МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

П. Г. КОЛИНЬКО

# ПОЛЬЗОВАТЕЛЬСКИЕ КОНТЕЙНЕРЫ

Учебно-методическое пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2022

УДК 004.424:004.422.63(07)
ББК З 973.2–018я7
К 60
**Колинько П. Г.**
К60 Пользовательские контейнеры: учебно-метод. пособие. СПб.: СПбГЭТУ «ЛЭТИ», 2022. 64 с. (вып.2202)
ISBN 978-5-7629-

Описывается цикл зачётных самостоятельных работ на ПЭВМ. Содержатся материалы для курсовой работы.

Предназначено для студентов бакалавриата по направлению 09.03.01 «Информатика и вычислительная техника» дневной, очно-заочной и заочной форм обучения.

УДК 004.424:004.422.63(07)
ББК З 973.2–018я7

Рецензент: канд. техн. наук, доцент СПбГУПТД Н. А. Мальгунова.

Утверждено
редакционно-издательским советом университета
в качестве учебно-методического пособия

ISBN 978-5-7629- 

# ВВЕДЕНИЕ

Цель методических указаний — поддержка второй части двухсеместрового курса «Алгоритмы и структуры данных». Пособие содержит три тематических главы. Первые две посвящены завершению знакомства с возможностями объектного программирования. На простом примере изучаются способы объединения нескольких классов в одном проекте — наследование и включение, а также механизм поддержки обработки особых ситуаций. Третья глава посвящена комбинированным структурам данных. В ней рассматриваются способы размещения множеств в памяти ЭВМ, оптимизированные под различные задачи работы с ними. Изучаются алгоритмы эффективного выполнения двуместных операций над множествами в этих структурах данных, а также применение их для поддержки произвольных последовательностей. Проводится эксперимент по созданию пользовательского контейнера, поддерживающего операции как с множеством, так и с последовательностью с помощью алгоритмов из стандартной библиотеки шаблонов.

В четвёртой главе даются материалы для курсовой работы, содержанием которой является эксперимент по прямому измерению временной сложности цепочки операций с пользовательский контейнером. В результате должно быть дано заключение об эффективности реализации набора операций с ним и предложения по совершенствованию алгоритмов отдельных операций.

В каждой главе даны ссылки на литературу, которую следует проработать для изучения темы.

Каждая из четырёх зачётных работ рассчитана на четыре учебных недели. Отчёты по работам должны содержать: название темы; текст индивидуального задания; тесты для проверки программ; результаты их работы; выводы, обязательно содержащие заключение о временной сложности реализованных алгоритмов; список использованных источников; перечень приложений. Обязательное приложение: исходный текст отлаженной программы на машинном носителе.

Все примеры, имеющиеся в пособии, проверены в оболочке Visual C++ 2017. Для обучения могут быть использованы любые программные оболочки типа Linux с поддержкой компилятора g++ с опцией C++17. Visual C++ 2017 и 2019 в варианте Community доступен для скачивания на сайте Microsoft. Актуальную информацию о стандартах C++11/14/17, необходимую для работы, и другую справочную информацию о языке и STL можно получить на сайте ru.cppreference.com. Нововведения подробно изложены в [11, с. 1049–1151] и кратко — в [16, с. 164–168]. Изучить программирование на C++ в новом стандарте можно также по [3] и [6]. С основами библиотеки STL можно ознакомиться по [2], [4] и [10]. Типовые ошибки в программах на C++ обсуждаются в [7] и [12].

# 1. РАБОТА С ИЕРАРХИЕЙ ОБЪЕКТОВ: НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Производные классы — это простое, гибкое и эффективное средство определения класса с целью повторного использования готового программного кода. Новые возможности добавляются к уже существующему классу, не требуя его перепрограммирования или перекомпиляции. С помощью производных классов можно организовать общий интерфейс с несколькими различными классами так, что в других частях программы можно будет единообразно работать с объектами этих классов. Понятие виртуальной функции позволяет использовать объекты надлежащим образом даже в тех случаях, когда их тип на стадии компиляции неизвестен. Основное назначение производных классов — упростить программисту задачу выражения общности классов.

Любое понятие существует не изолированно, а во взаимосвязи с другими понятиями, и мощность данного понятия во многом определяется наличием таких связей. Если класс служит для представления понятий, встает вопрос: как представить взаимосвязь понятий? Понятие производного класса и поддерживающие его языковые средства служат для представления иерархических связей, иными словами, для выражения общности между классами. Например, понятия окружности и треугольника связаны между собой, так как оба они представляют еще понятие фигуры, т. е. содержат более общее понятие. Чтобы представлять в программе окружности и треугольники и при этом не упускать из вида, что они являются фигурами, надо явно определять классы «окружность» и «треугольник» так, чтобы было видно, что у них есть общий класс — фигура. Эта простая идея, по сути, является основой того, что обычно называется объектно-ориентированным программированием.

Подробнее см. [3, с. 200–210], [15, с. 149–180].

Рассмотрим учебную программу, использующую некоторые из этих идей, прототип которой взят из [15]. Программа предназначена для вывода на экран картинок, составленных из набора заготовок — «фигур».

В программе объявлен абстрактный класс «фигура» (*shape*). Все конкретные фигуры — линия, прямоугольник и т. п. — являются производными от этого класса. Класс «фигура» определяет действия, необходимые для всех фигур. Он также создает из всех объявляемых фигур общий список, который может быть обработан функцией рисования при выдаче фигур на экран.

В классе «фигура» объявлен набор функций-членов, которые должны поддерживать все фигуры, чтобы из них можно было создавать картинки. Это функции для рисования, перемещения и изменения размеров фигуры, а также функции, возвращающие координаты всех крайних точек фигуры, по которым их можно будет стыковать. Функции — чисто виртуальные, они должны быть обязательно определены затем отдельно в каждой фигуре. Имеются также два дополнительных класса, уточняющие свойства фигур. Некоторые фигуры можно поворачивать. Для таких фигур имеется базовый класс *rotatable*, производный от *shape*. Для других фигур возможна операция отражения относительно горизонтальной или вертикальной оси. Эти фигуры можно строить на базе класса *reflectable*. Если для фигуры нужно и то и другое, то она должна быть производной от обоих классов.

Класс «фигура» является ядром библиотеки фигур *snape.h*. Имеется также библиотека поддержки работы с экраном *screen.h*, в которой определены размеры экрана, введено понятие точки и перечислены утилиты работы с экраном, конкретизированные затем в *shape.h*. Для простоты и универсальности работа с экраном реализована как формирование и построчный вывод матрицы символов.

Предполагается, что файлы *screen.h* и *shape.h* — покупные, а разработчик создает только третий файл — с самой прикладной программой.

### 1.1. Учебная программа «Библиотека фигур»

```
//=== Файл screen.h -- поддержка работы с экраном
const int XMAX=120; //Размер экрана
const int YMAX=50;
class point {      //Точка на экране
public:
  int x, y;
  point(int a = 0, int b = 0) : x(a), y(b) {  }
};
// Набор утилит для работы с экраном
void put_point(int a, int b); // Вывод точки (2 варианта)
void put_point(point p) { put_point(p.x, p.y); } //
void put_line(int, int, int, int); // Вывод линии (2 варианта)
void put_line(point a, point b)
   { put_line(a.x, a.y, b.x, b.y); }
void screen_init( );              // Создание экрана
void screen_destroy( );           // Удаление
```

```
void screen_refresh( );          // Обновление
void screen_clear( );            // Очистка
//======== Файл shape.h — библиотека фигур =========
#include <list>
//==1. Поддержка экрана в форме матрицы символов ==
char screen[YMAX] [XMAX];
enum color { black = '*', white = '.' };

void screen_init( )
{
  for (auto y = 0; y < YMAX; ++y)
    for (auto &x : screen[y])  x = white;
}
void screen_destroy( )
{
  for (auto y = 0; y < YMAX; ++y)
    for (auto &x : screen[y])  x = black;
}
bool on_screen(int a, int b) // проверка попадания точки на экран
{ return 0 <= a && a < XMAX && 0 <= b && b < YMAX; }
void put_point(int a, int b)
{ if (on_screen(a,b)) screen[b] [a] = black; }
void put_line(int x0, int y0, int x1, int y1)
/* Алгоритм Брезенхэма для прямой:
рисование отрезка прямой от (x0,y0) до (x1,y1).
Уравнение прямой: b(x-x0) + a(y-y0) = 0.
Минимизируется величина abs(eps), где eps = 2*(b(x-x0)) + a(y-y0).  */
{
  int dx = 1;
  int a = x1 - x0;   if (a < 0) dx = -1, a = -a;
  int dy = 1;
  int b = y1 - y0;   if (b < 0) dy = -1, b = -b;
  int two_a = 2*a;
  int two_b = 2*b;
  int xcrit = -b + two_a;
  int eps = 0;

  for (;;) { //Формирование прямой линии по точкам
     put_point(x0, y0);
     if (x0 == x1 && y0 == y1) break;
     if (eps <= xcrit) x0 += dx, eps += two_b;
     if (eps >= a || a < b) y0 += dy, eps -= two_a;
  }
```

```
}
void screen_clear( ) { screen_init( ); } //Очистка экрана
void screen_refresh( ) // Обновление экрана
{
  for (int y = YMAX-1; 0 <= y; --y) { // с верхней строки до нижней
    for (auto x : screen[y])    // от левого столбца до правого
      std::cout << x;
    std::cout << '\n';
  }
}
//== 2. Библиотека фигур ==
struct shape { // Виртуальный базовый класс "фигура"
  static list<shape*> shapes;// Список фигур (один на все фигуры!)
  shape( ) { shapes.push_back(this); } //Фигура присоединяется к списку
  virtual point north( ) const = 0;       //Точки для привязки
  virtual point south( ) const = 0;
  virtual point east( ) const = 0;
  virtual point west( ) const = 0;
  virtual point neast( ) const = 0;
  virtual point seast( ) const = 0;
  virtual point nwest( ) const = 0;
  virtual point swest( ) const = 0;
  virtual void draw( ) = 0;                //Рисование
  virtual void move(int, int) = 0;        //Перемещение
  virtual void resize(double) = 0;       //Изменение размера
  virtual ~shape( ) { shapes.erase(*this); } //Деструктор
};
list<shape*> shape::shapes;   // Размещение списка фигур
void shape_refresh( ) // Перерисовка всех фигур на экране
{
  screen_clear( );
  for (auto p : shape :: shapes) p->draw( ); //Динамическое связывание!!!
  screen_refresh( );
}
class rotatable : virtual public shape { //Фигуры, пригодные к повороту
public:
   virtual void rotate_left( ) = 0;        //Повернуть влево
   virtual void rotate_right( ) = 0;      //Повернуть вправо
};
class reflectable : virtual public shape { // Фигуры, пригодные
public:                                          // к зеркальному отражению
```

```
    virtual void flip_horisontally( ) = 0;          // Отразить горизонтально
    virtual void flip_vertically( ) = 0;            // Отразить вертикально
};
class line : public shape {
/* отрезок прямой ["w", "e"].
   north( ) определяет точку "выше центра отрезка и так далеко
   на север, как самая его северная точка", и т. п. */
protected:
    point w, e;
public:
  line(point a, point b) : w(a), e(b) { }; //Произвольная линия (по двум точкам)
  line(point a, int L) : w(point(a.x + L - 1, a.y)), e(a) {  }; //Горизонтальная линия
  point north( ) const { return point((w.x+e.x)/2, e.y<w.y? w.y : e.y); }
  point south( ) const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }
  point east( ) const { return point(e.x<w.x? w.x : e.x, (w.y+e.y)/2); }
  point west( ) const { return point(e.x<w.x? e.x : w.x, (w.y+e.y)/2); }
  point neast( ) const { return point(w.x<e.x? e.x : w.x, e.y<w.y? w.y : e.y); }
  point seast( ) const { return point(w.x<e.x? e.x : w.x, e.y<w.y? e.y : w.y); }
  point nwest( ) const { return point(w.x<e.x? w.x : e.x, e.y<w.y? w.y : e.y); }
  point swest( ) const { return point(w.x<e.x? w.x : e.x, e.y<w.y? e.y : w.y); }
  void move(int a, int b)          { w.x += a; w.y += b; e.x += a; e.y += b; }
  void draw( ) { put_line(w, e); }
  void resize(double d) // Изменение длины линии в (d) раз
  { e.x = w.x + (e.x - w.x) * d; e.y = w.y + (e.y - w.y) * d; }
};
// Прямоугольник
class rectangle : public rotatable {
/* nw ------ n ------ ne
   |                  |
   |                  |
   w          c       e
   |                  |
   |                  |
   sw ------- s ------ se */
protected:
  point sw, ne;
public:
  rectangle(point a, point b) :  sw(a), ne(b) { }
  point north( ) const { return point((sw.x + ne.x) / 2, ne.y); }
  point south( ) const { return point((sw.x + ne.x) / 2, sw.y); }
  point east( ) const { return point(ne.x, (sw.y + ne.y) / 2); }
```

```
  point west( ) const { return point(sw.x, (sw.y + ne.y) / 2); }
  point neast( ) const { return ne; }
  point seast( ) const { return point(ne.x, sw.y); }
  point nwest( ) const { return point(sw.x, ne.y); }
  point swest( ) const { return sw; }
  void rotate_right() // Поворот вправо относительно se
    { int w = ne.x - sw.x, h = ne.y - sw.y; //(учитывается масштаб по осям)
      sw.x = ne.x - h * 2; ne.y = sw.y + w / 2;}
  void rotate_left() // Поворот влево относительно sw
    { int w = ne.x - sw.x, h = ne.y - sw.y;
      ne.x = sw.x + h * 2; ne.y = sw.y + w / 2; }
  void move(int a, int b)
    { sw.x += a; sw.y += b; ne.x += a; ne.y += b; }
    void resize(int d)
  { ne.x = sw.x + (ne.x - sw.x) * d; ne.y = sw.y + (ne.y - sw.y) * d; }
  void draw( )
  {
    put_line(nwest( ), ne);   put_line(ne, seast( ));
    put_line(seast( ), sw);   put_line(sw, nwest( ));
  }
};
void up(shape& p, const shape& q) // поместить p над q
{   //Это ОБЫЧНАЯ функция, не член класса! Динамическое связывание!!
  point n = q.north( );
  point s = p.south( );
  p.move(n.x - s.x, n.y - s.y + 1);
}
//========== Файл shape.cpp (прикладная программа) ==========
// Пополнение и использование библиотеки фигур
#include "pch.h"          //связь с ОС (пример для Visual C++2017)
#include "screen.h"
#include "shape.h"
// ПРИМЕР ДОБАВКИ: дополнительный фрагмент - полуокружность
class h_circle: public rectangle, public reflectable {
  bool reflected;
public:
    h_circle(point a, point b, bool r=true) : rectangle(a, b), reflected(r) { }
    void draw();
    void flip_horisontally( ) { }; // Отразить горизонтально (пустая функция)
    void flip_vertically( ) { reflected = !reflected; };   // Отразить вертикально
};
```

```
void h_circle :: draw() //Алгоритм Брезенхэма для окружностей
{   //(выдаются два сектора, указываемые значением reflected)
    int x0 = (sw.x + ne.x)/2, y0 = reflected ? sw.y : ne.y;
    int radius = (ne.x - sw.x)/2;
    int x = 0, y = radius, delta = 2 - 2 * radius, error = 0;
    while(y >= 0) { // Цикл рисования
        if(reflected) { put_point(x0 + x, y0 + y*0.7); put_point(x0 - x, y0 + y*0.7); }
        else { put_point(x0 + x, y0 - y*0.7); put_point(x0 - x, y0 - y*0.7); }
       error = 2 * (delta + y) - 1;
       if(delta < 0 && error <= 0) { ++x; delta += 2 * x + 1; continue; }
       error = 2 * (delta - x) - 1;
       if(delta > 0 && error > 0) { --y; delta += 1 - 2 * y; continue; }
       ++x; delta += 2 * (x - y);  --y;
      }
}
// ПРИМЕР ДОБАВКИ: дополнительная функция присоединения…
void down(shape &p,  const shape &q)
{    point n = q.south( );
     point s = p.north( );
     p.move(n.x - s.x, n.y - s.y - 1); }

// Сборная пользовательская фигура - физиономия
class myshape : public rectangle { // Моя фигура ЯВЛЯЕТСЯ
     int w, h;                                  //       прямоугольником
     line l_eye; // левый глаз – моя фигура СОДЕРЖИТ линию
     line r_eye; // правый глаз
     line mouth; // рот
  public:
     myshape(point, point);
     void draw( );
     void move(int, int);
     void resize(int) { }
};
myshape :: myshape(point a, point b)
    : rectangle(a, b),      //Инициализация базового класса
      w(neast( ).x - swest( ).x + 1), // Инициализация данных
      h(neast( ).y - swest( ).y + 1), // - строго в порядке объявления!
      l_eye(point(swest( ).x + 2, swest( ).y + h * 3 / 4), 2),
      r_eye(point(swest( ).x + w - 4, swest( ).y + h * 3 / 4), 2),
      mouth(point(swest( ).x + 2, swest( ).y + h / 4), w - 4)
{ }
void myshape :: draw( )
```

```
{
    rectangle :: draw( ); //Контур лица (глаза и нос рисуются сами!)
    int a = (swest( ).x + neast( ).x) / 2;
    int b = (swest( ).y + neast( ).y) / 2;
    put_point(point(a, b)); // Нос – существует только на рисунке!
}
void myshape :: move(int a, int b)
{
    rectangle :: move(a, b);
    l_eye.move(a, b);  r_eye.move(a, b);
    mouth.move(a, b);
}
int main( )
{   setlocale(LC_ALL, "Rus");
   screen_init( );
//== 1.Объявление набора фигур ==
   rectangle hat(point(0, 0), point(14, 5));
   line brim(point(0,15),17);
   myshape face(point(15,10), point(27,18));
   h_circle beard(point(40,10), point(50,20));
   shape_refresh( );
   std::cout << "=== Generated... ===\n";
   std::cin.get(); //Смотреть исходный набор
//== 2.Подготовка к сборке ==
   hat.rotate_right( );
   brim.resize(2.0);
   face.resize(2.0);
   beard.flip_vertically();
    shape_refresh( );
   std::cout << "=== Prepared... ===\n";
   std::cin.get(); //Смотреть результат поворотов/отражений
//== 3.Сборка изображения ==
//  face.move(0, -10); // Лицо - в исходное положение
   up(brim, face);
   up(hat, brim);
   down(beard, face);
   shape_refresh( );
   std::cout << "=== Ready! ===\n";
   std::cin.get(); //Смотреть результат
   screen_destroy( );
   return 0; }
```

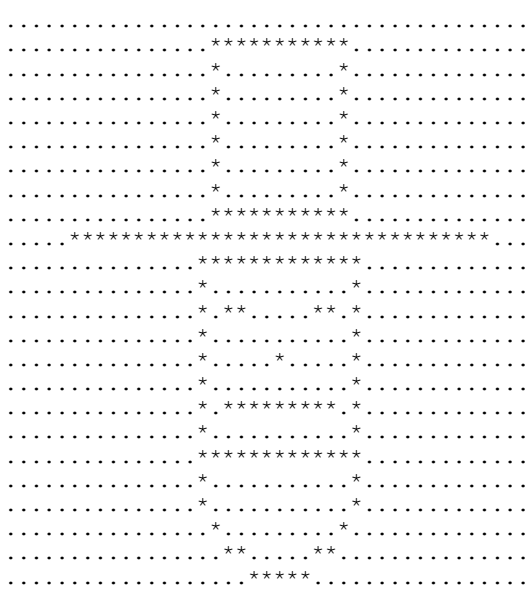


*Рис. 1.1.* Результат работы программы (с добавкой «бороды» в поз. 1)

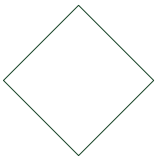
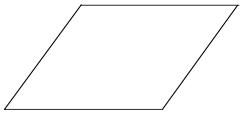
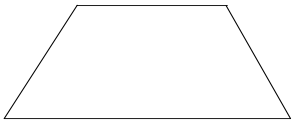
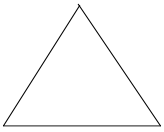
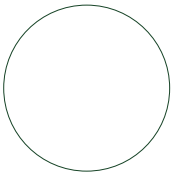
При запуске программы на экран сначала выводится объявленная коллекция фигур. Затем демонстрируется результат поворота/отражения некоторых фигур как подготовка к их использованию. Далее с помощью функций присоединения фигуры перемещаются и образуют заданную картинку: физиономию в шляпе (рис. 1.1). Для рисования использованы прямоугольники, линии и точки. Физиономия является пользовательской фигурой.

## 1.2. Практикум по гл. 1

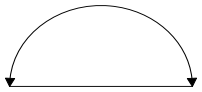
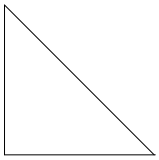
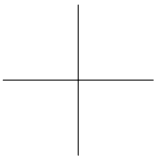
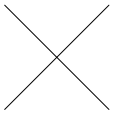
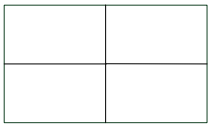
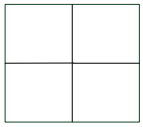
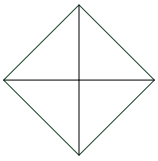
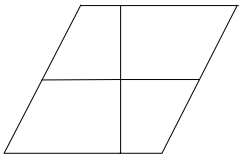
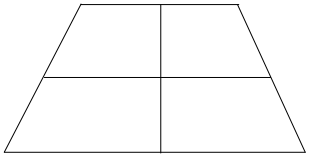
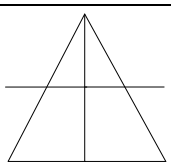
В табл. 1.1 собрана коллекция фигур, которыми можно дополнить рассмотренную прикладную программу для размещения на рис. 1.1.

*Таблица 1.1*

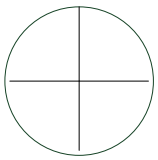
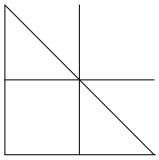
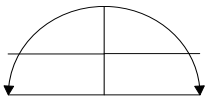
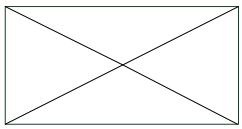
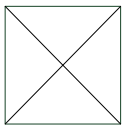
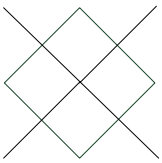
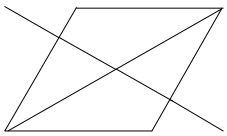
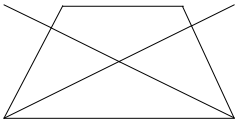
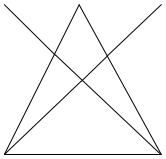
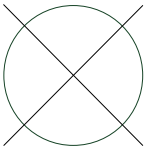
***Коллекция дополнительных фигур***

| № п/п | Наименование | Вид | Отражение | Поворот |
|---|---|---|---|---|
| 1 | Прямоугольник |  | Нет | Да |
| 2 | Квадрат |  | Нет | Нет |
| 3 | Ромб |  | Нет | Нет |
| 4 | Параллелограмм |  | Да | Да |
| 5 | Трапеция |  | Да | Да |
| 6 | Равносторонний треугольник |  | Да | Да |
| 7 | Кружок |  | Нет | Нет |

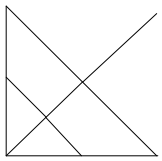
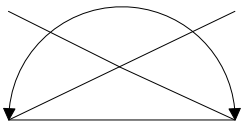
*Продолжение табл. 1.1*

| № п/п | Наименование | Вид | Отражение | Поворот |
|---|---|---|---|---|
| 8 | Полукруг |  | Да | Да |
| 9 | Прямоугольный треугольник |  | Да | Да |
| 10 | Крест |  | Нет | Нет |
| 11 | Косой крест |  | Нет | Нет |
| 12 | Прямоугольник с крестом |  | Нет | Да |
| 13 | Квадрат с крестом |  | Нет | Нет |
| 14 | Ромб с крестом |  | Нет | Нет |
| 15 | Параллелограмм с крестом |  | Да | Да |
| 16 | Трапеция с крестом |  | Да | Да |
| 17 | Треугольник с крестом |  | Да | Да |

*Продолжение табл. 1.1*

| № п/п | Наименование | Вид | Отражение | Поворот |
|---|---|---|---|---|
| 18 | Кружок с крестом |  | Нет | Нет |
| 19 | Прямоугольный треугольник с крестом |  | Да | Да |
| 20 | Полукруг с крестом |  | Да | Да |
| 21 | Прямоугольник с косым крестом |  | Нет | Да |
| 22 | Квадрат с косым крестом |  | Нет | Нет |
| 23 | Ромб с косым крестом |  | Нет | Нет |
| 24 | Параллелограмм с косым крестом |  | Да | Да |
| 25 | Трапеция с косым крестом |  | Да | Да |
| 26 | Треугольник с косым крестом |  | Да | Да |
| 27 | Кружок с косым крестом |  | Нет | Нет |

*Окончание табл. 1.1*

| № п/п | Наименование | Вид | Отражение | Поворот |
|---|---|---|---|---|
| 28 | Прямоугольный треугольник с косым крестом |  | Да | Да |
| 29 | Полукруг с косым крестом |  | Да | Да |

Для некоторых фигур возможны повороты на $90^{o}$ вправо или влево или отражение относительно горизонтальной и/или вертикальной оси симметрии, причем для части из них имеются обе возможности. Некоторые фигуры строятся как составные из более простых. Эти идеи можно отобразить показанной на рис. 1.2 иерархией классов.

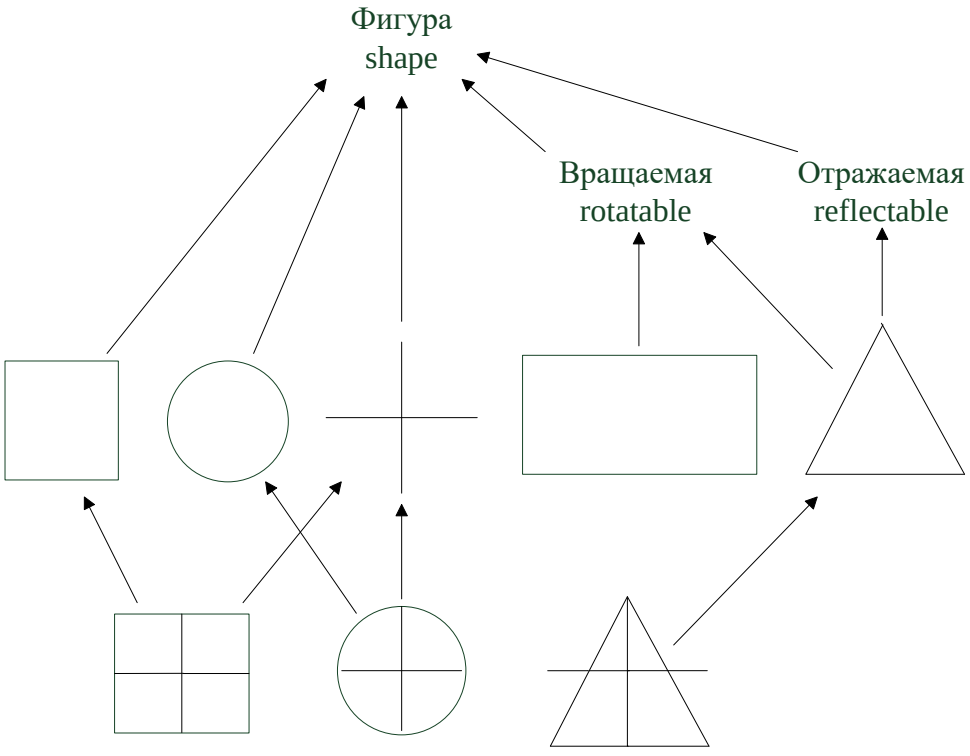



*Рис. 1.2*. Фрагмент возможной иерархии классов фигур

*Таблица 1.2*

***Индивидуальные задания к гл. 1***

| № вари-анта | Фигура | Расположение | № вари-анта | Фигура | Расположение |
|---|---|---|---|---|---|
| 1 | 11 | 1, 4, 5 | 26 | 25 | 7, 8, 14 |
| 2 | 15 | 1, 14 | 27 | 12 | 1, 10, 11 |
| 3 | 26 | 1, 13, 15 | 28 | 13 | 1, 7, 8, 12 |
| 4 | 7 | 10, 11, 12 | 29 | 2 | 4, 5, 9, 13, 15 |
| 5 | 8 | 2, 3, 7, 8 | 30 | 19 | 1, 7, 8 |
| 6 | 5 | 1, 4, 5 | 31 | 22 | 1, 2, 3, 14 |
| 7 | 4 | 2,3, 7, 8 | 32 | 21 | 1, 4, 5 |
| 8 | 9 | 1, 10, 11, 12 | 33 | 10 | 4, 5, 12 |
| 9 | 4 | 2, 3, 14 | 34 | 27 | 1, 7, 8, 14 |
| 10 | 15 | 7, 8, 12 | 35 | 3 | 2, 3, 7, 8 |
| 11 | 10 | 10, 11, 12 | 36 | 9 | 1, 4, 5, 12 |
| 12 | 25 | 4, 5, 9 | 37 | 29 | 4, 5, 6 |
| 13 | 6 | 1, 12, 13, 15 | 38 | 16 | 4, 5, 14 |
| 14 | 11 | 2, 3, 10, 11 | 39 | 8 | 1, 13, 15 |
| 15 | 28 | 1, 7, 8, 12 | 40 | 24 | 1, 2, 3, 14 |
| 16 | 17 | 6, 7, 8 | 41 | 16 | 2, 3, 9 |
| 17 | 12 | 2, 3, 7, 8 | 42 | 28 | 1, 13, 15 |
| 18 | 14 | 2, 3, 7, 8 | 43 | 22 | 1, 2, 3, 7, 8 |
| 19 | 18 | 1, 13, 15 | 44 | 7 | 2, 3, 12 |
| 20 | 13 | 7, 8, 12 | 45 | 18 | 10, 11, 14 |
| 21 | 24 | 2, 3, 12 | 46 | 3 | 2, 3, 6 |
| 22 | 19 | 2, 3, 14 | 47 | 17 | 9, 10, 11 |
| 23 | 14 | 7, 8, 14 | 48 | 5 | 1, 14 |
| 24 | 23 | 1, 4, 5 | 49 | 21 | 4, 5, 12 |
| 25 | 20 | 1, 7, 8 | 50 | 6 | 4, 5, 7, 8 |

**Задание: доработать модуль *shape.cpp***, добавив в коллекцию ещё одну фигуру, номер которой указан в табл. 1.2 в строке с вашим вариантом. Для этой фигуры нужно определить подходящее место в иерархии классов и написать недостающие функции-члены. Конструктор копии и другие генерируемые компилятором функции-члены, использование которых не предполагается, рекомендуется сделать недоступными. Грамотная иерархия наследования позволит сократить количество необходимых функций-членов. Переопределять память, имеющуюся в базовом классе, вообще не следует, но можно изменить её смысл в интересах наследуемых функций-членов.

Разработанной фигурой нужно дополнить картинку в указанных в варианте позициях (рис. 1.3). Позиция *1* обозначает галстук или воротник, *2* и *3* — бакенбарды, *4* и *5* — уши, *6* — кокарду, *7* и *8* — рога, *9* — нос, *10* и *11* — глаза, *12* — эмблему на шляпе, *13* и *15* — перья, *14* — шишак. Возможно, некоторые из фигур нужно будет повернуть или отразить. Так, в позициях *2, 4, 7, 10* и *13* фигура должна быть повернута (или отражена) влево; *3, 5, 8, 11* и *15* — вправо; *14* — отражена вверх; *1* и *9* — вниз.

Для фигур, назначенных в позиции *6, 9, 10, 11* и *12*, допускается замена отношения между классами ЯВЛЯЕТСЯ отношением СОДЕРЖИТ (см. комментарии в определении класса *myshape*).

Для примыкания фигур должны использоваться их габаритные точки. Необходимо написать аналоги функции *up* (поместить *p* над *q*), обеспечивающие примыкание очередной фигуры *p* с нужной стороны по отношению к уже размещенной *q*. Это должны быть обычные функции, а не функции-члены класса, чтобы их можно было применять для любых объектов. Категорически запрещается вычислять или подбирать координаты размещения фигуры вручную по готовой картинке и задавать их затем константами в функции move( ).

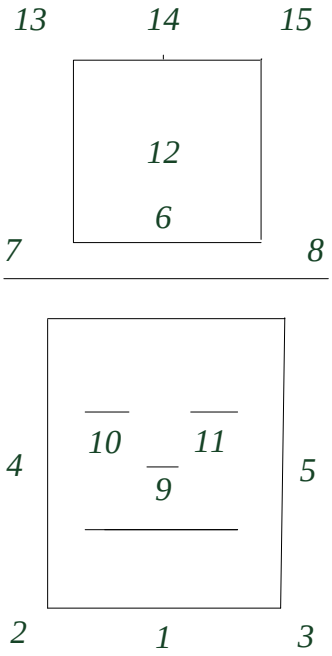



*Рис. 1.3*. Позиции для присоединения дополнительных фигур

При проектировании класса фигуры, допускающей поворот и/или отражение, рекомендуется отделить информацию, необходимую для рисования

фигуры с учетом её ориентации, и информацию для вычисления положения точек привязки, не зависящих от ориентации.

В результате прогона программа должна последовательно продемонстрировать:

1) исходный набор фигур, включая добавленные, которые следует расположить так, чтобы они по возможности не перекрывались; для добавления нескольких одинаковых фигур можно использовать конструктор копии, предусмотрев в нём смещение на экране изображения копии относительно оригинала;
2) результат подготовки фигур к сборке: изменение размеров, отражения, повороты; для вновь добавленных фигур должны быть продемонстрированы все заложенные для них возможности для трансформации;
3) собранное изображение физиономии со всеми заданными добавками.

### 1.3. Требования к отчёту

В отчёт по этой теме включите описание получившейся иерархии классов и пояснения:

1) какие классы пришлось добавить;

2) какие функции-члены пришлось переопределить и почему;

3) какие функции-члены сделаны недоступными и каким образом это осуществлено.

Приложите доработанный текст *shape.cpp* с выделением ваших добавок. Модули, в которых ничего не менялось, прикладывать не обязательно.

### 1.4. Контрольные вопросы

1. Какой базовый класс лучше всего использовать для производного класса «треугольник»?

2. То же — для класса «кружок».

3. То же — для класса «крестик».

4. Какой тип наследования следует выбрать: *private*, *public* или *protected*?

5. Можно ли вообще не указывать тип наследования?

6. В чём смысл объявления функций в базовом классе как виртуальных?

7. Нужно ли объявлять виртуальной функцию в производном классе, если в базовом она уже объявлена таковой?

8. Можно ли потребовать от компилятора проверить корректность объ-

явления виртуальной функции в производном классе и как это сделать?

9. Можно ли запретить виртуальную функцию в классах-наследниках?

10. Что такое «чисто виртуальная функция»?

11. Обязательно ли переопределять все функции-члены базового класса в производном классе?

12. Зачем может понадобиться создание набора (массива или списка) указателей на разные типы объектов в пределах некоторой иерархии?

13. Как запретить для объекта вызов конструктора по умолчанию?

14. Как запретить вызов конструктора для использования в качестве неявного преобразователя типа?

15. Каким образом можно установить значение переменных объекта, объявленных с модификаторами *const*?

16. Каким образом следует инициализировать объект базового класса в конструкторе производного класса? Всегда ли это нужно делать?

17. Каким требованиям должны удовлетворять классы, чтобы между ними можно было сформировать отношение «является»?

18. Как установить между двумя классами отношение «содержит»?

19. Каким образом на экране появляются глаза и рот, если функция *myshape* :: *draw*( ) не содержит кода для выдачи этих элементов фигуры?

20. Почему данные в классах *line* и *rectangle* сделаны *protected*?

## 2. ПОДДЕРЖКА ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Механизм исключительных ситуаций предоставляет пользователю возможность контроля хода выполнения программы и нейтрализации возможных ошибок. Очень часто исключительная ситуация — это не ошибка, а просто исчерпание какого-то ресурса, например, доступной памяти или времени ожидания сигнала, или даже один из предусмотренных вариантов завершения процесса. Механизм используется, если ситуация не может быть разрешена в той точке, где была выявлена, и требует перехода на более высокий уровень иерархии вызовов функций с завершением активных функций и освобождением ресурсов.

Для задействования механизма особых ситуаций в программе нужно проделать следующее:

— обнаружив в программе место, где особая ситуация может возникнуть, надо придумать для нее уникальное название, например *My_Error*, и объявить соответствующий класс ошибок, возможно, пустой:

```
class My_Error { };
```

— в точке программы, где обнаружена особая ситуация, поместить утверждение

```
throw My_Error( );
```

Это утверждение создает объект класса *My_Error*. Если в объекте предусмотрены поля для данных, через них можно передать информацию обработчику ошибок: аргумент утверждения *throw* — это конструктор объекта;

— точку вызова функции, которая может создать исключения, нужно поместить в блок контроля, за которым следуют обработчики особых ситуаций:

```
try{ //Начало блока контроля.
      Алгоритм, использующий функцию,
      которая может создать особую ситуацию
      (содержит утверждения throw My_Error( ));
}
catch (My_Error)
{ Обработка особой ситуации }
```

Предложение *catch* размещается на том уровне вложенности функций, где обработка ситуации *My_Error* возможна. Если нужно обрабатывать не-

сколько различных ошибок, после блока *try* последовательно размещаются соответствующие обработчики. При возникновении любой особой ситуации в блоке *try* его работа прерывается: происходит принудительный выход из всей цепочки вложенных функций, активных в точке особой ситуации, и вызов деструкторов для всех созданных при этом объектов, как это происходит при выходе из блока (области видимости). Этот процесс называется раскруткой стека: стек возвращается в состояние, в котором он был в момент входа в блок *try*.

Далее просматриваются блоки *catch* в том порядке, в каком они объявлены. Как только обнаруживается блок обработки ошибки нужного типа, управление передается ему. Остальные блоки *catch* не используются.

Если же выполнение блока *try* завершилось успешно, все блоки *catch* после него игнорируются.

Если для некоторого типа ошибки не обнаружено соответствующего блока *catch*, программа завершается аварийно. Чтобы этого избежать, последним в цепочке можно разместить блок *catch*(…), перехватывающий ошибки любого типа.

Как только подходящий блок *catch* будет вызван, особая ситуация будет считаться обработанной, даже если этот блок пуст. Однако чаще всего в него помещают выдачу на экран или в специальный файл (журнал) содержательного сообщения об ошибке. Возможно также одно из следующих действий:

— устранение причины ошибки (уменьшение запроса на выделение памяти, отказ от обработки несуществующего или испорченного файла и т. п.);

— аварийное завершение программы (вызов *abort*( ));

— возбуждение особой ситуации другого типа (вызов *throw* с соответствующим аргументом);

— перевозбуждение особой ситуации для передачи её на следующий уровень иерархии вызовов функций (вызов *throw* без аргумента).

Подробнее об особых ситуациях и их обработке см. [3, с. 222–230], [15, с. 232–256].

Правильный выбор уровня для размещения блока контроля позволяет сделать программу безопасной в смысле исключений (см. [12, с. 105–174]). Так, в учебном примере имеется следующая цепочка вызовов функций:

main( ) → screen_refresh( ) → myshape :: draw( ) → rectangle :: draw ( ) → put_line(a, b) → put_point(x, y) → on_screen(x, y).

Выход точки за пределы буферного массива *SCREEN* (экрана) выявляется функцией *on_screen*( ). Блок контроля вокруг вызова этой функции (или вызывающей её *put_point*) не имеет смысла: на этом уровне ничего, кроме выдачи сообщения об ошибке, сделать нельзя, а такое сообщение можно выдать и непосредственно, не прибегая к механизму *throw — cath*. На уровне *main*( ) или *screen_refresh*( ) обрабатывать ошибку поздно, здесь можно только прервать выполнение программы; без серьёзной доработки класса *shape* содержательное сообщение об ошибке получить нельзя. В то же время блок контроля внутри функции *myshape* :: *draw(* ) позволит локализовать ошибку при выводе прямоугольника — контура фигуры *myshape* и, возможно, попробовать изменить его размер. В общем случае проектирование реакции программной системы на ошибки должно выполняться одновременно с проектированием её самой. Так, в программе, рассмотренной в учебном примере, можно снабдить каждую фигуру автоматически формируемым порядковым номером, значение которого можно выводить как часть сообщения об ошибке «выход за пределы экрана».

Если ошибка выявлена в конструкторе фигуры, фигура не создаётся. Это не является проблемой для цепочки базовых классов-значений. Исключение — класс shape, управляющий цепочкой фигур для рисования. Этот класс должен иметь деструктор, удаляющий сбойную фигуру из цепочки или заменяющий ее специальной фигурой — значком ошибки. Деструктор для класса *shape* должен быть виртуальным, чтобы правильно удалять объекты всех производных классов.

Классы ошибок могут образовывать иерархию. В этом случае можно перехватом ошибки базового класса перехватить и все производные от него. Рекомендуется использовать в качестве базы исключений стандартное исключение *exception* (потребуется директива #*include* <*exception*>). Это позволит подключить программу к системному механизму перехвата исключений. Кроме того, можно использовать имеющийся в классе механизм для сообщений. Конструктор класса *exception* имеет аргумент типа *char** — строку (в стиле Си) для сообщения об ошибке. В блоке *catch* эта строка может быть получена вызовом виртуальной функции-члена *what*( ).

Например, класс ошибки перемещения фигуры может быть объявлен таким образом:

```
struct CantBeMoved : std::exception
```

```
{
   CantBeMoved(const std::string& s) : std::exception(s.c_str( )) { }
};
```

Возбуждение исключения:

```
throw CantBeMoved("Line can't be moved: out of screen");
```

Перехват:

```
catch (CantBeMoved &ex) {  std::cout << ex.what() << "\n\n";  }
```

### 2.1. Практикум по гл. 2

Переработать программу работы с библиотекой фигур, дополнив ее механизмом контроля исключительных ситуаций. Например, возможно выявление следующих ошибок:

— непопадание точки на экран;

— некорректные параметры при формировании фигуры;

— нехватка места на экране для размещения фигуры в одной из позиций (исходной, повернутой, отраженной, перемещенной);

— повторный поворот/отражение уже повернутой/отраженной фигуры и др.

Нужно реализовать генерацию и перехват не менее двух типов ошибок разного уровня сложности.

Если исключение генерируется в конструкторе фигуры, следует обеспечить исключение фигуры из списка для рисования или подмену её запасной фигурой — знаком ошибки.

Перехват исключения в той же функции, в которой оно возбуждено, не применяется. В этом случае, когда ошибку можно обработать в точке обнаружения, механизм исключений избыточен.

Организовать перехват исключений следует таким образом, чтобы искажения итоговой картинки были минимальны. Протестировать исключительные ситуации, результаты эксперимента поместить в отчёт.

### 2.2. Дополнительные требования к отчёту

В отчете по теме обоснуйте набор и вид классов для фиксации особых ситуаций, место расположения операторов *throw* и блоков контроля с целью получения безопасного кода. Приведите результаты тестирования.

## 2.3. Контрольные вопросы

1. Что такое исключительная ситуация при выполнении программы?

2. Как можно выявить исключительную ситуацию?

3. Что можно предпринять при выявлении исключительной ситуации?

4. Можно ли передать в обработчик особых ситуаций какую-либо информацию о произошедшем событии?

5. Можно ли обработать неизвестную особую ситуацию?

6. Можно ли сделать обработчик ситуации пустым?

7. Что можно предпринять, если для корректной обработки ситуации в данном месте программы у обработчика недостаточно данных?

8. Если требуется несколько обработчиков особых ситуаций, в каком порядке следует их размещать в программе?

9. Можно ли перехватывать одним обработчиком несколько различных особых ситуаций?

10. Как действуют обработчики в случае, когда никакой особой ситуации не произошло?

11. Как следует размещать блоки контроля, чтобы получить безопасный программный код?

12. Можно ли продолжить выполнение программы с точки, в которой выявлена ошибка, после внесения исправлений в данные?

# 3. КОМБИНИРОВАННЫЕ СТРУКТУРЫ ДАННЫХ И СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

В практических задачах множество часто используется как словарь. Основные операции со словарем — это поиск, добавление и удаление элементов множества (ключей). Двуместные операции со словарями выполняются сравнительно редко. При хранении множеств в форме упорядоченной последовательности операции поиска/вставки/удаления выполняются за линейное время. Если нужен только поиск, данные можно хранить в упорядоченном векторе (длиной $n$) с доступом к ключам за время $O(\log n)$. Для хранения словарей с возможностью пополнения применяются структуры данных, совмещающие быстрый поиск с быстрым добавлением и удалением ключей. Для небольших универсумов задачу решает вектор битов. Если же универсум велик, используются хеш-таблицы, деревья двоичного поиска и им подобные структуры данных.

## 3.1. Хеш-таблицы

Хеш-таблица — это обобщение способа хранения множества целых чисел (ключей) в форме вектора битов на случай, когда мощность универсума $U$ велика по отношению к мощности множеств, с которыми нужно работать. Функция отображения преобразует значения ключей к интервалу $[0, m - 1]$, где $m$ — размер хеш-таблицы, $m \ll |U|$. Очевидно, что при этом каждому индексу хеш-таблицы будет соответствовать много различных значений ключей. Поэтому, во-первых, в хеш-таблице приходится хранить не биты, а сами значения ключей, а во-вторых, имеется возможность размещать в ней более одного ключа для каждого значения функции отображения (разрешать коллизии).

Количество возможных коллизий можно уменьшить, если выполнить два условия:

1) выбрать размер хеш-таблицы с запасом. Если размер таблицы превышает мощность хранимого множества более чем вдвое, вероятность коллизии становится меньше 0.5. Если мощность множества заранее неизвестна, то выбирают некоторый начальный размер, а когда его оказывается недостаточно, таблицу перестраивают с увеличением размера (обычно вдвое);

2) подобрать функцию отображения (хеш-функцию) такую, чтобы все ячейки таблицы были востребованы по возможности с равной вероятностью,

независимо от того, какое распределение имеют хранящиеся в таблице ключи.

По способу разрешения коллизий различают хеш-таблицы двух типов:

1) с открытой адресацией. Конфликтующие значения ключей размещаются в свободных ячейках таблицы;

2) с цепочками переполнения. Каждая ячейка таблицы содержит указатель на список конфликтующих ключей.

Подробнее о хеш-таблицах — в [1, с. 115–128], [13, с. 529–556], [5, с. 316–338].

Таблицы второго типа применяются чаще, потому что для них не существует проблемы переполнения. Если мощность хранимого множества становится слишком большой, таблица просто начинает работать как $m$ списков. Если же таблица правильно построена и не переполнена, проверка принадлежности элемента множеству, а также вставка и удаление элемента выполняются в ней за постоянное время, примерно такое же, как и в массиве битов (рис. 3.1).

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 103 | 102 | 101 | 100 | 35 | 50 | – | 32 | 31 | 30 | – | 44 | 123 | 90 | 105 | 120 |
| 55 | 38 | 37 | 20 | – | – | – | 80 | – | – | – | 60 | – | 10 | – | 104 |
| – | 70 | – | – | – | – | – | – | – | – | – | – | – | – | – | 40 |
| – | – | – | – | – | – | – | – | – | – | – | – | – | – | – | – |

*Рис. 3.1.* Хеш-таблица из 16 экстентов с цепочками переполнения

За постоянное время (порядка размера таблицы) будут выполняться и двуместные операции над множествами: объединение, пересечение и разность: если хеш-функции для обоих множеств одинаковы, для этого пригоден такой же алгоритм попарного сравнения соответствующих ячеек, как и для массивов битов. В общем случае двуместная операция, организованная как просмотр первой таблицы, поиск каждого ключа во второй и вставка при необходимости в третью, имеет линейную сложность.

В хеш-таблице можно хранить и множество с повторениями: совпадающие значения ключей не создают никаких проблем, кроме гарантированных коллизий, которые разрешаются обычным образом.

К сожалению, всегда можно подобрать такие данные, что они все попадут в одну или несколько ячеек таблицы, образовав неупорядоченные списки (упорядочивание обычно не применяется). Это худший случай, для которого справедливы оценки временной сложности для списков: $O(n)$ — для поиска и удаления элемента; $O(n^2)$ — для двуместной операции над множествами.

Подбор подходящей хеш-функции — в общем случае достаточно сложная задача. Но если ключи представляют собой целые числа (или сводятся к таковым), хорошие результаты можно получить с хеш-функцией вида

$$h(x) = (a * x + b) \% m,$$

где $m$ – размер таблицы, $a$ и $b$ — простые числа.

Обычно $a$ выбирается близким по значению к $m$, а $b$ — к 1. Так, при $m = 100$ можно взять $a = 97$, $b = 11$. Такой выбор обеспечивает равномерное использование всех ячеек таблицы в большинстве практических случаев. Если размер таблицы $m$ предполагается изменять, можно взять в качестве $a$ любое достаточно большое простое число.

### *3.1.1. Контрольные вопросы*

1. Какой объём памяти нужно выделять под хеш-таблицу для хранения множеств со средней мощностью 50?
2. Хеш-таблица какого типа расходует больше памяти для хранения множества — с открытой адресацией или с цепочками переполнения?
3. Каким требованиям должна удовлетворять «хорошая» хеш-функция?
4. Можно ли построить хеш-таблицу, в которой не будет коллизий?
5. Каков оптимальный алгоритм выполнения двуместной операции над множествами в хеш-таблице? Какова его временная сложность?
6. Можно ли хранить в хеш-таблице множество с повторениями?
7. Какова временная сложность операций вставки и удаления элемента для хеш-таблицы?
8. Почему для операций с хеш-таблицей дают две оценки временной сложности — сложность в худшем случае и сложность в среднем?
9. Что такое вырождение хеш-таблицы и как его избежать?
10. Что нужно делать, если хеш-таблица переполнилась?

## 3.2. Деревья двоичного поиска

Дерево двоичного поиска (ДДП) — это способ хранения множества в форме расширяемого упорядоченного списка с сохранением упорядоченности при вставке новых элементов без перемещения уже имеющихся.

ДДП — это дерево с нагруженными узлами, вес в любом узле которого больше любого веса в левом его поддереве и не больше любого веса в правом поддереве. Количество шагов алгоритма поиска элемента множества в таком дереве не превышает его высоты, т. е. оценивается как $O(\log n)$. Такую же

сложность имеют операции вставки нового элемента в дерево и удаления элемента.

ДДП можно получить из упорядоченной последовательности ключей, если двоичное дерево соответствующей мощности разметить внутренним (симметричным) способом, а затем заменить номера узлов соответствующими элементами последовательности. Если последовательность хранится в массиве, то построить ДДП можно методом деления пополам: поместить в корень дерева средний по порядку элемент, затем рекурсивно создать левое поддерево из первой половины последовательности, а правое — из второй.

```
Node * Build(int a, int b, vector<int> data)
//Сборка ДДП из отрезка [a, b] упорядоченного вектора ключей data
{   if (b <= a) return nullptr;
    int  c = (a + b) /2;
    Node * s = new Node (data[ c ]);
    s->left = Build(a, c, data);
    s->right = Build(c+1, b, data);
    return s;
}
```

Недостаток ДДП в том, что оно хорошо работает только в том случае, если сбалансировано, т. е. длины путей из корня в любой лист примерно одинаковы. Однако при поэлементной вставке в дерево упорядоченной последовательности дерево вырождается, превращаясь в линейный список. Поиск, вставка и удаление в таком дереве будут выполняться не за логарифмическое, а за линейное время. Вероятность вырождения весьма велика. Так, из 7 узлов можно образовать только одно полностью сбалансированное дерево, а полностью вырожденных — 64 (рис. 3.2).

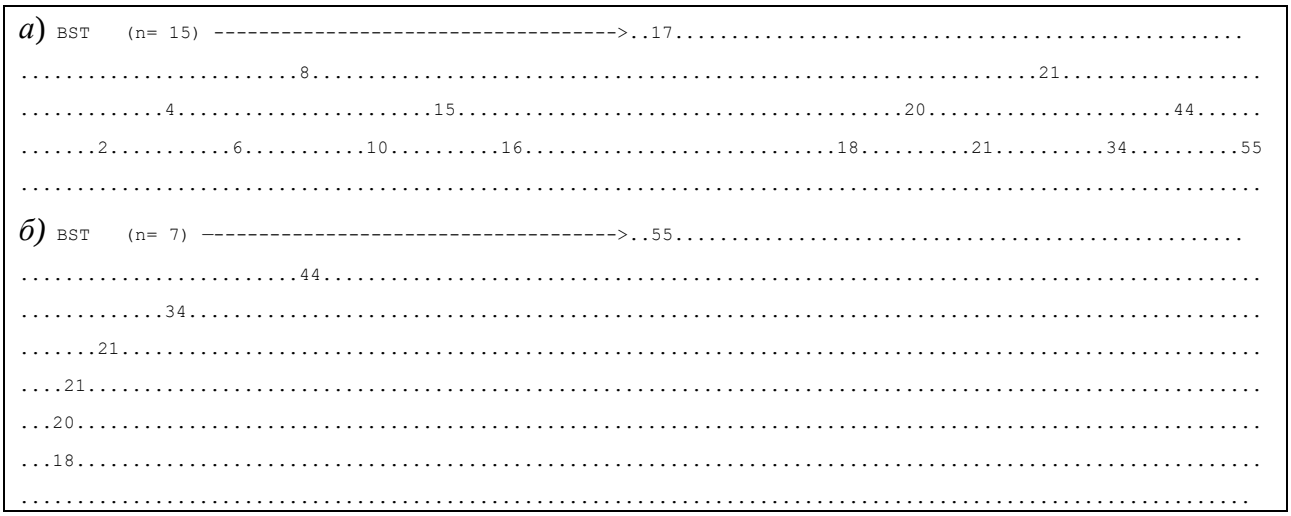



*Рис. 3.2*. Деревья двоичного поиска: *а* – сбалансированное, *б* – вырожденное

Поэтому алгоритмы работы с ДДП часто дополняют автобалансировкой после вставки и удаления. Наиболее употребительны следующие схемы.

1. АВЛ-деревья. Разность высот поддеревьев любого узла дерева не превышает 1. Информация о разности высот хранится в узле и используется для его перестройки после вставки и удаления узла (рис. 3.3).

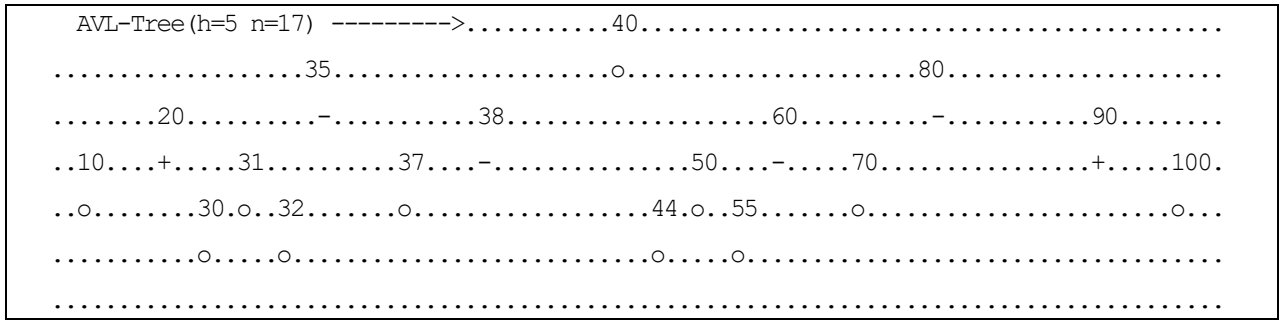



*Рис. 3.3*. АВЛ-дерево: –, o, + — значения баланса в узлах

2. Деревья с хранимой высотой. Если разместить в каждом узле ДДП поле со значением высоты поддерева, корнем которого является узел, возможна автоматическая поддержка балансировки способом, подобным АВЛ-дереву: балансы вычисляются на ходу сравнением высот поддеревьев в каждом проходимом узле (рис. 3.4).

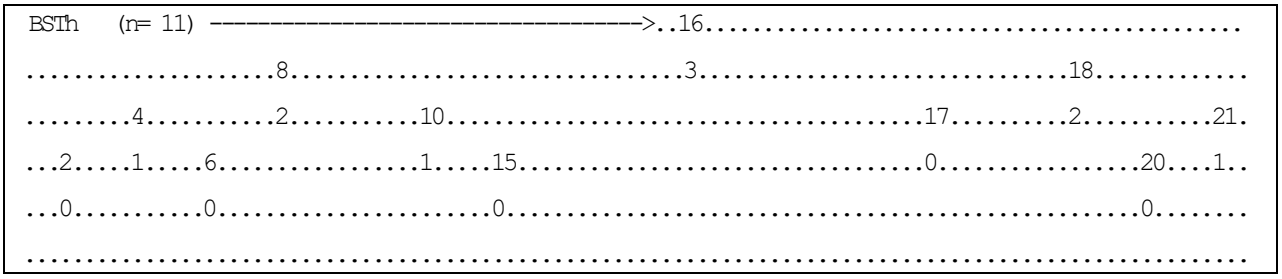



*Рис. 3.4.* ДДП с хранимой высотой (под каждым узлом — высота его поддерева)

3. Деревья с хранимой мощностью. Если в каждом узле имеется дополнительное поле, хранящее мощность поддерева, можно организовать вставку

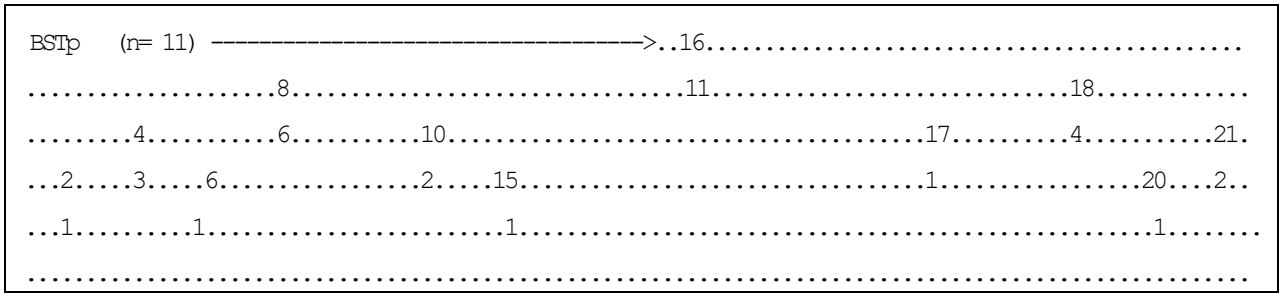



*Рис. 3.5.* ДДП с хранимой мощностью (под каждым узлом — мощность поддерева)

нового узла в корень или в случайное место, поддерживая тем самым относительную сбалансированность ДДП (рис. 3.5).

4. Красно-чёрные деревья. Узлы красятся в один из двух цветов — чёрный или красный. Если узел красный, его сыновья — обязательно чёрные. Вставляемый узел всегда красный. При появлении цепочки из двух красных узлов дерево перестраивается (рис. 3.6).

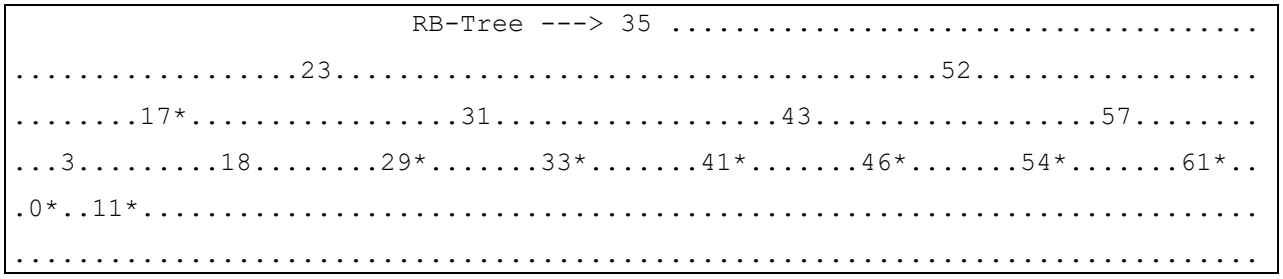



*Рис. 3.6.* Красно-чёрное дерево (красные узлы помечены *)

5. 2–3-деревья. Данные хранятся в листьях, над которыми делается надстройка из управляющих узлов, каждый из которых может иметь 2 или 3 сына и содержит наибольшие значения ключей в левом и в среднем поддеревьях, что необходимо для операции поиска. Если в результате вставки или удаления у управляющего узла оказывается 1 или 4 сына, дерево перестраивается (рис. 3.7).

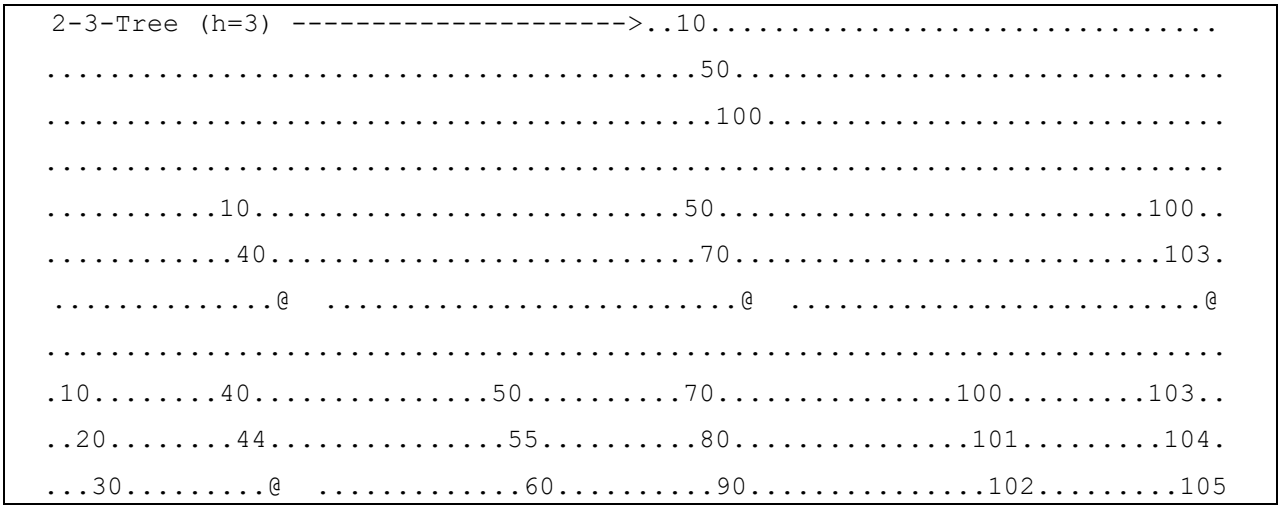



*Рис. 3.7.* 2–3-дерево (тройки из управляющих узлов; отсутствующие узлы помечены @)

6. 1–2-деревья. Узлы такого дерева объединяются в «горизонтальные группы» по два. При добавлении в группу третьего узла она разбивается на две подгруппы, выделяя корень, который передается вверх по иерархии, поддерживая тем самым общую сбалансированность дерева (рис. 3.8).

Подробнее о ДДП и деревьях с автобалансировкой — в [13, с. 457–468], [9, с. 341–344].

Двуместные операции над множествами в ДДП можно выполнять, используя примитивы проверка–вставка–удаление. Очевидно, что временная сложность двуместной операции при этом будет в среднем $O(n \log n)$. В худшем случае, при вырожденных деревьях, двуместная операция потребует $O(n^2)$ времени. Более того, если алгоритм порождает упорядоченную последовательность ключей, ДДП, полученное в результате последовательности вставок, всегда будет вырожденным.

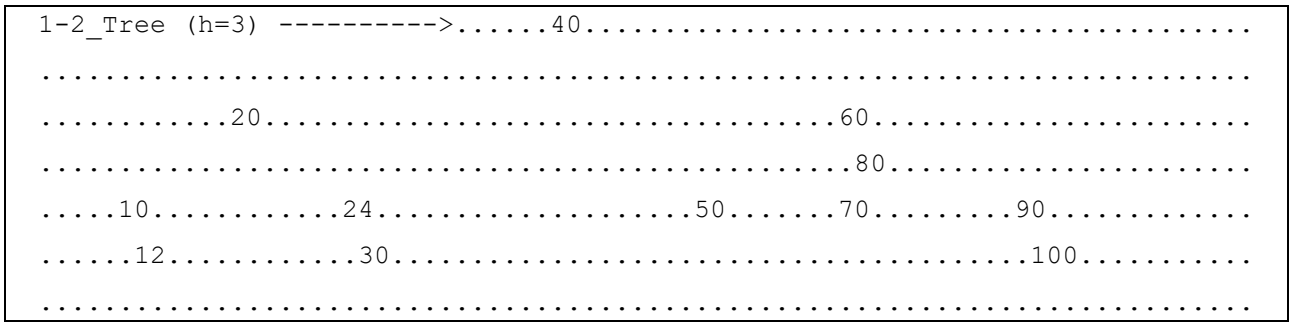



*Рис. 3.8.* 1–2-дерево (при подсчёте высоты дерева двойные узлы считаются за один)

Однако можно использовать тот факт, что из ДДП легко получить упорядоченную последовательность ключей внутренним обходом. Применив такой обход к двум ДДП одновременно, можно обработать последовательности алгоритмом слияния, модифицировав его для нужной операции над множествами. Результат в виде упорядоченной последовательности записывается в массив (вектор), из которого затем описанным ранее методом деления пополам строится новое ДДП. Этот способ обеспечивает получение результата за время $O(n)$ независимо от формы исходных ДДП, а результат всегда получается сбалансированным. Его недостаток — необходимость в буферной памяти. Без неё можно обойтись, если использовать деревья специального вида (с автобалансировкой и т. п.).

ДДП сами по себе мало пригодны для хранения множеств с повторениями, поскольку дубликаты ключей искажают форму дерева, образуя в нём мёртвые зоны: группы указателей, которые никогда не используются. Если дубликаты редки, проблемой можно пренебречь. Если же их может быть много, нужны специальные способы, применимость которых зависит от задачи. Так, вместо дубликатов ключей можно хранить в каждом узле дерева значение кратности (1 или больше). Если же дубликаты должны быть пред-

ставлены явно, их можно хранить в узлах как цепочки переполнения, по аналогии с хеш-таблицей.

### *3.2.1. Контрольные вопросы*

1. Каким способом следует разметить дерево, чтобы в нём был возможен двоичный поиск? Как его следует нагрузить?

2. Почему для операций с двоичным деревом дают две оценки сложности — «в худшем случае» и «в среднем»? Почему не рассматривается «лучший» случай?

3. Отчего деревья двоичного поиска вырождаются?

4. Можно ли воспрепятствовать вырождению ДДП?

5. Каков оптимальный алгоритм двуместной операции над множествами, представленными деревьями двоичного поиска?

6. Какова временная сложность такой операции и как сказывается на ней возможное вырождение деревьев?

7. Может ли при двуместной операции над множествами в ДДП получиться вырожденное дерево-результат?

8. Можно ли хранить в дереве двоичного поиска множество с повторениями?

9. Какая структура данных требует больше памяти для хранения множества: хеш-таблица или дерево двоичного поиска?

10. Какая из них быстрее работает?

11. Как сделать не вырождающееся ДДП? Зачем оно может понадобиться?

12. Какая структура данных является оптимальной для хранения дерева двоичного поиска?

## 3.3. Поддержка произвольной последовательности в структуре данных для множеств

К одной структуре данных могут применяться операции как для множества, так и для последовательности. Иногда требуется поддерживать в структуре данных для множеств произвольную последовательность элементов этих множеств. Например, можно фиксировать порядок появления элементов в множестве при его создании и работать с этим порядком.

Последовательность из структуры данных для множеств может быть получена как результат её обхода. Часто этого бывает достаточно: порядок элементов для результата операции над множествами можно назначить произ-

вольно. Однако для множества мощностью $n$ это будет только одна из $n!$ возможных последовательностей. Если нужно поддерживать любую последовательность, возможны следующие подходы:

1) присоединить к каждому ключу дополнительное поле для хранения порядкового номера. Способ не создаёт проблем при поиске номера по значению ключа и при вставке новых ключей, они просто нумеруются по порядку. Удаление ключа требует просмотра всей структуры данных для корректировки номеров, следующих за удаляемым. То же приходится делать при поиске ключа по порядковому номеру (сложность $O(n)$);

2) с помощью дополнительных указателей для каждого ключа сформировать из них список, возможно, двунаправленный. Проходом по этому списку можно как восстановить хранящуюся последовательность, так и получить номер для каждого ключа. Доступ к ключу по номеру и наоборот в этом случае имеет линейную сложность. Зато как вставка, так и удаление ключа требуют минимальных накладных расходов;

3) создать массив указателей на ключи. Если одновременно поддерживать в ключах дополнительное поле с обратным указателем на соответствующие элементы массива, можно избежать дополнительных расходов как для определения ключа по номеру, так и номера по ключу. Недостаток способа — необходимо заранее знать объём памяти для создания массива и перемещать часть массива в случае удаления ключей.

Операции над последовательностями, в отличие от операций с множествами, могут приводить к появлению дубликатов ключей. Структура данных для множеств должна обеспечивать соответствующую возможность.

Операции над последовательностями:

1. Слияние (*MERGE*). Объединение двух упорядоченных последовательностей в третью с сохранением упорядоченности. От операции объединения множеств отличается только возможностью появления дубликатов ключей. Если исходные последовательности не упорядочены, можно после их слияния просто упорядочить результат. Исходный порядок ключей в последовательностях в результате не сохраняется.

2. Сцепление (*CONCAT*). Вторая последовательность подсоединяется к концу первой, образуя её продолжение.

3. Размножение (*MUL*). Последовательность сцепляется сама с собой заданное количество раз.

4. Укорачивание (*ERASE*). Из последовательности исключается часть, ограниченная порядковыми номерами от $p1$ до $p2$.

5. Исключение (*EXCL*). Вторая последовательность исключается из первой, если она является её частью.

6. Включение (*SUBST*). Вторая последовательность включается в первую с указанной позиции $p$. Операция похожа на конкатенацию. Сперва берётся начало первой последовательности до позиции $p$, затем идёт вторая последовательность, а за ней — остаток первой.

7. Замена (*CHANGE*). Вторая последовательность заменяет элементы первой, начиная с заданной позиции $p$.

*Пример*. Пусть имеются две последовательности $A$ = <5, 3, 2, 4, 6, 7, 9, 1> и $B$ = <6, 7, 9>. Позиции считаются от 0.

Тогда операция *A.MERGE*(*B*) даст результат <1, 2, 3, 4, 5, 6, 6, 7, 7, 9, 9>;

*A.CONCAT*(*B*) — <5, 3, 2, 4, 6, 7, 9, 1, 6, 7, 9>;

*B.MUL*(3) — <6, 7, 9, 6, 7, 9, 6, 7, 9>;

*A.ERASE*(2, 4) — <5, 3, 7, 9, 1>;

*A.EXCL*(*B*) — <5, 3, 2, 4, 1>;

*A.SUBST*(*B,* 3) — <5, 3, 2, 6, 7, 9, 4, 6, 7, 9, 1>;

*A.CHANGE*(*B,* 2) — <5, 3, 6, 7, 9, 7, 9, 1>.

### *3.3.1. Контрольные вопросы*

1. Почему для хранения произвольной последовательности структуру данных для множества (хеш-таблицу или ДДП) приходится дорабатывать?

2. Какие доработки возможны?

3. Можно ли предложить оптимальный вариант доработки?

4. Влияет ли доработка структур данных для множеств для поддержки последовательностей на временную сложность операций над множествами?

5. Какую структуру данных проще дорабатывать — хеш-таблицу или ДДП?

6. Какова оптимальная доработка структуры данных и временная сложность для операции исключения части последовательности между указанными позициями?

7. То же — для операции вставки с указанной позиции.

8. То же — для замены.

### 3.4. Использование стандартной библиотеки шаблонов

Стандартная библиотека шаблонов (*STL*) поддерживает большинство типовых операций со структурами данных.

Мы уже использовали последовательные контейнеры *vector*, *list* и *deque* и их производные (адаптеры) *stack* и *queue*. В *STL* имеются ассоциативные контейнеры: *set* — для множеств, *map* — для отображений, *multiset*, *multimap* — для множеств и отображений с повторениями, основанные на деревьях двоичного поиска, и их аналоги *unordered_set, unordered_map*, *unordered_multiset* и *unordered_multimap* на базе хеш-таблиц. Для обработки данных используются возможности библиотеки алгоритмов (*algorithm*).

Каждому контейнеру соответствует заголовочный файл, который нужно подключать директивой #*include*.

Контейнеры *set* и *map* хранят множества в виде дерева двоичного поиска с автобалансировкой (красно-черное дерево). Контейнер *set* хранит множество ключей, а *map* — пары <ключ, значение>, причем все ключи в них уникальны. Для множеств с повторениями используются контейнеры *multimap* и *multiset*. При просмотре всех этих контейнеров их содержимое выдается в виде упорядоченной последовательности (внутренний обход дерева двоичного поиска).

При просмотре *unordered* контейнеров будет выдана неупорядоченная последовательность ключей.

Возможно много вариантов приспособления контейнеров для работы с последовательностями: использование *map* (или *multimap*) вместо *set*, чтобы хранить вместе с ключами их порядковые номера, комбинирование контейнера для множеств с контейнером последовательностей (*vector* или *forward_list*), хранящим итераторы, и т. п.

Конструкторы контейнеров позволяют уже при их объявлении сформировать множество заданной мощности. Для этого достаточно в качестве инициализатора содержимого контейнера использовать датчик случайных чисел.

Все необходимое для операций с контейнерами можно найти в библиотеке алгоритмов (*algorithm*). В частности, в ней имеются функции *set_union*, *set_intersection, set_difference, set_symmetric_difference,* вычисляющие объединение, пересечение, вычитание и симметрическую разность множеств. Функции принимают в качестве аргументов отрезки из двух контейнеров и формируют новый контейнер с результатом. В них реализуется схема слия-

ния, поэтому входные отрезки должны быть упорядочены. Это справедливо по умолчанию для контейнеров *set*, *map* и аналогичных. Для *unordered_set* аналогичные результаты дает одновременный просмотр двух контейнеров с применением функций проверки наличия и вставки элемента множества в результат. В библиотеке *STL* имеются функции для выполнения любых операций с последовательностями.

Подробнее — в [3, с. 295–368], [11, с. 835–962]. Полезно также посмотреть библиотечные файлы в каталоге *include* компилятора C++: только там содержится исчерпывающая информация о том, какие на самом деле объявляются классы и какие функции-члены они содержат. Информация в литературных источниках, как правило, запаздывает и содержит неточности. Важно и то, что в сообщениях компилятора об ошибках обычно присутствует информация из текстов каталога *include*, поскольку эти тексты компилируются вместе с программой пользователя.

## 3.5. Превращение в контейнер пользовательской структуры данных

Пользовательскую структуру данных для хранения множества/последовательности можно превратить в подобие библиотечного контейнера и тем самым обеспечить возможность применения к нему стандартных алгоритмов библиотеки *algorithm*. Для этого достаточно соблюдать соглашения о кодировании, принятые для контейнеров: объявить классы итераторов для просмотра и вставки и функции для их инициализации и контроля.

Можно ограничиться только теми средствами, которые действительно понадобятся для работы с пользовательским контейнером.

Для просмотра контейнера в цикле необходимы и достаточны функции *begin*( ) и *end*( ), возвращающие прямой итератор чтения, который указывает на первый элемент контейнера и элемент «сразу за последним» соответственно. Итератор должен поддерживать операцию разыменования для доступа к элементам контейнера, операцию инкремента для перемещения по контейнеру и операцию сравнения с результатом *end*( ). Для вставки нового значения необходим итератор вставки и средство для его инициализации (инсертер). Итератор вставки должен обеспечивать вставку нового элемента одновременно и в множество, и в последовательность.

Наличие этих средств позволит, например, стандартным образом получить копию не только стандартного, но и пользовательского контейнера *A*

в произвольном контейнере *B*:

```
std::copy(A.begin( ), A.end( ), back_inserter(B));
```

**Пример** объявления итераторов для пользовательского контейнера (хеш-таблица из массива *bct*, содержащего указатели на цепочки переполнения):

```
#include <iterator>
using namespace std;
//ИТЕРАТОР ЧТЕНИЯ — нужны сравнения, разыменования, инкремент
struct myiter : public std::iterator<std::forward_iterator_tag, int>
{ //В качестве базы использован стандартный прямой итератор
   myiter(Node *p) : bct(nullptr), pos(0), Ptr(p) {}
   bool operator == (const myiter & Other) const { return Ptr == Other.Ptr; }
   bool operator != (const myiter & Other) const { return Ptr != Other.Ptr; }
   myiter operator++(); //Ключевая операция — инкремент по контейнеру
   myiter operator++(int) { myiter temp(*this); ++*this; return temp; }
   pointer operator->() { return & Ptr->key; } //Разыменование косвенное
   reference operator*() { return Ptr->key; } //Разыменование прямое
   //protected:
//  Container& c;
   Node **bct; //Указатель на хеш-таблицу (массив экстентов)
   size_t pos; //Номер текущего экстента
   Node * Ptr; //Реальный указатель на элемент контейнера
};
//ИТЕРАТОР ВСТАВКИ — нужно только присваивание!
template <typename Container, typename Iter = myiter>
class outiter : public std::iterator<std::output_iterator_tag, typename Contain-
er::value_type>
{
protected:
   Container& container;    // Контейнер для вставки элементов
   Iter iter;        // Текущее значение итератора чтения
public:
   // Конструктор
   explicit outiter(Container& c, Iter it) : container(c), iter(it) { }
   // Присваивание = вставка ключа в контейнер
   const outiter<Container>&
            operator = (const typename Container::value_type& value) {
            iter = container.insert(value, iter).first;
            return *this;
   }
```

```
    const outiter<Container>& //Присваивание копии — фиктивное
            operator = (const outiter<Container>&) { return *this; }
    // Разыменование — пустая операция
    outiter<Container>& operator* ( ) { return *this; }
    // Инкремент — пустая операция
    outiter<Container>& operator++ ( ) { return *this; } //префиксный
    outiter<Container>& operator++ (int) { return *this; } //постфиксный
};
```

// ИНСЕРТЕР — функция для создания итератора вставки — аргумент для алгоритма, создающего новый контейнер (универсальная)

```
template <typename Container, typename Iter>
inline outiter<Container, Iter> outinserter(Container& c, Iter it)
{  return outiter<Container, Iter>(c, it); }

//Функции, превращающие хеш-таблицу в контейнер
myiter HT::begin( )const { //Получение итератора на начало
    myiter p(nullptr); //Инициализация итератора значением «на конец»
    p.bct = bucket; //Установка на массив экстентов хеш-таблицы
    for (; p.pos < Buckets; ++p.pos) { //Поиск первого элемента в XT
            p.Ptr = bucket[p.pos];
            if (p.Ptr) break; //Нашли!
    }
    return p;
}

myiter HT::end( )const { return myiter(nullptr); } //Итератор на конец

myiter myiter::operator++( ) //Инкремент итератора: пример для XT
{
    if (!Ptr) { //Инициализация сделана?
        return *this; //Не работает без предварительной установки на XT
    }
    else
    { //Текущий уже выдан
            if(Ptr->down) { //Есть следующий, вниз
                    Ptr = Ptr->down;
                    return (*this);
            }
             while(++pos < HT::Buckets)       //Поиск очередного элемента
                    if(Ptr = bct[pos]) return *this; //Нашли — выход
             }
```

```
            Ptr = nullptr; //Не нашли — выход с пустым итератором
            return *this;
        }
    }
```

Детальная информация о пользовательских итераторах уровня стандарта C++17 — в [2], [3] и [10]. Справку о текущем состоянии вопроса можно получить в Интернете на сайте ru.cppreference.com.

## 3.6. Практикум по гл. 3

Реализовать индивидуальное задание темы «Множества + последовательности» в виде программы, используя свой контейнер для заданной структуры данных (хеш-таблицы или одного из вариантов ДДП), и доработать его для поддержки операций с последовательностями. Для операций с контейнером рекомендуется использовать возможности библиотеки алгоритмов. Программа должна реализовывать цепочку операций над множествами, имеющимися в выражении, взятом по номеру варианта задания из табл. 3.1 с базовым контейнером и операциями с последовательностью из табл. 3.2. Результат каждого шага цепочки операций выводится на экран.

Реализация каждой операции должна обеспечивать расширенные гарантии устойчивости к исключениям.

### *3.6.1. Дополнительные требования к отчету*

В отчете опишите свой контейнер и функции *STL,* использованные для работы с ними. Приведите результат выполнения цепочки операций для случайного набора данных заданной мощности. Оцените предполагаемую временную сложность выполнения цепочки операций.

### *3.6.2. Контрольные вопросы*

1. Что такое стандартный контейнер библиотеки *STL*? Чем он отличается от обычного объекта?

2. Какой стандартный контейнер можно считать наиболее подходящим для работы с множествами?

3. Можно ли использовать стандартные контейнеры для множеств, на которых не определено отношение полного порядка?

4. Существуют ли ограничения на применение стандартных алгоритмов двуместных операций над множествами в контейнерах?

5. Можно ли реализовать двуместную операцию над множествами

в контейнерах без применения стандартного алгоритма?

6. Можно ли выполнять операции над последовательностями для множеств, хранящихся в стандартном контейнере?

7. Можно ли обеспечить поддержку произвольных последовательностей в контейнере для множеств?

8. Какова ожидаемая временная сложность при выполнении стандартным алгоритмом операции объединения двух множеств в стандартных контейнерах *set*?

9. С какой целью может понадобиться оформить пользовательскую структуру данных как стандартный контейнер?

10. Каков минимально необходимый набор средств для превращения пользовательской структуры данных в полноценный контейнер?

11. Зачем нужны гарантии устойчивости алгоритмов относительно исключений?

12. Почему базовых гарантий устойчивости алгоритма к исключениям может быть недостаточно?

13. В чём заключаются расширенные гарантии устойчивости алгоритма к исключениям и как их можно обеспечить?

*Таблица 3.1*

***Индивидуальные задания к практикуму по гл. 3.. Операции над множествами***

| № вари-анта | Мощность множества | Что надо вычислить | № вари-анта | Мощность множества | Что надо вычислить |
|---|---|---|---|---|---|
| 1 | 10 | A ∩ B ∪ C ∪ (D ⊕ E) | 26 | 26 | A \ (B ∩ C ∩ D) ⊕ E |
| 2 | 26 | (A \ B \ C) ⊕ D ∪ E | 27 | 16 | (A ∪ B ⊕ C ∩ D) ∩ E |
| 3 | 16 | A ∩ B ⊕ C ∩ D ∩ E | 28 | 26 | A ⊕ (B ∪ C ∪ D) ∩ E |
| 4 | 26 | (A ⊕ B \ C) ∪ D ∩ E | 29 | 10 | (A \ B) ∪ C ⊕ D ∪ E |
| 5 | 10 | A ⊕ B \ (C ∪ D) \ E | 30 | 32 | A ∪ B ∩ C ∪ D ⊕ E |
| 6 | 32 | (A ∩ B) ⊕ C ∪ D \ E | 31 | 16 | (A ⊕ B) ∪ (C \ D) \ E |
| 7 | 16 | A ∪ B ∪ C ⊕ D ∩ E | 32 | 32 | (A ∪ B ∪ C) \ D ⊕ E |
| 8 | 26 | A \ (B ∩ C) ∪ D ⊕ E | 33 | 10 | (A ∩ B) ⊕ (C ∪ D) \ E |
| 9 | 10 | A ⊕ B ∩ C ∪ D ∪ E | 34 | 26 | A \ B \ (C ⊕ D) \ E |
| 10 | 26 | ((A ∪ B) \ C) ∩ D ⊕ E | 35 | 16 | (A ∪ B) \ (C ∪ D) ⊕ E |
| 11 | 16 | A ∪ B ⊕ C \ D \ E | 36 | 32 | (A ⊕ B ∩ C) \ D ∩ E |
| 12 | 26 | A ∪ B ∪ C ⊕ D \ E | 37 | 10 | A \ (B ∩ C ∪ D) ⊕ E |
| 13 | 10 | A ⊕ B \ C \ D ∩ E | 38 | 32 | (A ∪ B ⊕ C) \ D ∩ E |
| 14 | 32 | A \ B ⊕ (C \ D \ E) | 39 | 16 | A ∩ B ∪ C ∩ D ⊕ E |
| 15 | 16 | (A ∪ B) \ C \ (D ∪ E) | 40 | 52 | (A \ B) ∪ C ⊕ D ∩ E |
| 16 | 32 | (A ⊕ B ∩ C) \ D ∩ E | 41 | 10 | A ⊕ B ∪ C ∩ D ∪ E |
| 17 | 10 | A \ (B ∩ C) \ D ⊕ E | 42 | 66 | (A ∩ B) \ (C ∩ D) ⊕ E |
| 18 | 32 | A ∪ (B ⊕ C) \ D ∩ E | 43 | 16 | A \ (B ⊕ C ∩ D) ∪ E |
| 19 | 16 | A ∩ B ⊕ C ∩ D ∪ E | 44 | 52 | (A ∪ B) ⊕ (C ∩ D) \ E |
| 20 | 26 | (A \ B) ∪ C ∩ D ⊕ E | 45 | 10 | A ∩ B ∩ C ⊕ D ∩ E |
| 21 | 10 | A ⊕ B ∪ C ∩ D ∪ E | 46 | 26 | A \ (B ∩ C ∩ D) ⊕ E |
| 22 | 32 | (A ∩ B) \ (C ∩ D) ⊕ E | 47 | 16 | A ∪ B ⊕ C ∩ D ∩ E |
| 23 | 16 | A ⊕ B \ C ∩ D \ E | 48 | 26 | A ∩ (B ∪ C ∪ D) ⊕ E |
| 24 | 32 | A ∪ B ⊕ (C ∩ D \ E) | 49 | 10 | (A \ B) ⊕ C ∪ D ∪ E |
| 25 | 10 | A ∩ B ⊕ C ∩ D ∩ E | 50 | 40 | A ∪ B ∩ C ∪ D ⊕ E |

*Таблица 3.2*

***Индивидуальные задания к практикуму по гл. 3.***
***Базовая структура данных и операции над последовательностями***

| № вари-анта | Базо-вая СД | Дополнительные операции | № вари-анта | Базо-вая СД | Дополнительные операции |
|---|---|---|---|---|---|
| 1 | ДДПв | MERGE, EXCL, CHANGE | 18 | АВЛд | CONCAT, EXCL, SUBST |
| 2 | АВЛд | CONCAT, EXCL, MUL | 19 | К-ч-д | MERGE, EXCL, SUBST |
| 3 | 1-2д | EXCL, SUBST, MUL | 20 | ДДПм | CONCAT, SUBST, CHANGE |
| 4 | 2-3д | MERGE, ERASE, SUBST | 21 | 1-2д | MERGE, CONCAT, CHANGE |
| 5 | ДДПм | MERGE, CONCAT, SUBST | 22 | 2-3д | MERGE, CONCAT, MUL |
| 6 | К-ч-д | MERGE, EXCL, SUBST | 23 | ДДП | EXCL, SUBST, CHANGE |
| 7 | ДДП | CONCAT, EXCL, SUBST | 24 | ХТ | ERASE, EXCL, MUL |
| 8 | ХТ | MERGE, ERASE, EXCL | 25 | ДДПв | MERGE, CONCAT, SUBST |
| 9 | ДДП | MERGE, CONCAT, EXCL | 26 | АВЛд | ERASE, EXCL, CHANGE |
| 10 | ДДПм | CONCAT, ERASE, CHANGE | 27 | 1-2д | CONCAT, ERASE, EXCL |
| 11 | 1-2д | CONCAT, SUBST, CHANGE | 28 | ДДПм | EXCL, SUBST, MUL |
| 12 | 2-3д | ERASE, EXCL, SUBST | 29 | ХТ | CONCAT, EXCL, CHANGE |
| 13 | АВЛд | CONCAT, EXCL, SUBST | 30 | К-ч-д | MERGE, CONCAT, EXCL |
| 14 | ХТ | CONCAT, EXCL, CHANGE | 31 | 2-3д | MERGE, EXCL, CHANGE |
| 15 | К-ч-д | MERGE, CONCAT, MUL | 32 | ДДП | CONCAT, EXCL, MUL |
| 16 | ДДПв | ERASE, SUBST, CHANGE | 33 | ХТ | MERGE, CONCAT, ERASE |
| 17 | АВЛд | MERGE, SUBST, MUL | 34 | ДДП | CONCAT, EXCL, SUBST |

*Окончание табл. 3.2*

| № варианта | Базовая СД | Дополнительные операции | № варианта | Базовая СД | Дополнительные операции |
|---|---|---|---|---|---|
| 35 | ДДПв | MERGE, SUBST, CHANGE | 43 | 2-3д | ERASE, EXCL, SUBST |
| 36 | ХТ | CONCAT, ERASE, EXCL | 44 | АВЛд | CONCAT, ERASE, CHANGE |
| 37 | 2-3д | MERGE, CONCAT, CHANGE | 45 | ДДПм | MERGE, ERASE, SUBST |
| 38 | ДДПм | EXCL, SUBST, CHANGE | 46 | ДДПв | MERGE, SUBST, MUL |
| 39 | ДДП | ERASE, EXCL, CHANGE | 47 | 1-2д | MERGE, ERASE, CHANGE |
| 40 | 1-2д | MERGE, CONCAT, ERASE | 48 | К-ч-д | MERGE, SUBST, CHANGE |
| 41 | К-ч-д | ERASE, EXCL, MUL | 49 | ДДП | ERASE, SUBST, CHANGE |
| 42 | ДДПв | MERGE, ERASE, CHANGE | 50 | К-ч-д | MERGE, ERASE, EXCL |

*Примечание*: В таблице использованы следующие обозначения: ХТ — хеш-таблица; ДДП — дерево двоичного поиска (без автобалансировки); ДДПв — то же, с хранением в каждом узле высоты поддерева; ДДПм — то же, с хранением мощности поддерева; АВЛд — АВЛ-дерево (с автобалансировкой); К-ч-д — красно-чёрное дерево (с автобалансировкой); 2–3д — 2–3-дерево (всегда сбалансированное); 1-2д — 1-2-дерево.

Итератор чтения для просмотра структуры данных или её части можно реализовать для всех перечисленных выше структур данных. Итератор вставки, обеспечивающий добавление ключей в множество, может без проблем быть реализован только для хеш-таблицы. Указание места вставки для неё игнорируется как совершенно излишнее. Для ДДП вставка без указания места начала поиска может быть выполнена только за логарифмическое время, поскольку корректный поиск места вставки в общем случае должен начинаться от корня дерева. Вставка за константное время возможна, только если итератор вставки укажет для начала поиска места вставки на ключ, вставленный последним. В случае вставки произвольного ключа такое указание при-

ведет к хаосу. Оно допустимо и целесообразно только для вставки упорядоченной последовательности ключей в пустое дерево, как это происходит в двуместных операциях с множествами по схеме слияния. Чтобы форма ДДП такими вставками не искажалась, оно должно быть автобалансирующимся. Чтобы обеспечить двуместную операцию за линейное время с обычным ДДП — без автобалансировки, необходимо помещать результат операции в промежуточный буфер (вектор) и затем восстанавливать дерево из него.

ДДП с хранением в узлах высоты работает аналогично самобалансирующемуся АВЛ-дереву: балансы вычисляются динамически сравнением высот поддеревьев.

Для деревьев с автобалансировкой итератор вставки обеспечивает в качестве точки начала очередного поиска узел, на котором балансировка закончилась. Для этого в нём хранится стек с путем от корня к точке вставки.

ДДП с хранением в узлах мощности поддеревьев само по себе не имеет преимуществ перед обычным ДДП при случайных вставках. Но для такого дерева существует алгоритм вставки в корень. После создания, как обычно, нового узла — листа он с помощью серии вращений пар отец-сын перемешается в корень дерева. При двуместных операциях итератор вставки может выполнять вставку нового узла сразу в  корень (за константное время). Дерево получается вырожденным, но по завершении двуместной операции может быть сбалансировано за линейное время, как и обычное ДДП, но без использования дополнительной памяти.

Для 2–3-дерева возможна как схема с автобалансировкой после каждой вставки, так и получение промежуточного списка упорядоченных узлов, из которого по завершении двуместной операции дерево восстанавливается за линейное время без использования дополнительной памяти. Узлы дерева при балансировке не перемещаются, и итераторы на них остаются действительны.

Оптимальный способ дополнения дерева двоичного поиска для обеспечения операций с последовательностями зависит от реализуемого набора таких операций. Например, если требуется доступ к порядковому номеру по значению ключа, оптимальным является хранение номеров в узлах дерева вместе с ключами. Но это означает, что ДДП должно обеспечивать хранение дубликатов ключей, которые могут появиться как результат операции с последовательностью. Снять проблему дубликатов можно присоединением номеров к ключам в операции сравнения. Но такой подход сделает невозмож-

ным оперативную замену номеров: ключ со старым номером придется удалять и вставлять заново с новым — за логарифмическое время.

Реализация последовательности в виде вектора итераторов на узлы дерева обеспечивает доступ к ключу по порядковому номеру в последовательности за константное время, а обратную операцию — за линейное. Обход дерева даёт упорядоченную последовательность ключей, а обход вектора — произвольную, что позволяет работать со структурой данных и как с множеством, и как с последовательностью.

Способ обхода выбирается таким образом, чтобы временная сложность двуместной операции получалась линейной. Это легко сделать для хеш-таблицы. Для дерева двоичного поиска следует избегать последовательности случайных вставок с поиском места вставки от корня дерева, поскольку сложность такой последовательности будет $O(n \log n)$.

# 4. КУРСОВАЯ РАБОТА. «ИЗМЕРЕНИЕ ВРЕМЕННОЙ СЛОЖНОСТИ АЛГОРИТМА В ЭКСПЕРИМЕНТЕ НА ЭВМ»

На основе программы, составленной по гл. 3, выполнить статистический эксперимент по измерению фактической временной сложности алгоритма обработки данных.

Программа дорабатывается таким образом, чтобы она генерировала множества мощностью, меняющейся, например, от 10 до 200, измеряла время выполнения цепочки операций над множествами и последовательностями и выводила результат в текстовый файл. Каждая строка этого файла должна содержать пару значений «размер входа — время» для каждого опыта. Затем эти данные обрабатываются, и по результатам обработки делается заключение о временной сложности алгоритма.

Для повышения надежности эксперимента следует предусмотреть в программе перехват исключительных ситуаций. Можно сделать так, чтобы любой сбой сводился просто к пропуску очередного шага эксперимента. В частности, рекомендуется перехватывать ситуацию *bad_alloc*, возбуждаемую конструктором при нехватке памяти.

## 4.1. Пример программы для эксперимента

Далее приводится пример программы для тестирования цепочки операций с заданной структурой данных. В качестве базовой структуры данных для множества ключей использован контейнер *set* (ДДП). Поддержка последовательностей обеспечивается вектором итераторов на ключи в *set*. Если заменить *set* на *unordered_set*, программа будет работать с хеш-таблицей.

Взяв эту программу за основу, необходимо подставить в нее свой контейнер и свою поддержку последовательностей, выбросить ненужные операции, а для нужных обеспечить оптимальные алгоритмы. После отладки нужно сделать цикл для перебора значений мощности *p*, отключить отладочный вывод (*debug=false*), провести статистический эксперимент и обработать его результаты.

```
// demo1STL.cpp: упражнения с ДДП/вектором указателей.
//(c)lgn, 28.04/11.05.16/27.01.17/05.02.18/05.02.19
#include "pcb.h"
#include <iostream>
#include <algorithm>
#include <set>
```

```
#include <ctime>
#include <iterator>
#include <chrono>
#include <vector>

using  MySet = std::set<int>;
using MyIt  = std::set<int>::iterator;
using MySeq = std::vector<MyIt>;

const int lim = 1000; //ОГРАНИЧИТЕЛЬ для множества ключей
class MyCont {
   int power;
   char tag;
   MySet A;
   MySeq sA;
   MyCont& operator = (const MyCont &) = delete;
   MyCont& operator = (MyCont &&) = delete;
public:
   MyCont ( int, char );
   MyCont (const MyCont &);
   MyCont (MyCont &&);
   MyCont& operator |= (const MyCont &);
   MyCont operator | (const MyCont & rgt) const
      { MyCont result(*this); return (result |= rgt); }
   MyCont& operator &= (const MyCont &);
   MyCont operator & (const MyCont & rgt) const
      { MyCont result(*this); return (result &= rgt); }
   MyCont& operator -= (const MyCont &);
   MyCont operator - (const MyCont & rgt) const
      { MyCont result(*this); return (result -= rgt); }
   void Merge (const MyCont &);
   void Concat (const MyCont &);
   void Mul (int);
   void Erase (size_t, size_t);
   void Excl (const MyCont &);
   void Subst (const MyCont &, size_t);
   void Change (const MyCont &, size_t);
   void Show ( ) const;
   size_t Power( ) const { return sA.size( ); }
   void PrepareExcl(const MyCont& );        // подготовка excl
friend     void PrepareAnd
```

```
      (MyCont &, MyCont&, const int);         // подготовка and и sub
};

MyCont::MyCont( int p = 0, char t = 'R') : power(p), tag(t)
 {  for(int i = 0; i < power; ++i)
     { sA.push_back(A.insert(std::rand()%lim).first); }
 }

MyCont::MyCont (MyCont && source)   //Копия "с переносом"
   : power(source.power), tag(source.tag),
     A(std::move(source.A)), sA(std::move(source.sA)) { }

MyCont::MyCont (const MyCont & source) //Конструктор копии
   : power(source.power), tag(source.tag) {
   for (auto x : source.A) sA.push_back(A.insert(x).first);
}

void MyCont::Show( ) const {
   using std::cout;
   cout << "\n" << tag << ": ";
/* unsigned n = A.bucket_count( );  //Вариант: выдача для XT
   for (auto i = 0; i < n; ++i)
      if (A.bucket_size(i))
           {
                cout << "\n" <<  i << "("<< A.bucket_size(i) << "):" ;
   //           auto it0 = A.begin(i), it1 = A.end(i);
                for (auto it = A.begin(i); it != A.end(i); ++it) cout << " " << *it;
   } */
   for(auto x : A) cout << x << " ";    //Выдача множества
   cout << "\n < ";
   for(auto x : sA) cout << *x << " "; //Выдача последовательности
   cout << ">";
}

void PrepareAnd(MyCont & first, MyCont& second, const int quantity) {
   for (int i = 0; i < quantity; ++i) { //Подготовка пересечения:
           int x = rand( )%lim;           // добавление общих эл-тов
           first.sA.push_back(first.A.insert(x));
           second.sA.push_back(second.A.insert(x).first);
   }
}
```

```
MyCont& MyCont::operator -= (const MyCont & rgt){ //Разность мн-в
   MySet temp;
   MySeq stemp;
   for (auto x : A)
          if(rgt.A.find(x) == rgt.A.end( ))
                stemp.push_back(temp.insert(x).first);
   temp.swap(A);
   stemp.swap(sA);
   return *this;
}

MyCont& MyCont::operator &= (const MyCont & rgt){ //Пересечение
   MySet temp;
   MySeq stemp;
   for (auto x : A) if(rgt.A.find(x) != rgt.A.end( ))
       stemp.push_back(temp.insert(x).first);
   temp.swap(A);
   stemp.swap(sA);
   return *this;
}

MyCont& MyCont::operator |= (const MyCont & rgt) { //Объединение
   for (auto x : rgt.A) sA.push_back(A.insert(x).first);
   return *this;
}

void MyCont::Erase (size_t p, size_t q) { //Исключение фр-та от p до q
   using std::min;
   size_t r(Power( ));
   p = min(p, r); q = min(q+1, r);
   if(p <= q) {
          MySet temp;
          MySeq stemp;
          for(size_t i = 0; i < p; ++i)
                stemp.push_back(temp.insert(*sA[i]).first);
          for(size_t i = q; i < r; ++i)
                stemp.push_back(temp.insert(*sA[i]).first);
          A.swap(temp);
          sA.swap(stemp);
   }
```

```
}

void MyCont::Mul(int k) { //Размножение (не более чем в 5 раз)
   auto p = sA.begin( ), q = sA.end( );
   if(p != q && (k = k%5) > 1) { //Пропуск, если мн-во пусто или k < 2
         std::vector<int> temp(A.begin( ), A.end( ));
      MySeq res(sA);
      for(int i = 0; i < k-1; ++i) {
         std::copy(p, q, back_inserter(res));
         A.insert(temp.begin( ), temp.end( ));
      }
      sA.swap(res);
   }
}

void MyCont::Merge(const MyCont & rgt) { //Слияние
   using std::sort;
   MySeq temp(rgt.sA), res;
   auto le = [ ] (MyIt a, MyIt b)->bool { return *a < *b; };//Критерий
   sort(sA.begin( ), sA.end( ), le);
   sort(temp.begin( ), temp.end( ), le);
   std::merge(sA.begin( ), sA.end( ), temp.begin( ), temp.end( ),
      std::back_inserter(res), le); //Слияние для последовательностей...
   A.insert(rgt.A.begin( ), rgt.A.end( )); //... и объединение множеств
   sA.swap(res);
}

void MyCont::PrepareExcl( const MyCont& rgt ) {
  //Подготовка объекта исключения в пустом контейнере...
   int a = rand( )%rgt.Power( ), b = rand( )%rgt.Power( );
  //... из случайного [a, b] отрезка rgt
   if (b>a) {
         for (int x = a; x <= b; ++x)  {
               int y =*(rgt.sA[x]); sA.push_back(A.insert(y).first);
         }
   }
}

void MyCont::Excl (const MyCont & rgt)
{ //Исключение подпоследовательности
   size_t n(Power( )), m(rgt.Power( ));
```

```
   if(m) for (size_t p = 0; p < n; ++p) { //Поиск первого элемента
            bool f(true);
//          int a(*sA[p]), b(*rgt.sA[0]); //ОТЛАДКА
            if(*sA[p] == *rgt.sA[0]) { //Проверка всей цепочки
                  size_t q(p), r(0);
                  if (m > 1) do {
                        ++q, ++r;
                        size_t c(*sA[q]), d(*rgt.sA[r]);
                        f &= c == d;
                  } while ((r<m-1) && f);
                  if(f) {//Цепочки совпали, удаляем
                        MySet temp;
                        MySeq stemp;
                        for(size_t i = 0; i < p; ++i)
                  stemp.push_back(temp.insert(*sA[i]).first);
                        for(size_t i = p+m; i < Power( ); ++i)
                  stemp.push_back(temp.insert(*sA[i]).first);
                        A.swap(temp);
                        sA.swap(stemp);
                        break;
   } } } }
void MyCont::Concat(const MyCont & rgt) { //Сцепление
      for(auto x : rgt.sA) sA.push_back(A.insert(*x).first);
}

void MyCont::Subst (const MyCont & rgt, size_t p)
{ //Подстановка
   if(p >= Power( )) Concat(rgt);
   else {
            MySeq stemp(sA.begin( ), sA.begin( ) + p);      //Начало
            std::copy(rgt.sA.begin( ), rgt.sA.end( ), back_inserter(stemp)); //Вставка
            std::copy(sA.begin( )+p, sA.end( ), back_inserter(stemp)); //Окончание
            MySet temp;
            sA.clear( );
            for (auto x : stemp) sA.push_back(temp.insert(*x).first);
            A.swap(temp);
   }
}
void MyCont::Change (const MyCont & rgt, size_t p)
{ //Замена
   if(p >= Power( )) Concat(rgt);
```

```
    else {
            MySeq stemp(sA.begin( ), sA.begin( ) + p);      //Начало
            std::copy(rgt.sA.begin( ), rgt.sA.end( ), back_inserter(stemp));
 //Замена
            size_t q = p + rgt.Power( );
            if (q < Power( ))
              std::copy(sA.begin( )+q, sA.end( ), back_inserter(stemp));
//Окончание
            MySet temp;
            sA.clear( );
            for (auto x : stemp) sA.push_back(temp.insert(*x).first);
            A.swap(temp);
    }
}

int main( )
{  using std::cout;
   using namespace std::chrono;
    setlocale(LC_ALL, "Russian");
   srand((unsigned int)7); //Пока здесь константа, данные повторяются
//    srand((unsigned int)time(nullptr)); //Разблокировать для случайных данных
    bool debug = true; //false, чтобы запретить отладочный вывод
    auto MaxMul = 5;
    int middle_power = 0, set_count = 0;
    auto Used = [&] (MyCont & t){ middle_power += t.Power( );
   ++set_count; };
    auto DebOut = [debug] (MyCont & t) { if(debug) { t.Show( ); system("Pause");}};
    auto rand = [ ] (int d) { return std::rand( )%d; }; //Лямбда-функция!
    ofstream fout("in.txt"); //Открытие файла для результатов
    int p = rand(20) + 1; //Текущая мощность (место для цикла по p)
    //=== Данные ===
    MyCont A(p, 'A');
    MyCont B(p, 'B');
    MyCont C(p, 'C');
    MyCont D(p, 'D');
    MyCont E(p, 'E');
    MyCont F(0, 'F');   //Пустая заготовка для Excl
    MyCont G(p, 'G');
    MyCont H(p, 'H');
    MyCont R(p);
    int q_and(rand(MaxMul) + 1);
```

```
    PrepareAnd(A, R, q_and);
    if (debug) A.Show( ); Used(A);
    if (debug) R.Show( ); Used(R);
//=== Цепочка операций ===
// (Операция пропускается (skipped!), если аргументы некорректны)
//Идет суммирование мощностей множеств и подсчет их количества,
// измеряется время выполнения цепочки
    auto t1 = std::chrono::high_resolution_clock::now( );
    if (debug) cout << "\n=== R&=A ===(" << q_and << ") ";
    R&=A;   DebOut(R); Used(R);

    if (debug) B.Show( ); Used(B);
    if (debug) cout << "\n=== R|=B ===";
    R|=B;  DebOut(R); Used(R);

    int e = rand(R.Power( ));
    if (debug) cout << "\n=== R.Change (H, " << e << ") ===";
    H.Show( ); Used(H);
    R.Change(H, e);   DebOut(R);     Used(R);

    int q_sub(rand(MaxMul) + 1);
    PrepareAnd(C, R, q_sub);
        if (debug) R.Show( ), C.Show( ); middle_power += q_sub; Used(C);
    if (debug) cout << "\n=== R-=C ===(" << q_sub << ") ";
    R-=C;  DebOut(R); Used(R);

    int a = rand(R.Power( )), b = rand(R.Power( ));
    if (debug) cout << "\n=== R.Erase (" << a << "," << b << ")===";
    if (a>b) cout << "(skipped!)";
    R.Erase(a, b);   DebOut(R);         Used(R);

    if (debug) cout << "\n=== R.Concat(D) ===";
    D.Show( ); Used(D);
    R.Concat(D);   DebOut(R);Used(R);

    if (debug) cout << "\n=== R.Merge(E) ===";
    E.Show( ); Used(E);
    R.Merge(E);   DebOut(R); Used(R);

    if (debug) cout << "\n=== R.Excl(F) ===";
    F.PrepareExcl(R);
    if(debug && !F.Power( )) cout << "(skipped)!";
```

```
    F.Show( );  Used(F);
    R.Excl(F);   DebOut(R);     Used(R);

    int d = rand(R.Power( ));
    if (debug) cout << "\n=== R.Subst (G, " << d << ") ===";
    G.Show( ); Used(G);
    R.Subst(G, d);   DebOut(R);        Used(R);

    int c = rand(MaxMul);
    if (debug) cout << "\n=== R.Mul(" << c << ")===";
    if (c < 2) cout << "(skipped!)";
    R.Mul(c);   DebOut(R);      Used(R);
            auto t2 = std::chrono::high_resolution_clock::now( );
            auto dt = duration_cast<duration<double>>(t2-t1);
    middle_power /= set_count;
fout << p << '  ' << dt.count() << endl; //Выдача в файл
    cout << "\n=== Конец === (" << p << " : " << set_count << " * " <<
       middle_power << " DT=" <<      (dt.count()) <<")\n";
    cin.get( );
   return 0;
}
```

### 4.2. Обработка результатов эксперимента

По результатам эксперимента выполняется регрессионный анализ: подбор уравнения регрессии, наиболее соответствующего полученным данным. Для выполнения этого пункта следует получить у преподавателя набор данных «*stat*», состоящий из программы *RG*41, файлов с примерами ее входа и выхода (*in.txt, out.txt*) и заготовки электронной таблицы *EXAMPLE*.

Программа *RG*41 представляет собой консольное *Windows*-приложение и запускается из-под интерфейса командной строки (*CMD*). В качестве аргумента программа принимает имя файла с результатами измерений. Имя и расширение этого файла могут быть любыми. Рекомендуется поместить программу и файл измерений в один каталог. Если файл назван «*IN.txt*», строка запуска будет выглядеть таким образом:

```
RG41 in.txt
```

В программе *RG*41 имя входа «*in.txt*» принято по умолчанию и его можно опустить. В этом случае программу можно запускать и в окне проводника.

Каждая из строк содержит результат одного опыта: размер входа и время, разделенные хотя бы одним пробелом. Упорядочивать данные по размеру

входа необязательно. Можно выполнить по несколько опытов для некоторых или для всех размеров входа. Программа подсчитает и использует фактически имеющееся количество результатов. Она откажется работать только в случае, если опытов менее 10.

Программа *RG*41 выдаст на экран значения коэффициентов регрессии для набора функций, начиная от константы (отсутствие регрессии) и кончая полиномом четвертой степени. Общий вид уравнения регрессии выводится программой в первой строке. Далее выводятся результаты подбора с поочередным подключением коэффициентов слева направо. Не используемые в уравнении коэффициенты выводятся нулями. Кроме коэффициентов дается значение выборочной дисперсии (и среднеквадратичного отклонения — СКО). Последняя колонка таблицы коэффициентов — оценочный код, вычисленных программой по результату сравнения отношения дисперсий с квантилем Фишера. Код состоит из двух символов — для допустимой ошибки 5% и 1%. Знак «+» указывает, что текущая модель может быть улучшена, знак «–» означает «нет». Недостоверные модели (с отрицательным старшим коэффициентом) обозначаются знаком «*».

Результат работы программы *RG*41 включается в отчёт в виде таблицы.

Одновременно с выводом на экран программа *RG*41 создает текстовый файл *OUT.txt*, пригодный для импорта в электронную таблицу *EXCEL*.

Для упрощения обработки результатов можно воспользоваться полученной у преподавателя заготовкой электронной таблицы *EXAMPLE.xls*. В эту таблицу импортируются файлы с результатами измерений и коэффициентами уравнений регрессии — ввод и вывод программы *RG*41. Данные из файлов помещаются на отведённые для них места.

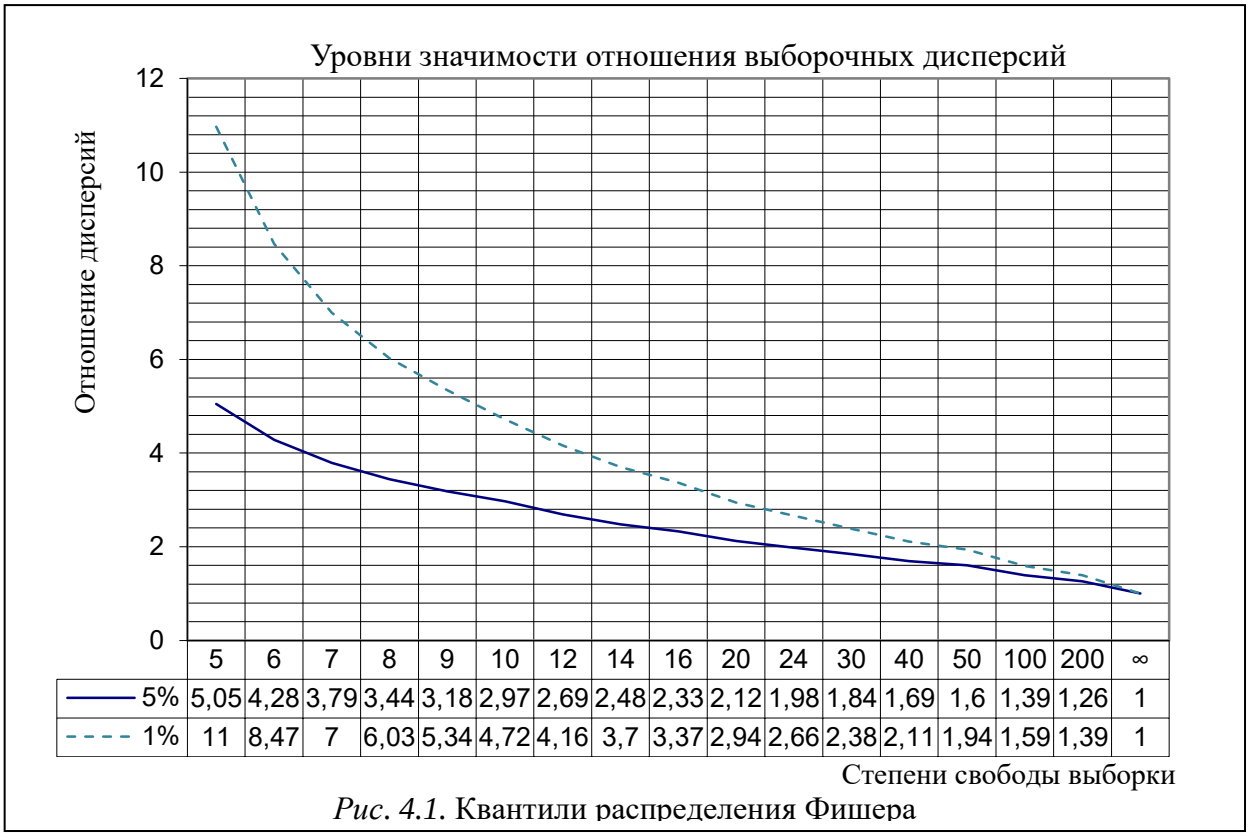
Вся необходимая обработка данных уже заложена в таблицу *EXAMPLE*. Содержимое файла с измерениями и файла *OUT.txt* нужно импортировать на свободное место в таблице (закладка «Данные», команда «Из текста»), а затем скопировать на штатное: измерения — в две крайние левые колонки, а коэффициенты уравнений регрессии — на соответствующее место в правой части (см. пример в табл. 4.1).

Для корректного вида графиков измерения должны быть упорядочены по возрастанию мощности (выделить область с измерениями, в закладке «Данные» выполнить команду «А->Я»).

Рядом с колонками результатов эксперимента находятся расчёты значе-

ний уравнений регрессии, по которым уже построены графики, а правее места расположения коэффициентов регрессии находится расчёт отношений каждой выборочной дисперсии ко всем остальным (табл. 4.2). По значениям отношений дисперсий можно сразу указать наиболее подходящее уравнение регрессии, самый старший коэффициент которого определит временную сложность алгоритма. Наиболее подходящим будет то уравнение, при дальнейшем усложнении которого уменьшение выборочной дисперсии прекращается или перестает быть значимым.

Поскольку выборочные дисперсии являются случайными величинами, значения дисперсий можно считать различными только в том случае, если их отношение превосходит значение квантиля распределения Фишера (рис. 4.1).




| | 5 | 6 | 7 | 8 | 9 | 10 | 12 | 14 | 16 | 20 | 24 | 30 | 40 | 50 | 100 | 200 | ∞ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5% | 5,05 | 4,28 | 3,79 | 3,44 | 3,18 | 2,97 | 2,69 | 2,48 | 2,33 | 2,12 | 1,98 | 1,84 | 1,69 | 1,6 | 1,39 | 1,26 | 1 |
| 1% | 11 | 8,47 | 7 | 6,03 | 5,34 | 4,72 | 4,16 | 3,7 | 3,37 | 2,94 | 2,66 | 2,38 | 2,11 | 1,94 | 1,59 | 1,39 | 1 |

*Рис. 4.1.* Квантили распределения Фишера

Входом в таблицу является количество степеней свободы выборки, равное количеству опытов минус количество оцениваемых коэффициентов регрессии. Во второй строке дано значение отношения выборочных дисперсий, которое можно считать значимым с ошибкой не более 5 %, в третьей — не более 1 %.

Как видно из табл. 4.2, отношение дисперсий перестает быть значимым, начиная со строки 6 (квантиль Фишера для 100 опытов равен 1,39/1,59).

*Таблица 4.1.*

***Пример результатов работы программы RG41 (100 точек)***

Результаты статистической обработки (из файла *out.txt*)

| Вариант | *D* | *S* | *k* | *c*0 (1) | *c*1 (ln *n*) | *c*2 (*n*) | *c*3 (*n* ln *n*) | *c*4 ($n^2$) | *c*5 ($n^3$) | *c*6 ($n^4$) | | Код |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1,24E+10 | 111362 | 1 | 124727 | 0 | 0 | 0 | 0 | 0 | 0 | 101 | ++ |
| 2 | 4,66E+09 | 68290 | 2 | -342638 | 106072 | 0 | 0 | 0 | 0 | 0 | 343 | ++ |
| 3 | 1,04E+09 | 32221 | 2 | -66364 | 0 | 1819,91 | 0 | 0 | 0 | 0 | 334 | ++ |
| 4 | 3,83E+08 | 19568 | 3 | 172451 | -78718 | 2848,72 | 0 | 0 | 0 | 0 | 776 | ++ |
| 5 | 1,33E+08 | 11548 | 3 | 55453,3 | 0 | -7333,3 | 1655,1 | 0 | 0 | 0 | 686 | ++ |
| 6 | 5,28E+07 | 7268,6 | 4 | -98088 | 86118 | -16553 | 3118,74 | 0 | 0 | 0 | 1407 | -- |
| 7 | 4,14E+07 | 6430,9 | 3 | 12040,5 | 0 | -339,42 | 0 | 10,2825 | 0 | 0 | 778 | -- |
| 8 | 3,25E+07 | 5700,5 | 4 | -9884,4 | 0 | 2860,12 | -740,946 | 14,5592 | 0 | 0 | 1251 | -- |
| 9 | 3,30E+07 | 5741,3 | 5 | -14934 | 4692,59 | 1852,37 | -542,874 | 13,849 | 0 | 0 | 1511 | -- |
| 10 | 3,34E+07 | 5780 | 4 | 2655,59 | 0 | 151,691 | 0 | 4,54349 | 0,018219 | 0 | 1260 | -- |
| 11 | 3,33E+07 | 5767,3 | 5 | -3966,4 | 4086,27 | -115,83 | 0 | 6,42406 | 0,013566 | 0 | 1524 | -- |
| 12 | 3,36E+07 | 5794,7 | 5 | 2069,12 | 0 | 410,075 | -78,1557 | 5,86721 | 0,015537 | 0 | 1501 | -- |
| 13 | 3,38E+07 | 5810,6 | 6 | 124,434 | 945,166 | 440,46 | -103,375 | 6,63938 | 0,013857 | 0 | 1513 | -- |
| 14 | 3,32E+07 | 5757,8 | 5 | -360,32 | 0 | 405,187 | 0 | -0,71675 | 0,056915 | -9,25E-05 | 1356 | ** |
| 15 | 3,36E+07 | 5792,6 | 6 | -410,66 | 0 | 252,468 | 59,9499 | -2,72468 | 0,066514 | -0,0001108 | 1431 | ** |
| 16 | 3,38E+07 | 5815,2 | 7 | -214,72 | -574,42 | 883,646 | -151,851 | 3,15105 | 0,039663 | -6,02E-05 | 2792 | ** |

*Таблица 4.2*

***Результаты расчета отношений выборочных дисперсий (фрагмент)***

| Вариант | Отношение дисперсий | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 1 | 1,00 | 0,38 | 0,08 | 0,03 | 0,01 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 |
| 2 | 2,66 | 1,00 | 0,22 | 0,08 | 0,03 | 0,01 | 0,01 | 0,01 | 0,01 | 0,01 | 0,01 | 0,01 | 0,01 |
| 3 | 11,95 | 4,49 | 1,00 | 0,37 | 0,13 | 0,04 | 0,04 | 0,03 | 0,03 | 0,03 | 0,03 | 0,03 | 0,03 |
| 4 | 32,39 | 12,18 | 2,71 | 1,00 | 0,35 | 0,11 | 0,11 | 0,08 | 0,09 | 0,09 | 0,09 | 0,09 | 0,09 |
| 5 | 92,99 | 34,97 | 7,78 | 2,87 | 1,00 | 0,32 | 0,31 | 0,24 | 0,25 | 0,25 | 0,25 | 0,25 | 0,25 |
| 6 | 289,53 | 108,88 | 24,24 | 8,94 | 3,11 | **1,00** | 0,97 | 0,76 | 0,77 | 0,78 | 0,78 | 0,78 | 0,79 |
| 7 | 299,87 | 112,77 | 25,10 | 9,26 | 3,22 | **1,04** | 1,00 | 0,79 | 0,80 | 0,81 | 0,80 | 0,81 | 0,82 |
| 8 | 381,64 | 143,51 | 31,95 | 11,78 | 4,10 | **1,32** | 1,27 | 1,00 | 1,01 | 1,03 | 1,02 | 1,03 | 1,04 |
| 9 | 376,24 | 141,48 | 31,50 | 11,62 | 4,05 | **1,30** | 1,25 | 0,99 | 1,00 | 1,01 | 1,01 | 1,02 | 1,02 |
| 10 | 371,21 | 139,60 | 31,08 | 11,46 | 3,99 | **1,28** | 1,24 | 0,97 | 0,99 | 1,00 | 1,00 | 1,01 | 1,01 |
| 11 | 372,85 | 140,21 | 31,21 | 11,51 | 4,01 | **1,29** | 1,24 | 0,98 | 0,99 | 1,00 | 1,00 | 1,01 | 1,02 |
| 12 | 369,33 | 138,89 | 30,92 | 11,40 | 3,97 | **1,28** | 1,23 | 0,97 | 0,98 | 0,99 | 0,99 | 1,00 | 1,01 |
| 13 | 367,30 | 138,12 | 30,75 | 11,34 | 3,95 | **1,27** | 1,22 | 0,96 | 0,98 | 0,99 | 0,99 | 0,99 | 1,00 |
| 14 | 374,07 | 140,67 | 31,32 | 11,55 | 4,02 | **1,29** | 1,25 | 0,98 | 0,99 | 1,01 | 1,00 | 1,01 | 1,02 |
| 15 | 369,60 | 138,99 | 30,94 | 11,41 | 3,97 | **1,28** | 1,23 | 0,97 | 0,98 | 1,00 | 0,99 | 1,00 | 1,01 |
| 16 | 366,73 | 137,91 | 30,70 | 11,32 | 3,94 | **1,27** | 1,22 | 0,96 | 0,97 | 0,99 | 0,98 | 0,99 | 1,00 |

Если с учётом уровня значимости из всех полученных уравнений регрессии невозможно однозначно указать наиболее подходящее, следует по*пробовать увеличить количество опытов.*

При выборе подходящего уравнения регрессии следует также принимать во внимание следующие соображения:

1. Если базовой структурой данных является хеш-таблица, в уравнении регрессии неоткуда взяться логарифмам, и варианты с логарифмами принимать во внимание вообще не следует. Возможное исключение — использование сортировки в операции слияния.

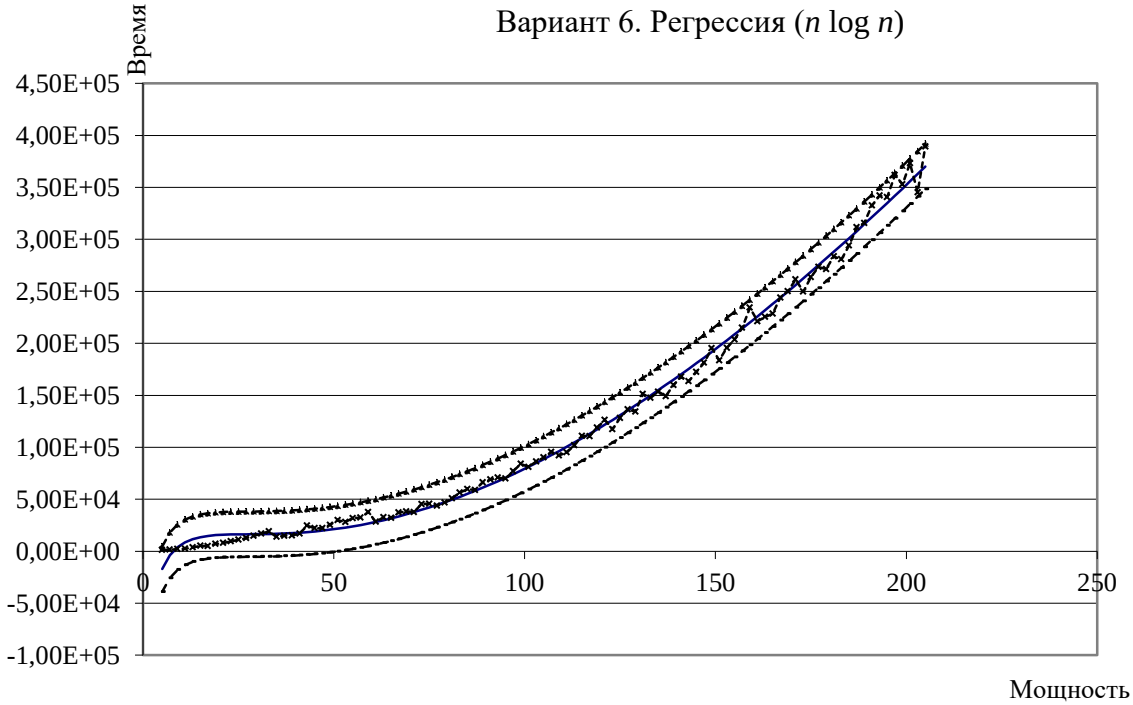



*Рис. 4.2.* Результат статистического эксперимента

2. Старший коэффициент корректного уравнения регрессии не может быть отрицательным. Такое уравнение не имеет физического смысла. Оно может получиться из-за смещения: отсчёты времени привязаны не к неправильному значению размера входа. Это может получиться, если используются размеры входа, заданные при генерации множеств, а не те, с которыми имеет дело каждая операция. Для нейтрализации смещения в примере программы эксперимента используется функция *Used*. Вызываемая после каждой операции с множествами, она подсчитывает их количество *set_count* и накапливает сумму мощностей *middle_power*, по которой в итоге вычисляется средняя мощность.

3. Необходимо также специально позаботиться о том, чтобы пересече-

ния множеств были не пусты, а цепочка, исключаемая из последовательности операцией *Excl*, действительно в ней встречалась с достаточно большой вероятностью. В эксперименте это обеспечивается двумя вспомогательными функциями *PrepareAnd* и *PrepareExcl*.

4. Рекомендуется также перед экспериментом с измерением времени понаблюдать за результатами каждой из операций исследуемой цепочки, чтобы вовремя заметить и предотвратить вырождение (обработку пустых множеств и т. п.). Для этого в программе для эксперимента используется функция *DebOut*, выдающая результат на экран, если управляющая переменная *debug* имеет значение *true*.

### 4.3. Оформление результатов эксперимента

Результаты эксперимента следует оформить в виде графика зависимости времени решения задачи как функции размера входа. На графике должны быть представлены:

— точки, соответствующие измеренным значениям;

— кривая регрессии для уравнения, отобранного по результатам сравнения выборочных дисперсий;

— границы доверительного интервала (регрессия ±3 СКО).

Значения СКО выдаются программой *RG*41. Ожидается, что все измеренные значения попадут в доверительную область (с вероятностью 99,7 %), а кривая регрессии будет усреднять их.

График можно взять из заготовок в электронной таблице *EXAMPLE*. Возможно, для этого таблицу придётся подкорректировать под фактический объём обработанных измерений (размножить строки в расчётной части и исправить диапазоны данных в графике). Достаточно доработать только тот график, который пойдёт в отчёт (см. рис. 4.2).

### 4.4. Выводы

По итогам эксперимента даётся заключение о том, насколько экспериментальная оценка временной сложности алгоритма обработки данных соответствует теоретической. Исследуются особенности реализации алгоритмов, позволяющие объяснить полученные результаты.

Алгоритмы отдельных операций следует доработать таким образом, чтобы получилась итоговая линейная сложность или обосновать, почему это невозможно.

## Список литературы


1. Ахо Дж., Хопкрофт А., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. 536 с.

2. Галовиц Я. C++17 STL. Стандартная библиотека шаблонов. СПб.: Питер, 2018. 432 с.: ил.

3. Готтшлинг П. Современный C++ для программистов, инженеров и ученых. М.: ИД «Вильямс», 2016. 512 с.: ил.

4. Джоссатис Н. М. Стандартная библиотека C++: справочное руководство. 2-е изд. / пер. с англ. М.: ИД «Вильямс», 2014. 1136 с.: ил.

5. Кнут Д. Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск. М.: Мир, 2013. 736 с.

6. Липпман С. Б., Лажойе Ж., Му Б. Э. Язык программирования C++. Базовый курс. 5-е изд. / пер. с англ. М.: ИД «Вильямс», 2014. 1120 с.: ил.

7. Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов. / пер. с англ. М.: Изд-во «ДМК Пресс», 2012. 294 с.: ил.

8. Мэйерс С. Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14 / пер. с англ. — М.: ИД «Вильямс», 2018. 304 с.: ил.

9. Новиков Ф. А. Дискретная математика: учеб. для вузов. 2-е изд. Стандарт третьего поколения. СПб.: Питер, 2013. 432 с.: ил.

10. О'Двайр, А. Осваиваем C++17 STL. М.: Изд-во «ДМК Пресс», 2019. 352 с.: ил.

11. Прата С. Язык программирования C++. 6-е изд. М.: ИД «Вильямс», 2011. 1244 с.: ил.

12. Саттер Г. Решение сложных задач на C++ / пер. с англ. М.: ИД «Вильямс», 2015. 400 с.: ил.

13. Седжвик Р. Алгоритмы на C++ / пер. с англ. М.: ИД «Вильямс», 2011. 1156 с.: ил.

14. Страуструп Б. Программирование: принципы и практика с использованием C++. 2-е изд. / пер. с англ. СПб.: ООО «Диалектика», 2019. 1328 с.: ил.

15. Страуструп Б. Язык программирования C++. 2-е изд., доп. М.: Изд-во «Бином-пресс», 2001. 1098 с.

16. Страуструп Б. Язык программирования C++ (стандарт C++11). Краткий курс / пер. с англ. М.: Изд-во «Бином-пресс», 2019. 176 с.: ил.

## ПРИЛОЖЕНИЕ

### Измерение времени запросом внутреннего счётчика тактов процессора

Современные процессоры (начиная с *Pentium* II последних серий) поддерживают команду *RDTSC*, возвращающую 64-битное значение внутреннего счетчика тактов. Это достойная альтернатива применению функции *clock*( ): можно измерить время выполнения даже одной команды процессора. Самый простой и универсальный способ добраться до счетчика тактов состоит в применении возможностей библиотеки *std::chrono*. В ней доступна функция *now*( ), снимающая текущие показания часов высокого разрешения. Её можно применить тем же способом, что и *clock*( ), но без необходимости зацикливать измеряемый процесс. Чтобы избежать ненужных сложностей, рекомендуется действовать следующим образом:

1) перед началом измеряемого процесса снимается отсчет времени *t*1:

```
auto t1 = std::chrono::high_resolution_clock::now( );
```

2) по завершении процесса таким же образом снимается отсчет *t*2;

3) для получения разности отсчетов *dt* используется выражение

```
auto dt = duration_cast<duration<double>>( t2–t1 );
```

4) в результате используется выражение *dt.count*( ):

```
cout << " DT=" << (dt.count( )) <<")\n";
```

Именно этот способ измерения времени использован в примере программы для статистического эксперимента (см. 4.1).

## СОДЕРЖАНИЕ

Колинько Павел Георгиевич

**Пользовательские контейнеры**

Учебно-методическое пособие

Редактор Е. А. Ушакова

---

Подписано в печать . .2020 . Формат 60×84 1/16.
Бумага офсетная. Печать цифровая. Печ. л. 4,0.
Гарнитура «Times New Roman». Тираж 151 экз. Заказ

---

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5