нн нн нн	нн нн нн нн нн	нн нн нн нн нн
3 5 6 нн 2	нн 3 4 нн 0	нн нн нн нн нн	нн нн нн нн нн
6 2 3 8 нн	нн 0 1 5 нн	нн 0 1 5 нн	нн 0 1 нн нн

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

Результат решения задачи коммивояжера:

3->1->2->4->5->3
весь путь:
18

24.4. Решение задачи коммивояжера

Ввод матрицы осуществляется путем ввода двумерного массива через два вложенных цикла For. Конечным условием циклов будет количество городов, введенное пользователем заранее.

Поскольку расстояния из города А в город А нет, при совпадении итераторов двух циклов расстояние между городами заменяем на -1 для последующего удобства в программе:

```
void CreateMap (int **map) { // Функция ввода матрицы
    for (int i = 0; i < col; i++) { // Цикл ввода строк
        for (int j = 0; j < col; j++) { // Цикл ввода столбцов
            if (i != j) { // Пути из А в А не должно быть!
                cout << "из города " << i + 1 << " в " << j + 1;
                cin >> map[i][j]; // Вводим расстояние
            }
            else map[i][j] = -1; // Меняем путь из А в А на -1
        }
    }
}
```

Поскольку матрица по ходу выполнения программы будет изменяться, копируем ее для подсчета расстояния всего пути. Аналогично вводу матрицы копируем ее через два цикла For:

```
for (int i = 0; i < col; i++)
    for (int j = 0; j < col; j++)
        map1[i][j] = map[i][j]; // Поэлементно
                                // копируем матрицу
```

Функции редукции строк и столбцов – `min_line` и `min_column` соответственно – на вход принимают указатель на двумерный массив. Поскольку данные функции отличаются незначительно, рассмотрим в качестве примера функцию редукции строк.

Создаем цикл прохождения по всем строкам, переменную `min`, хранящую значения минимального элемента, изначально присваивая значение 1000. Проходим по каждому столбцу в строке, за исключением столбца, значение которого равно -1 (расстояние из города А в город А); если расстояние меньше `min`, то `min` присваиваем это расстояние. Следующим циклом вычитаем значение переменной `min` из каждого значения в строке, не равного -1. Пройдя таким образом все строки, функция завершается:

```

void min_line(int **map) { // Редукция строк
    int min; // Переменная для нахождения минимума
    for (int i = 0; i < col; i++) { // Проходим по
        // каждой строке
        min = 1000; // Обновляем значение каждой
        // новую строку
        for (int j = 0; j < col; j++) // Обходим
            // столбцы в строке
            if ((min > map[i][j]) && (map[i][j] >=0))
                /* Элемент должен быть меньше переменной
                 и больше нуля */
                min = map[i][j]; /* Условие совпадает,
                // обновляем значение переменной */
        for (int j = 0; j < col; j++) // Обходим
            // столбцы в строке
        if (map[i][j] != -1) // Путь не из A в A
            map[i][j] -= min; // Проводим редукцию
        // строк
    }
}

```

Функция вывода матрицы out_map на вход принимает указатель на двумерный массив. Вывод осуществляется с помощью двух вложенных циклов For, все элементы матрицы, не равные -1 (они заменяются на « nn » для удобства пользователя), выводятся:

```

void out_map(int **map) { // Функция вывода матрицы
    for (int i = 0; i< col; i++) { // Цикл вывода строк
        for (int j = 0; j < col; j++) { // Цикл вывода
            // столбцов
            if (map[i][j] == -1)
                cout << "nn "; // Путь из A в A заменяем на «nn»
            else cout<< map[i][j] << " "; // Выводим значение
            }
            cout << endl; //После вывода всей строки,
            //переходим на новую
    }
}

```

Функции оценки нулей – min_str и min_stl соответственно – принимают на вход указатель на двумерный массив; номер строки,

в которой ищем минимальный элемент; номер столбца или строки, в которых находится оцениваемый 0. Поскольку данные функции отличаются незначительно, рассмотрим в качестве примера функцию минимального элемента в строке.

Переменной `min` присваиваем значение 1000, проходим по всем элементам строки, исключая полученный на входе в функцию и равный -1 ; если элемент меньше `min`, записываем его значение в переменную `min`. После прохождения всей строки функция возвращает переменную `min`:

```
int min_str(int **map, int i, int l) {      //Оцениваем 0 по
                                             строке
    int min=1000;      // Создаем переменную минимального
                       // элемента
    for (int j = 0; j < col; j++)
        // Проходим каждый столбец в заданной строке
        if (j!=l)          // Координаты столбца не совпали
                           // с заданными
        if ((min > map[i][j])&(map[i][j] >=0))      // Ищем
                                                       // минимум
        min = map[i][j];           // Обновляем значение
                                     // переменной
    return min;           // Возвращаем минимальное значение
}
```

Функция «чистка карты `cleaner_map`» находит 0 с максимальной оценкой и заменяет все элементы в столбце и строке, в которых он находится, на -1 . На вход принимается указатель на двумерный массив. Возвращает функция структуру, содержащую координаты строки и столбца, в которых находится 0 с максимальной оценкой.

Ищем 0 с максимальной оценкой, создаем переменную `max`, присваиваем ей -1 , проходим по всей матрице: если находим 0, даем его полную оценку, складывая результаты работы функций `min_str` и `min_stl`; если данная сумма оказалась больше переменной `max`, записываем в нее данную сумму, координаты строки и столбца записываем в структуру. После прохождения всей матрицы будут иметься координаты строки и столбца, содержащие 0 с максимальной оценкой.

Теперь требуется вычеркнуть эту строку и столбец, заменяя все элементы строки и столбца на -1 , возвращаем структуру:

```
rebro cleaner_map(int **map) {      // Функция чистки карты
    rebro a;                      // Создаем переменную
    int max = -1, k, m;
    /* Создаем переменные координаты строки и столбца */
    for (int i = 0; i < col; i++) {      // Проходим по
        for (int j = 0; j < col; j++) {      // Проходим по
            if (map[i][j] == 0) {          // Если элемент равен 0
                if (max <= min_str(map, i, j) + min_stl(map, j, i))
                    // Оцениваем данный 0
            }
            max = min_str(map, i, j) + min_stl(map, j, i);
            // Обновляем значение переменной max
            k = i;
            m = j;
            // Сохраняем координаты строки и столбца
        }
    }
    for (int i = 0; i < col; i++) {      // Проходим по
        for (int j = 0; j < col; j++) {      // Проходим по
            if (i == k) map[i][j] = -1;      // Чистим строку
            if (j == m) map[i][j] = -1;      // Чистим столбец
        }
    }
    a.a = k;
    a.b = m;
    /* Одному из полей структуры присваиваем координату
       строки, другому - столбца */
    return a; // Возвращаем структуру
}
```

Создаем цикл нахождения пути. Количество повторений цикла будет равно 3, так как исходную матрицу мы сводим к матрице 2×2 . В данном цикле поочередно вызываем функции:

1. Редукция строк.
2. Вывод карты.
3. Редукция столбцов.
4. Вывод карты.
5. Чистка карты.
6. Вывод карты.
7. Увеличиваем счетчик.

Счетчик отвечает за учет количества повторений цикла и за проход по массиву структур, в который будут записываться результаты работы функции чистки карты:

```
int i = 0; // Устанавливаем начальное значение счетчика
while (i != col-2) { // Цикл поиска пути
    min_line(map); // Редукция строк
    cout << "редукция строк\n";
    out_map(map); // Вывод карты
    min_column(map); // Редукция столбцов
    cout << "редукция столбцов\n";
    out_map(map); // Вывод карты
    rez[i] = clear_map(map); // Записываем результаты
    чистки карты
    cout << "новая матрица\n";
    out_map(map); // Вывод карты
    i++; // Увеличиваем счетчик
}
```

Результаты чистки карты – ребра. Сортировка ребер. В результате работы программы были получены ребра следующего вида:

```
1→3
4→5
2→4
```

Но, так как по условию задачи коммивояжер не может пройти через один город дважды, необходимо поменять ребра таким образом, чтобы 1-е поле одной структуры не было равно 2-му полю другой структуры:

```
for (int i=1; i < z; i++)
    /* z содержит количество элементов массива структур */
```

```

// Так как первое ребро неизменно, цикл начинаем с 1
// Сравниваем первое поле i-й структуры со вторым
// полем j-й

for (int j = i; j < z; j++) { // Проходим по вторым полям
    if (rez[i].a == rez[j].b) {
        /* Если поля совпадают, то меняем структуры местами */
        a = rez[i].a;
        b = rez[i].b;
        rez[i] = rez[j];
        rez[j].a = a;
        rez[j].b = b;
    }
}

```

Подсчет расстояний. Первое ребро и расстояние от последнего элемента до 1-го считаем сразу, далее считаем расстояние оставшихся ребер, затем считаем расстояние между ребрами:

```

a = map1[rez[i-1].b][rez[0].a]; /* Расстояние от последней
                                 вершины до первой */
a += map1[rez[0].a][rez[0].b]; // Расстояние первого ребра
for (int i = 1; i < z; i++)           // Проходим по массиву
                                         структур
a += map1[rez[i].a][rez[i].b];       /* Прибавим расстояние
                                         i-го ребра */
for (int i = 0; i < z-1; i++)         // Проходим по массиву
                                         структур

    if (rez[i].b!=rez[i+1].a)      /* Если поля равны, то
                                         расстояние от вершины A до A считаем
                                         равным 0 */
    a += map1[rez[i].b][rez[i+1].a]; /* Считаем расстояние
                                         между ребрами */

```

ГЛАВА 25. STL-БИБЛИОТЕКИ В С++

STL – 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.

25.1. 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, сохраненные в контейнере, *принадлежат ему*. Это означает, что, когда время существования контейнера истекает, то же самое происходит с сохраненными в нем 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`, `forward_list`, `priority_queue` – являются последовательностями. Уточнение заключается в том, что переход от элемента к элементу последовательности должен быть по меньшей мере односторонним, что гарантирует размещение элементов в определенном порядке, который не меняется от одного цикла итераций к другому.

Поскольку элементы в последовательности размещены в определенном порядке, становятся возможными такие операции, как вставка значений в определенную 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.

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()`[9].

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 [9].

list:

Класс шаблона `list` представляет собой двусвязный список. Каждый его элемент, за исключением первого и последнего, связан как с предшествующим элементом, так и с последующим, 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 др. [9].

forward_list:

В C++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 [9].

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 ее элементам. Взамен *queue* ограничивается базовыми операциями, определяющими очередь [9].

stack:

Класс шаблона *stack* предоставляет типичный интерфейс стека. Он так же, 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* [9].

Ассоциативные контейнеры

Ассоциативный контейнер – еще 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, т.е. обеспечивают быстрый поиск данных по ключу. Подобно последовательности, ассоциативный контейнер позволяет вставлять элементы, однако нельзя указать определенное 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, т.е. 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, причем 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.

25.2. Контейнер *vector*

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

Создадим вектор, заполним его элементами, удалим некоторые элементы.

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;
```

Функция *pop_back()* удаляет последний элемент вектора.

```
#include <iostream>
#include <vector> // Подключаем вектор
using namespace std;
void main(){
    vector <int> a; // Создаем пустой вектор
    for(int i=0; i<10; i++)
        a.push_back(i); // Заполняем вектор
    // элементами от 0 до 9
    a.pop_back(); // Удаляем последний элемент вектора
    for(int i=0; i<a.size(); i++)
        /* a.size() - возвращает размер вектора */
        cout<<a[i]<<" "; // Обращаться к элементам
    // можно так: a[i]
}
```

Создание вектора и удаление последнего элемента:

```
Заполненный вектор
0 1 2 3 4 5 6 7 8 9
Удаляем последний элемент
Выводим новый вектор
0 1 2 3 4 5 6 7 8
```

25.3. 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* аналогична тому, как работают указатели. Итератор может быть объектом, для которого определены операции над указателями, такие как разыменование 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 необязательно находятся в ячейках памяти последовательно. В этом случае создается класс «итератор», который за счет своей внутренней реализации позволит последовательно перемещаться по списку и будет иметь интерфейс, аналогичный указателю.

Рассмотрим возможности итератора на примере вектора.

Напишем программу, где обращение к элементам вектора будет осуществляться при помощи итератора.

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
    }
}

```

Выводим вектор через итератор:

```

Заполненный вектор
Обращаемся через итератор
0 1 2 3 4 5 6 7 8 9

```

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++)                   // Заполняем вектор значениями
        a.push_back(i);                           // от 0 до 9
    vector<int>::iterator it = a.begin();          // Создаем
                                                // итератор
    advance(it, 5); // Перемещаем итератор на 5 элементов
    cout « *it « endl; // Выводим новое значение итератора
}

```

Смещение итератора:

```

Заполненный вектор
Обращаемся через итератор
0 1 2 3 4 5 6 7 8 9
Выводим элемент после смещения итератора
5

```

Последующее изучение итератора будет происходить по ходу изучения новых контейнеров.

25.4. Предопределенные итераторы

Библиотека STL предоставляет ряд заранее определенных 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 разыменования 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, " ");
// Использование 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 вместо `begin()` и `end()` использовали методы `rbegin()` и `rend()`, тогда программа вместо `1|2|3|4|5|6` вывела бы `6|5|4|3|2|1`.

Итераторы вставки

Существует три итератора вставки: `back_insert_iterator`, `front_insert_iterator` и `insert_iterator`. Они отличаются от обычных

итераторов тем, что автоматически изменяют размер контейнера. Например, при копировании данных из массива в контейнер нет необходимости заранее знать нужный размер контейнера. Кроме того, итераторы вставки необходимы также в случаях, когда нужно не замещать существующие элементы, а вставлять новые.

Итератор `back_insert_iterator` вставляет элементы в конец контейнера, а `front_insert_iterator` – в его начало. Итератор `insert_iterator` вставляет элементы перед позицией, указанной в аргументе конструктора `insert_iterator`.

Несмотря на удобство итераторов вставки, существуют некоторые ограничения. Так, например, `back_insert_iterator` может применяться только с контейнерными типами, которые допускают быструю вставку в конец (`push_back`). Итератор `front_insert_iterator` может использоваться только с контейнерными типами, допускающими вставку в начало за постоянное время (`push_front`). Итератор `insert_iterator` не обладает этими ограничениями, но другие два итератора вставки выполняют свои задачи (вставку в начало и конец) на порядок быстрее.

Рассмотрим создание итераторов вставки для объекта типа `vector<double>`:

```
vector<double> v; // Создание объекта v типа
vector<double>
// Создание итератора back_iter типа back_insert_iterator
back_insert_iterator<vector<double>> back_iter(v);
// Создание итератора iter типа insert_iterator
insert_iterator<vector<double>> iter(v, v.begin());
// Вставка будет осуществляться в начало контейнера
```

Пример. Использование итераторов вставки

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <iterator>

using namespace std;
```

```

int main() {
    /* Создание объектов points и points_copy
       типа vector<int> и их инициализация
       списком значений */
vector<int> points = { 1,2,3,4,5,6 }, points_copy =
{3,2,1};
    /* Вставка в объект points значений
       из контейнера points_copy с
       помощью функции copy */
copy(points.begin(), points.end(),
back_insert_iterator<vector<int>>(points_copy));
    /* Вывод элементов контейнера points_copy
       в обратном порядке с помощью обратного итератора */
for (vector<int>::reverse_iterator p =
points_copy.rbegin();
p != points_copy.rend(); p++) {
    cout << *p << "|";
}
system("pause");
return 0;
}

```

В результате программа выведет на экран: 6|5|4|3|2|1|2|3|.

25.5. 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.

Напишем программу для создания словаря.

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`:

```
void main() {
    map<string, string> slov; // Создаем контейнер map
    for (int i = 0; i < 2; i++) { // Заполняем его двумя
        значениями
        string a;
        cin >> a; // Вводим ключ (слово на английском)
        cin >> slov[a]; // Вводим значение (слово на русском)
    }
    slov.insert(pair<string, string>("hi", "привет"));
    //Добавим элемент */
    map<string, string>::iterator it = slov.begin();
    // Создаем итератор
    // Указываем на первый элемент контейнера
    for (it; it != slov.end(); it++)
        // Обходим все записи в контейнере */
    cout << it->first << " - " << it->second; // Выводим записи
    // Слово на английском - слово на русском
    // it->first обращаемся к ключу
    // it->second обращаемся к значению
}
```

Реализация словаря:

```
слово - перевод
сар
машина
карт
карта
Вывод содержимого контейнера
сар-машина
hi-привет
карта-карта
```

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 [9].

```
slov.erase(slov.find("hi")); // Удаляем элемент с ключом hi
for (it= slov.begin(); it != slov.end(); it++)
    // Выводим содержимое
cout « it->first « "-" « it->second « endl;
```

Удаление элемента из словаря:

```
слово - перевод
car
машина
map
карта
Вывод содержимого контейнера
саг-машина
hi-привет
мар-карта
Удаляем из словаря слово hi
саг-машина
мар-карта
```

Ассоциативный контейнер set

Для работы с различными множествами используется контейнер *set*. Данные, попадающие в данный контейнер, сортируются по возрастанию и удаляются одинаковые значения, однако контейнер *multiset* не удаляет одинаковые значения, а по возможности идентичен *set*.

Подключение данного контейнера осуществляется добавлением библиотеки *<set>*. Инициализация происходит таким образом:

```
Set <тип данных> название;
```

Напишем программу, сортирующую введенную последовательность. Поскольку контейнер *set* сам сортирует введенные значения, сортировки не требуется:

```
void main() {
    set<int>n; // Инициализируем контейнер
    for (int i = 0; i< 5; i++) {
        // Цикл ввода последовательности
        int d;
        cin » d; // Вводим значение
        n.insert(d); // Записываем значение в контейнер
```

```
// Данная функция уже знакома
}
set<int>::iterator it = n.begin(); // Создаем итератор
cout « "Вывод содержимого контейнера" « endl;
for (it;it != n.end();it++)
    /* Обходим все содержимое контейнера */
    cout « *it « " "; // Выводим содержимое
```

Работа контейнера set:

```
9 5 6 3 1
Вывод содержимого контейнера
1 3 5 6 9
```

ГЛАВА 26. WINDOWS FORMS В С++

Windows Forms – платформа для создания кроссплатформенных графических интерфейсов. Обычно приложение Windows Forms строится перемещением элементов на форму и написанием кода для каждого отдельного элемента.

26.1. Разработка проекта

Создадим Windows-форму: для этого создаем новый проект в VisualStudio. Выбираем «Создать проект CLR» – создаем пустой проект (рис. 26.1).

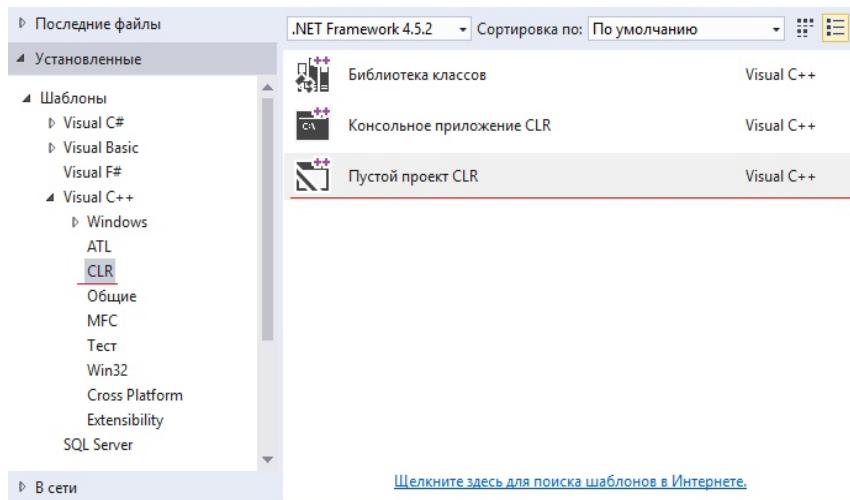


Рис. 26.1. Создание пустого проекта CLR

Добавляем в пустой проект Windows-форму, но, прежде чем начать работу, заходим в свойства проекта и меняем следующие параметры (рис. 26.2).

После установки параметров перезапускаем форму. Добавляем на нее кнопки или метки, установив их свойства (рис. 26.3, 26.4).

Подсистема		Windows (/SUBSYSTEM:WINDOWS)
Минимальная требуемая версия		
Резервируемый размер кучи		
Фиксируемый размер кучи		
Резервируемый размер стека		
Фиксируемый размер стека		
Включить большие адреса		
Сервер терминалов		
Запускать с компакт-диска с помощью ф	Нет	
Запускать из сети с помощью файла подк	Нет	
Драйвер		Не задано

Рис. 26.2. Установка подсистемы

Точка входа		Main
Без точки входа		Нет
Установить контрольную сумму		Нет
Базовый адрес		
Внесение случайности в базовый адрес	Да (/DYNAMICBASE)	
Фиксированный базовый адрес	Нет (/FIXED:NO)	
Предотвращение исполнения данных (DE)	Да (/NXCOMPAT)	
Отключить создание сборки	Нет	
Выгрузить отложено загружаемые DLL		
Не включать отложено загружаемые DLL		
Библиотека импорта		
Объединить разделы		
Конечный компьютер	MachineX86 (/MACHINE:X86)	
Профиль	Нет	
Атрибут потока CLR		
Тип CLR-образа		Тип образа по умолчанию

Рис. 26.3. Установка точки входа

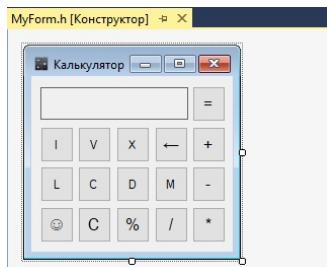


Рис. 26.4. Готовая форма

26.2. Проектирование формы, описание свойств

После создания формы разместим на ней нужные нам элементы, их можно найти на панели элементов (рис. 26.5).

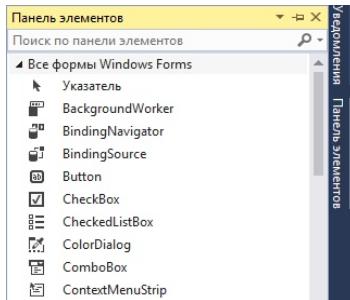


Рис. 26.5. Панель элементов

Элементы размещаем на форме и устанавливаем размер вручную, или можно обратиться к свойствам данного объекта и изменить его некоторые параметры (рис. 25.6).

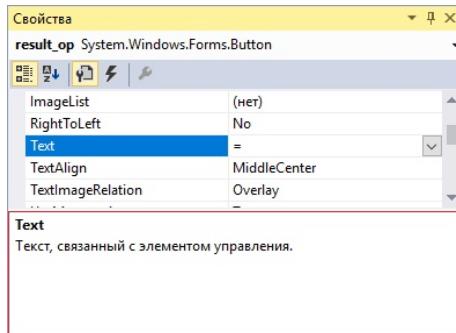


Рис. 26.6. Свойства элемента формы

Нажимая на любое свойство, можем получить краткое его описание. Помимо свойств элементов формы есть свойства самой формы, которые тоже можно менять.

На панели элементов находим форму Button, перетаскиваем ее на форму, увеличиваем количество кнопок на форме до нужного количества (рис. 26.7).

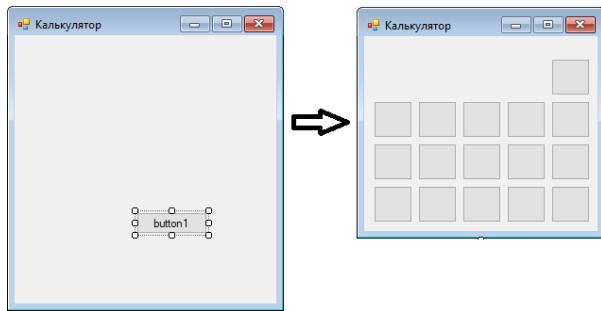


Рис. 26.7. Моделирование формы

Установим свойства каждого элемента (назовем его). Перетаскиваем на форму с панели элементов объект TextBox (будет выводить значение пользователю) (рис. 26.8).

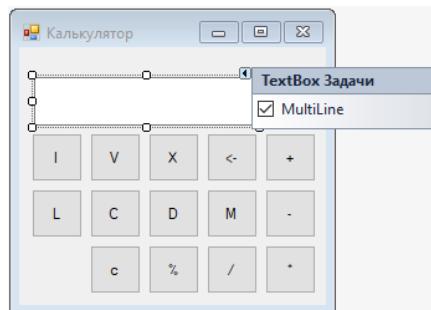


Рис. 26.8. Установка TextBox
и его масштабирование

После создания формы со всеми ее элементами нажимаем на элемент дважды, чтобы прописать действия.

26.3. Разработка калькулятора для чисел римской системы счисления

Для калькулятора римской СС создадим форму из рис. 26.8. В калькуляторе реализовать все математические операции и функцию возврата действия.

Напишем действия при нажатии на число в римской СС (рис. 26.9–26.13). После нажатия на клавишу с числом программа должна вывести его на экран, значит, требуется написать функцию вывода текста на экран:

```
private:  
System::Void IR_Click(System::Object^sender,  
System::EventArgs^ e) {  
    // Код, созданный конструктором форм  
    inputLetter("I");  
    // Вызываем функцию вывода текста на экран  
}
```

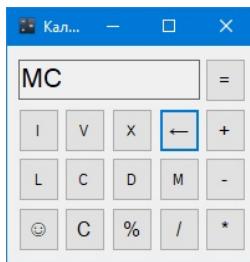


Рис. 26.9. Ввод первого числа

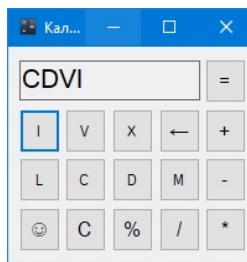


Рис. 26.10. Ввод второго числа

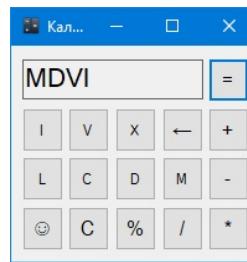


Рис. 26.11. Результат суммы двух чисел

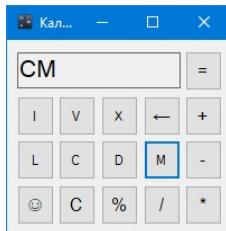


Рис. 26.12. Ввод вычитаемого

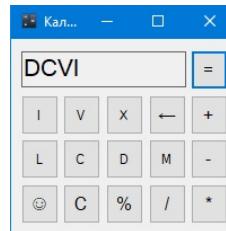


Рис. 26.13. Разность двух чисел

Для вывода текста используется элемент формы TextBox. Функция `inputLetter` принимает значение, которое она должна вывести пользователю, проверяет, возможно ли такое число в римской СС, и выводит его, если это возможно, иначе возникает ошибка.

Чтобы использовать действия, заранее пронумеруем их. Это значит, что после нажатия на действие в функцию будет передаваться номер этого действия:

```
private:  
System::Void  
PLUSOP_Click(System::Object^sender, System::EventArgs^e){  
    // Код, созданный конструктором форм  
    useOperation(1);           // Функция обработки действий  
}
```

После нажатия на действие TextBox очищается для следующего значения, а номер операции запоминается.

Для удобной работы с числами переведем их в десятичную систему по правилам перевода. Аналогичным образом создадим контейнер map для хранения числа в римской и в десятичной СС:

```
const map<char, int> Calculator::alphabet = {  
    // Записываем значения в контейнер */  
    { 'I', 1 },  
    { 'V', 5 },  
    { 'X', 10 },  
    { 'L', 50 },  
    { 'C', 100 },  
    { 'D', 500 },  
    { 'M', 1000 }  
};
```

После создания функций перехода в разные системы счисления, функций сохранения предыдущих значений напишем функцию, реализующую математические операции:

```
switch (last_op) { // Ключ – номер операции  
case 1:  
    a_n += b_n;  
    /* Складываем два значения, результат  
     записываем в первое */  
    // Числа уже переведены в десятичную систему  
    break;  
    // Аналогичным образом реализуем оставшиеся операции  
}
```

Проверяем результат операции на существование (строго больше нуля и меньше 3999 – нет чисел в римской СС, не удовлетворяющих этому условию). Результат переводим в римскую СС и выводим пользователю.

В качестве результата работы сложим римские числа MC (1100) и CDVI (406), затем вычтем из них число CM (900). После суммы MC и CDVI мы должны получить число MDVI ($1100 + 406 = 1506$); из результата предыдущей операции, MDVI, вычитаем CM, ответ будет равен DCVI ($1506 - 900 = 606$).

ГЛАВА 27. ВЕРТИКАЛЬНОЕ ПРЕДСТАВЛЕНИЕ ДЕРЕВА. ИСПОЛЬЗОВАНИЕ ГРАФИЧЕСКОЙ БИБЛИОТЕКИ OPengl

27.1. Графическая библиотека OpenGL

OpenGL (Open Graphics Library) – спецификация, определяющая платформонезависимый (независимый от языка программирования) программный интерфейс для написания приложений, использующих двумерную и трехмерную компьютерные графики. Включает более 300 функций для реализации сложных двумерных и трехмерных сцен из простых примитивов.

Учтем, что OpenGL – это спецификация, т.е. она лишь определяет набор обязательных возможностей. Реализация же зависит от конкретной платформы.

OpenGL является кроссплатформенным, независимым от языка программирования API для работы с графикой. OpenGL – низкоуровневый API, поэтому для работы с ним необходимо иметь некоторое представление о графике в целом и знать основы линейной алгебры.

Наименования функций в OpenGL

Имена всех функций, предоставляемых непосредственно OpenGL, начинаются с приставки `gl`. Функции, задающие некоторый параметр, характеризующийся набором чисел (например, координату или цвет), имеют суффикс вида [число параметров + тип параметров + представление параметров].

Число параметров указывает число принимаемых параметров. Принимает следующие значения: 1, 2, 3, 4.

Тип параметров указывает тип принимаемых параметров. Возможны следующие значения: `b`, `s`, `i`, `f`, `d`, `ub`, `us`, `ui`, т.е. `byte` (char в C, 8-битное целое число), `short` (16-битное целое число), `int` (32-битное целое число), `float` (число с плавающей запятой), `double` (число с плавающей запятой двойной точности), `unsigned byte`, `unsigned short`, `unsigned int` (последние три – беззнаковые целые числа).

Представление параметров указывает, в каком виде передаются параметры: если каждое число по отдельности, то ничего не пишется, если же параметры передаются в виде массива, то к названию функции дописывается буква v.

Пример: glVertex3iv задает координату вершины, состоящую из трех целых чисел, передаваемых в виде указателя на массив.

Представление графических объектов в OpenGL

Все графические объекты в OpenGL представляют собой набор точек, линий и многоугольников (рис. 27.1). Существует 10 различных примитивов, при помощи которых строятся все объекты, как двухмерные, так и трехмерные. Все примитивы, в свою очередь, задаются точками – вершинами.

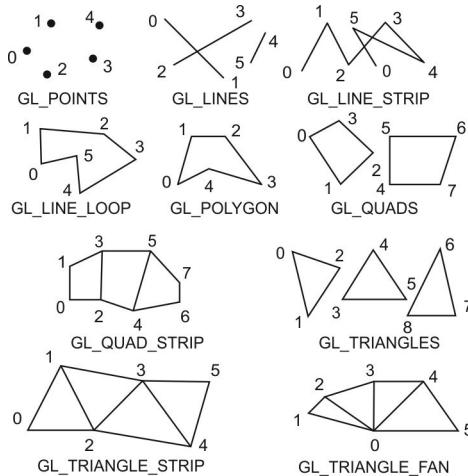


Рис. 27.1. Примеры представления графических объектов

Обозначения на рисунке:

GL_POINTS – каждая вершина задает точку.

GL_LINES – каждая отдельная пара вершин задает линию.

GL_LINE_STRIP – каждая пара вершин задает линию (т.е. конец предыдущей линии является началом следующей).

GL_LINE_LOOP – аналогично предыдущему, за исключением того, что последняя вершина соединяется с первой и получается замкнутая фигура.

GL_TRIANGLES – каждая отдельная тройка вершин задает треугольник.

GL_TRIANGLE_STRIP – каждая следующая вершина задает треугольник вместе с двумя предыдущими (получается лента из треугольников).

GL_TRIANGLE_FAN – каждый треугольник задается первой вершиной и последующими парами (т.е. треугольники строятся вокруг первой вершины, образуя нечто похожее на диафрагму).

GL_QUADS – каждые четыре вершины образуют четырехугольник.

GL_QUAD_STRIP – каждая следующая пара вершин образует четырехугольник вместе с парой предыдущих.

GL_POLYGON – задает многоугольник с количеством углов, равным количеству заданных вершин.

Для задания примитива используется конструкция `glBegin` (тип_примитива)...`glEnd` (). Вершины задаются `glVertex*`. Вершины задаются против часовой стрелки. Координаты задаются от верхнего левого угла окна. Цвет вершины задается командой `glColor*`. Цвет задается в виде RGB или RGBA. Команда `glColor*` действует на все вершины, что идут после, до тех пор, пока не встретится другая команда `glColor*`, или же на все, если других команд `glColor*` нет.

Ниже код, рисующий квадрат с разноцветными вершинами [23]:

```
glBegin(GL_QUADS);
glColor3f(1.0, 1.0, 1.0);
glVertex2i(250, 450);
glColor3f(0.0, 0.0, 1.0);
glVertex2i(250, 150);
glColor3f(0.0, 1.0, 0.0);
glVertex2i(550, 150);
glColor3f(1.0, 0.0, 0.0);
glVertex2i(550, 450);
glEnd();
```

Основы программы на OpenGL

Для независимой от платформы работы с окнами используется библиотека GLUT, что упрощает работу с OpenGL.

Для инициализации GLUT в начале программы надо вызвать glutInit (&argc, argv). Для задания режима дисплея вызывается glutInitDisplayMode (режим), где режим может принимать следующие значения:

GLUT_RGB – включает четырехкомпонентный цвет (используется по умолчанию);

GLUT_RGBA – то же, что и GLUT_RGB;

GLUT_INDEX – включает индексированный цвет;

GLUT_DOUBLE – включает двойной экранный буфер;

GLUT_SINGLE – включает одиночный экранный буфер (по умолчанию);

GLUT_DEPTH – включает Z-буфер (буфер глубины);

GLUT_STENCIL – включает трафаретный буфер;

GLUT_ACCUM – включает буфер накопления;

GLUT_ALPHA – включает альфа-смешивание (прозрачность);

GLUT_MULTISAMPLE – включает мультисемплинг (сглаживание);

GLUT_STEREO – включает стереоизображение.

Для выбора нескольких режимов одновременно нужно использовать побитовое ИЛИ "|".

Например: glutInitDisplayMode (GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH) включает двойную буферизацию, Z-буфер и четырехкомпонентный цвет. Размеры окна задаются glutInitWindowSize (ширина, высота). Его позиция – glutInitWindowPosition (x, y). Создается окно функцией glutCreateWindow (заголовок_окна).

GLUT реализует событийно-управляемый механизм, т.е. есть главный цикл, который запускается после инициализации, и в нем уже обрабатываются все объявленные события. Например, нажатие клавиши на клавиатуре или движение курсора мыши и т.д. Зарегистрировать функции-обработчики событий можно при помощи следующих команд:

void glutDisplayFunc (void (*func) (void)) – задает функцию рисования изображения;

void glutReshapeFunc (void (*func) (int width, int height)) – задает функцию обработки изменения размеров окна;

void glutVisibilityFunc (void (*func)(int state)) – задает функцию обработки изменения состояния видимости окна;

void glutKeyboardFunc (void (*func)(unsigned char key, int x, int y)) – задает функцию обработки нажатия клавиш клавиатуры (только тех, что генерируют ASCII-символы);

void glutSpecialFunc (void (*func)(int key, int x, int y)) – задает функцию обработки нажатия клавиш клавиатуры (тех, что не генерируют ASCII-символы);

void glutIdleFunc (void (*func) (void)) – задает функцию, вызываемую при отсутствии других событий;

void glutMouseFunc (void (*func) (int button, int state, int x, int y)) – задает функцию, обрабатывающую команды мыши;

void glutMotionFunc (void (*func)(int x, int y)) – задает функцию, обрабатывающую движение курсора мыши, когда зажата какая-либо кнопка мыши;

void glutPassiveMotionFunc (void (*func)(int x, int y)) – задает функцию, обрабатывающую движение курсора мыши, когда не зажато ни одной кнопки мыши;

void glutEntryFunc (void (*func)(int state)) – задает функцию, обрабатывающую движение курсора за пределы окна и его возвращение;

void glutTimerFunc (unsigned int msecs, void (*func)(int value), value) – задает функцию, вызываемую по таймеру.

Затем можно запускать главный цикл glutMainLoop() [23].

Проектирование программ в OpenGL

Напишем простую программу для закрепления знаний.

Начнем с того, что нужно подключить заголовочный файл glut.h. Учтем, что может пригодиться подключение заголовочного файла stdafx.h:

```
#include"stdafx.h"
#ifndefdefined(linux) || defined(_WIN32)
#include<GL/glut.h> /*Для Linux и Windows*/
#else
#include<GLUT/GLUT.h> /*Для Mac OS*/
#endif
```

В main зарегистрируем два обработчика: для представления содержимого окна и обработки изменения его размеров. Эти два обработчика, по сути, регистрируются в любой программе, использующей OpenGL и GLUT. Ниже приводится пример main:

```
void reshape(int, int);
void display();
int main(int argc, char * argv[]) {
    glutInit(&argc, argv);
    /* Включаем двойную буферизацию
       и четырехкомпонентный цвет */
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(800, 600);
    glutCreateWindow("OpenGL lesson 1");
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Необходимо написать функцию – обработчик изменений размеров окна. Зададим область вывода изображения размером во все окно при помощи команды glViewport (x , y , ширина, высота). Затем загрузим матрицу проекции glMatrixMode (GL_PROJECTION), заменим ее единичной glLoadIdentity() и установим ортогональную проекцию. И наконец, загрузим модельно-видовую матрицу glMatrixMode (GL_MODELVIEW) и заменим ее единичной.

В итоге получим:

```
void reshape(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, w, 0, h);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

Последующим шагом нужно написать функцию рисования содержимого окна. Рисовать будем тот квадрат, что был приведен до

этого в качестве примера. Добавить придется совсем немного кода. Во-первых, перед рисованием надо очистить различные буфера при помощи `glClear` (режим). Используется так же, как и `glutInitDisplayMode`. Возможные значения:

`GL_COLOR_BUFFER_BIT` – для очистки буфера цвета;
`GL_DEPTH_BUFFER_BIT` – для очистки буфера глубины;
`GL_ACCUM_BUFFER_BIT` – для очистки буфера накопления;
`GL_STENCIL_BUFFER_BIT` – для очистки трафаретного буфера.

В нашем случае нужно очистить только буфер цвета, так как другие не используются. Во-вторых после рисования нужно сменить экранные буфера при помощи `glutSwapBuffers()`, ведь включена двойная буферизация. Все рисуется на скрытом от пользователя буфере, и затем происходит смена буферов. Делается это для получения плавной анимации и для того, чтобы не было эффекта мерцания экрана.

Получаем разноцветный квадрат на рис. 27.2, соответствующий коду листинга:

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_QUADS);
    glColor3f(1.0, 1.0, 1.0);
    glVertex2i(250, 450);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2i(250, 150);
    glColor3f(0.0, 1.0, 0.0);
    glVertex2i(550, 150);
    glColor3f(1.0, 0.0, 0.0);
    glVertex2i(550, 450);
    glEnd();
    glutSwapBuffers();
}
```

OpenGL – удобный инструмент для создания кроссплатформенных приложений, использующий графику. OpenGL легко использовать с тем языком программирования, который более удобный. Привязки к OpenGL есть для множества популярных языков, таких как C, C++, C#, Java, Python, Perl, VB и др. [23].

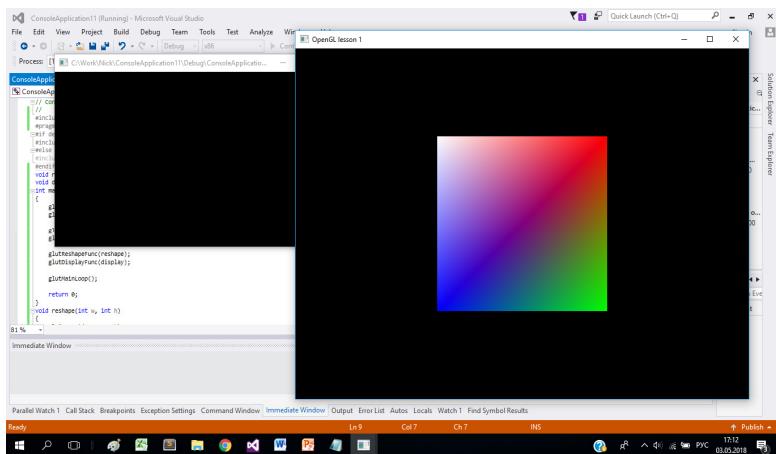


Рис. 27.2. Разноцветный квадрат

27.2. Представление бинарного дерева в OpenGL

Переменные и прототипы функций

Для представления бинарного дерева в OpenGL необходимо знать координаты каждого из элементов. Неслучайно до этого была выведена формула вычисления координат каждого узла для функции вертикальной печати.

То, сколько пробелов нужно поставить перед данным элементом, отсчитывая с начала печатаемой в консоли строки, предназначенней для печати отдельного уровня дерева, зависит от уровня, на котором сейчас находится элемент, от уровня, который может быть в принципе в качестве максимального, и, естественно, от минимальной ширины дерева, требуемой для печати какого-либо полного дерева (если печатается неполное дерево, то оно пополнится значениями NULL). Ширина данных, хранимых в дереве, при печати может занимать более одного символа, вследствие чего введен дополнительный коэффициент, выбираемый пользователем. И это очень удобно.

Можно считать, что это количество пробелов – эквивалент координаты x , а текущий уровень – это y . Следовательно, мы действительно имеем дело с координатами.

Но эти значения координат пока что не готовы к использованию в представлении дерева в OpenGL. Их нужно преобразовать в зависимости от ширины, высоты окна, возможного отступа (от краев окна), а также от собственных характеристик дерева, как, например, от введенной до этого понятия минимальной ширины печатаемого дерева и его высоты (будут использоваться некоторые понятия из функции вертикальной печати). Так, будущее дерево будет отображаться в некоторых рамках, т.е. предусматриваем факт того, что пространство для изображения дерева ограничено. При этом будем стараться разместить дерево по максимуму в данном пространстве, соблюдая отступы, не нарушая границ окна. И распространяться дерево будет как в ширину, так и в высоту, но в основном размеры элементов будут зависеть от ширины.

Будут введены новые коэффициенты, которые помогут расширять дерево динамически. Имеется в виду, что с изменением факторов, от которых зависит рисунок дерева, будут меняться и координаты элементов, и размеры фигур, внутри которых будут размещены какие-либо данные. Кроме того, учтем то, что ширина данных может быть больше единицы.

Пусть такое дерево на рис. 27.3 имеет высоту, равную 3, минимальную ширину (если бы мы его печатали), равную 7, и максимальную ширину данных 3. Ширина и высота окна соответственно 800 и 600, а отступ от краев 10.

При этом радиусы кругов (размещаться данные будут в кругах) таковы, что между любыми элементами можно расположить как минимум один такой же.

Понадобится несколько аргументов, от которых в дальнейшем будет зависеть то, как дерево будет выглядеть на рисунке.

Кроме того, что в функцию отображения дерева придется занести само дерево, необходимо воспользоваться и знанием о ширине, высоте окна, величине отступа от краев окна (дабы дерево не находилось вплотную к его краям), а также о знакомом нам из функции вертикальной печати коэффициенте ширины данных (КШД). Вообще, у нас был еще коэффициент расширения дерева, преобразующий координаты элементов при вертикальной печати таким же образом, что и КШД,

но в данном случае использовать его бессмысленно, ведь он будет ухудшать видимость элементов (незачем еще раз масштабировать дерево, раз это приведет к дополнительным неудобствам даже для пользователя, желающего посмотреть на рисунок дерева).

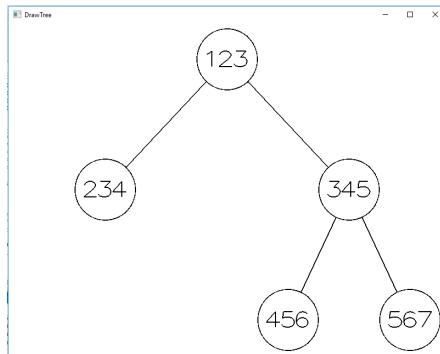


Рис. 27.3. Изображение дерева в окне 800×600

На основании вышеизложенных аргументов нужно просчитать в первую очередь радиусы кругов элементов, затем координаты самих элементов, а затем координаты текста.

Обозначения:

`window_width` – ширина окна; `window_height` – высота окна; `tree_width` – минимальная ширина дерева (МШД), используемая для вертикальной печати; `tree_height` – высота дерева; `shift` – отступ от краев (решено сделать его одинаковым с двух сторон); R – радиус круга; `node_x` – x -координата элемента; `node_y` – y -координата элемента; k_x – коэффициент пропорциональности по оси Ox ; k_y – коэффициент пропорциональности по оси Oy ; `text_x` – x -координата текста; `text_y` – y -координата текста; x – индекс узла на данном уровне, если представить уровень как массив; y – текущий уровень; k – коэффициент расширения данных.

Если элемент в дереве один (а это означает, что высота и МШД равны), то $R = \frac{\text{window_height} - 2 * \text{shift}}{2 * \text{tree_width}}$, иначе $R = \frac{\text{window_width} - 2 * \text{shift}}{2 * \text{tree_width}}$.

Если элемент в дереве один, то $\text{node_x} = \frac{\text{window_width}}{2}$,

$\text{node_y} = \frac{\text{window_height}}{2}$, иначе считаем коэффициенты пропор-

циональности по осям Ox и Oy , чтобы получить координаты:

$$k_x = \frac{\text{window_width} - 2 * (\text{shift} + R)}{\text{tree_width} - 1},$$

$$k_y = \frac{\text{window_height} - 2 * (\text{shift} + R)}{\text{tree_height} - 1},$$

$$\text{node_x} = k_x * x + \text{shift} + R, \quad \text{node_y} = \text{window_height} - k_y * y - \text{shift} - R.$$

$$\text{text_x} = \text{node_x} - \frac{3R}{4}, \quad \text{text_y} = \text{node_y} - \frac{3R}{4k}.$$

Главное свойство дерева, которое будет нарисовано, – масштабируемость. При этом меняются не только расстояние между элементами и размеры кругов, но и шрифт (необходимо заострить на этом внимание, так как в будущем на основании этого свойства будет рисоваться текст определенным шрифтом).

На рис. 27.4 представлены деревья разных размеров в окошке одних и тех же размеров для показательности масштабирования.

Кроме того, при наведении курсора на элемент цвет окружности поменяется на синий, что продемонстрирует работу с мышью (рис. 27.5).

Поскольку некоторые актуальные функции OpenGL принимают только прописанные в них аргументы, что вызывает некоторые неудобства, придется воспользоваться глобальными переменными.

Ниже создается структура в заголовочном файле “tree.h”, в которой обозначены такие глобальные переменные, как tree – копия дерева, а также ширина, высота окна, отступ, коэффициент ширины данных, радиус круга, координаты чего-либо и переменная состояния, которая в будущем пригодится нам при работе с мышью:

```
struct SGlutContextStruct {  
    void* tree;  
    int window_width, window_height, shift, k, R, x, y,  
state;  
};
```

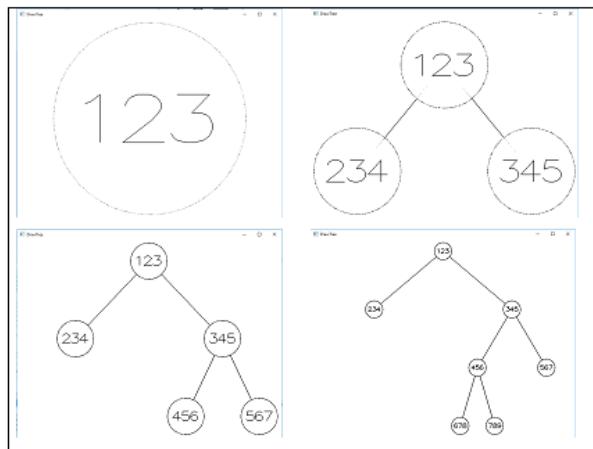


Рис. 27.4. Изображения вида деревьев в окне 800×600

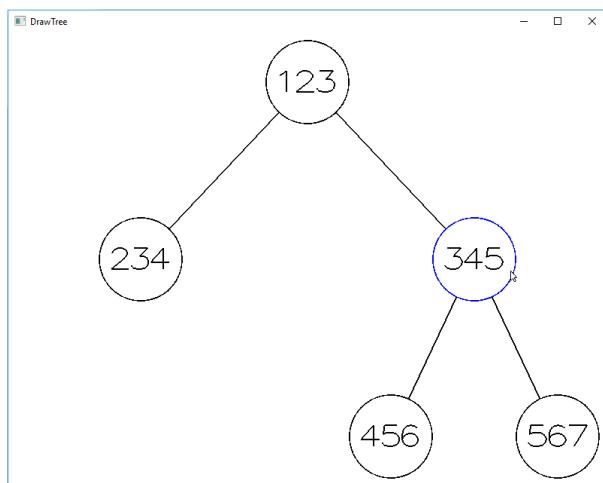


Рис. 27.5. Изменение цвета окружности при наведении мыши

В классе Tree появятся новые поля и функции. Главная функция – drawTree():

```
int node_x;
int node_y;
int text_x;
int text_y;
/* Установить координаты для данного
   узла при рисовании */
void setCoordsForNode(int window_width, int window_height,
int shift, int tree_width, int tree_height, int x, int y,
int R);
Tree* Tree<T>::getNodeByCoords(int x, int y, int R);
/* Установить координаты для текста
   текущего узла при рисовании */
void setCoordsForText(int k, int shift);
void drawTree(int argc, char** argv, int window_width, int
window_height, int shift, int k);           //рисовать дерево
```

Реализация функций

Подключается структура, позволяющая инициализировать переменные и выполнять с ними какие-либо операции в дальнейшем, в файле “tree.cpp” – он же файл описания классов Tree/SearchTree:

```
extern SGlutContextStruct glutContext;
```

Реализуются три первые новые функции, показанные до этого. Функции-сеттеры используют формулы, разработанные ранее, а функция-геттер, используя вспомогательную функцию, проходится по дереву и сверяется со стандартной формулой окружности в координатной плоскости xOy для того, чтобы найти узел:

```
template<class T>
void Tree<T>::setCoordsForNode(int window_width, int
window_height, int shift, int tree_width, int tree_height,
int x, int y, int R) {
    /* Это условие не выполняется, когда
       дерево состоит из одного элемента */
if (tree_width != tree_height) {
    // Коэффициент пропорциональности по оси Ox
```

```

int k_x = (window_width - 2 * (shift + R)) / (tree_width - 1);
                                // Коэффициент пропорциональности по оси Oy
int k_y = (window_height - 2 * (shift + R)) / (tree_height - 1);
node_x = k_x*x + shift + R;           // x-координата узла
node_y = window_height - k_y*y - shift - R;
                                         // y-координата узла
} else {
    node_x = window_width/2;           // x-координата узла
    node_y = window_height/2;           // y-координата узла
}
}

```

Ниже представлена реализация функции установления координат для текста и функции получения узла по координатам и радиусу.

Во вспомогательной функции `get_help()` применяется прямой обход по дереву с проверкой справедливости формулы окружности: $(x - x_0)^2 + (y - y_0)^2 = R^2$, где x и y – координаты мыши; x_0 и y_0 – координаты центра окружности элемента; R – радиус данной окружности.

Если ни один элемент дерева не будет подходить по данной формуле, функция вернет `NULL`:

```

template<class T>
void Tree<T>::setCoordsForText(int k, int R) {
    text_x = node_x - 3*R / 4;
                                         // x-координата первого символа текста
    text_y = node_y - 3*R / (4 * k);
                                         /* y-координата первого символа текста */
}
template<class T>
Tree<T>* Tree<T>::getNodeByCoords(int x, int y, int R) {
    Tree<T>* node = this;
    node = get_help(node, x, y, R);
    return node;
}
template<class T>
Tree<T>* get_help(Tree<T>* node, int x, int y, int R) {

```

```

if (pow(x - node->node_x, 2) + pow(y - node->node_y, 2)
<= pow(R,
2)) return node;
Tree<T>* temp = NULL;
if (node->getLeft() != NULL) {
    temp = get_help(node->getLeft(), x, y, R);
}
if (temp != NULL) return temp;
if (node->getRight() != NULL) {
    temp = get_help(node->getRight(), x, y, R);
}
return temp;
}

```

Чтобы формула нахождения координат элементов была справедлива, дерево должно быть полным. Для этого не случайно была объявлена переменная структуры tree, которая отныне будет хранить практически копию дерева, которое необходимо отобразить, только пополненного значениями NULL, не превышая высоту дерева. Сделаем мы это с помощью функции replaceNULLForEmpty(), ранее использованной в функции вертикальной печати.

Помимо изменений в переменной дерева нужно инициализировать и другие глобальные переменные, обозначенные в структуре.

Первые два аргумента функции drawTree() берутся из аргументов main() и необходимы в дальнейшей инициализации в связи с тем, что в данном примере приложение консольное, соответственно, может работать с консолью. Первый аргумент – это какое-либо число, а второй – массив символов:

```

template<class T>
void Tree<T>::drawTree(int argc, char** argv, int
window_width, int window_height, int shift, int k) {
    Tree<T>* temp = this->copyTree();
    temp = temp->replaceNULLforEmpty();
    glutContext.tree = temp;
    glutContext.window_width = window_width;
    glutContext.window_height = window_height;
    glutContext.shift = shift;
    glutContext.k = k;
    initWindow<T>(argc, argv);
}

```

Далее опишем работу функции `initWindow<T>()`.

`glutInit()` инициализирует GLUT. Мы обязательно должны ее вызвать до того, как начнем использовать любые другие функции данной библиотеки.

`glutInitDisplayMode()` задает режим дисплея, параметры `GLUT_DOUBLE|GLUT_RGB` обозначают двойную буферизацию и четырехкомпонентный цвет.

`glutInitWindowSize()` по заданным ширине и высоте определяет окно.

`glutCreateWindow()` создает его с заголовком в качестве аргумента.

`glutDisplayFunc()` подключает дисплей.

`glutReshapeFunc()` подключает функцию обработки изменений размеров окна.

`glutPassiveMotionFunc()` подключает работу с мышью, в данном случае – пассивное движение мыши, обозначающее, что мы просто двигаем мышью, не совершая никаких нажатий на ее клавиши.

`glutMainLoop()` запускает главный цикл.

```
template<class T>
void initWindow(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(glutContext.window_width,
                      glutContext.window_height);
    glutCreateWindow("DrawTree");
    glutDisplayFunc(display<T>);
    glutReshapeFunc(reshape);
    glutPassiveMotionFunc(mouseMove<T>);
    glutMainLoop();
}
```

Функции изменения размеров окна, изображения линий и окружности

Реализуем функцию обработки изменений размеров окна.

`glViewport()` отвечает за вывод изображения, начиная с первых двух координат и заканчивая последними двумя.

glMatrixMode(GL_PROJECTION) и glMatrixMode(GL_MODELVIEW) подключают матрицу проекции и модельно-видовую матрицу, а glLoadIdentity() загружает единичную матрицу.

gluOrtho2D() по четырем аргументам определяет систему координат. Первый аргумент – абстрактное лево, второй – право, третий – низ, четвертый – верх. Так, слева направо задается привычная нам ось Ox , а снизу вверх – ось Oy . В данном случае Ox начинается нулем, а заканчивается численным значением ширины окна. А ось Oy начинается нулем и заканчивается численным значением высоты окна.

glutContext.window_width и glutContext.window_height инициализируют переменные размеров окна, а glutPostRedisplay() будет перерисовывать изображение в окне, если его два параметра будут меняться:

```
static void reshape(int w, int h) {
    glViewport(0, 0, (GLsize i)w, (GLsize i)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, (GLsize i)w, 0, (GLsize i)h);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glutContext.window_width = w;
    glutContext.window_height = h;
    glutPostRedisplay();
}
```

Создадим и реализуем простейшую функцию рисования линии по координатам первой и второй точек.

glBegin(GL_LINES) говорит о том, что все отдельные пары вершин, написанные после него, будут соединяться линиями.

glColor3f(0.0, 0.0, 0.0) задает черный цвет.

glVertex2i(x1, y1) и glVertex2i(x2, y2) определяют соответственно конечные первую и вторую точки линии.

glEnd() говорит об окончании действия glBegin().

```
static void drawLine(int x1, int y1, int x2, int y2) {
    glBegin(GL_LINES);
    glColor3f(0.0, 0.0, 0.0);
```

```

glVertex2i(x1, y1);
glVertex2i(x2, y2);
glEnd();
}

```

Реализуем функции изображения кругов и окружностей.

Обе функции используют три аргумента: x - и y -координаты в сочетании с радиусом R .

`drawFillCircle()` рисует сначала белый круг, а затем окаймляет его черным «ободком» – окружностью. Белый круг – это просто точки, x - и y -координаты которых задаются формулами $x_1 = i \cdot \sin(t) + x$ и $y_1 = i \cdot \cos(t) + y$, где i – это часть или весь радиус, а t – градусная мера угла.

`drawBlueCircle()` рисует только окружность синего цвета.

```

static void drawFillCircle(int x, int y, int R) {
    glColor3f(1.0, 1.0, 1.0);
    float x1, y1;
    glBegin(GL_POINTS);
    for (int i = 0; i <= R; i++) {
        for (int t = 0; t <= 360; t++) {
            x1 = i*sin(t) + x;
            y1 = i*cos(t) + y;
            glVertex2f(x1, y1);
        }
    }
    glEnd();
    glColor3f(0.0, 0.0, 0.0);
    glBegin(GL_POINTS);
    for (int i = R-1; i <= R; i++) {
        for (int t = 0; t <= 360; t++) {
            x1 = R*sin(t) + x;
            y1 = R*cos(t) + y;
            glVertex2f(x1, y1);
        }
    }
    glEnd();
}
static void drawBlueCircle(int x, int y, int R) {
    glColor3f(0.0, 0.0, 1.0);
}

```

```
float x1, y1;
glBegin(GL_POINTS);
for (int i = R - 1; i <= R; i++) {
    for (int t = 0; t <= 360; t++) {
        x1 = R*sin(t) + x;
        y1 = R*cos(t) + y;
        glVertex2f(x1, y1);
    }
}
glEnd();
}
```

Шрифты в GLUT

GLUT предоставляет возможность использовать текст в программе для работы с OpenGL. Для того чтобы изобразить текст в GLUT, имеется две команды:

```
void glutBitmapCharacter(void *font, int character);
void glutStrokeCharacter(void *font, int character);
```

Первая из них рисует так называемый Bitmap-текст, а вторая Stroke-текст. Различие между Bitmap и Stroke состоит в том, что Stroke-текст можно как угодно масштабировать и изменять, а также поворачивать и так далее, но он немного хуже выглядит на экране, чем Bitmap-текст. Bitmap-текст, в отличие от Stroke, нельзя масштабировать, так как он определен в виде растровых букв, зато он красивее на экране и у него более богатый выбор разнообразных шрифтов. И еще, Bitmap-текст выводится на экран медленнее, чем Stroke-текст.

Вот примеры вызова этих функций:

```
draw_string_bitmap(GLUT_BITMAP_HELVETICA_18, "Hello");
draw_string(GLUT_STROKE_ROMAN, "Hello");
```

Осталось рассмотреть, какие шрифты имеются в GLUT (т.е. параметр font).

Для Stroke-текста параметр font может принимать значения

```
GLUT_STROKE_ROMAN
GLUT_STROKE_MONO_ROMAN
```

Для Bitmap определено большее число шрифтов (из-за невозможности видоизменяться) [24]:

```
GLUT_BITMAP_9_BY_15  
GLUT_BITMAP_8_BY_13  
GLUT_BITMAP_TIMES_ROMAN_10  
GLUT_BITMAP_TIMES_ROMAN_24  
GLUT_BITMAP_HELVETICA_10  
GLUT_BITMAP_HELVETICA_12  
GLUT_BITMAP_HELVETICA_18
```

Реализуем функцию изображения текста

Функция использует в качестве аргументов текст неопределенного типа, который затем переводится в строковый; шрифт, координаты текста, радиус и коэффициент ширины данных.

glPushMatrix() – положить в стек текущую матрицу, а glPopMatrix() – вытащить со стека положенную туда матрицу.

Между этими двумя командами мы будем делать различные вычисления и рисовать текст.

glTranslatef() – это переход в точку, с которой нужно начать что-либо рисовать.

После нее следует преобразование строки из типа string, сконвертированного из типа T, в тип char*.

Затем идет прохождение по массиву char*, и с помощью функции glutStrokeWidth() получается целочисленное значение ширины каждого символа. Из всех значений выбирается максимальное.

$$\text{По формулам } \text{expand_x} = \frac{1.5R}{k * \text{max_char_width}},$$

$\text{expand_y} = \frac{1.5R}{k * 100}$ определяется коэффициент расширения для осей Ox и Oy , чтобы масштабировать текст.

Условимся, что максимальная высота символа равна 100 пикселям, но это только для цифр. Можно, конечно, посчитать и для других символов, но нет необходимости вычислять пиксели и для них, пусть пока что для ознакомления будет достаточно цифр.

glScalef() масштабирует символы текста, а glStrokeCharacter() выводит.

```

template<class T>
static void drawText(T text, void* font, int text_x, int
text_y, int R, int k) {
    glColor3f(0.0, 0.0, 0.0);
    glPushMatrix();
    glTranslatef(text_x, text_y, 0.0);
    string s = to_string(text);
    char* s1 = newchar[s.size() + 1];
    for (int i = 0; i < s.size(); i++) {
        s1[i] = s.at(i);
    }
    s1[s.size()] = 0;
    char* c;
    int max_char_width = 0;
    int char_width = 0;
    for (c = s1; *c != '\0'; c++) {
        char_width = glutStrokeWidth(font, *c);
        if (max_char_width < char_width) max_char_width =
char_width;
    }
    float expand_x = (float)1.5*R /
(float)(k*max_char_width);
    float expand_y = (float)1.5*R / (float)(k*100);
    glScalef(expand_x, expand_y, 1.0);
    for (c = s1; *c != '\0'; c++)
        glutStrokeCharacter(font, *c);
    glPopMatrix();
}

```

Простейший пример использования glutBitmapCharacter():

```

void draw_string_bitmap(void *font, const char* string)
{
    while (*string)
        glutBitmapCharacter(font, *string++);
}
glRasterPos2f(0, 0);
draw_string_bitmap(GLUT_BITMAP_HELVETICA_18, "Hello!");

```

Для рисования текста с помощью Bitmap достаточно указать позицию с помощью glRasterPos2f() и посимвольно вывести каким-

либо шрифтом, например GLUT_BITMAP_HELVETICA_18, какую-либо строчку. Однако этот шрифт нельзя масштабировать самостоятельно, из-за чего приходится пользоваться уже готовыми вариантами размеров данного шрифта.

Легче уследить по математической зависимости за поведением векторного шрифта, чем за поведением растрового, учитывая его ограничения в масштабировании. Вследствие этого оперируем именно glutStrokeCharacter().

Организация функции display()

Первая часть функции – это инициализация и вычисление.

Во временные переменные копируются значения соответствующих глобальных переменных. Определяется высота дерева и минимальная ширина дерева – как в функции вертикальной печати. Инициализируются первоначальные значения известных переменных текущего уровня, индекса и числа пробелов для корня.

Задается структура и два списка.

Далее идет вычисление радиуса для каждого из элементов дерева по ширине, высоте окна и ширине дерева. Именно здесь устанавливается глобальная переменная R .

Просчитываются координаты для первого узла – корня, обновляются списки и структура элемента:

```
void display(void) {
    Tree<T>* tree = (Tree<T>*)glutContext.tree;
    int k = glutContext.k;
    int window_width = glutContext.window_width;
    int window_height = glutContext.window_height;
    int shift = glutContext.shift;
    int height = tree->getHeight();
        /* Максимальное число листов на
           нижнем уровне (нумерация с нуля) */
    int maxLeafs = pow(2, height - 1);
        /* Минимальная ширина дерева для
           печати (не конечная, но необходимая) */
    int width = 2 * maxLeafs - 1;
    int curLevel = 0;                      // Номер строки (на выводе)
    int index = 0;
        //номер элемента в строке (нумерация с нуля)
```

```

    // Позиция корня (число пробелов перед ним)
    int factSpaces = getPos(index, width, curLevel, height -
1);
    pos node;
    vector<Tree<T>*> V;
    vector<pos> Vi;
    int R;                                // Радиус круга
    R = (window_width - 2 * shift) / (2 * width);
    if (2*R*height > (window_height - 2*shift)) R =
(window_height - 2 *
    shift) / (2 * height);
    glutContext.R = R;
    // Установили координаты корня при рисовании
    tree->setCoordsForNode(window_width, window_height,
shift, width,
height, factSpaces, curLevel, R);
    V.push_back(tree);
    node.col = factSpaces;
    node.str = curLevel;
    Vi.push_back(node);

```

`glClearColor()` устанавливает белый цвет экрана, `glClear()` очищает его, `glLineWidth()` устанавливает ширину линии, с помощью которой будут отображаться в дальнейшем соединительные линии между вершинами, `glEnable()` с данным параметром устанавливает режим сглаживания.

Далее реализуется обычное поперечное прохождение по дереву, какое использовалось в функции вертикальной печати.

При условии, что потомок слева имеет данные, не равные `NULL`, будет отображаться линия в его сторону начиная с текущего элемента. Иными словами, если у узла A слева узел с данными `NULL`, линия между ними чертиться не будет:

```

glClearColor(1.0, 1.0, 1.0, 1.0);
glClear(GL_COLOR_BUFFER_BIT);
glLineWidth(2);
 glEnable(GL_POINT_SMOOTH);
for (int i = 0; i < tree->getAmountOfNodes(); i++) {
    if (pow(2, curLevel) <= index + 1) {
        index = 0;

```

```

    curLevel++;
}
if (V.at(i)->getLeft() != NULL) {
    V.push_back(V.at(i)->getLeft());
    factSpaces = getPos(index, width, curLevel, height - 1);
    node.col = factSpaces;
    node.str = curLevel;
    Vi.push_back(node);
    index++;
    V.at(i)->getLeft()->setCoordsForNode(window_width,
window_height,
shift, width, height, factSpaces, curLevel, R);
    if (V.at(i)->getLeft()->getData() != NULL) {
        int x1 = V.at(i)->node_x;
        int y1 = V.at(i)->node_y;
        int x2 = V.at(i)->getLeft()->node_x;
        int y2 = V.at(i)->getLeft()->node_y;
        drawLine(x1, y1, x2, y2);
    }
}

```

Аналогичным образом идет работа с правыми потомками:

```

if (V.at(i)->getRight() != NULL) {
    V.push_back(V.at(i)->getRight());
    factSpaces = getPos(index, width, curLevel, height - 1);
    node.col = factSpaces;
    node.str = curLevel;
    Vi.push_back(node);
    index++;
    V.at(i)->getRight()->setCoordsForNode(window_width,
window_height, shift, width, height, factSpaces,
curLevel, R);
    if (V.at(i)->getRight()->getData() != NULL) {
        int x1 = V.at(i)->node_x;
        int y1 = V.at(i)->node_y;
        int x2 = V.at(i)->getRight()->node_x;
        int y2 = V.at(i)->getRight()->node_y;
        drawLine(x1, y1, x2, y2);
    }
}

```

Только после того, как нарисовались линии, нужно отобразить круги, а затем и текст.

Делается это все при условии, что данные таких элементов не равны NULL.

Затем проверяется значение переменной state. Если она равна 0, нарисуется просто белый круг, окаймленный черной окружностью.

В противном случае допустим 1, нарисуется белый круг, окаймленный синей окружностью, т.е. при наведении мыши (функцию работы с мышью мы рассмотрим позже). Но произойдет это, если координаты данного узла будут соответствовать области, в которой сейчас находится курсор.

Далее рисуется текст и заканчивается цикл.

Напоследок после рисования нужно попросить OpenGL сменить экранные буфера при помощи glutSwapBuffers(), ведь у нас включена двойная буферизация.

glDisable() отключает сглаживание при таком параметре.

```
if (V.at(i)->getData() != NULL) {
    if (glutContext.state == 0) {
        drawFillCircle(V.at(i)->node_x, V.at(i)->node_y, R);
    }
    else {
        drawFillCircle(V.at(i)->node_x, V.at(i)->node_y, R);
        if ((tree->getNodeByCoords(glutContext.x, glutContext.y,
R)->
        ->node_x == V.at(i)->node_x)
            &&(tree->getNodeByCoords(glutContext.x, glutContext.y,
R)->node_y == V.at(i)->node_y))
            drawBlueCircle(V.at(i)->node_x, V.at(i)->node_y, R);
    }
    V.at(i)->setCoordsForText(k, R);
    drawText(V.at(i)->getData(), GLUT_STROKE_ROMAN, V.at(i)->text_x,
    V.at(i)->text_y, R, k);
}
}
glutSwapBuffers();
glDisable(GL_POINT_SMOOTH);
}
```

Дополнительный вариант вычисления координат узла

Деление экрана пополам также справедливо для рисования дерева отдельно для каждого уровня. Например, корень дерева должен располагаться по центру, соответственно, его x -координата – численно половина ширины окна (учтем то, что нужно вычесть два отступа). Его потомок слева имеет x -координату, расположенную посередине между левой границей рисования дерева и x -координатой своего родителя – корня. Потомок справа находится ровно посередине между x -координатой корня и правой границей. Все узлы на последующих уровнях также будут располагаться в середине пространства, ограниченного слева и справа x -координатами ближайших по оси Ox узлов.

Для этого в функцию `setCoordsForNode()` вместо аргумента `factSpaces`, представляющего собой количество пробелов перед узлом (если бы это был вывод дерева в консоли), нужно поместить аргумент `index`, идентифицирующий индекс элемента отдельного уровня-массива.

И теперь `node_x` – x -координата узла будет высчитываться по такой формуле:

$$\text{node_}_x = \frac{(\text{window_width} - 2 * \text{shift})|2x - 1|}{2^{y+1}} + \text{shift}.$$

Соответственно, в самой функции `node_x` будет высчитываться иначе:

$$\text{node_}_x = (\text{window_width} - 2 * \text{shift}) * \text{abs}(2 * x - 1) / \text{pow}(2, y + 1) + \text{shift}.$$

В отличие от предыдущего варианта вычисления `node_x`, этот справедлив как для корня, так и для любого другого узла.

Работа с мышью в OpenGL

Регистрация нажатий мыши

GLUT предлагает вам способ, чтобы зарегистрировать функцию, которая будет отвечать за обработку событий, создаваемых нажатиями клавиш мыши. Название этой функции `glutMouseFunc()`. Синтаксис выглядит следующим образом:

```
void glutMouseFunc(void (*func)(int button, int state, int x, int y));
```

Параметры:

*func – имя функции, которая будет обрабатывать события мыши.

Как видно, с момента подписания glutMouseFunc() функция, которая будет обрабатывать события мыши, должна иметь четыре параметра. Первый из них касается того, какая кнопка была нажата или отпущена. Этот аргумент может иметь одно из трех значений:

```
GLUT_LEFT_BUTTON;  
GLUT_MIDDLE_BUTTON;  
GLUT_RIGHT_BUTTON;
```

Второй аргумент относится к состоянию кнопки, т.е. идентификации момента нажатия и отпускания кнопки. Возможные значения:

```
GLUT_DOWN;  
GLUT_UP;
```

Если ответный вызов генерируется со статусом GLUT_DOWN, приложение может предположить, что GLUT_UP будет после этого события, даже если мышь перемещается за пределы окна. Остальные два параметра обеспечивают x-, у-координаты мыши относительно левого верхнего угла рабочей области окна.

Определение перемещения мыши

GLUT дает возможность обнаружения движения мыши для нашего приложения. Есть два типа движения для GLUT: активные и пассивные. Активное движение происходит при перемещении мыши и нажатии кнопки. Пассивные движения – когда мышь движется, но ни одна кнопка не нажата. Если приложение отслеживает движение манипулятора, будет сгенерировано событие в кадре во время периода, когда мышь движется.

Нужно зарегистрировать функцию, которая будет отвечать за обработку событий движения. Определено две различных функции: одна для отслеживания пассивных движений, а другая, чтобы отслеживать активные движения.

Синтаксис GLUT-функций слежения за перемещением:

```
void glutMotionFunc(void (*func) (int x,int y));  
void glutPassiveMotionFunc(void (*func) (int x, int y));
```

Параметры:

*func – функция, которая будет отвечать за соответствующий тип движения.

Параметрами функции обработки движения мыши являются x - , y -декартовы координаты курсора мыши относительно левого верхнего угла рабочей области окна.

Обнаружение попадания или выхода из рабочей области окна курсора мыши

GLUT может определять, когда мышь покидает или входит в рабочую область окна. Зарегистрируем функцию обратного вызова для обработки этих двух событий. GLUT-функцией регистрации является обратный вызов glutEntryFunc(), и синтаксис выглядит следующим образом:

```
void glutEntryFunc(void (*func)(int state));
```

Параметры:

*func – функция, которая будет обрабатывать эти события.

Параметр функции, которая будет обрабатывать эти события, сообщает нам, если мышь попадает в левую область окна. GLUT определяет две константы, которые можно использовать в приложении [22]:

```
GLUT_LEFT;  
GLUT_ENTERED;
```

Реализуем функцию работы с мышью.

Функция работы с мышью в качестве аргументов принимает текущие координаты курсора. Затем по функции нахождения узла по этим координатам определяется, попадает ли курсор в нужную область.

О том, что он не попадает, сообщит функция getNodeByCoords(), вернув значение NULL. Тогда придадим глобальной переменной state значение 0. Учтем, в функции display() это значение говорило о том,

что соответствующий узел будет нарисован по умолчанию, т.е. белый кружок и черная окружность вокруг него.

Если значение не равно NULL, обновятся глобальные переменные *x* и *y* полученными значениями, обновится переменная state, допустим, значением 1.

В обоих случаях необходимо перерисовать изображение с помощью функции glutPostRedisplay():

```
template <class T>
static void mouseMove(int x, int y) {
    Tree<T>* tree = (Tree<T>*)glutContext.tree;
    int R = glutContext.R;
    Tree<T>* node = tree->getNodeByCoords(x,
                                              glutContext.window_height-y, R);
    if (node != NULL) {
        glutContext.x = x;
        glutContext.y = glutContext.window_height-y;
        glutContext.state = 1;
        glutPostRedisplay();
    } else {
        glutContext.state = 0;
        glutPostRedisplay();
    }
}
```

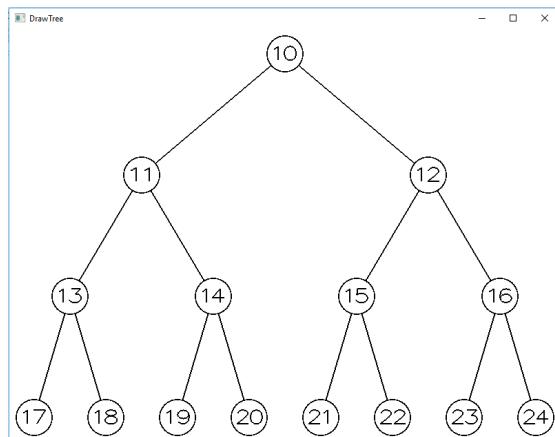


Рис. 27.6. Изображение полного дерева

В итоге составим дерево на рис. 27.6 по следующему программному коду:

```
vector<int> arr = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24};
Tree<int>* tree = newTree<int>(arr.at(0));
for (int i = 0; i < arr.size(); i++) {
    int left = 2 * i + 1;
    int right = left + 1;
    if (left < arr.size()) {
        tree->findElement_insertLeft(tree, arr.at(i),
arr.at(left));
    }
    if (right < arr.size()) {
        tree->findElement_insertRight(tree, arr.at(i),
arr.at(right));
    }
}
tree->drawTree(argc, argv, 800, 600, 10, 2);
```

СПИСОК ЛИТЕРАТУРЫ

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами на С++. – М.: БИНОМ, 1998. – 560 с.
2. Ваныкина Г.В., Сундукова Т.О. Алгоритмы компьютерной обработки данных: учеб. пособие. – Тула: Изд-во Тул. гос. пед. ун-та им. Л.Н. Толстого, 2011. – 219 с.
3. Гольдштейн А.Л. Теория принятия решений. Задачи и методы исследования операций и принятия решений: учеб. пособие. – 2-е изд., испр. – Пермь: Изд-во Перм. гос. техн. ун-та, 2009. – 361 с.
4. Давыдов В.Г. Программирование и основы алгоритмизации. – М., 2003. – 447 с.
5. Джамса К. Учимся программировать на языке С++. – М.: Мир, 1997. – 320 с.
6. Крупник А.Б. Самоучитель С++. – СПб.: Питер, 2005. – 233 с.
7. Мамонова Т.Е. Информатика. Общая информатика. Основы языка С++: учеб. пособие. – Томск: Изд-во Томск. политехн. ун-та, 2011. – 206 с.
8. Майн М., Савитч У. Структуры данных и другие объекты в С++: пер. с англ. – 2-е изд. – М.: Вильямс, 2002. – 832 с.
9. Ноткин А.М. Объектно-ориентированное программирование: ООП на языке С++: учеб. пособие. – Пермь: Изд-во Перм. нац. исслед. политехн. ун-та, 2013. – 230 с.
10. Павловская Т.А. С/С++. Программирование на языке высокого уровня. – СПб.: Питер, 2007 – 461 с.
11. Павловская Т.А., Щупак Ю.А. С++. Объектно-ориентированное программирование: практикум. – СПб., 2006. – 265 с.
12. Подбельский В.В. Язык Си++: учеб. пособие. – М.: Финансы и статистика, 1996. – 560 с.
13. Скиена С. Алгоритмы. Руководство по разработке: пер. с англ. – 2-е изд. – СПб.: БХВ-Петербург, 2011. – 720 с.
14. Прата С. Язык программирования С++. Лекции и упражнения: пер. с англ. – 6-е изд. – М.: Вильямс, 2012. – 1248 с.

15. Страуструп Б. Языки программирования C++. – СПб.: БИНОМ, 1999. – 991 с.
16. Топп У., Форд У. Структуры данных в C++: пер. с англ. – М.: БИНОМ, 1999. – 816 с.
17. Шмидский Я.К. Программирование на языке C/C++. Самоучитель. – М.: Вильямс, 2004. – 368 с.
18. C++ Reference. – URL: <http://www.cplusplus.com/reference> (accessed 20 April 2018).
19. Двусвязные списки в C++ [Электронный ресурс] // Cppstudio.com. – URL: <http://cppstudio.com/post/8482/> (дата обращения: 16.07.2018).
20. Очередь в C++ [Электронный ресурс] // Cppstudio.com. – URL: <http://cppstudio.com/post/8487/> (дата обращения: 16.07.2018).
21. Структура данных: стеки [Электронный ресурс] // Cppstudio.com. – URL: <http://cppstudio.com/post/5155/> (дата обращения: 17.07.2018).
22. Библиотека GLUT. Урок 7. Работа с мышью [Электронный ресурс] // grafika.me. – URL: <http://grafika.me/node/133> (дата обращения: 16.05.2018).
23. Знакомимся с OpenGL [Электронный ресурс] // habr.com. – URL: <https://habr.com/post/111175/> (дата обращения: 16.05.2018).
24. Как использовать шрифты в GLUT? [Электронный ресурс] // programmingcpp. narod.ru. – URL: http://programmingcpp.narod.ru/font_in_GLUT.htm (дата обращения: 16.05.2018).
25. Лекция 38: Алгоритмы поиска в линейных структурах [Электронный ресурс] / НОУ ИНТУИТ. – URL: <https://www.intuit.ru/studies/courses/648/504/lecture> (дата обращения: 17.08.2018).

Учебное издание

Полякова Ольга Андреевна,
Викентьева Ольга Леонидовна

ТЕХНОЛОГИИ РАЗРАБОТКИ
ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ
ПРОГРАММ НА ЯЗЫКЕ С++

В трех частях

ЧАСТЬ III. ПРЕДСТАВЛЕНИЕ ГРАФИЧЕСКИХ ОБЪЕКТОВ
И ПРОЕКТИРОВАНИЕ ПРОГРАММ
НА АЛГОРИТМИЧЕСКОМ ЯЗЫКЕ С++

Учебное пособие

Редактор и корректор Е.Б. Денисова

Подписано в печать 09.03.2021. Формат 60×90/16.
Усл. печ. л. 12,6. Тираж 43 экз. Заказ № 30/2021.

Издательство
Пермского национального исследовательского
политехнического университета.
Адрес: 614990, г. Пермь, Комсомольский пр., 29, к. 113.
Тел. (342) 219-80-33