ФЕДЕРАЛЬНОЕ АГЕНСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение
высшего профессионального образования
УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет математики и информационных технологий

В.В. Угаров

Технология программирования
часть 2

Учебно-методическое пособие

Ульяновск – 2010

УДК 681.142.2

Печатается по решению Ученого Совета
факультета математики и информационных технологий
Ульяновского государственного университета

Рецензенты:
доктор технических наук, профессор И.В. Семушин;
кандидат педагогических наук Г.А. Жаркова

Угаров В.В.
Технология программирования. Часть 2: учебно-методическое пособие / В.В.Угаров.–Ульяновск: УлГУ, 2010.–83 с.

Данное учебное пособие представляет собой курс лекционного материала по дисциплине «Технология программирования» с некоторыми сокращениями. Кроме того, в пособие добавлено значительное количество примеров, заданий для самостоятельного выполнения и приложений.

Пособие состоит из введения, 6 глав, списка литературы и приложения.

В первых главах пособия рассматриваются некоторые вопросы информационных технологий, затем краткое изложение основ языка программирования C++. В основном же пособие посвящено вопросам формирования структур данных и алгоритмов их обработки.

Отличие данного пособия от многих аналогичных состоит в том, что в нем более глубоко рассматриваются структуры данных и методы их обработки, а не собственно язык программирования, который служит в основном для реализации алгоритма. Тем не менее, средства языка C++ в значительной мере влияют на способы реализации тех или иных алгоритмов обработки. Естественно, в отдельных случаях такие моменты в данном пособии оговариваются.

В пособии рассматриваются свойства как простых типов данных, так и более сложных структур данных: последовательностей, одномерных и двухмерных массивов, структур, файлов. Параллельно с рассмотрением структур данных подробно анализируются алгоритмы их обработки и связанные с этим средства языка программирования: механизм функций, модулей, библиотек. Представлен широкий спектр алгоритмов обработки последовательностей, массивов и файлов.

Примеры программ по текущим темам могут быть использованы для изучения раздела темы студентами соответствующих курсов, а задания - для получения навыков в практике составления программ.

СОДЕРЖАНИЕ

Введение

Учебная дисциплина "Технология программирования-2" читается на 1 курсе 2 семестре университета для информационных специальностей.

Назначение дисциплины: изучение основных (базовых) структур данных, грамотное программирование задач объемом до нескольких (2-5) тысяч строк, освоение и изучение базовых алгоритмов, лежащих в основе большинства программных продуктов, освоение методики создания программ прикладного назначения.

Этот курс посвящен программирования как таковому.

Не рассматриваются вопросы вычислительных методов приближенных вычислений, операционных систем, методы построения компиляторов, и т.д. Эти темы могут слегка затрагиваться при изучении основных разделов.

В этой дисциплине предполагается изучение текстов готовых программ и создание новых, учебных программ объемом несколько тысяч строк.

Дисциплина "Технология программирования" читается в 1 семестре учебного года, а во втором семестре читается тесно связанная с ней "Технология программирования-2".

Во втором семестре рассматриваются более сложные вопросы построения прикладных программ. В частности, происходит знакомство с элементами объектно-ориентированного программирования, освоение работы с динамическими структурами данных, рассмотрение разнообразных алгоритмов на графах, знакомство с основами программирования в среде Windows на основе инструментального пакета Visual C++, изучение методов индустриального производства программных продуктов.

Наряду с лекционным курсом параллельно проводятся семинарские и практические занятия, в том числе в компьютерных классах. На них студенты выполняют работы по созданию учебных программ, осваивают среду программирования, получают навыки грамотного программирования.

В конце каждого семестра студент сдает экзамен. К экзамену допускаются только студенты, выполнившие необходимый объем работ как на семинарах (контрольные работы), так и на практических занятиях (лабораторные проекты).

Основные требования к программным проектам, которые создаются в ходе учебного процесса: во-первых, это работоспособность программ, т.е. программы должны выполнять то, что задано, без каких либо исключений, во-вторых, проект должен быть выполнен полностью самостоятельно.

Количество защищенных работ влияет на оценку экзамена.

Для работы в дисплейном классе студентам необходимо будет приобрести средства для хранения своих учебных программ либо пару дискет, либо флэш-память, если в компьютерном классе есть вход USB.

Поскольку время работы в компьютерном классе ограничено учебным планом, то для выполнения домашних заданий и доработки проектов, студентам необходимо получить доступ к компьютеру вне университета. Это может быть личный домашний компьютер, компьютер на работе у родителей, компьютер у друзей и т.д.

1. Графическая подсистема языка C++

1.1. Видеосистема компьютера.

Видеосистема компьютера имеет два режима вывода информации: текстовый и графический.

В текстовом режиме на экране дисплея отображается 25 строк по 80 знакомест в каждой строке. Одно знакоместо предназначено для одного символа из таблицы ASCII.

В графическом режиме экран представлен в виде совокупности из отдельных элементов – пикселов (pixels), из которых строится графическое изображение. Количество пиксел определяется разрешением видеосистемы и текущим режимом видеомонитора. Разрешение видеомонитора задается в виде: Rx*Ry, где Rx – количество пиксел по оси x, а Ry – по оси y. Например:

1. CGA 320 x 200 пиксел, 4 цвета
2. QVGA 320 x 240 пиксел,
3. VGA 640 x 480 пиксел, 16 цветов
4. SVGA 800 x 600 пиксел, 256 цветов
5. XVGA 1024 x 768, 1600 x 1200 пиксел, 16 млн. цветов.

Чем больше разрешение экрана, тем выше качество изображения. На качество изображения влияет также количество цветов, которое может отображать один пиксел.

Если сопоставить 1 бит оперативной памяти с 1 пикселом, то при значении бита, равного 0 пиксел будет погашен (черный цвет), при значении, равном 1, пиксел будет светится (белый цвет). Таким образом можно получить черно-белое изображение без полутонов.

При сопоставлении с пикселом 4 бит возможно отображение $2^4 = 16$ цветов. Такой способ применен в режиме VGA. Оценка количества требуемой видеопамяти в режиме VGA дает:

Vop = 640 x 480 x 4 = 1228800 (бит) = 153600 (байт).

0 – черный	8 – серый
1 – синий	9 – светло-синий
2 – зеленый	10 – светло-зеленый
3 – голубой	11 – светло-голубой
4 – красный	12 – светло-красный
5 – фиолетовый	13 – светло-фиолетовый
6 – коричневый	14 – желтый
7 – светло-серый	15 – белый

Табл.1. Цвета видеорежима VGA.

Если на один пиксел отвести 1 байт памяти, тогда можно отобразить 256 цветов. Дальнейшее улучшение качества изображения требует все возрастающего количества ячеек памяти. Например, режим TrueColor сопоставляет каждому пикселу 3 байта (24 бит), что позволяет отобразить 16777216 цветов и требует памяти при разрешении 1024x768 размером:

Vop = 1024 x 768 x 3 = 2359296 байт или 2,4 Мбайт.

Для работы видеосистемы компьютера в графическом режиме необходима специальная программа-драйвер, которая управляет графическим адаптером. Для установки режима VGA она называется: **"egavga.bgi"** и находится обычно в подкаталоге "BGI". Графический экран имеет следующую систему координат (Рис 1):

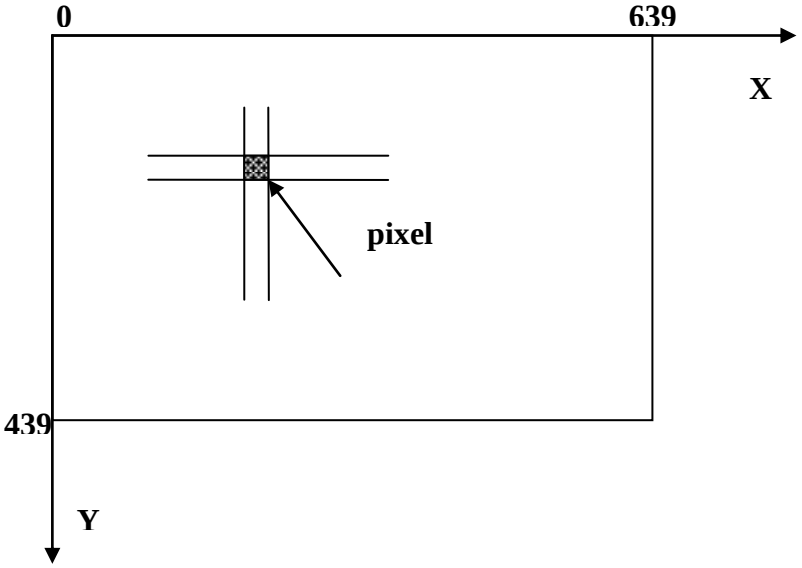


Рис 1. Система координат графического экрана.

1.2. Основные функции графической библиотеки.

Для работы с графикой в языке Borland C++ создана библиотека **"graphics.h".**В ней собрано большое количество функций, позволяющих строить в графическом режиме векторные графические изображения. Приведем некоторые из них.

1. **initgraph(&gdriver,&gmode,"")** – описание и пример в Ctrl+F1
2. **closegraph()** – закрытие графического режима
3. **moveto(int x, int y)** – устанавливает курсор в точку (x,y)
4. **getmaxx(), getmaxy()** – возвращает максимальное значение по оси x и по оси y
5. **line(int x1, int y1, int x2, int y2)** – линия от точки (x1, y1) до точки (x2, y2)

6. **rectangle(int x1, int y1, int x2, int y2)** – прямоугольник от левой верхней точки (x1, y1) до нижней правой точки (x2, y2)
7. **circle(int x, int y, int r)** – круг с центром в точке (x,y) и радиусом r
8. **arc(int x, int y, int stangle, int endangle, int r)** – дуга окружности с центром в точке (x,y), начальный угол **stangle,** конечный угол **endangle,** (в градусах), радиус **r;**
   arc(300,100,45,185,50);
9. **setcolor(int color)** – установка цвета рисования:
   0 - черный, 1 - синий, 2 - зеленый, 3 - бирюзовый,
   4 - красный, 5 - фиолетовый, 6 - коричневый, 7 - светло-серый,
   8 - темно-серый, 9 - голубой, 10 - светлозеленый, 11 - светлобирюзовый,
   12 - розовый, 13 - пурпурный, 14 - желтый, 15 - белый.
10. **getcolor()** – возвращает текущий цвет
11. **setbkcolor(int color)** – установка цвета фона
12. **getbkcolor()** – возвращает цвет фона
13. **putpixel(int x, int y, int color)** – пикселу в точке (x,y) присваивает цвет **color**
14. **outtextxy(int x, int y, char st)** – выводит текст **st**, начиная с точки (x,y):
    outtextxy(200,300,"ABCD-text");
15. **setlinestyle(int linestyle, int upattern, int thickness)** – линии рисуются стилем **linestyle**,
    по шаблону **upattern** (если **linestyle=4**),
    толщиной **thickness** (1 или 3);
    Параметры **linestyle**: 0 - сплошная, 1 - точечная, 2 - штрихпунктирная, 3 - пунктирная, 4 - по шаблону
16. **setfillpattern(char upattern, int color)** – установка шаблона заливки **upattern** цветом **color;**
    Пример:

```
// Создаем шаблон с названием "Strelka":
char Strelka[8]=
        {0x00,0x1e,0x06,0x0a,0x12,0x20,0x40,0x80};
// Устанавливаем заливку по шаблону "Strelka" желтым цветом:
setfillstyle(12,14);
setfillpattern(Strelka,14);
// Рисуем круг цветом 15, как пример замкнутой области:
circle(200,300,15);
// Область (круг), ограниченную цветом 15, с внутренней точкой
// (x=200, y=300), заливаем рисунком по шаблону "Strelka":
floodfill(200,300,15);
```

Полезные функции для работы в графическом режиме:

- Функция округления вещественного числа, возвращает целое число, округленное до целого:

```
int round(float a)
 { if (a-floor(a)<0.5) return floor(a);
                       return ceil(a);};
```

- Функция преобразования **x** координаты из мировой системы координат в систему координат устройства (в данном случае - VGA):

```
int Nx(float x)
  {return round(((x-Xmin)/(Xmax-Xmin))*639);};
```

- Функция преобразования **y** координаты из мировой системы координат в систему координат устройства (в данном случае - VGA):

```
int Ny(float y, float Ymin, float Ymax)
  {return 479-round(((y-Ymin)/(Ymax-Ymin))*479);};
```

1.3. Построение статичных векторных изображений.

В векторной графике изображения строятся из простых объектов — прямых линий, дуг, окружностей, эллипсов, прямоугольников, областей однотонного или изменяющегося цвета (заполнителей) и т. п., называемых примитивами. Из простых векторных объектов создаются различные рисунки

Комбинируя векторные объекты-примитивы и используя заливку различными цветами, получают более сложные рисунки. Графические примитивы строятся на экране с помощью функций графической библиотеки.

Достоинством векторных изображений является возможность формировать рисунок разного размера, т.е. масштабировать, без потери качества.

При построении статичных векторных рисунков используется подход, при котором, изменяя параметры, можно устанавливать этот рисунок в любое заданное место на экране, и строить его в любом, заданном масштабным коэффициентом, размере. Назовем эти свойства перемещаемостью и масштабируемостью изображения.

Для этого используется метод опорной точки.

В качестве примера рассмотрим создание рисунка, который можно размещать на экране в любом месте экрана в любом размере.

Нарисуем тележку на бумаге и определим размеры в некоторых относительных единицах (Рис 2.). На рисунке выберем опорную точку с координатами x, y в центре оси левого колеса тележки (по рисунку). Длина кузова тележки равна 40 единиц, высота 20 единиц, радиус колеса - 10 единиц и т.д. Для того, чтобы можно было менять положение рисунка и его размеры, оформим программу построения рисунка в виде функции с параметрами опорной точки **x,y** и масштабного коэффициента **mk**.

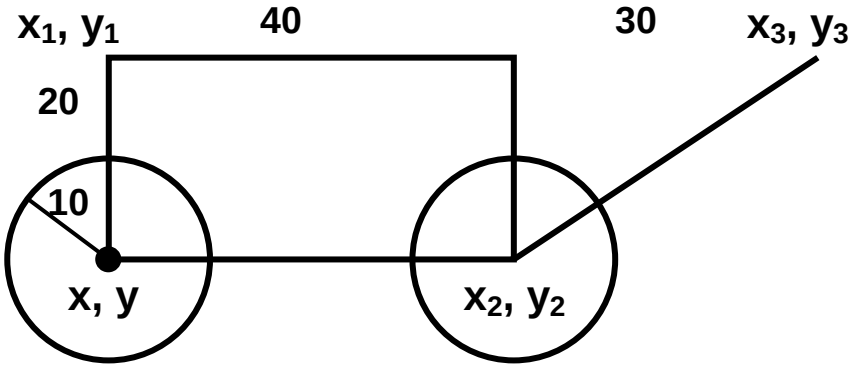


Рис. 2. Пример векторного рисунка.

```
void tachka(int x, int y, float mk)
     {    int x1, y1, x2, y2, x3, y3;
          circle(x,y,round(10*mk));      // Левое колесо
          x1=x; y1=y-round(20*mk);
          x2=x+round(40*mk); y2=y;
          rectangle(x1,y1,x2,y2);        // Кузов
          circle(x2,y2,round(10*mk));    // Правое колесо
          x3=x+round(70*mk); y3=y1;
          line(x2,y2,x3,y3);             // Тяга
     }
```

При вызове функции в списке параметров задаются значения опорной точки и масштабный коэффициент, и на экране будет построено изображение тележки, причем опорная точка будет находится в точке (100,200):

```
tachka(100,200,3.7);
```

1.4. Работа с фрагментами изображений.

При построении графических изображений достаточно часто требуется оперировать отдельными фрагментами изображений, нередко называемых спрайтами. Это позволяет заранее подготовить отдельные элементы изображения и затем собрать из них необходимую картину. Поскольку вывод на экран фрагментов происходит очень быстро, появляется возможность создания анимированных изображений.

Рассмотрим процесс создания анимированного изображения на примере движения некоторого объекта на фоне статичной картины.

На первом этапе на видеостранице тем или иным способом строится фрагмент изображения, который затем сохраняется в оперативной памяти. Затем этот фрагмент вызывается из оперативной памяти на экран для построения изображения. Эффект движения создается поочередным выводом фрагмента изображения, задержке его на некоторое время на экране, затем его удаление и вывод этого же фрагмента со смещением.

```
// Сцена анимации: пролет НЛО под лучами радиолокатора
#include <graphics.h>
#include <alloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{  int gdriver = DETECT, gmode, ecode;
   initgraph(&gdriver, &gmode, "");
   ecode = graphresult();
   if (ecode != grOk)
   {printf("Graphics error: %s\n", grapherrormsg(ecode));
      getch();  exit(1);     //выход, если ошибка графики
   }
// Задается шаблон заполнителя НЛО
char shab1[8]={0x00,0x1e,0x06,0x0a,0x12,0x20,0x40,0x80};
   void far *buf;
   setcolor(12);
   setfillstyle(8,14);
   pieslice(500,100,0,360,30);              //отрисовка НЛО
   long size=imagesize(470,70,530,130);    //объем рисунка
   buf=farmalloc(size);
   getimage(470,70,530,130,buf);  // Рисунок НЛО в память
   cleardevice();                 // Очистка экрана
   setcolor(2);
   line ( 0,400,500,0);           // Лучи радиолокатора
   line ( 0,400,640,100);
   setfillstyle(10,3);
   setfillpattern(shab1,5);
   floodfill( 500,10,2);

   for (int i=1;i<650;i++)        // Цикл анимации НЛО
       {putimage(5+i,150, buf, XOR_PUT);
        delay(5);                 // Задержка
        putimage(5+i,150, buf, XOR_PUT);
        if (i>350)       //Если НЛО обнаружен,то сигнал!
          { outtextxy(200,400,"ALERT !!!  NLO !!!!");
            if (i%70==0)printf("\a");
          }
       }
  getch();
  farfree(buf);                   // Очистка памяти
  closegraph();                   // Закрытие графики
  return 0;
}
```

В программе используется метод **putimage(x,y,buf,op)**, который вызывает из памяти фрагмент изображения **buf** и записывает его в видеопамять с координатами (x,y). Параметр **op** метода задает режим взаимодействия с фоном при выводе фрагмента на экран. Поскольку эта операция побитовая, то возможны следующие значения этого параметра:

```
Const    XOR_PUT = 1;      { xor }
         OR_PUT  = 2;      { or }
         AND_PUT = 3;      { and }
         NOT_PUT = 4;      { not }
```

Следует отметить, что вывод в режиме **XOR_PUT** обеспечивает видимость фрагмента на любом фоне с восстановлением фонового изображения.

1.5. Преобразование координат при построении графиков

Система координат устройства (СКУ) вывода на графический экран представляет собой прямоугольный массив адресуемых пикселов (pixels) на плоскости:

0 <= Nx <= Nxmax; 0 <= Ny <= Nymax;

Здесь **Nx, Ny** – целые, для режима VGA **Nx=639**, **Ny=479**. Точка (0,0) расположена вверху слева экрана.

Другая система координат, называемая мировой или декартовой, является независимой от типа устройства отображения:

Xmin <= X <= Xmax; Ymin <= Y <= Ymax;

Данные соотношения определяют некоторую прямоугольную область в двухмерном пространстве. Здесь величины **X,Y,Xmin,Xmax,Ymin,Ymax** – имеют вещественный тип. Ширина этой области равна **W = Xmax – Xmin**, а высота равна **H = Ymax – Ymin.**

Таким образом, ставится задача перевода координат некоторой точки **(x,y)** мировой системы координат в систему координат устройства **(Nx,Ny)**. Такое преобразование координат позволит отобразить прямоугольную область в мировой системе на устройстве без искажений. (Рис.3)

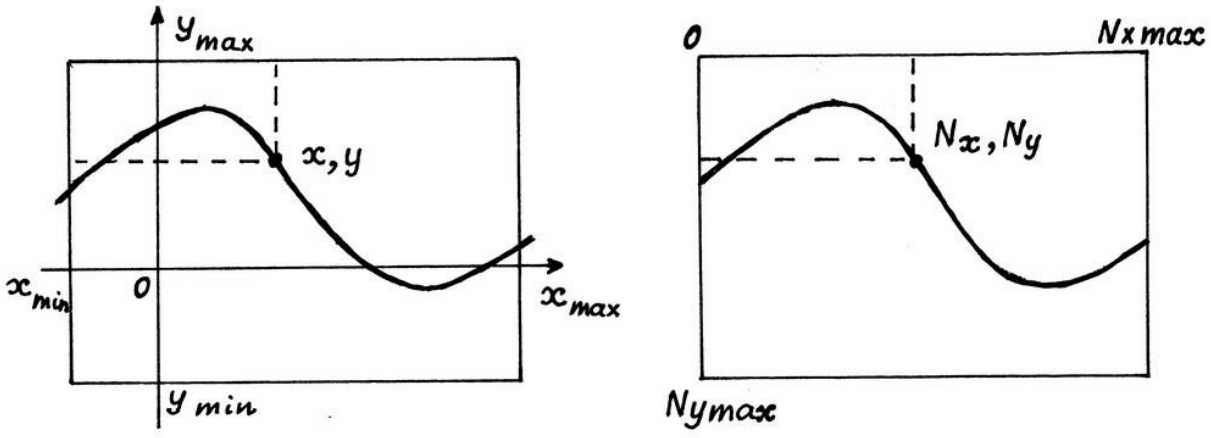


Рис. 3. График в декартовой системе и в системе устройства.

Преобразование выполняется на основе пропорциональных соотношений:

$$\frac{x - x_{\min}}{N_x} = \frac{x_{\max} - x_{\min}}{N_{x\max}};$$

$$\frac{y_{\max} - y}{y_{\max} - y_{\min}} = \frac{N_y}{N_{y\max}};$$

Пример построения графического изображения заданной функции

y=(8+12*sin(x/0.34)-4.6*cos(x/0.14))

при изменении переменной **x** в диапазоне от **Xmin** до **Xmax**, таким образом, чтобы оно полностью поместилось на экране устройства. Процесс разделяется на два этапа:

- Определение значений **Ymin** и **Ymax** в заданном диапазоне от **Xmin** до **Xmax;**
- Вычисление значений **Y(x)**, преобразование этих значений в **Nx** и **Ny**, и вывод пикселя с координатами **Nx** и **Ny** на экран.

```
// Построение графика функции
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>

int NxL=20, NyV=20, NxP=600, NyN=400; //Размеры окна

float modul(float x){if(x<0)return -x;return x;};

float Y(float x){return(8+12*sin(x/0.3)-4*cos(x/0.1) );};

int round(float a)        // Округление числа a
   {if(a-floor(a)<0.5) return floor(a); return ceil(a);};

// Функции преобразования из мировой системы в систему координат устройства
int Nx(float x, float Xmin, float Xmax)
  {return NxL+round(((x-Xmin)/(Xmax-Xmin))*NxP);};
int Ny(float y, float Ymin, float Ymax)
  {return NyN-round(((y-Ymin)/(Ymax-Ymin))*(NyN-NyV));};
```

```
//-------------------------------------------------------
int main(void)
{ int gdriver = DETECT, gmode, errorcode;
  initgraph(&gdriver, &gmode, "C:\\LANG\\BC\\BGI");
  errorcode = graphresult();
  if (errorcode != grOk)
  { printf("Graphics error!\n"); getch(); exit(1);}
  setbkcolor(1);  clearviewport();

  // Определение Ymin и Ymax
  float x, y, Ymin, Ymax;
  float Xmin=-1.8;        float Xmax=1.7;
  float t=(Xmax-Xmin)/((NxP-NxL)*5);
  x=Xmin;  Ymin=Y(Xmin); Ymax=Y(Xmin);
  while (x<=Xmax)
      { y=Y(x);
        if (y>Ymax) Ymax=y;
        if (y<Ymin) Ymin=y;
        x+=t;
      };
  // Рамка
  setcolor(8);
  rectangle(Nx(Xmin, Xmin,Xmax), Ny(Ymin, Ymin,Ymax),
            Nx(Xmax, Xmin,Xmax), Ny(Ymax, Ymin,Ymax));
  // Построение графика на экране
  x=Xmin;
  while (x<=Xmax)
      { y=Y(x);
        putpixel(Nx(x,Xmin,Xmax),Ny(y,Ymin,Ymax),15);
        x+=t;
      };
  setcolor(7);
  getch();
  closegraph();
  return 0;
}
```

В том случае, если затраты на вычисление значений функции перед построением значительны, можно полученные значения **Y(x)** записать в бинарный файл, одновременно определив **Ymin** и **Ymax**, а затем использовать эти данные при построении графика.

2. Объектно-ориентированное программирование

2.1. Определение объектов

В основе объектно-ориентированного программирования (далее ООП) лежит идея объединения в одной структуре набора данных и действий над ними (методов). В технологии ООП данные и методы связаны намного точнее, чем в традиционном структурном программировании.

Основная цель ООП – повышение эффективности разработки программ. Процесс проектирования программ, как правило, сопровождается неоднократным изменением как текста программ, так и её спецификации на каждом этапе проектирования. Причем, чем ближе к завершению, тем большие трудозатраты требуются для корректировки программы.

Нарисуем диаграмму, показывающую процесс разработки ПО (Рис. 4).

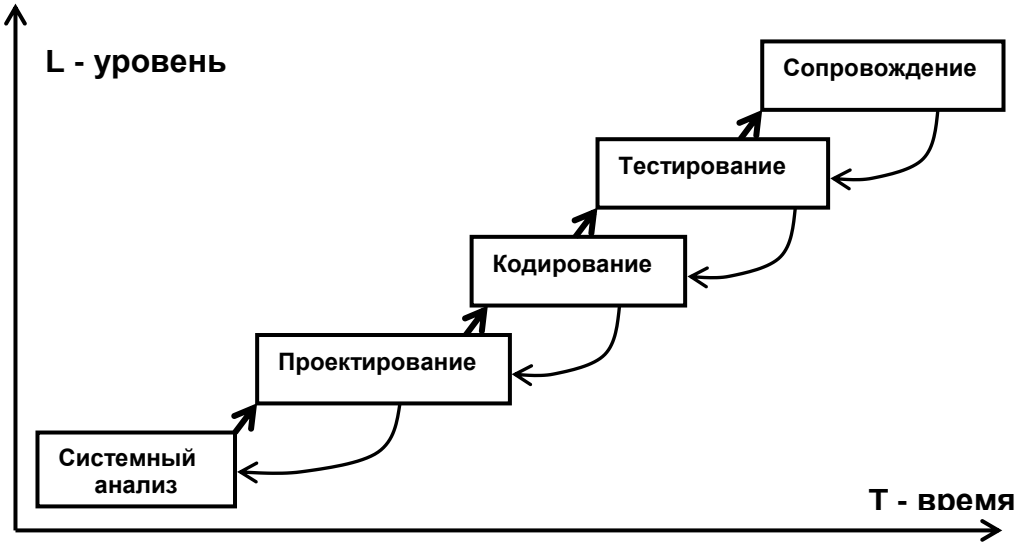


Рис.4. Процесс создания программного продукта.

Эта диаграмма (Рис.4) показывает, что при отклонении проекта от заданных требований возможен возврат на предыдущие этапы. Причем, чем ближе программный проект к завершению, тем большие трудозатраты требуются для корректировки проекта.

Использование ООП регулирует этот процесс, т.к. позволяет изменить только необходимый компонент (объект), не затрагивая его интерфейса.

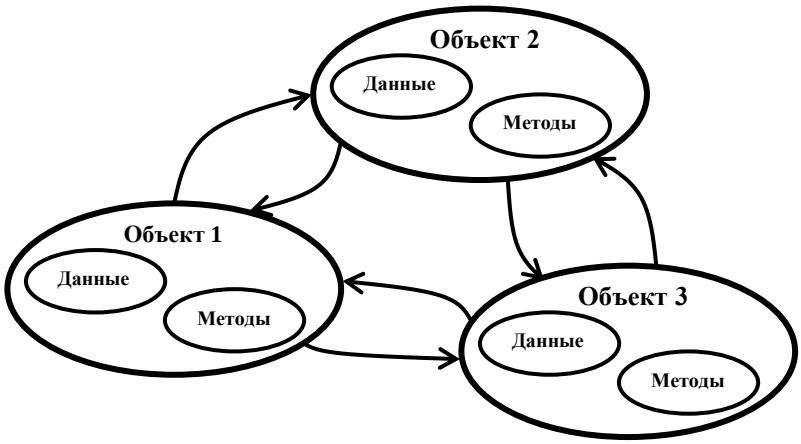


Рис.5. Связи объектов.

При создании программного продукта с использованием ООП следует уменьшать до минимума количество связей между объектами (Рис.5).

Точно также как, в свое время, использование механизма процедур и функций позволило перейти к структурированному программированию, так и ООП явилось основой перехода на более высокий уровень технологии создания программ.

Технология ООП базируется на трех понятиях:

1. Инкапсуляция – комбинирование данных и методов (процедур и функций) в единой структуре (объекте).
2. Наследование – использование свойств объекта для построения иерархии объектов-наследников. Объекты-наследники имеют доступ к методам и данным предыдущих «родительских» объектов.
3. Полиморфизм – возможность создания метода, имеющего одно имя, но применимого ко всем объектам иерархии наследования, причем каждый объект выполняет это действие соответствующим для него образом.

ООП позволяет упростить создание сложных программ, расширить возможности объектов и программ, не переделывая уже готовых, отлаженных элементов, а только добавляя к ним новые возможности.

В ООП применятся следующая нотация.

1. Применение метода к объекту: **<Объект>.<Метод>;**
   Например: **point.init(x,y); line.clear; form2.showmodal;**
2. Присвоение свойству объекта некоторого значения:
   <Объект>.<Свойство> := <Значение>;
   Например: b**utton1.color := Red;**

2.2 Структура объекта

Объект	
Данные:	**Методы:**
x;	**init();**
y;	**move();**
radius;	**add();**
…	**...**

Рис. 6. Объект.

Объект - самый высокий уровень абстракции данных. Эта структура, объединяющие данные различных типов и методов их обработки (Рис.6).

Компоненты данных называются полями, а процедуры и/или функции методами.

В качестве примера можно привести такие общеизвестные объекты:

- Телевизор: регулятор громкости, коммутатор. Методы: переключить, выключить.
- Автомобиль: руль, тормоз, свет. Методы: повернуть, включить, нажать.

2.3 Понятие класса

Класс - это описание множества объектов, которые имеют одинаковые свойства, операции, отношения и семантику (смысл). Любой объект - это про-

сто экземпляр класса. Различают внутреннее представление класса (реализацию) и внешнее представление класса (интерфейс). Реализация класса описывает поведение класса и включает реализацию всех операций, определенных в интерфейсе класса.

Основной принцип ООП состоит в том, что данные класса должны быть доступны только через методы этого класса. Специальные метки **`public, protected и private`** определяют режим доступа к данным класса:

- **`private`** - данные доступны только для методов данного класса;
- **`protected`** - данные доступны также и наследникам этого класса;
- **`public`** - данные доступны для других классов.

Другом класса называют класс, который имеет доступ ко всем частям этого класса (метки public, protected, private). Иными словами, от друга у класса нет секретов.

2.4 Иерархия объектов

Создавая иерархию объектов, разработчик, прежде всего, определяет их общие черты и отличия. В качестве примера рассмотрим класс геометрических фигур, разделив их, к примеру, на две группы (Рис. 7):

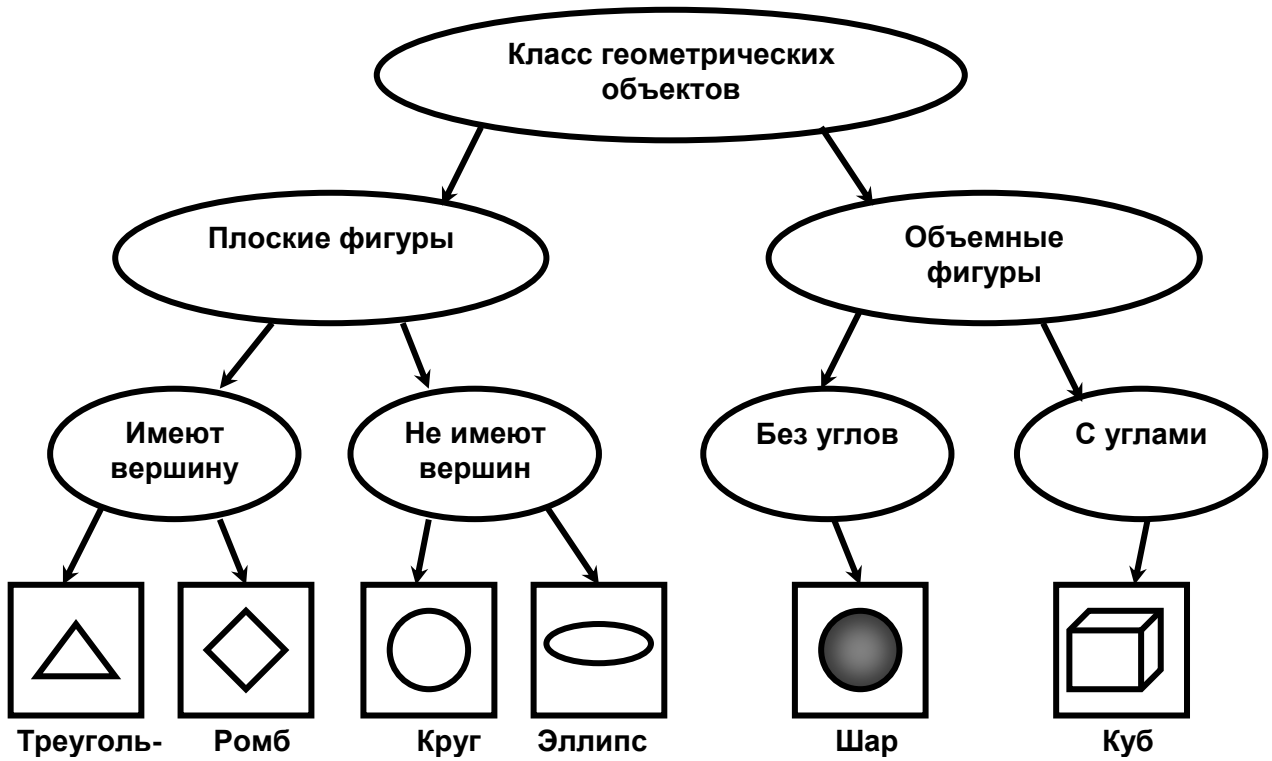


Рис.7. Иерархия объектов.

На верхнем уровне свойства более абстракты, на каждом следующем свойства объектов более конкретны, на самом нижнем уровне иерархии разработчик оп-

ределяет конкретные параметры объекта: цвет, размер, положение, толщину линий и т.д.

2.5 Пример создания иерархии объектов

В качестве примера рассмотрим программу построения графических объектов на экране, в которой использованы инкапсуляция, наследование, полиморфизм, а также вызов методов. В программе описан класс **cPoint** ("Точка"), затем, используя наследование, описывается класс **cCircle** ("Круг"), который отличается только добавленным параметром - радиусом круга. В главной функции создаются объект **p1** ("Точка") и объект **c1** ("Круг"). Затем происходит перемещение на экране точки и круга использованием метода **Move**.

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>

class cPoint                          // класс Точка
  {
   protected: int x; int y;
   public:
     void InitPoint(int A, int B) {x=A; y=B;}
     virtual void On()  { putpixel(x,y,getcolor());}
     virtual void Off() { putpixel(x,y,getbkcolor());}
     void Move(int dx, int dy)
       { Off(); x+=dx; y+=dy; On();}
   };

 class cCircle: public cPoint     // Класс Круг
  {
   protected:   int r;
   public:
     void InitCircle(int A,int B,int C) {x=A; y=B; r=C;}
     virtual void On() { circle(x,y,r);}
     virtual void Off()
       {int  Col=getcolor(); setcolor(getbkcolor());
             circle(x,y,r);  setcolor(Col);}
   };
//-----------------------------------------------------
int main(void)  { cPoint p1;  cCircle c1;
   int gd = DETECT, gm, err;
   initgraph(&gd, &gm, "A:");  err = graphresult();

   setcolor(7); setbkcolor(1); clrscr; randomize;
```

```
   for (int k=1; k<100; k++)
     { p1.InitPoint(random(640),random(480)); p1.On();};
   setcolor(14);
   p1.InitPoint(100,100); p1.On();
   p1.Move(90,70);              // перемещение точки
   setcolor(12);
   c1.InitCircle(100,120,40); c1.On();
   c1.Move(190,70);             // перемещение круга
   getch();
   closegraph();
   return 0;
}
```

Полиморфизм демонстрируется использованием виртуальных функций. Если в некотором классе имеется функция, описанная как **`virtual`**, то в такой класс компилятором добавляется скрытый указатель на таблицу виртуальных функций (VMT). На основе этой таблицы происходит выбор виртуальной функции во время выполнения программы, а не во время компиляции. Эта способность называется поздним связыванием, а методы класса, переопределяемые таким способом, называются полиморфными.

Иерархия классов подразумевает наличие базовых классов (base) и производных классов (derived). Для объектов производного класса доступны все открытые данные и открытые методы базового класса.

3. Динамические структуры данных

3.1. Связанные динамические структуры

Динамическими называют такие структуры данных, размеры, связи и местоположение которых меняется в процессе функционирования программы. Доступ к таким данным, как правило, выполняется по ссылкам (указателям).

В практике применения компьютерных технологий встречается большое количество задач, для которых невозможно заранее определить размер памяти для хранения этих данных, их местоположение в памяти, а также и связи между ними. Например, задачи, в основе которых лежат сетевые модели данных, САПР, системы мультимедиа, распределенные базы данных, реализация процессов в сети Интернет и т.д.

Поэтому большой практический и теоретический интерес (см. Д.Кнут, т.1, стр. 296) представляют структуры данных, имеющие связи между отдельными элементами, их называют связанными динамическими структурами данных. Так, например, в памяти компьютера можно создать такое расположение данных, чтобы каждый элемент имел ссылку или указатель на последующий элемент. В результате получаем линейный список со следующей структурой (Рис.8):

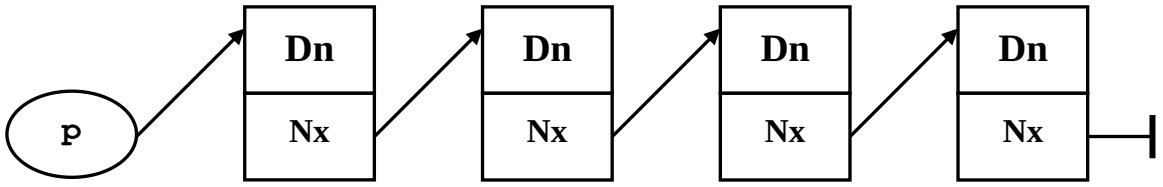


Рис.8. Линейный однонаправленный список.

В памяти компьютера элементы этого списка совершенно необязательно должны располагаться друг за другом, как в массиве, а могут находиться в разных местах этой памяти.

Такие структуры удобно строить на базе структур или классов, в которых возможно объединение полей различных типов.

Мы рассмотрим несколько видов динамических структур, а именно:

1. Линейные: список, стек, очередь, дек;
2. Деревья: бинарные сбалансированные деревья, бинарные деревья поиска;
3. Матрицы, в том числе разреженные;
4. Графы, в основном связанные.

Рассматривая эти виды структур, обязательно определим:

1. Методы, необходимые для реализации доступа к элементам данной структуры;
2. Наиболее известные алгоритмы, определенные на той или иной структуре;
3. Размещение в оперативной памяти структуры данных;
4. Примеры некоторых типичных задач, решаемых с применением той или иной структуры.

3.2. Способы выделения памяти для структур.

В языке **C++** используется несколько функций для выделения памяти под динамические структуры. Перечислим некоторые из них:

1. **`malloc()`** - определяет область памяти под данные;
2. **`free()`** - освобождает область памяти;
3. **`new()`** - выделяется область памяти;
4. **`delete()`** - освобождается память.

Поскольку по окончанию работы программы память должна быть очищена, то на каждый вызов функции **`new()`** необходим вызов функции **`delete().`**

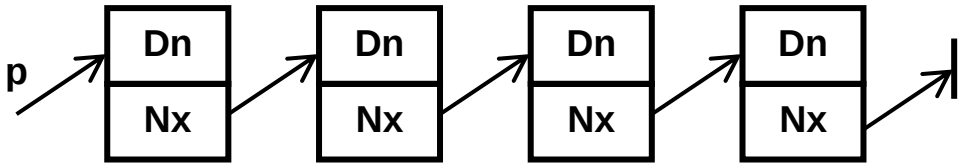
3.3. Линейный список.

Линейный список представляет собой совокупность элементов, связанных друг с другом указателями.

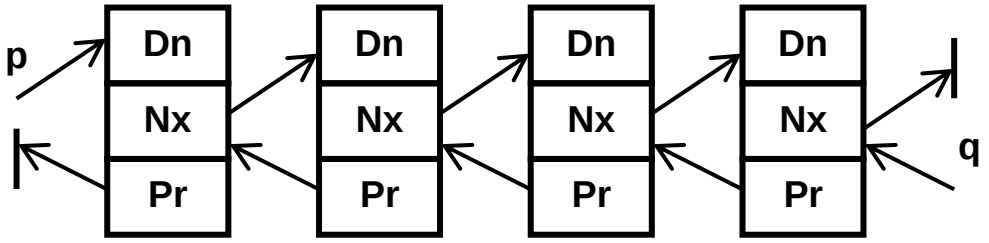
Списки являются базой для целого класса структур, имеющих линейную топологию. Списки различаются по способу доступа к компонентам:

- L1 - однонаправленный список;
- L2 - двунаправленный список;
- LC - кольцевой список.

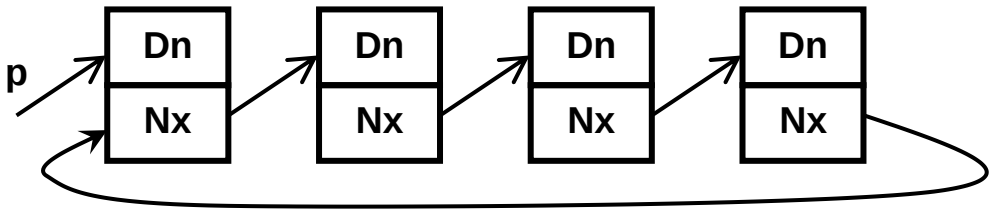
Изобразим их в графическом виде (Рис. 9):



Однонаправленный список – L1;



Двунаправленный список – L2;



Кольцевой список – LC;

Рис. 9. Виды списковых структур.

На базе списковых структур могут быть построены производные структуры, доступ к элементам которых ограничен особыми правилами. Таким структурам даны специальные наименования в связи с их широким применением:

1. Стек (**`Stack`**) – структура, в которой включение и исключение элементов выполняется с одного конца. Доступ к внутренним элементам запрещен. Первый элемент называется вершиной стека. Графическое изображение стека может быть представлено следующим образом (Рис.10):

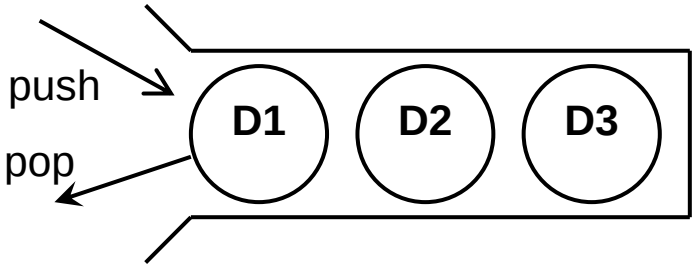


Рис.10. Структура «Стек».

Структура “Stack” функционирует по принципу LIFO (Last Input - First Output), т.е. последним вошел, первым вышел.

2. Очередь (**`Queue`**) – структура, в которой включение элементов выполняется с одного конца, а исключение с другого. Доступ к внутренним элементам запрещен. Графическое изображение очереди может быть представлено следующим образом (рис.11):

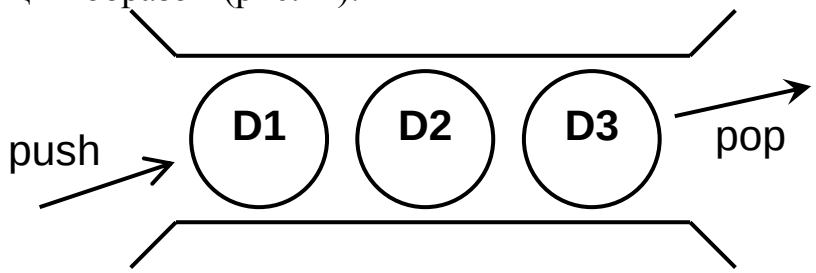


Рис.11. Структура «Очередь».

Структура “Queue” функционирует по принципу FIFO (First Input - First Output) , т.е. первым вошел, первым вышел.

3.Дек (**`Deck`**) – структура, в которой включение и исключение элементов выполняется с двух концов. Доступ к внутренним элементам запрещен. Графическое изображение дека может быть представлено следующим образом (Рис.12):

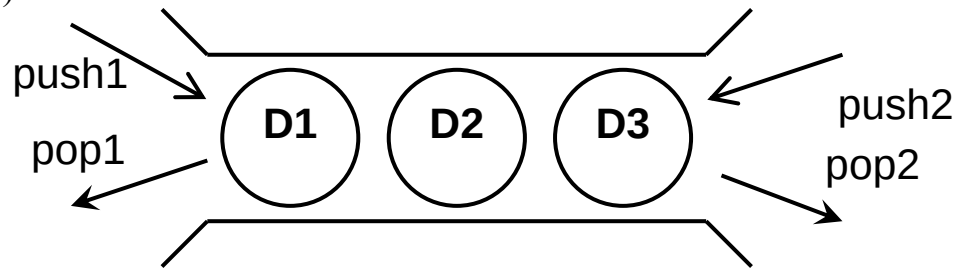


Рис.12. Структура «Дек».

Особенностью построения программ по обработке данных, хранящихся в списковых структурах, состоит в том, что сначала необходимо разработать набор функций, необходимых для решения данной прикладной задачи.

Опишем несколько основных свойств списковых структур:

1. Список может быть пустым. Существуют два варианта построения списка: с головным элементом и без головного элемента. В первом случае методы обработки элементов списка имеют более простую реализацию, а во втором – экономится память.

2. Длина списка конечна, но неограниченна. (Ограничение аппаратное).

3. Доступ к элементам однонаправленного списка возможен только с начала списка по указателю.

4. Связь между номером элемента в списке и его адресом в памяти отсутствует. Поэтому список является структурой с последовательным доступом.

5. Перед выполнением операции необходима проверка ее выполнимости, поскольку в списке может не хватить элементов для этой операции. Проверка может выполняться внешним или внутренним образом. В первом случае, например, с помощью функции Empty():

```
if (L1.Empty()) L1.DelFirst();
```

Во втором случае необходимо возвращать информацию об успешности операции с помощью некоторого параметра. Например:

```
int Method(<список параметров>);
int Swap(int i, int j); //обмен i-того с j-ым элементом
```

6. Преимущество в использовании списковых структур состоит в том, что для модификации списков нет необходимости перемещать в памяти поля данных, а достаточно только менять указатели. Как известно, в практических задачах поля данных очень нагружены, т.е. поля данных имеют большие размеры и могут достигать нескольких мегабайт. Поэтому при операциях модификации списков достаточно изменить содержимое указателя (размером всего 2, 4 или 8 байт), но не перемещать элемент, размер поля данных которого достигает нескольких мегабайт. Это резко повышает быстродействие при операциях вставки, удаления, перемещения и обмена данных, и к тому же скорость работы почти не зависит от размера данных. Особенно это важно в операциях сортировок.

3.4. Пример реализации линейного списка.

```
// Dinamic List
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#define List struct list
List { int Dn; List* Nx;}; //Структура элемента списка
class cList
 { protected: List* p;
   public:
   void Init    (){p=NULL;};
   int  Empty   (){return (p==NULL);};
   void AddHead (int D)
        {List*q=new(List); q->Dn=D; q->Nx=p; p=q;};
   void AddEnd  (int D);
   void Display ();
   void Insert  (int D,int k);
```

```
   void DelFirst(){ List*q=p; p=p->Nx; delete(q); };
   long Len();
   void Done();
 };
void cList::Display()          // Печать списка
   { List* t=p; if(p)while(t){cout<<t->Dn<<" ";t=t->Nx;}
                else cout<<"->|";
     cout<<"\n";
   };

void cList::AddEnd(int D)   //Добавить элемент в список
   { List* q=new(List); q->Dn=D; q->Nx=NULL; List* t=p;
     if (p) { while (t->Nx) t=t->Nx; t->Nx=q;} else p=q;
   };

long cList::Len()           // Длина списка
   {if(p){List*t=p;long i=0;while(t){t=t->Nx;i++;};
    return i;};
    return 0;};

void cList::Done() { while (!Empty()) DelFirst();};

void cList::Insert (int D,int k) // Вставить на k место
   { List*t=p; int i=1;
     if(k==1) AddHead(D);
     else { while (i<k-1) {t=t->Nx;i++;};
     List* q=new(List); q->Dn=D; q->Nx=t->Nx; t->Nx=q;};
   };

//----- Main Program -----------
void main()
 { clrscr();
   cout<<_memavl()<<"\n"; randomize();
   cList L1;
   L1.Init();
   cout<<L1.Empty()<<endl;
   for (int i=1; i<17; i++) L1.AddEnd(random(99));
   L1.Insert(-1,5);   L1.Display();
   L1.DelFirst();     L1.Display();
   L1.Done();         L1.Display();
   cout<<_memavl()<<"\n";
   getch();
 }
```

3.5. Модели доступа к элементам линейного списка

При обращении к данным, которыми нагружены структуры, базовым понятием является интерфейс – это строго определенный набор операций (или методов в терминологии ООП), необходимый для взаимодействия объектов между собой. Таким образом, если объект поддерживает определенный интерфейс, то следует, что он выполняет все, описанные в данном интерфейсе, операции. Поэтому для работы с той или иной структурой данных необходимо создавать функционально полный набор методов для реализации того интерфейса, который необходим для решения конкретной прикладной задачи или класса задач.

Поскольку непосредственный доступ к полям объекта в ООП недопустим, то при разработке набора методов объекта необходимо создание и методов доступа к полям объекта.

В частности, для линейного однонаправленного списка L1 возможно создание таких наборов методов, которые обеспечивают либо последовательный доступ, как в файлах, либо произвольный доступ, как в массиве, т.е. по индексу.

3.5.1. Модель произвольного доступа.

Реализация объектов "Список" (List), в которых доступ к элементам списка выполняется по индексу (произвольный доступ), должна иметь следующий набор полей и методов:

1. В разделе данных описываются переменные для индексирования.
2. В разделе методов описываются все необходимые для произвольного доступа процедуры и функции.

В качестве примера рассмотрим выполнение некоторых операций с моделью произвольного доступа:

```
// Random Access
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>
#include <math.h>
#define List struct list

List { int Dn; List*Nx;};// Элемент списка

class cList                // Класс Список
 { protected: List*p;
   public:
   void Init() {p=NULL;};
   int  Empty() {return (p==NULL);};
   void AddEnd (int D);   // Добавить в конец списка
```

```
   int  GetDn  (long k); // Получить k-тый элемент
   void SetDn  (long k, int D);// Заменить k-тый элемент
   long Len    ();   // количество элементов в списке
   void Insert (long k, int D);// Вставить на k место
   void Del    (long k);      // Удалить k-тый элемент

   void Display();            // Печать списка
   void Done();               // Удаление списка
 };


//-------------------------
 void cList::AddEnd (int D)
      { List*q=new(List); q->Dn=D; q->Nx=NULL; List*t=p;
        if(p) { while (t->Nx) {t=t->Nx;}; t->Nx=q;}
        else p=q;
      };

 long cList::Len()
 { if(p)
    {List*t=p;long i=0;while(t){t=t->Nx;i++;};return i;};
   return 0; };

 void cList::Insert(long k, int D)
   { List*t=p; int i=1;
     if(k==1) {List*q=new(List); q->Dn=D; q->Nx=p; p=q;}
     else { while (i<k-1) {t=t->Nx;i++;};
     List* q=new(List); q->Dn=D; q->Nx=t->Nx; t->Nx=q;};
   };

 void cList::Display()
   { List*t=p; while (t) {cout<<t->Dn<< " ";t=t->Nx; };
    cout<<"\n";
   };

 void cList::Done() { while (!Empty()) DelFirst();};

//-------------------------------------
void main()
 {clrscr();  randomize();  int N=12;
 cout<<_memavl()<<endl;
 cList L1; L1.Init();
 int b=0;
 for(int i=1; i<=N; i++) { L1.AddEnd(i*i);}
 L1.Display();
```

```
 cout<<L1.Len()<<endl;
 L1.Insert(5,-1);
 L1.AddEnd(101);
 L1.Display();
 L1.Done();
 cout<<_memavl()<<endl;
 getch();
}
```

На основе данного набора методов возможна работа со списком, как с массивом переменной длины, причем интересно то, что структура хранения данных, в нашем случае – список, скрыта от пользователя.

3.5.2. Модель последовательного доступа.

Если в модели произвольного доступа необходима переменная, которая играет роль индекса элемента, то в модели последовательного доступа требуется переменная, исполняющая роль текущего указателя, аналогично указателю последовательного файла. Этот указатель сначала устанавливается в начало списка, а затем передвигается все время вперед во время обработки, до конца списка.

Для определения положения указателя в списке необходима функция **EOL (End of List)**. Таким образом, для реализации модели последовательного доступа необходим следующий набор методов:

```
class cListSA                    // Класс Список
 { protected: List*p;
   public:
   void Init();              // Инициализация списка
   int  Empty();             // Пустой ли список
   void GoTop();             // Переход в начало списка
   void GoBottom();          // Переход в конец списка
   int  EOL();               // Достигнут ли конец списка
   void Skip();              // Переход к след. элементу
   void AddEnd (int D);      // Добавить в конец списка
   int  GetData();           // Получить значение элемента
   void Display();           // Печать списка
   void Done();              // Удаление списка
 };
```

Рассмотрим пример модели последовательного доступа в рамках следующей задачи: Дан список целых чисел. Вычислить сумму положительных чисел списка.

```
void main()
 { clrscr();  randomize();  int N=120;
   cList L1; L1.Init();
```

```
   for (int i=1; i<=N; i++) L1.AddEnd(50-random(99));
   L1.GoTop();         // Указатель в начало списка
   long Sum=0;
   while (!EOL())      // Пока не конец списка
     { if (L1.GetData()>=0) Sum+=L1.GetData();
       L1.Skip();     // Переход на следующий элемент
     };
   cout<<"Sum = "<<Sum<<endl;
   L1.Done();          // Удаление списка
   getch();
}
```

3.6. Способы конструирования структур данных

При создании программного обеспечения большое значение приобретает правильное, адекватное решаемой задаче представление данных.

Как правило, структуры данных конструируются на базе конечной совокупности объектов одного и того же типа. Например, матрицы, массивы, стеки, очереди, множества и т.д. Если структуры нагружены данными, то каждый её элемент должен иметь информационное поле соответствующего типа.

Между собой структуры отличаются тем, как именно они работают с элементами. Например, из стека элементы можно получать в порядке, обратном их поступлению в стек, а из очереди, наоборот, в том же порядке, в котором они поступили в очередь.

При создании структур стараются сделать их динамическими, так, чтобы местоположение и количество элементов в них могло изменяться в процессе выполнения программы.

В любой динамической структуре имеются некоторые сходные методы, которые могут быть реализованы в объектах-родителях, но использованы в объектах-наследниках. Например, такие методы, как определение длины списка, очистка элементов списка и т.д.

Давая имена методам из набора, необходимых для решения некоторого класса задач, при их вызове можно считать, что мы используем определенный набор операторов. То есть, задавая набор методов для выбранной или сконструированной структуры данных, мы тем самым создаем проблемно-ориентированный язык для решения задач, ограничивая определенными рамками возможности доступа к элементам структуры.

В качестве иллюстрации к этому положению приведем примеры реализации некоторых структур данных и конструирования для них набора методов.

3.6.1. Структура "Перечисление"

Для объекта, заданного перечислением, должны быть определены операции с упорядоченным дискретным типом:

- Присваивание;
- Сравнение на равенство и неравенство;

- Сравнение по порядку;
- Операции перехода: Pred и Suss;

Например, возьмем такой объект, как tDay. В языке C++ перечисление описывается следующим образом:

```
enum tDay{PN, VT, SR, CH, PT, SB, VS};
```

Однако размер данной структуры ограничен типом. В этой структуре каждое значение имеет свой внутренний номер, начинающий с 0. Порядок нумерации соответствует порядку перечисления, т.е.:

```
PN < VT < SR < CH < PT < SB < VS
```

Поскольку данная структура упорядочена, её элементы можно получить в циклическом процессе:

```
for (tDay t=PN;t<=VS;t++) { cout<<t<<" ";};
```

Анализирую функции, необходимые для обработки этой структуры, приходим к выводу, что наиболее приемлемым для объекта "Перечисление" является двунаправленный список с двумя концевыми указателями **p** и **q**, с последовательным доступом, с возможностью перехода на соседний элемент в прямом и обратном направлении (Рис.13). Для выбора текущего элемента используется указатель **t**.

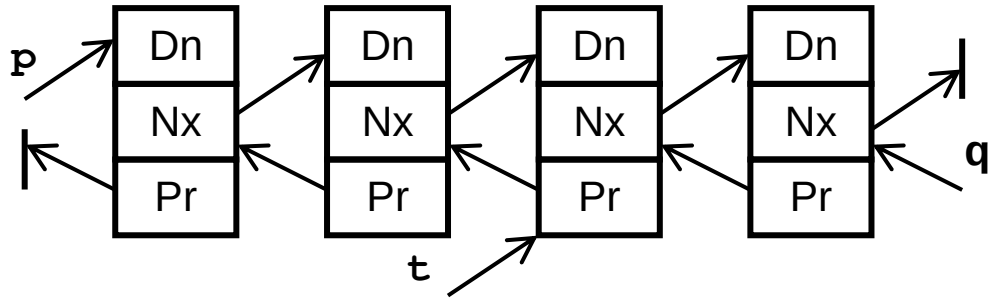


Рис. 13. Двунаправленный список.

Тогда набор методов для этой структуры будет следующий:

```
class cEnum                  // Класс Перечисление
 { protected: List*p; List*q; List*t;
   public:
   void Init();              // Инициализация списка
   int  Empty();             // Пустой ли список
   void GoTop(){t=p;};       // Переход в начало списка
   void GoBottom(){t=q;};// Переход в конец списка
   int  EOL();               // Достигнут ли конец списка
   void Succ(){t=t->Nx;};// Переход к след. элементу
   void Pred(){t=t->Pr;};// Переход к предыд. элементу
   void AddEnd (int D);   // Добавить в конец списка
   int  GetDn(){return t->Dn;};// Получить значение
   void Display();           // Печать списка
   void Done();              // Удаление списка
 };
```

3.6.2. Структура "Множество"

В отличие от других структур, множество является совокупностью неупорядоченных элементов. Кроме того, при выполнении операции "Взять элемент из множества" этот элемент из него удаляется. Операция "Добавить элемент" выполняется, даже если во множестве такой элемент есть. То же и с операцией "Удалить", которая выполняется, даже если во множестве такого элемента нет.

Для создания структуры "Множество" можно использовать линейный динамический однонаправленный список, причем элементы, отсутствующие во множестве, в список не включаются. Использование списка позволяет снять ограничение на размер множества.

Для этой структуры необходимо реализовать следующий набор методов:

```
class cSet                  // Класс Перечисление
 { protected: List*p;
   public:
   void Init();             // Инициализация списка
   int  Empty();            // Пустой ли список
   void PlusSet(i:inf);     // Добавить во множество
   void MinusSet(i:inf);    // Удалить из множества
   int  InSet(i:inf);       // Есть ли элемент во множестве
   void AddEnd (int D);     // Добавить в конец списка
   void Display();          // Печать списка
   void Done();             // Удаление списка
 };
```

Добавление элемента может выполняться в любое место списка, например в начало списка, или в конец списка, но только в случае отсутствия такого элемента в списке.

Тем не менее, при добавлении, извлечении, поиске элемента приходится просматривать весь список, чтобы определить, содержит ли множество этот элемент. Т.е. используется неэффективный линейный поиск. Более эффективную модель "Множество" можно построить на динамической структуре "Дерево".

На основе полученной структуры возможно создание таких производных методов, как "Объединение множеств", "Разность множеств", "Пересечение множеств", "Проверка на равенство множеств" и т.д.

3.6.3. Структура "Стек".

Стек работает по принципу LIFO. В любой момент времени мы можем изъять из стека только один элемент – верхний.

Стек широко используется для реализации системных и прикладных программ, например в системе прерываний в ОС, рекурсивные процессы, преобразование формульных выражений, создание локальных переменных и т.д.

Существует целый ряд языков программирования, основанных на стековых структурах, наиболее известным из них является язык FORTH.

Объект Stack описывается следующим набором данных и методов:

```
class cStack                // Класс Стек
 { protected: List*p;
   public:
   void Init();             // Инициализация списка
   int  Empty();            // Пустой ли список
   void Push(int D);        // Добавить элемент в стек
   int  Pop();              // Взять элемент из вершины
   void Display();          // Печать стека
   void Done();             // Удаление стека
 };
```

Пример решения задачи с использованием структуры "Stack":

Дана последовательность положительных целых чисел. Удалить из последовательности нечетные числа, используя структуры "Stack".

Задача может быть решена следующим образом.

Создадим два стека. В первый стек запишем последовательность целых чисел. Затем из него будем извлекать число, и определять, является ли оно четным. Если является, то перепишем его во второй стек, иначе переписывать не будем. Процесс продолжится до тех пор, пока первый стек не опустеет. Нам останется только переписать числа из второго стека в первый и напечатать результат.

```
// Stack
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#define Stack struct stack

Stack { int Dn; Stack* Nx;};

class cStack
  { protected:  Stack* p;
    public:
    void Init (){p=NULL;};
    int  Empty(){return (p==NULL);};
    void Push (int D)
           {Stack*q=new(Stack); q->Dn=D; q->Nx=p; p=q;};
```

```
    int  Pop  ();
    void Display();
    void Done();
  };


    int  cStack::Pop()
     { Stack*q=p;
       p=q->Nx; int B=q->Dn; delete(q); return B;
     };

    void cStack::Done() { while (!Empty()) Pop();};

    void cStack::Display()
     { Stack*t=p;
       if (p)
       while (t) {cout<<t->Dn<<" "; t=t->Nx;}
       else cout<<"->|";
       cout<<"\n";
     };
//----- Main Program -----------
void main()
 { clrscr();
   cStack St1; cStack St2;
   St1.Init(); St2.Init();
   for (int i=1; i<17; i++) St1.Push(random(99));
   St1.Display();
   while (!St1.Empty())
     { int Buf = St1.Pop();
       if (Buf%2==0) St2.Push(Buf);
     };
   while (!St2.Empty()) St1.Push(St2.Pop());
   St1.Display();
   St1.Done();  St2.Done();
   getch();
 }
```

3.6.4. Структура "Очередь"

Очередь (**Queue**) работает по принципу FIFO. В любой момент времени мы можем изъять из очереди только один элемент. Размер очереди меняется во время выполнения программы.

Очередь широко используется для реализации системных и прикладных программ, в тех алгоритмах обработки данных, когда данные надо извлекать в том же порядке, в котором они поступили.

Отличие от стека состоит в том, что добавление в очередь происходит, как и у стека, в голову списка, а извлечение, в отличие от стека, с конца списка.

Объект **Queue** описывается следующим набором данных и методов:

```
class cQueue                // Класс Стек
 { protected: List*p;
   public:
   void Init();             // Инициализация списка
   int  Empty();            // Пустой ли список
   void Push(int D);        // Добавить элемент в стек
   int  Pop();              // Взять элемент из вершины
   void Display();          // Печать стека
   void Done();             // Удаление стека
 };
```

Пример решения задачи с использованием структуры "**Queue**":

Дана некоторая конечная последовательность символов, в которой цифр и малых латинских букв содержится одинаковое количество, например:

"ab56cd143acd78"

Необходимо создать последовательность, в которой эти же буквы и цифры образуют чередование, без изменения порядка их следования:

"a5b6c1d4a3c7d8"

Задача может быть решена следующим образом (Рис.14).

Создадим две очереди, **Qu1** и **Qu2**. Из входного потока буквы будем записывать в первую очередь **Qu1**, а цифры во вторую очередь **Qu2**. По окончании потока будем попеременно извлекать символы из первой и второй очереди, до тех пор, пока они не опустеют, и направлять эти символы в выходной поток.

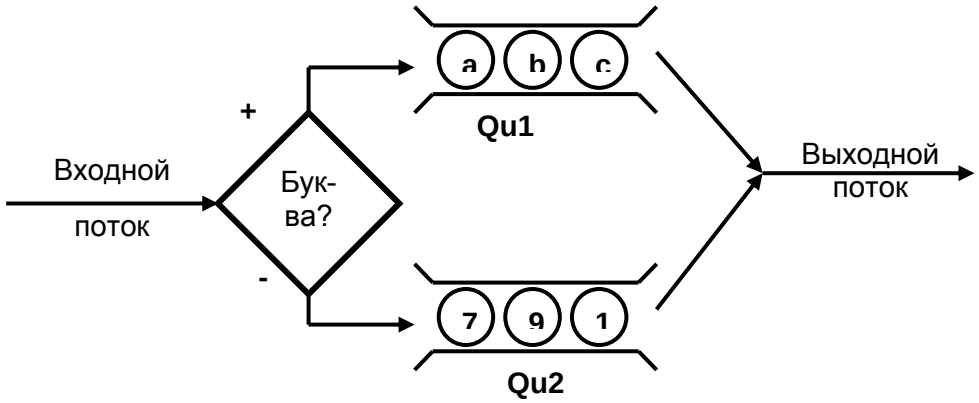


Рис.14. Диаграмма алгоритма.

```
void main()
{ FILE *fin; FILE *fout; char fname[20]; char st[80];
  cQueue Qu1; cQueue Qu2; Qu1.Init(); Qu2.Init();
```

```
  int i;
  cout<<"file name = "; cin>>fname;
  if ((fin=fopen(fname,"rt"))==NULL)
     {printf("Error");getch();return;};
  fout=fopen("d_rez.cpp","wt");

  while (!feof(fin))
    { fgets(st,80,fin);
      for (i=0; st[i]; i++)
        {if ((st[i]>=48)&&(st[i]<=57)) Qu1.push(st[i]);
         else Qu2.push(st[i]);
     };
    };
  while ((!Qu1.empty())||(!Qu2.empty()))
    { fprintf(fout,"%c",Qu1.pop());
      fprintf(fout,"%c",Qu2.pop());}
  fprintf(fout,"\n");
  Qu1.Done(); Qu2.Done();
  fclose(fin);     fclose(fout);
  getch();
}
```

3.6.5. Формальные языки и грамматики.

Краткое определение языка.

Пусть A - некоторый алфавит, т.е. произвольное непустое множество. Элементы этого множества назовем символами. Произвольная конечная последовательность символов алфавита (в том числе и пустая) называется цепочкой.

Произвольное подмножество LA множества всех возможных цепочек называется языком над A. Над одним и тем же алфавитом можно определить много различных языков.

Пусть **`A = 0,1`** - алфавит из двух символов. Тогда можно определить такие языки, как язык пустых цепочек, язык непустых цепочек, цепочек, начинающихся с 1 и т.д.

ПРИМЕР. Пусть определен язык LA, состоящий из 26 строчных латинских букв, двух скобок и 4-х знаков арифметических действий:

`A = a,b,c,d,...z,(,),+,-,*,/`

Примем ограничение: в цепочках используем только односимвольные переменные a,b,c,...z, а также запретим унарные операции.

Тогда цепочки: **`'a', 'b-a*c', '(((a+b)*(c-d))-a)/(e+f)'`** будут формулами, т.е. принадлежать языку LA.

Другие цепочки: **`'))+a***', 'aa+b', '*a'`**, языку LA не принадлежат. Но до сих пор правильность формулы, т.е. принадлежность языку LA, определяется только интуитивным представлением.

Но например: **delta, a+(-d), (((a))),()-** являются ли эти цепочки формулами ?

Поэтому сложные языки задаются не словесно, в формально, с тем, чтобы принадлежность цепочки можно определить или опровергнуть, исходя из определения, т.е. заданного в виде /грамматики/.

Грамматикой называется система правил (т.е. конечная последовательность строк), записанных с помощью метасимволов.

Содержательно каждое правило грамматики имеет смысл подстановки.

Например, **alpha --> beta, alpha --> alpha gamma alpha.**

ПРИМЕР: Пусть A - алфавит, над которым рассматривается /язык/ правильных арифметических формул: M = alpha,beta,gamma, где alpha- главный метасимвол.

Тогда грамматика:

```
alpha-->(alpha)                beta-->a      gamma--> +
alpha-->beta                   beta-->b      gamma--> -
alpha-->alpha gamma alpha      beta-->c      gamma--> *
                               . . . . .
                               beta-->d
                               beta-->z
```

Т.е. строка **alpha-->(alpha gamma alpha)** означает возможность замены метасимвола **alpha** на цепочку **alpha gamma alpha.** Тогда подстановка порождает множество цепочек, например:

alpha-->(alpha)-->(alpha gamma alpha)-->(beta gamma beta)-->(x+y)

Обычные символы: **f,b,c,d,...z,+,-,*** и т.д. называются терминалами, а метасимволы - нетерминалами. Все множество цепочек над A, которые можно получить с помощью грамматики, называется языком.

В данной интерпретации метасимволы можно понимать как имена понятий. Поэтому удобней их представлять в угловых скобках, как в БНФ.

Известно, что одной из форм грамматики является нормальная форма Бэкуса-Наура или НФБН, или БНФ. В нотации БНФ метасимволы обозначаются понятиями в угловых скобках, знаком объединения служит **::=**, варианты определяются вертикальной чертой **|.**

Пусть **alpha** - обозначает правильную формулу, **beta** - имя переменной, **gamma** - знак операции. Тогда можно определить некоторую грамматику, задающую язык арифметических формул:

```
<формула>::=(<формула>)|<имя перем>|<формула><знак> <формула>
<имя перем>::=a|b|c|d|..|z
<знак>::=+|-|*|
```

Тогда для анализа строки символов, определяющего, является ли данная строка правильной формулой, можно использовать следующую подстановку: **((x+y)-a)-->((beta gamma beta) - beta)-->(alpha gamma alpha)-->alpha.** Если возможно путем подстановки достичь главного метасимвола, то данная строка символов является правильной формулой.

В качестве примера приведем БНФ-запись грамматики формулы. Дана строка символов, заканчивающаяся точкой. Определить, является ли она правильной формулой. Для данной формулы задана следующая грамматика:

```
<Form>::=<Cifra>|(<Form><Znak><Form>)
<Cifra>::=0|1|2|3|4|5|6|7|8|9
<Znak>::=+|-|*|
```

3.8.2. Стековый калькулятор.

В качестве примера использования рассмотрим конструирование стекового калькулятора. В основе работы СК лежит преобразование обычной формульной записи в обратную польскую (постфиксную).

Поясним это на примере.

Пусть мы имеем выражение: **(a+b/c)*(d-e*f)**

Существуют 3 вида упорядочения:

1. Код операции, 1-й операнд, 2-й операнд, (КОП, X, Y) – префиксная форма: *** + a / b c – d * e f**

2. 1-й операнд, код операции, 2-й операнд, (X, КОП, Y) – инфиксная форма: **a + b / c * d – e * f**

3. 1-й операнд, 2-й операнд, код операции, (X, Y, КОП) – постфиксная форма: **a b c / + d e f * – ***

Обратная польская запись формулы позволяет реализовать простой алгоритм вычисления по формуле.

Например, имеем формулу вида: **(((A+B)*C)-(D+H)**.

Представив эту формулу в постфиксной записи, получим:

A B + C * D H + -

Алгоритм вычислений по этой формуле выглядит следующим образом: Последовательность операндов и операций записываются в стек в постфиксной форме. Затем из стека извлекается группа из двух операндов и одной операции и выполняется вычисление, результат которого записывается в стек вместо этой группы. Этот процесс заканчивается, когда в стеке остается одно число, которое и является окончательным результатом.

```
A B +
AB + ) C *
D H +
( A B + ) C * )( D H + ) -
```

Рассмотрим один из алгоритмов формирования обратной польской записи этого арифметического выражения и вычисления с использованием стековых структур.

Для работы требуется два стека: **St1** и **St2**.

Во время просмотра выражения левые скобки пропускаются и не рассматриваются. Идентификаторы A,B,C,D,H отправляем в стек ST1, знаки операций в стек St2.

Если встречается правая скобка, то знак операции из St2 переписываются в St1.

```
St1:  [ A B + C * D H + -
St2:  [     +   *     - +
```

Затем все переписываем в стек St2, получаем:

```
St2:  [ - + H D * C + B A
```

Реализация:

```
     . . . . . .
ch:=' ';
while ch <> '.' do
  begin
    read(ch);
    case ch of
     'A'..'Z':   St1.Push(ch);
     '+','-','*':St2.Push(ch);
     ')':        St1.Push(St2.Pop);
    end {case};
  end;
while not St1.Empty do St2.Push(St1.Pop);
     . . . . . .
```

Во время вычислений выполняется подстановка значений переменных.

```
     . . . . . .
X:=St2.Pop; Y:=St2.Pop; Op:=St2.Pop;
iX:=ord(X)-48; iY:=ord(Y)-48;
case Op of
     '+':R:=X+Y;
     '-':R:=X-Y;
     '*':R:=X*Y;
end {case};
. . . . . .
```

Стековый калькулятор используется при построении компиляторов, интерпретаторов, при создании вычислительных алгоритмов.

3.7. Конструирование сложных динамических структур

3.7.1. Массивы данных.

Ранее рассмотренные линейные структуры могут быть основой для конструирования более сложных структур данных, необходимых для решения того или иного класса задач.

Например, для текстового редактора наиболее применима конструкция "Двунаправленный список двунаправленных списков", позволяющая хранить символы текста в строках. Для решения задач обхода графов часто требуются структуры "Стек стеков" и "Стек очередей". Для решения сложных задач могут создаваться и трехмерные структуры, например "Список списков очередей" и т.п.

В любом случае для решения конкретного класса задач необходимо сконструировать структуру, наиболее полно отображающую решаемую проблем-

ную ситуацию. Затем, на базе созданной структуры необходимо реализовать набор методов, представляющих в некоторой степени проблемно-ориентированный язык. И только после этого можно приступить к реализации программы.

В качестве примера рассмотрим конструирование структуры "Однонаправленный список однонаправленных списков" **(L1 to L1)** для хранения данных некоторого целочисленного двухмерного массива. По вертикали расположим элементы основного списка (стволового - **"Main"**), а по горизонтали элементы подсписков (**"Sub"**). В данной структуре используются элементы двух видов: **"ListM"** и **"ListP"**. Элемент **"ListM"** состоит из трех полей: **Dm, Nm** и **Np**, а элемент **"ListP"** состоит из двух полей: **Dp** и **Np**. Чтобы компилятор при создании элемента **"ListM"** основного списка **"Main"** знал о типе указателя на элемент **"ListP"** подсписка **"Sub"** необходимо **"ListP"** определить первым.

Введем следующие допущения:

1. Добавление элементов в основной список **"Main"** и подсписок **"Sub"** будем выполнять только в начало списков.
2. Считывание данных из списков выполним, используя произвольный доступ, т.е. по индексам: **i** - для строк **("Main")**, **j** - для столбцов **("Sub")**.
3. Все данные хранятся в подсписках **"Sub"**.
4. Поля **"Dm"** основного списка используются как заголовочные, например, для записи номера строки.
5. Проверку выполнимости операций будем производить внешним образом.

Теперь опишем некоторый минимальный набор методов для работы с этой структурой:

```
int  Empty   ();          // Пуст ли список
void AddMain (int D);     // Добавить в основной
void AddSub  (int D, int i);  // Добавить в подсписок
int  GetDp   (int i, int j);  // Получить значение
```

На этой основе реализуем программу, позволяющую обрабатывать сложную структуру "Список списков", предназначенную для хранения двухмерного массива.

```
// Динамическая структура «List to List»
// p --> [Dm]
//       [Np] --> [Dp|Np] --> [Dp|Np] --> NULL
//       [Nm]
//        |
//       [Dm]
//       [Np] --> [Dp|Np] --> NULL
```

```
//       [Nm]
//        |
//       [Dm]
//       [Np] --> [Dp|Np] --> [Dp|Np] --> [Dp|Np] -->NULL
//       [Nm]
//        |
//       NULL

#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>

#define ListP struct listp
#define ListM struct listm

ListP {int Dp; ListP*Np;};   // - Sub List --
ListM {int Dm;
       ListM*Nm;
       ListP*Np;};           // - Main List -

class cL2L
  { protected: ListM* p;
    public:
    void Init    () {p=NULL;};
    int  Empty   () {return (p==NULL);};
    void AddMain (int D);
    void AddSub  (int D, int i);
    int  GetDp   (int i, int j);
    void Display ();
    void Done();
  };

    void cL2L::Display()          // Печать структуры;
     { ListM* t=p; ListP* q;
       while (t)
        { printf("[%3i]",t->Dm);
          q=t->Np;
          while (q) {printf("%4i",q->Dp); q=q->Np;};
          t=t->Nm;
          printf("\n");
        };
     };
```

```
    void cL2L::AddMain (int D)    // Добавить в главный;
     {ListM* q=new(ListM);
       q->Dm=D; q->Np=NULL; q->Nm=p; p=q;
      };

    void cL2L::AddSub(int D, int i)//Добавить в подсписок
     { ListP* q=new(ListP); q->Dp=D;
       ListM* t=p;
       int k=1;
       while (k<i) {t=t->Nm; k++;};
       q->Np=t->Np; t->Np=q;
     };

    int cL2L::GetDp (int i, int j)    //Получить элемент;
     { ListM* t=p;
       int ii=1;
       while (ii<i) {t=t->Nm; ii++;};
       int jj=1;
       ListP* q=t->Np;
       while (jj<j) {q=q->Np; jj++;};
       return q->Dp;
     }

    void cL2L::Done ()            //Удалить структуру;
     { ListM* t; ListM* s; ListP* q;
       t=p;
       while (t)
         { while (t->Np)
              { q=t->Np; t->Np=q->Np; delete(q);};
           t=t->Nm;
         }
       while (p) { t=p; p=p->Nm; delete(t);}
     }

//----- Main Program -----------
void main()
 { clrscr();
   cL2L LL1;  LL1.Init();     // Инициализация структуры;
   LL1.AddMain(44); LL1.AddMain(33);   LL1.AddMain(22);
   LL1.AddMain(11);
   LL1.AddSub(-3,1);LL1.AddSub( 15,2); LL1.AddSub( -8,3);
   LL1.AddSub(17,2);LL1.AddSub( 25,2); LL1.AddSub(-19,1);
   LL1.Display();
   cout<<LL1.GetDp(2,3)<<endl;//Получить элемент i=2,j=3;
```

```
    LL1.Done();
    getch();
}
```

3.7.2. Разреженные матрицы.

Второй пример иллюстрирует методику "экономного" размещения данных в памяти компьютера на примере так называемых "разреженных" матриц.

"Разреженными" называют матрицы высоких порядков, большинство элементов которых равны нулю или не значимы (отсутствуют). В этом случае необходимо создать такую структуру данных, чтобы не занимать память незначимыми элементами[1].

Например, при обычном распределении данных в массиве небольшой по размерам матрицы уже требуется 200 x 200 = 40000 элементов, что очень много.

Одним из вариантов реализации может быть предложена структура из циклически связанных списков для каждой строки и каждого столбца матрицы. На пересечении строки и столбца, т.е. в узлах находятся элементы, состоящие из пяти полей.

Left		**Up**
ROW	**COL**	**Dn**

Здесь: **Left** - ссылка на левый элемент, **Up** - ссылка на верхний элемент, **ROW** - номер строки, **COL** - номер столбца, **Dn** - поле данных.

Структура подобных матриц включают в себя специальные головные элементы для каждой строки и каждого столбца. Если в некоторой строке матрицы элементов нет, то вся строка отсутствует, то же и для столбцов. Поэтому количество занимаемой памяти для разреженной матрицы может быть небольшим. Схема разреженной матрицы показана на рисунке 3.7.2. Указатель **p1** указывает на головные элементы строк, а **p2** – столбцов (Рис.15).

[1] Дональд Кнут. Искусство программирования для ЭВМ. М.Мир. 1976 г. стр. 375.

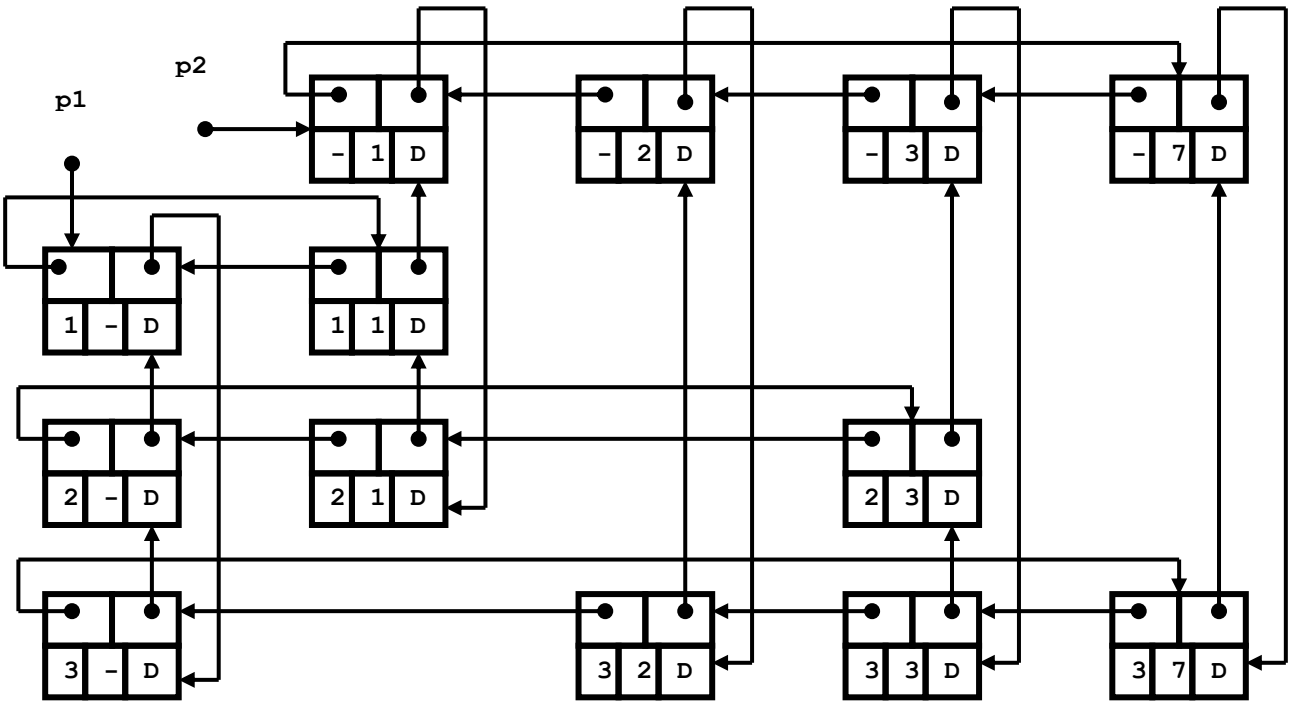


Рис. 15. Структура "разреженной" матрицы.

Примеры использования данной формы достаточно обширны: алгоритмы решения линейных уравнений, обращение матриц и линейное программирование (симплекс метод). Из известных программных средств такую форму представления имеют электронные таблицы SuperCalc, Lotus 1-2-3, MS Excel и др.

Для работы с разреженными матрицами необходимо определить модель доступа и разработать набор методов.

Например, выберем модель произвольного доступа (т.е. с использованием индексов). Тогда набор методов может быть следующий:

```
int  GetRM(int i, int j);
void PutRM(int i, int j, int D);
void DelRM(int i, int j);
```

Метод **GetRM** (Получить значение элемента) проверяет наличие элемента в матрице, и если он существует, то возвращается его значение, иначе возвращается незначимое значение, например 0.

Метод **PutRM** (Включить элемент в матрицу) проверяет наличие элемента в матрице, и если его там нет, то элемент создается, и в него записывается значение **D**. Создается, если это необходимо, вместе с головными элементами. Если элемент с такими координатами есть, то просто меняется его значение.

Метод **DelRM** (Удалить элемент из матрицы) проверяет наличие элемента в матрице, и если он имеется, то этот элемент удаляется. Если этот элемент был единственный в строке, то удаляется и головной элемент этой строки, то же происходит и со столбцами. Если же удаляемый элемент в структуре отсутствует, то никаких действий не выполняется.

Для более подробного ознакомления с особенностями реализации разреженных матриц рекомендуется прочитать:

Кнут Дональд, Искусство программирования для ЭВМ М.:Мир, 1976.
Влах И., Сингхан К. Машинные методы анализа и проектирования электронных схем М:РиС, 1988.

4. Сложные структуры данных - деревья

4.1. Несколько определений.

Многие структуры данных иногда удобно определять рекурсивно.

Например, список или последовательность с базовым типом **`L`** это:

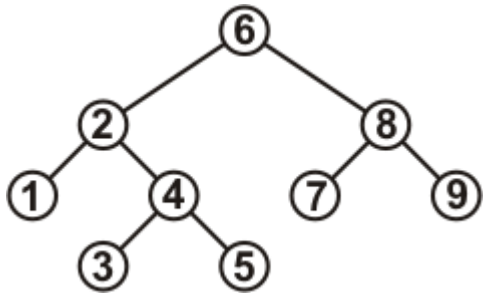
- либо пустая последовательность;
- либо соединение (конкатенация) элемента типа **`L`** и некоторой последовательности того же типа.

Дерево можно определить таким же образом. Дерево с базовым типом **`T`** это:

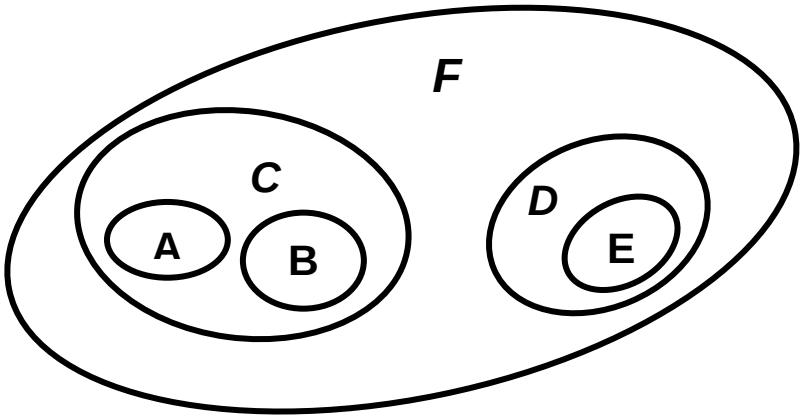
- либо пустое дерево;
- либо некоторая вершина типа **`T`** с конечным числом связанных с ней отдельных деревьев с базовым типом **`T`**, называемых поддеревьями.

Существуют несколько способов изображения деревьев:

1. В виде графа:



2. В виде выражения со вложенными скобками: **`(F(C(A,B),D(E))).`**
3. В виде вложенных множеств:



Дадим несколько определений.

1. Упорядоченным называют дерево, у которого ребра (ветви), исходящие из каждой вершины, упорядочены.

2. Вершина А, которая находится ниже вершины С, называется потомком, а С в этом случае – предком.

3. Если вершина не имеет потомков, ее называют терминальной или листом.

4. Считается, что корень дерева находится на уровне 0. Максимальный уровень дерева называют глубиной или высотой дерева.

5. Максимальное количество потомков у некоторой вершины называют степенью дерева.

6. Число ветвей, которые нужно пройти от корня до вершины А, называют длиной пути к А. Корень имеет длину пути, равную 0.

4.2. Бинарные деревья.

Бинарные или двоичные деревья имеют степень, равную 2. Каждая вершина такого дерева содержит либо пустые ссылки, либо ссылки на поддеревья. Деревья степени более двух, называют сильно ветвящимися деревьями.

Примеры двоичных деревьев:

- родословная или генеалогическое дерево;
- футбольный или теннисный турнир;
- арифметическое выражение;
- и множество других примеров.

Деревья также различаются по способу построения на деревья поиска и сбалансированные деревья.

4.3. Деревья поиска.

Дерево поиска определяется следующим образом: Для каждой вершины t_i все ключи левого поддерева меньше по значению ключа вершины, а ключи правого поддерева больше его. Дерево поиска позволяет выполнить поисковые операции с числом шагов **~ log2N**, если оно сбалансировано и имеет **N** вершин. Деревья поиска подразделяются на 2 вида:

- повторные элементы во входном потоке игнорируются, тогда это простое дерево поиска;
- в состав каждой вершины дерева включается счетчик, определяющий количество вхождений каждого значения из входного потока. Такое дерево называют деревом поиска со счетчиком.

Рассмотрим пример построения простого дерева поиска. Пусть входной поток содержит следующие элементы:

8, 3, 10, 1, 6, 14, 5, 7, 13.

Тогда корень дерева будет иметь значение – 8. Следующий элемент – 3, поэтому он включается в левое поддерево. Очередной элемент – 19 больше 8, поэтому он включается в правое поддерево. Таким же образом включаются и остальные элементы (Рис.16).

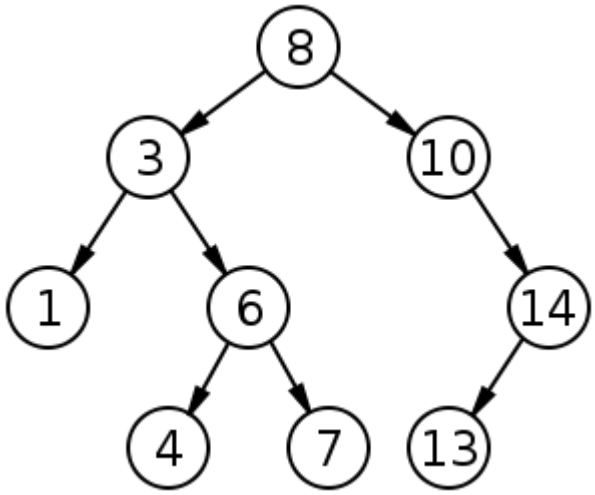


Рис. 16. Структура дерева поиска.

Программа построения деревьев поиска реализована на основе рекурсивных алгоритмов с использованием объектно-ориентированной технологии.

```
// Dinamic BinaryTree
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>

class btree
 { protected: int Dn; int B; btree* Ln; btree* Rn;
   public:
   void Init(){B=1;};
   void Add(int D);
   void Pri(int k);
   void Done();
 };

 void btree::Add(int D)    // Добавить в дерево очередной элемент
   { if (B)
     {B=0; Dn=D; Ln=new(btree); Ln->Init();
          Rn=new(btree); Rn->Init();}
     else {if (D<Dn) Ln->Add(D); else Rn->Add(D);};
   };

 void btree::Pri(int k)          // Печатать дерево поиска
   { if(!B)
     { Ln->Pri(k+4);
       for(int i=1;i<=k;i++)cout<<" ";
```

```
        cout<<Dn<<" "<<endl;
        Rn->Pri(k+4);
      };
   };

 void btree::Done()              // Удаление дерева поиска
   { if(!B)
     {Ln->Done(); delete(Ln);
      Rn->Done(); delete(Rn); };
   };

//----- Main Program -----------
void main()
 { clrscr(); randomize();
   cout<<_memavl()<<"\n";        // Количество свободной памяти
   btree T1;
   T1.Init();                    // Инициализация дерева
   T1.Add(50);
   for (int i=1; i<20; i++) T1.Add(random(99));
   T1.Pri(1);                    // Печать дерева
   T1.Done();                    // Удаление дерева поиска
   cout<<_memavl()<<"\n";
   getch();
 }
```

Часто в задачах по обработке данных, нагруженных на древовидные структуры, требуется определить максимальный уровень дерева. Алгоритм основан на обходе всех вершин дерева, причем при переходе на более низкий уровень происходит сравнение текущего уровня с ранее достигнутым максимальным. Если текущий уровень окажется больше, тогда максимальный уровень заменяется на текущий. После посещения всех вершин метод возвращает максимально достигнутый уровень.
Пример программы, в которой определяется максимальный уровень дерева.

```
// Максимальный уровень дерева
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>

class btree
 { protected:
```

```
    int Dn;
    int B;
    btree* Ln;
    btree* Rn;
  public:
  void Init(){B=1;};
  void Add(int D);
  void Pri(int k);
  void Done();
  int  MLevel(int &mL, int u);
};

void btree::Add(int D)       // Добавить в дерево
  { if (B)
    {B=0; Dn=D; Ln=new(btree); Ln->Init();
        Rn=new(btree); Rn->Init();}
    else { if (D<Dn) Ln->Add(D); else Rn->Add(D);};
  };

void btree::Pri(int k)       // Печать дерева
  { if(!B)
    { Ln->Pri(k+4);
      for(int i=1;i<=k;i++)
           cout<<" "; cout<<Dn<<" "<<endl;
      Rn->Pri(k+4);
    };
  };

void btree::Done()           // Удаление дерева
  { if(!B) { Ln->Done(); delete(Ln);
             Rn->Done(); delete(Rn); };
  };

int btree::MLevel(int &mL, int u) //максимальный уровень
  { if(!B) { if (u>mL) mL=u;
         Ln->MLevel(mL,u+1);
         Rn->MLevel(mL,u+1);
       };
    return mL;
  };
//----- Main Program -----------
void main()
 { clrscr(); randomize(); int mL=0;
   cout<<_memavl()<<"\n";
   btree T1;  T1.Init();
```

```
    T1.Add(50);
    for (int i=1; i<12; i++) T1.Add(1+random(99));
    T1.Pri(1);
    cout<<T1.MLevel(mL,0)<<endl;
    T1.Done();
    cout<<_memavl()<<"\n";
    getch();
}
```

4.4. Создание дерева-формулы.

Поскольку грамматика построения формул позволяет создавать многоуровневые скобочные выражения, то имеется возможность представить формулу в виде дерева. Построение дерева-формулы ведется следующим образом:

- Каждая открывающая скобка создает левое поддерево;
- Каждый операнд создает вершину дерева и заполняет ее значением операнда;
- Каждый операнд записывается в текущую вершину;
- Каждая правая скобка перемещает текущую вершину на шаг вверх.

Например, для выражения: **((5+3)*7)** строится следующее дерево – формула (Рис.17):

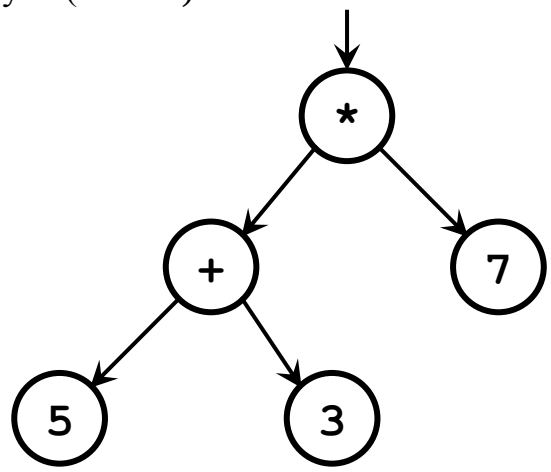


Рис.17. Структура дерева – формулы.

4.5. Операции обхода вершин дерева.

Наряду с операцией построения дерева очень важной является и операция обхода дерева, т.е. посещения всех вершин дерева по одному разу.

Существует три способа обхода дерева, которые можно выразить в терминах рекурсии. Обозначим текущую вершину буквой R, левое поддерево – A, правое поддерево – B. Порядок обхода может быть обозначен следующим образом:

1. Сверху-вниз (R, A, B) – префиксный. Сначала посещаем корень, затем левое поддерево, а потом – правое.

2. Справа-налево (A, R, B) – инфиксный. Порядок посещения: левое поддерево, корень, правое поддерево.
3. Снизу-вверх (A, B, R) – постфиксный. Сначала посещаются поддеревья, затем корень.

В качестве примера рассмотрим обход дерева поиска на рис. 18:

В результате получим следующие упорядоченные последовательности:

1. **(R, A, B) – [* + 5 3 7]** – Префиксный способ обхода.
2. **(A, B, R) – [5 + 3 * 7]** – Инфиксный способ обхода.
3. **(A, B, R) – [5 3 + 7 *]** – Постфиксный способ обхода.

Порядок обхода можно менять, переставляя в методе **Pri(int k)** расположение рекурсивных вызовов. В результате, применяя постфиксный способ обхода (A, B, R) дерева – формулы, получим обратную польскую запись формулы, которая в дальнейшем может быть использована для вычислений с помощью стекового калькулятора.

Другой пример рассматривает способы обхода дерева поиска, построенного на основе числовой последовательности:

35 83 78 31 67 69 12 27 98 33

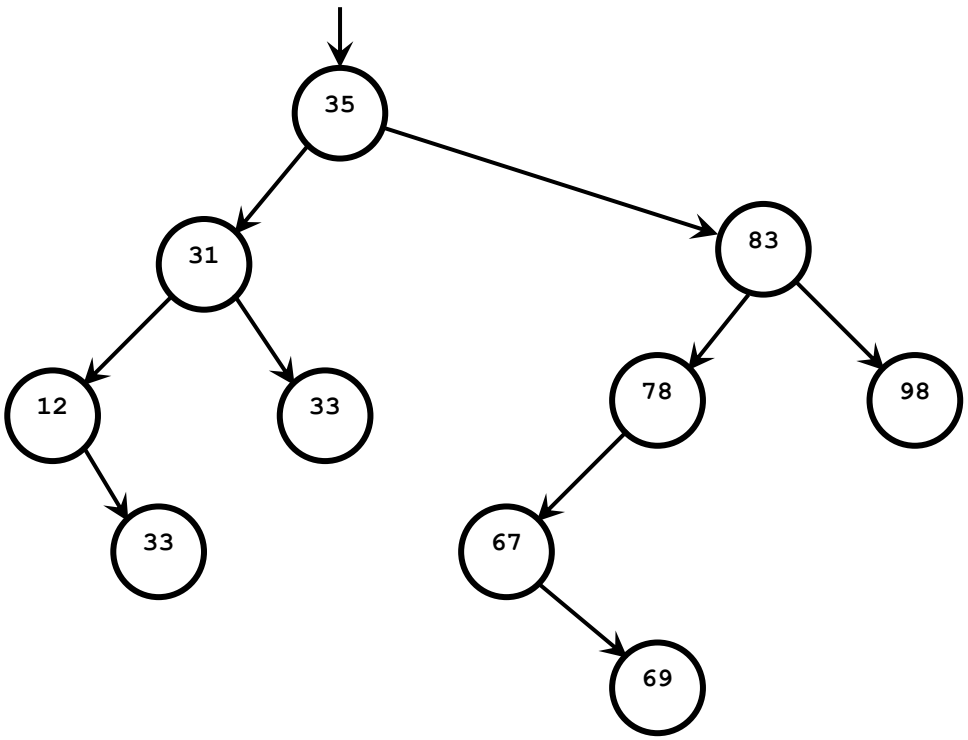


Рис.18. Методы обхода дерева поиска.

Выполним обход дерева тремя способами:

1. (R, A, B) – Префиксный способ обхода:
   35 31 12 27 33 83 78 67 69 98.
2. (A, B, R) – Инфиксный способ обхода:
   12 27 31 33 35 67 69 78 83 98.

3. (A, B, R) – Постфиксный способ обхода:

27 12 33 31 69 67 78 98 83 35.

Отметим, что инфиксный способ обхода дерева возвращает упорядоченный список данных. Этот способ является основой для создания эффективных алгоритмов сортировки, требующих на весь процесс количества операций, пропорциональных $\sim N*\log_2 N$.

4.6. Сбалансированные деревья.

Различают сбалансированные и несбалансированные деревья.

Определение 1. Дерево идеально сбалансировано, если для каждого его узла количество узлов в левом и правом поддереве различается не более чем на единицу. Отметим, что эффективный поиск возможен только в сбалансированном дереве поиска.

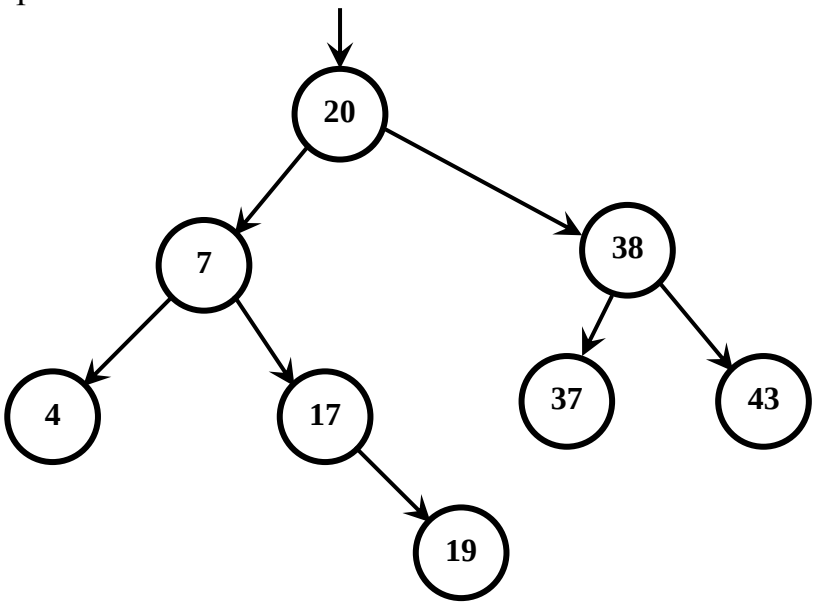


Рис.19. Идеально сбалансированное дерево.

Поскольку при добавлении нового элемента в дерево баланс может нарушиться, то для его сохранения применяют алгоритмы балансировки. Эти алгоритмы меняют взаимное расположение элементов так, чтобы сохранялось условие существования дерева поиска и, в то же время, условие сбалансированности.

Рис 20. Пример балансировки дерева.

Операции балансировки достаточно сложны, и следовательно, для их выполнения требуется затратить значительное машинное время и ресурсы, особенно для больших по объему структур данных.

Одним из возможных выходов из этого положения может быть введение менее строгого условия сбалансированности. В результате это позволит получить более простой набор операций для переупорядочения элементов. Одно из таких условий сбалансированности было предложено в 1962 году советскими учеными Адельсоном-Вельским и Ландисом.

Определение 2. Дерево называется сбалансированным тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более, чем на 1.

Такие деревья называются AVL-деревья, по имени авторов. Отметим, что идеально-сбалансированные деревья являются также и AVL деревьями. Доказана теорема, что AVL дерево по высоте никогда не превысит идеально-сбалансированное дерево более, чем на 45%.

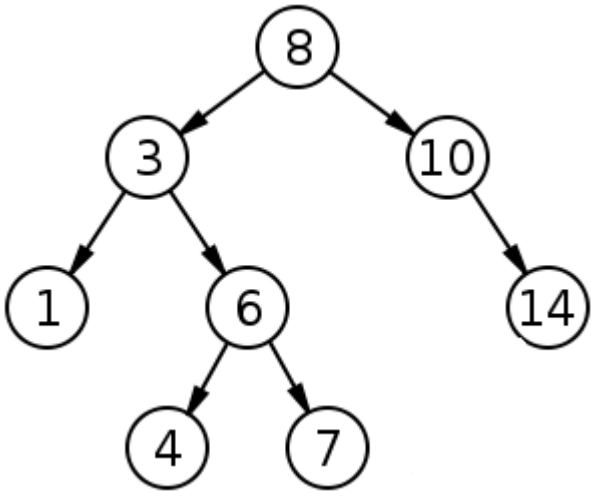


Рис 21. Пример дерева AVL.

На практике достаточно часто используют так называемые красно-чёрные деревья (Red-Black-Tree, RB-Tree). Они относятся к классу самобалансирующихся двоичных деревьев поиска, которые гарантируют логарифмическую зависимость времени поиска от количества элементов. Сбалансированность дерева достигается за счет введения в узел дерева дополнительного параметра — «цвет». Этот параметр принимает одно из двух возможных значений — «чёрный» или «красный». На рис.4.7 серый цвет соответствует красному цвету узла, а черный –черному цвету узла.

Красно-чёрное дерево обладает следующими свойствами[1]:

1. Все листья черны.
2. Все потомки красных узлов черны (т.е. запрещена ситуация с двумя красными узлами подряд).
3. На всех ветвях дерева, ведущих от его корня к листьям, число чёрных узлов одинаково. Это число называется чёрной высотой дерева.

При этом для удобства, листьями красно-чёрного дерева считаются фиктивные «нулевые» узлы, не содержащие данных.

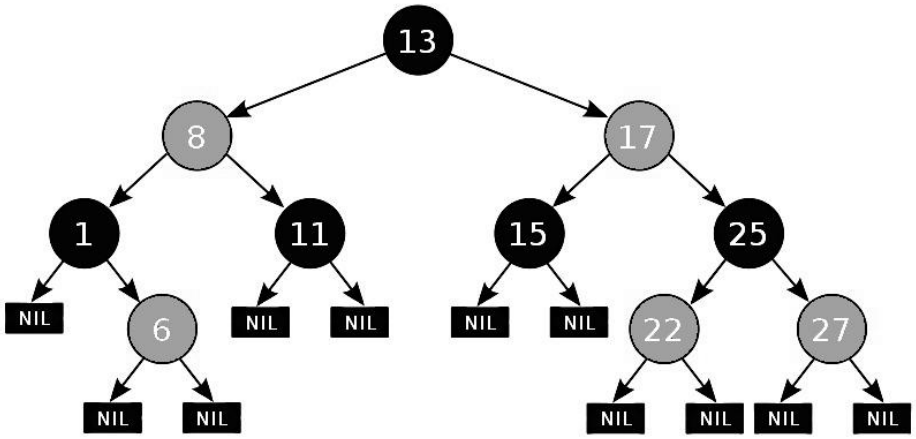


Рис. 22. Красно-черное дерево.

Каждое добавление нового элемента потребует, по меньшей мере, до 2 поворотов в дереве. Несмотря на то, что красно-чёрное дерево в общем случае выше, и поиск в нём медленнее, тем не менее, проигрыш по времени не превышает 39% в худшем случае.

Реализация метода для построения идеально-сбалансированного дерева:

```
// Идеально сбалансированные деревья
    #include <iostream.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <conio.h>
    #include <malloc.h>
    #include <math.h>

    int maxl; int u=0;

    class BTree
        { protected:
            int Dn;
             int B;
            BTree *Ln;
            BTree *Rn;
         public:
         void init() {B=1;};
         void addbalans(int N);
         void done();
         void pri (int k);
          };

    void BTree::pri (int k)     // Печать дерева
      { if (!B) { Ln->pri (k+4);
```

```
            for (int j=0;++j<k;)cout<<" ";
            printf("%3i\n",Dn);
            Rn->pri(k+4);
          };
    };
void BTree::done ()
    { if (!B) { Ln->done(); delete(Ln);
                Rn->done(); delete(Rn); };
    };
          //  Построение сбалансированного дерева
void BTree::addbalans (int N)
    { if (N!=0)
      { B=0; Dn=Dt[u]; u++;
        Ln=new(BTree); Ln->init();
        Rn=new(BTree); Rn->init();
        int NL=N/2;
        int NR=N-NL-1;
        Ln->addbalans(NL);
        Rn->addbalans(NR);
      }
    };
//-------------------------------------------
int D[12]={45,23,28,19,12,7,56,4,22,73,11,63};
void main()
{ clrscr();  int N;
  BTree T1; T1.init();
  cout<<"N = "; cin>>N;
  T1.addbalans(N);
  T1.pri(4);
  T1.done();
  getch();
}
```

4.7. Деревья поиска с включением.

Если во входном потоке имеются повторяющиеся элементы, то построение дерева поиска может быть выполнено двумя способами: либо игнорировать повторные вхождения элементов, либо учитывать количество повторных вхождений элементов в специальных счетчиках, устанавливаемых в каждую вершину дерева.

Деревья поиска с учетом количества повторных вхождения элементов, называют деревьями поиска с включением. В состав вершины дерева добавляется счетчик **Count**, который принимает значение, равное 1 при создании очередной вершины. При повторном вхождении этого же элемента в счетчик добавляется единица.

```
class btree
```

```
{ protected:
     int Dn;
     int B;
     int Count;
     btree*Ln; btree*Rn;
     public: ……..;
};
```

Использование динамических структур позволяет получить новые качества в обработке данных, например, получение значений в лексикографическом порядке при обходе дерева поиска.

В практике программирования часто встречаются задачи, для решения которых необходимо построение так называемых частотных словарей. Он создается при чтении текста, выборке из текста символов, определении количества вхождений каждого символа в текст. Имеют практическое значение и словари символов, и словари слов данного текста.

При создании частотных словарей используют деревья поиска с включением. При повторном вхождении символа к счетчику добавляется единица. В результате получаем дерево поиска (рис.23), каждая вершина которого содержит некоторый символ, который встречается в тексте, и количество этих символов в данном тексте. На этом рисунке изображено дерево поиска с включением для последовательности целых чисел, полученных из входного потока, среди которых встречаются повторные вхождения. Например, значение корня – "10" встречается в потоке 2 раза, а число "14" встречается 4 раза.

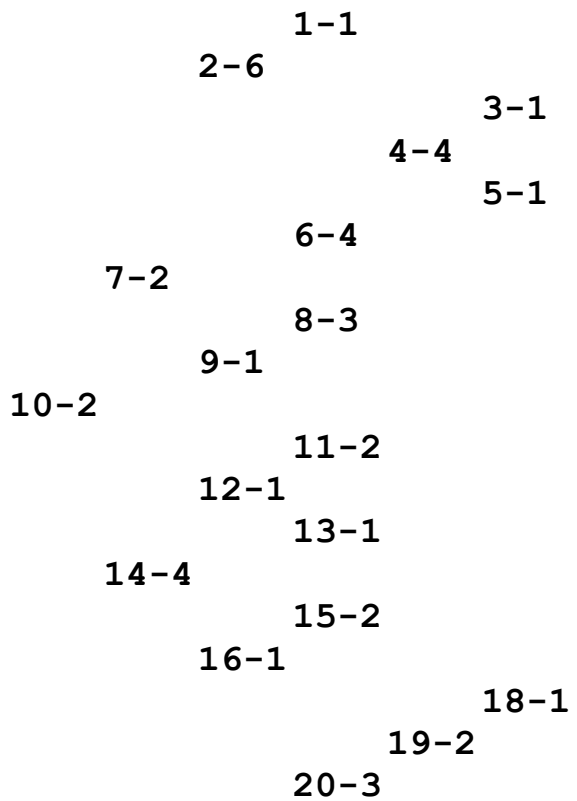


Рис.23. Дерево поиска с включением, построенное на экране.

В качестве примера рассмотрим построение дерева поиска с включением.

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>

class btree
 { protected:
     int Dn;          //  Поле данных
     int B;           //  Поле барьера
     int C;           //  Поле счетчика
     btree* Ln;       //  Указатель на левое поддерево
     btree* Rn;       //  Указатель на правое поддерево
   public:
   void Init(){B=1;};//  Инициализация
   void AddVkl(int D);    //  Добавление нового элемента
   void PriVkl(int k);    //  Печать дерева со счетчиком
   void Done();       //  Удаление дерева из памяти
 };
 void btree::AddVkl(int D)
   { if (B)
     {B=0; Dn=D; C=1;
          Ln=new(btree); Ln->Init();
          Rn=new(btree); Rn->Init();}
     else { if (D<Dn)       Ln->AddVkl(D);
        else if (D>Dn) Rn->AddVkl(D); else C++;};
   };
 void btree::PriVkl(int k)
   { if(!B)
     { Ln->PriVkl(k+4);
       for(int i=1;i<=k;i++) cout<<" ";
       cout<<Dn<<"-"<<C<<endl;
       Rn->PriVkl(k+4);
     };
   };
 void btree::Done()
   { if(!B)
     {Ln->Done(); delete(Ln);
      Rn->Done(); delete(Rn); };
   };
```

```
//----- Main Program -----------
void main()
 { clrscr(); randomize();
   btree T1;  T1.Init();
   T1.AddVkl(10);
   for (int i=1; i<42; i++) T1.AddVkl(1+random(20));
   T1.PriVkl(1);
   T1.Done();
   getch();
 }
```

Несмотря на то, что значения во входном потоке неупорядочены, метод **PriVkl()** возвращает элементы потока в упорядоченном виде с указанием их частотности.

4.8. Алгоритм оптимального кодирования Шеннона-Фано

В качестве примера рассмотрим использование частотных словарей в алгоритме оптимального кодирования Шеннона-Фано. При передаче по каналу связи текстовую информацию необходимо кодировать. Если использовать равномерные коды, то в этом случае различная частота вхождения символов в текст не будет учитываться.

Пусть по каналу связи передается следующее сообщение:

BBABBBABCBACBDAD

В этом сообщении имеется 16 букв из алфавита в 4 символа: A, B, C, D. Если символы алфавита закодировать равномерным кодом, то для каждой буквы потребуется 2 бита или 2 двоичных разряда: A - 00; B - 01; C - 10; D - 11. Таким образом для передачи сообщения потребуется 2*16 = 32 (бита).

Однако Шеннон в 1948 году показал, что при неравной вероятности появления символов количество бит на сообщение может быть меньше. Он же, совместно с Фано, разработал алгоритм, позволяющий создавать оптимальный код для кодирования символов сообщения.

Сущность метода состоит в том, что часто встречающиеся буквы кодируются коротким кодом, а редко встречающиеся буквы более длинным кодом. В результате общее количество бит для сообщения уменьшается. Подобное наблюдается в коде Морзе, применяемый в радиосвязи. Вот некоторые фрагменты кода Морзе:

```
A  .-    I  ..    S ...
E  .     M  --    F ..-.
T  -     O  ---   Q --.-
```

Рассмотрим создание кода Шеннона-Фано. Для этого создадим частотный словарь букв на основе дерева с включением. Каждый узел этого дерева содержит букву и количество вхождений ее в текст.

Однако для создания частотного словаря необходимо упорядочить данные не по алфавиту, а по их частоте вхождения в текст.

Итак, в нашем сообщении имеем:

Количество символов "B" - 8, вероятность вхождения - 8/16 = 0,5;
Количество символов "A" - 4, вероятность вхождения - 4/16 = 0,25;
Количество символов "C" - 2, вероятность вхождения - 2/16 = 0,125;
Количество символов "D" - 2, вероятность вхождения - 2/16 = 0,125;

А теперь построим таблицу (Табл. 4.8.), в которой буквы отсортированы по частоте вхождения в текст. На основе этой таблицы формируется неравномерный код, обладающий свойством однозначного декодирования. Такие коды называют префиксными, потому что среди них нет ни одного кодового слова, которое было бы префиксом (началом) любого другого кодового слова данного кода.

Буква	Вероятность, P	1-я итерация	2-я итерация	3-я Итерация	Код Шеннона
B	$P_b = 0,5$	0			0
A	$P_a = 0,25$	1	0		10
C	$P_c = 0,125$		1	0	110
D	$P_d = 0,125$			1	111

Табл. 4.8.

Формирование кода выполняется по следующей схеме. Список букв делится на две части таким образом, чтобы сумма вероятностей верхней половины списка была как можно ближе к сумме нижней половины списка. Верхней части поделенного списка присваивается - 0, а нижней - 1. Если в верхней или нижней части списка количество букв больше единицы, то этот алгоритм рекурсивно применяется к этим частям до тех пор, пока количество букв не станет равным единице. Код Шеннона-Фано формируется из обозначенных частей по всем проведенным итерациям.

Закодируем наше сообщение полученным кодом Шеннона-Фано:

BBABBBABCBACBDAD

0010000100110010110011110111

В этом случае для передачи сообщений потребуется уже 28 бит, а не 32, как это было для равномерного кода, то есть меньше. Неравная вероятность вхождения символов в текст позволяет сократить количество бит для передачи сообщений. Алгоритм Шеннона-Фано, а также его более совершенная версия, предложенная Хаффманом, широко используется в программах архивации текстов, файлов, изображений, звуковых последовательностей и других видов информации.

4.9. Удаление вершин из дерева

Удаление вершины из дерева является операцией, обратной включению. Главным требованием для этой операции является сохранение упорядоченности после удаления элемента из дерева.

Рассмотрим эту операцию применительно к дереву с упорядоченными ключами. Пусть мы имеем следующее дерево поиска (Рис. 24). При удалении вершины в дереве без нарушения упорядоченности возможны 3 случая.

1. Удаляемая вершина является терминальной, или листом (на рис. 24 это вершины 17, 28, 33, 55). Удаление такой вершины состоит в присвоении ссылке на эту вершину значения NULL. Условие упорядоченности сохраняется.

2. Удаляемый элемент имеет только одного потомка (на рис. 24 это вершины 19 и 35). В этом случае необходимо переопределить ссылку с (20) вершины на (17) вершину, а вершину (19) удалить. И в этом случае условие упорядоченности сохраняется.

3. Удаляемый элемент имеет двух потомков, например, вершина (50). Правило удаления этого элемента следующее. Удаляемый элемент (50) следует заменить на самый правый элемент его левого поддерева (35), или на самый левый элемент его правого поддерева (55). Эти элементы не могут иметь второго потомка, поэтому они удаляются по первому или по второму способу, то есть ссылка с (30) элемента переопределяется на (33) элемент, а элемент (35) удаляется. После такого удаления условие упорядоченности сохраняется.

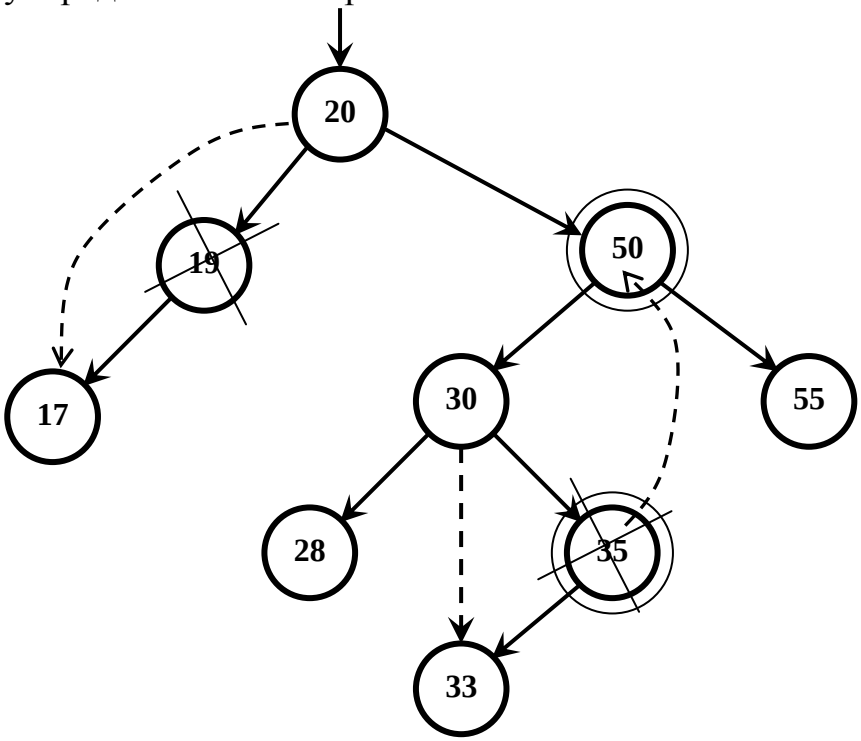


Рис.24. Способы удаления элементов из дерева.

Для реализации алгоритмы необходима процедура поиска ссылки на самый правый элемент в поддереве.

Рассмотренные способы удаления вершин в дереве применимы также и к сбалансированным деревьям. Однако в этом случае необходимо проверить дерево по критерию сбалансированности. В случае нарушения необходимо после операции удаления выполнить повторную балансировку дерева.

Сложность операций балансировки ограничивает сферу применения сбалансированных деревьев случаями, когда поиск информации выполняется значительно чаще, чем включение новых элементов или их удаление. Можно привести в качестве примера телефонный справочник города, словарь ключевых слов в компиляторе и т.д.

4.10. Таблицы перекрестных ссылок.

При создании прикладных и системных программ часто требуется создавать автоматический предметный указатель. Это таблица, в которой указано, в каких строках текста встречаются указанные слова.

Таблица создается следующим образом:

1. Из входного потока читается текст по строкам, каждой строке присваивается порядковый номер.
2. Для каждого слова запоминаются номера строк, в которых встречается это слово.

Например, для некоторого программного текста:

```
void printtree(Tree **t,int k)
{ if (*t!=NULL) {
printtree(&((*t)->Ln),k+4);
for (int i=1;i<=k;i++) cout<<" ";
printtree(&((*t)->Rn),k+4);
};};
```

создается таблица следующего вида:

```
printtree    1,3,5
Tree         1
k            1,3,4,5
for          4
i            4,4,4
*t           1,2,3,5
k+4          3,5
```

Очевидно, что для хранения ссылок на строки для каждого слова необходима динамическая списковая структура.

Поэтому элемент дерева, в котором хранится слово из входного потока, должен содержать поле для ссылки на список хранения ссылок. Название структуры для формирования и хранения таблицы перекрестных ссылок (ТПС) - "дерево поиска списков".

Процесс создания ТПС состоит из двух этапов:

1. Просмотр текста и формирование дерева поиска с включением в список ссылок на номера строк, в которых встречается данное слово.
2. Печать дерева в алфавитном порядке на основе инфиксного порядка обхода вершин дерева.

Таблица перекрестных ссылок применяется для создания таблиц идентификаторов, формируемых компиляторами, в алфавитных указателях, в предметных указателях, словарях и т.д.

Элемент дерева списков может быть задан следующим образом:

```
#define ListP struct listp
ListP {int Dp, ListP *Np;};
#define TreePL struct treepl
TreePL {int Dn; ListP *Np; TreePL *Ln; Tree *Rn;};
```

При построении дерева списков следует учитывать, что при создании новой вершины следует создать и элемент списка, в который записать номер текущей строки.

5. Графы и алгоритмы на графах

Графы являются удобным языком для описания программных моделей.

5.1. Основные определения.

Графом G(V,E) называется совокупность двух множеств V и E.

Элементы первого множества V: v_1, v_2, … v_m называются вершинами графа, элементы второго множества E: e_1, e_2, … e_n - ребрами. В случае ориентированного графа элементы множества V называют узлами, а множества E – дугами.

Если ребро графа определяется вершинами V_i и V_j , т.е. $e_k = (v_i,v_j)$, тогда ребро e_k инцидентно вершинам v_i и v_j.

Два ребра, инцидентные одной вершине, называются смежными.

Замкнутая цепь называется циклом. Цепь – это совокупность смежных ребер. На рис. 25 приведен пример неориентированного графа.

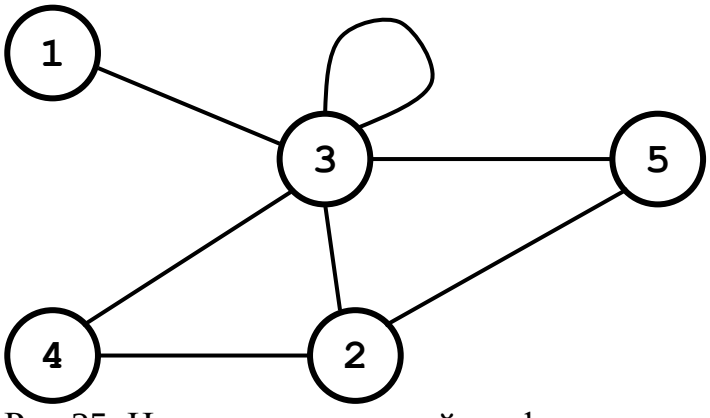


Рис.25. Неориентированный граф.

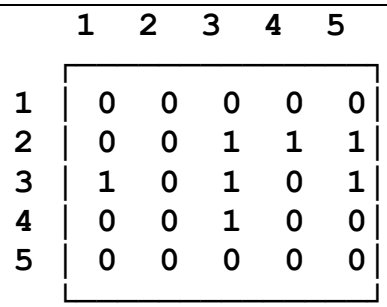
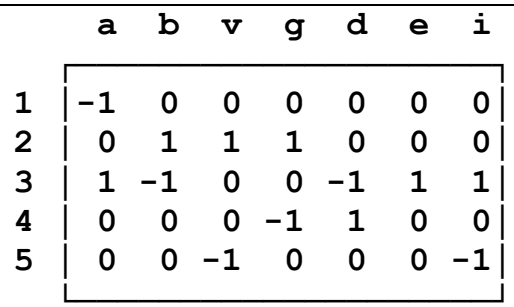
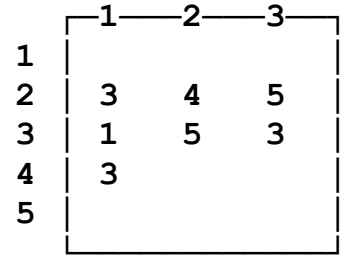
5.2. Машинное представление графов.

Одной из важных задач программирования является адекватное представление графа в памяти компьютера. Как правило, графы представляются динамическими структурами, на которых должны быть определены методы

создания, добавления, удаления ребер и вершин, доступа к вершинам, анализа связности, наличия циклов в графе и т.д.

Существует несколько способов представления графов, имеющих свои достоинства и недостатки.

Приведем некоторые из них.

1.Перечень ребер. Граф представляется совокупностью пар вершин, связанных ребром. Для орграфов первая вершина пары есть начало дуги, а вторая-конец.	(3,1) (3,3) (2,3) (4,3) (2,5) (2,4) (3,5)
2.Матрица смежности. Строки и столбцы матрицы помечены номерами вершин: "1" в (i,j) клетке матрицы соответствует наличию ребра, инцидентного j вершине. Для ненаправленного графа "1" присутствует как в (j,i) клетке, так и в (i, j).	
3.Матрицы инцидентности. Строки матрицы помечены номерами вершин, а столбцы - номерами ребер. "1" в (i ,j) клетке для ненаправленного графа означает факт их инцидентности. Для ориентированного графа "1" означает, что начало дуги в i - ой вершине, а "-1" – конец.	
4.Массив векторов смежности позволяет сжать разреженные матрицы смежности и инцидентности. Строки помечены номерами вершин и содержат номера смежных вершин. Таким образом, количество столбцов равно максимальной степени вершины графа.	

Машинное представление графа может быть выполнено на основе двумерных массивов, однако, лучше использовать динамические структуры типа "Список списков" с произвольным доступом по индекса. В этом случае снимается ограничения на размер структуры.

6. Технологии создания программных продуктов

6.1. Производство программных продуктов

В настоящее время создание программных продуктов возможно только в результате деятельности крупных фирм с численностью сотрудников до нескольких тысяч человек. Это связано с тем, что современный программный продукт обладает очень высокой сложностью, и не может быть создан небольшой группой программистов.

Важным условием успешной реализации программного проекта является правильный выбор стратегии проектирования программного продукта. Из всего множества стратегий наиболее применимы следующие: однократный процесс, инкрементная стратегия, эволюционная стратегия.

На основе стратегий проектирования строятся различные модели разработки программных продуктов. Рассмотрим некоторые из них.

1. Однократный процесс (другое название – водопадная модель) - это линейная последовательность этапов проектирования с минимальным количеством возвратов на предыдущие этапы. Эта стратегия трактуется как последовательность законченных этапов, приводящая к полностью завершенному проекту. Применяется в основном для небольших проектов, основные этапы которых уже формализованы, например математические или статистические расчеты.

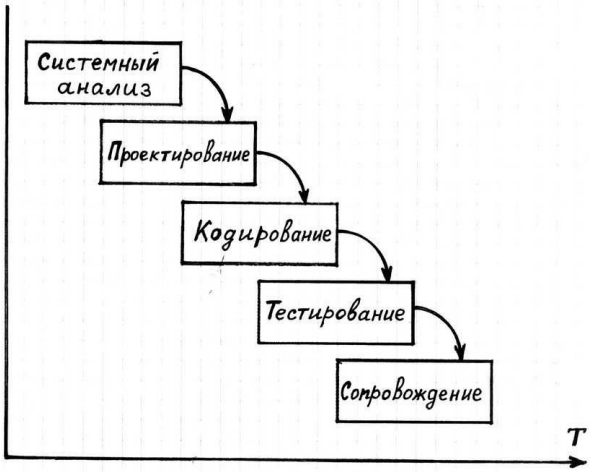


Рис 26. Этапы создания программного продукта.

2. Инкрементная модель объединяет элементы водопадной модели с итерационным способом макетирования. Первая итерация создает базовый продукт с малой функциональностью – прототип.

Вторая итерация модифицирует базовый продукт с добавлением дополнительной функциональности. Последующие итерации добавляют все новые элементы функциональности до тех пор, пока очередная итерация позволит удовлетворить требования заказчика.

3. Модель быстрой разработки - Rapid Application Development (RAD). Эта модель обеспечивает короткий цикл разработки за счет использования компонентно-ориентированного конструирования.
RAD – подход состоит из нескольких этапов:
а) Бизнес-моделирование потоков между бизнес-функциями.
б) Моделирование данных, определяется набор объектов данных, определяются их свойства и отношения между объектами.
в) Моделирование обработки, когда определяются процедуры добавления, модификации, удаления или поиска данных.
г) Генерация приложения с использованием современных компонентных языков программирования.
д) Тестирование и объединение готовых блоков в законченный проект.
Применение RAD позволяет создавать систему за 60-90 дней при условии, что каждый блок создается отдельной группой программистов, а затем интегрируется в целую систему.

4. Спиральная модель – классический пример применения эволюционной стратегии конструирования. Автор – Барри Боэм (1988). Модель базируется на свойствах классического жизненного и макетирования, к которым добавлен новый элемент – анализ риска.

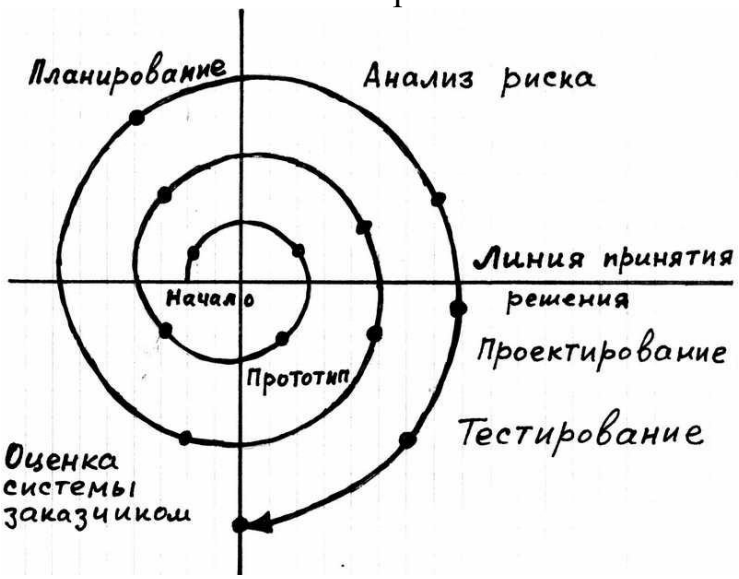


Рис. 27. Спиральная модель разработки ПО.
Процесс проектирования состоит из следующих этапов:
а) Начальный сбор сведений.
б) Соответствие требованиям заказчика.
в) Анализ риска на основе начальных требований.
г) Проектирование начального макета системы.
д) Переход к следующему уровню проектируемой системой.
е) Получение готовой системы.
ж) Оценка полученной системы заказчиком.
Если на определенном этапе риск слишком велик, то реализация проекта может быть приостановлена.

Достоинства модели: наиболее реально (в виде эволюции) отображает разработку программного продукта, явно учитывает риски на каждом витке разработки, использует моделирование для уменьшения риска.
Недостатки: повышенные требования к заказчику, трудности контроля и управления временем разработки.

5. Модель экстремального программирования – XP (eXtreme Programming) ориентирована на группы малого и среднего размера, создающих программы в условиях неопределенных или быстро изменяющихся требований. Обычно XP-группа состоит из десятка человек, работающих в одном помещении. XP-процесс должен быть высокоэкономичным, поэтому каждый цикл разработки состоит из очень коротких итераций. В каждую входят кодирование, тестирование, анализ модели заказчиком, проектирование.

XP-процесс широко использует принципы минимальности, простоты, эволюционности. Отличается малой длительностью итераций, участием заказчиков, но в предельной, "экстремальной" форме.

Кроме того, характерной особенностью XP является парное программирование. Две группы программистов создают тот или иной фрагмент программы параллельно, а затем происходит сравнение полученных продуктов.

И хотя, на первый взгляд, это удваивает ресурсы, но на самом деле затраты согласованной группы возрастают всего на 15-20%, в то время как время цикла сокращается на 40-50%, а качество продукта резко повышается.

Другой особенностью данного способа является опережающее тестирование отдельных блоков. Сначала создается тестовая программа, затем выполняется кодирование блока, который затем тестируется. Кроме того происходит исчерпывающее документирование каждого этапа разработки, позволяющее осуществить корректный откат при обнаружении несогласованности проекта. Особенно широко XP-процесс используется в Интернет программировании.

6.2. Модели качества процессов проектирования программных продуктов

В современных условиях жесткой конкуренции важно гарантировать высокое качество создаваемых программных продуктов. Такую гарантию дает сертификат качества процесса, соответствующий принятым международным стандартам, например ISO 9001-2000, IEC 15504, модель зрелости процесса CMM (Capability Maturity Model), предложенный американским институтом инженерии Карнеги-Меллон.

Базовым понятием CMM является зрелость компании. Основной принцип оценивания: хороший и сложный продукт может быть создан только в очень хорошей компании, но не наоборот.

То есть, в "незрелой" компании все решения по созданию программных продуктов зависят только от таланта и опыта разработчиков, в то время как в "зрелой" компании созданы механизмы управления проектом и снижены зави-

симости от конкретных исполнителей. Это создает среду, обеспечивающую качественную разработку проектов.

В модели СММ зафиксированы 5 уровней зрелости и предусмотрен плавный, поэтапный подход к совершенствованию процессов:

1. Начальный уровень зрелости: процесс проектирования не формализован, строго не планируется, успех проекта носит случайный характер, результат полностью зависит от личных качеств программистов. При увольнении этих сотрудников проект чаще всего останавливается.
2. Повторяемый уровень: имеются формальные процедуры процесса проектирования, контролируется и документируется каждый этап, что позволяет повторить достигнутый успех.
3. Определенный уровень: основные этапы разработки определены, стандартизованы и документированы. Качество проекта мало зависит от способностей отдельных личностей.
4. Управляемый уровень: в компании приняты количественные показатели качества как программных продуктов, так и процесса проектирования, что позволяет более точно планировать процесс создания программных продуктов.
5. Оптимизирующий уровень: Главной задачей компании становится постоянное улучшение и повышение эффективности существующих процессов.

Сертификацию компании проводит независимая экспертная организация.

6.3. Параметры качества программных продуктов

Качество программных продуктов - это совокупность свойств, определяющих полезность изделия (программы) для пользователей в соответствии с функциональным назначением и требованиями с их стороны.

Характеристики качества - понятие, отражающие отдельные факторы, влияющие на качество программ и поддающиеся измерению.

Критерии качества включают разные характеристики, например: экономичность, надежность, документированность, гибкость, модульность, тестируемость, точность,

Критерий должен:

- численно характеризовать целевую функцию программы;

- обеспечивать возможность определения затрат для достижения требуемого уровня качества;

- быть по возможности простым, хорошо измеримым и иметь малую дисперсию.

Метрика качества программ - это система измерений параметров качества программ.

В настоящее время известно несколько сотен метрик оценки качества программ, которые можно сгруппировать по следующим направлениям:

- оценки топологической и информационной сложности программ;

- оценки надежности программных систем;

- оценки производительности программ;

- оценки уровня языковых средств;

- оценки понимания программных текстов, необходимых для модификации и сопровождения программ;

- оценки производительности труда программистов.

Метрики сложности программ можно разделить на метрики сложности потоков управления программ - метрики Мак Кейба, Джилба, Майерса и др., и метрики сложности потока данных программ - Чепина, Сиена и др.

Метрики размера программ:

- оценка LOC (Lines of Code) - количество строк (операторов) в тексте программного продукта. На основе LOC-оценок вычисляются значения производительности и качества для каждого проекта:

Производительность = Длина (тыс. LOC) / Затраты (Чел. - месяц)

Достоинство LOC -оценок в простоте, распространенности, недостаток в зависимости от языка программирования, в необходимости получения исходных данных на начальных этапах проектирования.

- оценка длины программы на основе метрики М. Холстеда, в которой источником данных является текст программы. Холстед вводит следующие величины:

n1 - число уникальных операторов программы (словарь операторов);

n2 - число уникальных операндов программы (словарь операндов);

N1 - общее число операторов в программе;

N2 - общее число операндов в программе;

Тогда n=n1+n2 - словарь программы, а N=N1+N2 длина программы.

В этом случае по Холстеду объем программы равен

$$V = N * \log_2(n);$$

Если N - эмпирическая длина программы, то Холстед вводит понятие теоретической длины программы:

$$Nt = n1*\log_2(n1) + n2*\log_2(n2);$$

В результате исследований большого количества программных текстов М.Холстед показал, что зависимость между эмпирической и теоретической длиной программы достаточно сильна и приближается к 1. При этом отмечено, что если логическое совершенство программы высокое, то и зависимость выше. Таким образом, параметр зависимости может служить параметром логического совершенства программ.

Параметр объема V показывает такой объем, который соответствует максимально компактному тексту программы, реализующий данный алгоритм.

6.4. Категории специалистов в области программирования

В программной индустрии востребованы специалисты самого различного направления, поскольку любой серьезный проект является многоплановой разработкой, требующей для своей реализации применения многих технологий.

Вот небольшой перечень специальностей, применяемых в программной индустрии:

1. Системный программист (System Software Programming) - ориентируется на создание компиляторов, интерпретаторов, драйверов, утилит, редакторов, операционных систем и т.д.
2. Прикладной программист (Application Programming) - разработка и отладка задач в прикладных областях: вычисления, базы данных, офисные и промышленные приложения.
3. Системный аналитик (Analyst) - выполняет анализ и проектирование систем на уровне спецификаций.
4. Постановщик задач - это своего рода технолог в процессе создания программного продукта.
5. Администратор БД - обеспечивает организационную поддержку работоспособности систем и баз данных.

Литература

1. Шилд Г. Программирование на BORLAD C++ для профессионалов. Мн.1998
2. Керниган Б, Ричи Д. Язык программирования Си. М. 1992.
3. Основы алгоритмизации и программирования. Язык Си: учеб. пособие / Демидович Е.М.–СПб.:БХВ-Петербург, 2006.–440с.
4. Страуструп Б. Язык программирования С++.: Бином, Невский Диалект, 2004 г.–1104 стр.
5. Седжвик Р. Фундаментальные алгоритмы на С++.: Diasoft. М. 2004.–1136
6. Стенли Б. Липпман. С++ для начинающих: Пер. с англ. 2тт. - Москва: Унитех; Рязань: Гэлион, 1992,–345с.
7. М. Эллис, Б. Строуструп. Справочное руководство по языку С++ с комментариями: Пер. с англ. - Москва: Мир, 1992. 445с.
8. В.В. Подбельский. Язык С++: Учебное пособие. - Москва: Финансы и статистика, 1995. 560с.
9. У. Сэвитч. С++ в примерах: Пер. с англ. - Москва: ЭКОМ, 1997. 736с.

Приложение 1. Практическое программирование

Тема 1: *Элементы машинной графики.*

Задание. На графическом экране построить изображение согласно варианту из рис.1, используя базирование по опорной точке. Рисунок должен быть реализован в виде функции с параметрами: координаты опорной точки и масштабный коэффициент:

```
void Picture(int x, int y, float mk);
```

Элементы изображения должны быть закрашены выбранными Вами цветами и заполнены видами заливок согласно рис.2.

Изображение должно обладать свойствами перемещаемости и масштабируемости.

Координаты опорной точки (x,y) и масштабный коэффициент (Mk) для рисунка вводятся в диалоговом режиме.

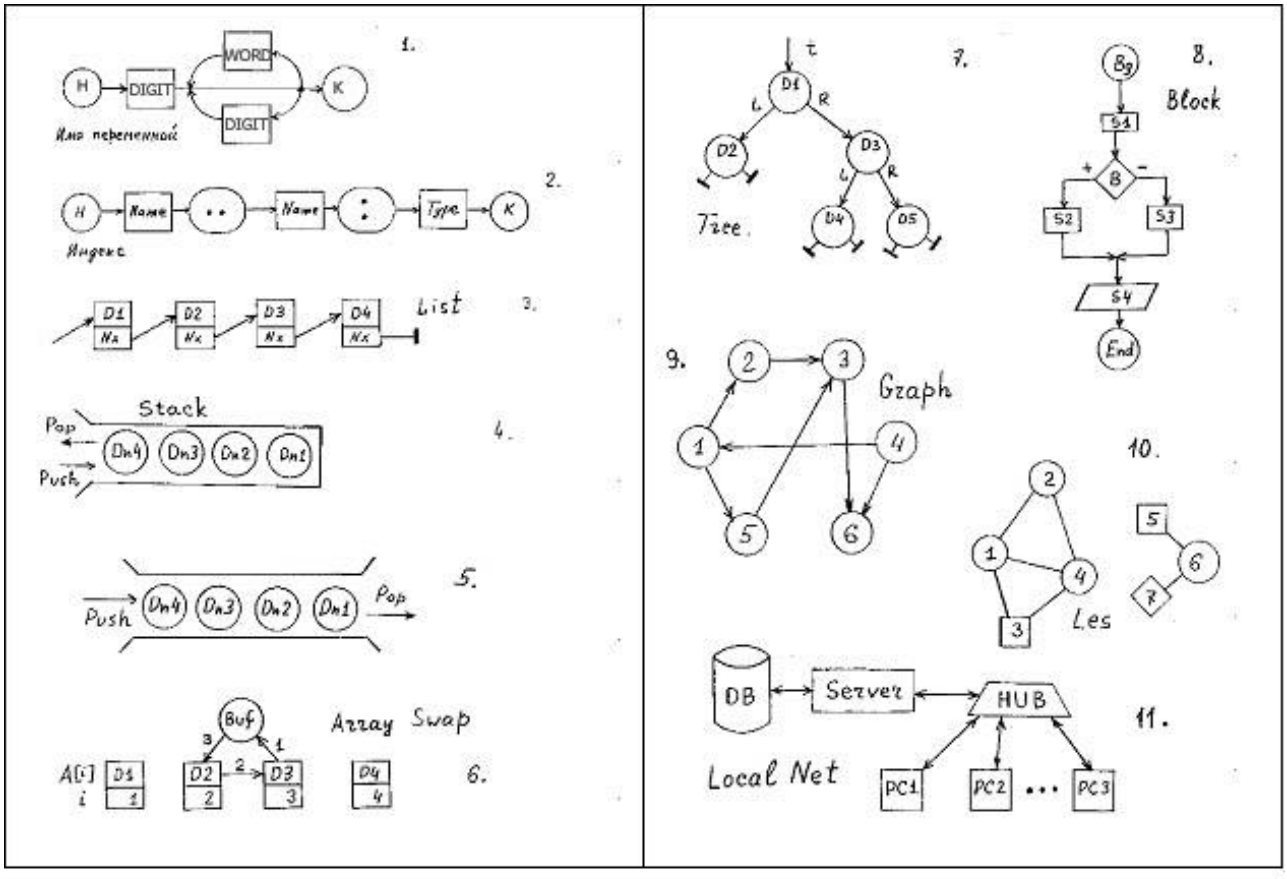


Рис. 1. Простые векторные изображения .

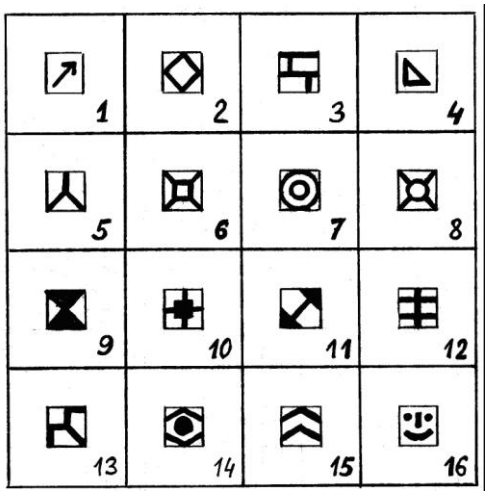


Рис. 2. Матрица заполнения 8x8.

Тема 2: Построение зависимостей на графическом экране.

Задание. Построить на графическом экране график функции, оформив ее такими атрибутами, как:

- обрамляющая рамка,
- оси координат,
- надписи и масштабные отметки хотя бы на одной оси,
- наименование графика.

При построении графика обязательно следует выбрать масштабные коэффициенты по осям координат так, чтобы график или другое изображение заполнял весь экран, но не выходил за его пределы.

1. На экране построить график (Спираль), заданный функцией:
x = R·cos(t+F); y = R·sin(t+F); R = t/2; [0 < t < 2*pi*n]
Параметры F,R для построения графика задаются в диалоговом режиме.

2. На экране построить график, заданный функцией:
Y = A·sin(0.98·x) + B·cos(0.37·x); [-5 < x < 5]
Параметры A,B для построения графика задаются в диалоговом режиме.

3. На экране построить график, заданный функцией:
Y = K·x - F·sin(0.93x) + S·cos(W·x); [-6 <x < 6]
Параметры K,F,S,W для построения графика задаются в диалоговом режиме.

4. На экране построить график, заданный функцией:
Y = Q·cos(2x) - J·x + ln(|x+3|); [-2 < x < 2]
Параметры Q,J для построения графика задаются в диалоговом режиме.

5. На экране построить график, заданный функцией:
Y = P·0.56·x3 - 11.3·V·x + 4·M·sin(X+1) + 3; [-5 < x < 5]
Параметры P,V,M для построения графика задаются в диалоговом режиме.

6. На экране построить график, заданный функцией:
Y = Z·sin(x)- 1.3·D·x^2 + exp(1.6·L·(x-1)) - 2; [-3 < x < 3]
Параметры Z,D,L для построения графика задаются в диалоговом режиме.

7. На экране построить график, заданный функцией:
Y = 0.4·N·x - 2.3·R·x2 + ln(|x+5|) + T; [-4 < x < 4]
Параметры N,R,T для построения графика задаются в диалоговом режиме.

8. На экране построить график (Улитка Паскаля), заданный функцией:
X = A·cos(fi)·cos(fi) + 4·K·cos(fi);
Y = A·sin(fi)·cos(fi) + 4·K·sin(fi); [0 <= fi <= 2·pi]
Параметры A,K для построения графика задаются в диалоговом режиме.

9. На экране построить график, заданный функцией:
Y = 1.7·H·x + 7.8·U·sin(2.1·x) + 3.4·(x-1)2 - 7.2; [-3 < x < 3]
Параметры H,U для построения графика задаются в диалоговом режиме.

10. На экране построить график, заданный функцией:
Y = 0.5·T·exp(1.7·(x-1)) + 0.4·E·x2 - 2x - 8.9; [-8 < x < 3]
Параметры T,E для построения графика задаются в диалоговом режиме.

11. На экране построить график, заданный функцией:
Y = 4.6·G·sin(x/F) + 1.9·M·cos(x+1.8); [-4 < x <4]
Параметры G,M,F для построения графика задаются в диалоговом режиме.

12. На экране построить график (Астроида), заданный функцией:
X = sin(t)^{alpha}; Y = cos(t)^{alpha}; [0 <= t <= 2·pi]
Параметр **alpha** для построения графика задается в диалоговом режиме и может принимать только целочисленные нечетные значения: 1, 2, 3, … .

13. На экране построить график (Эпициклоида), заданный функцией:
X = (1+A)·cos(A·t) + R·cos(1+A)·t; [0 <= t <= 2·pi]
Y = (1+A)·sin(A·t) - R·sin(1+A)·t;
Параметры A,R для построения графика задаются в диалоговом режиме.

14. На экране построить график (Гипоциклоида), заданный функцией:
X = A·cos(t) + D·cos(A·t); [0 <= t <= 2·pi]
Y = A·sin(t) - D·sin(A·t);
Параметры A,D для построения графика задаются в диалоговом режиме.

15. На экране построить график, заданный функцией:
Y = B·x3 - A·x + SQRT(x+1) + 3; [-1 < x < 4]

Параметры A,B для построения графика задаются в диалоговом режиме.

16. На экране построить график (Строфоида), заданный функцией:
P = -(A·cos(2·Fi) / (cos(Fi); [-2·pi < Fi < 2·pi]
Параметры A,Fi для построения графика задаются в диалоговом режиме.

17. На экране построить график (Декартов лист), заданный функцией:
R=(3·A·cos(Fi)·sin(Fi))/(cos3(Fi)+sin3(Fi));[-2·pi<Fi<2·pi]
Параметры A,Fi для построения графика задаются в диалоговом режиме.

2.1. Некоторые методы графического режима.

1. **initgraph(&gdriver, &gmode, "")** – описание и пример в Ctrl+F1;
2. **closegraph()** – закрытие графического режима;
3. **moveto(int x, int y)** – устанавливает курсор в точку (x,y);
4. **getmaxx(), getmaxy()** – возвращает максимальное значение по оси x и по оси y;
5. **line(int x1, int y1, int x2, int y2)** – линия от точки (x1,y1) до точки (x2,y2);
6. **rectangle(int x1, int y1, int x2, int y2)** – прямоугольник от левой верхней точки (x1, y1) до нижней правой точки (x2, y2);
7. **circle(int x, int y, int r)** – круг с центром в точке (x,y) и радиусом r;
8. **arc(int x, int y, int stangle, int endangle, int r)** – дуга окружности с центром в точке (x,y), начальный угол stangle, конечный угол endangle, (в градусах), радиус r:
   arc(300,100,45,185,50);
9. **setcolor(int color)** – установка цвета рисования: 0 - черный, 1 – синий, 2 – зеленый, 3–бирюзовый, 4-красный, 5-фиолетовый, 6-коричневый, 7-светло-серый, 8-темно-серый, 9-голубой, 10-светлозеленый, 11-свелобирюзовый, 12-розовый, 13-пурпурный, 14-желтый, 15-белый.
10. **getcolor()** – возвращает текущий цвет
11. **setbkcolor(int color)** – установка цвета фона
12. **getbkcolor()** – возвращает цвет фона
13. **putpixel(int x, int y, int color)** – пикселу в точке (x,y) присваивает цвет color
14. **outtextxy(int x, int y, char st)** – выводит текст st, начиная с точки (x,y):
    outtextxy(200,300,"ABCD-text");
15. **setlinestyle(int linestyle, int upattern, int thickness)** – линии рисуются стилем **linestyle**, по шаблону **upattern** (если linestyle=4), толщиной **thickness** (1 или 3):

linestyle: 0-сплошная, 1-точечная, 2-штрихпунктирная, 3-пунктирная, 4-по шаблону

16. **setfillpattern(char upattern, int color)** – установка шаблона заливки цветом color;

Пример:

```
char
Strelka[8]={0x00,0x1e,0x06,0x0a,0x12,0x20,0x40,0x80};
```

// шаблон 1 – “Strelka”

// **setfillstyle(int pattern, int color)** – устанавливает стиль заливки;

```
setfillstyle(12,14);
setfillpattern(Strelka,14);
```

// устанавливаем заливку по шаблону **Strelka** желтым цветом;

// **floodfill(int x, int y, int color);** заливается область, ограниченная цветом color, имеющая внутреннюю точку (x,y);

```
circle(200,300,15);//рисуем замкнутую область цветом 15 (круг);
floodfill(200,300,15);// заливаем ее рисунком по шаблону 1;
```

17. Операция округления вещественного числа:

```
int round(float a) {if(a-floor(a)<0.5) return
floor(a);
return ceil(a);};
```

18. Преобразования координат:

```
int Nx(float x)
    {return round(((x-Xmin)/(Xmax-Xmin))*639);};
int Ny(float y, float Ymin, float Ymax)
    {return 479-round(((y-Ymin)/(Ymax-Ymin))*479);};
```

Тема 3: Динамические структуры данных - списки.

Задание: Создать объект - однонаправленный список L1 без головного элемента.

Добавить в стандартный набор объекта два Ваших метода согласно варианту и варианту со сдвигом = 13.

Ваш метод должен быть выполнен при условии, что в списке имеется достаточное количество элементов для выполнения операции, иначе вывести сообщение о невозможности выполнения метода.

Все операции выполнять только изменением указателей на элементы.

Значения элементов задать в строке констант или в диалоговом режиме.

Проверить размеры свободной памяти до выполнения программы и после функцией **`_memavl()`**, эти значения должны совпадать.

Обязательно выводить список на экран до и после выполнения заданной операции. Элементы, меняющие свое местоположение в списке, окрасить отличающимся цветом.

Например при удалении второго элемента:

```
Before Memaval = 280960
12.34  -5.67  431.32  98.46 -->|
12.34  431.32  98.46  -->|
After  Memaval = 280960
```

Возможный вид метода:

```
int Method(<список параметров>) {...};
```

Функция возвращает 1, если выполнение прошло успешно и 0 в противном случае.

Варианты.

1. Переместить максимальный элемент из первых 3-х на первое место, если среди них нет равных, иначе оставить только один элемент из равных, удалив остальные.
2. Меньший из 3-х первых элементов переместить на первое место, если этот элемент четный, иначе все нечетные элементы из этих трех удалить.
3. Удалить два первых элемента, если они равны, иначе модуль их разности поместить на третье место.
4. Удалить меньший элемент из первых трех, если среди них нет равных, иначе удалить только равные элементы.
5. Максимальный элемент из первых трех перенести на 3-е место, если их два, то один из них удалить и в начало списка вставить номер удаленного элемента.
6. Из первых трех элементов удалить больший из них, если он четный, иначе вставить на его место сумму нечетных элементов.

7. Вставить на первое место больший из 3-х первых элементов, если он кратен третьему элементу, иначе на третье место вставить их сумму.
8. Поменять местами первый и четвертый элементы, если они не равны, иначе на 2-е место вставить сумму этих элементов.
9. Если среди первых трех элементов нет равных, то упорядочить их по возрастанию, иначе в начало списка вставить номера равных элементов.
10. Удалить наибольший элемент из первых трех, если все они четные, иначе удалить из них только нечетные элементы.
11. Удалить максимальный элемент из первых четырех, если он единственный, иначе количество равных максимальных элементов поместить в начало списка.
12. Минимальный элемент из первых трех перенести на первое место, если их два, то один из них удалить. Номер удаленного элемента вставить в начало списка.
13. Если среди первых четырех элементов есть один нечетный, то перенести его на второе место, иначе в начало списка поместить номера четных элементов.
14. Если среди первых трех элементов есть один, кратный трем, то перенести его на первое место, иначе в начало списка поместить сумму этих элементов.
15. Если все три элемента не равны между собой, упорядочить их по возрастанию, иначе удалить равные.
16. Поменять местами второй и четвертый элементы, если они положительные или второй элемент с третьим в противном случае.
17. Выполнить ротацию (сдвиг с переносом) 3-х первых элементов списка, если они не равны между собой, иначе удалить равные элементы.
18. Если среди первых трех элементов есть два равных, то удалить первый из них, иначе вставить в начало списка сумму трех элементов.
19. Заменить минимальный элемент из первых четырех его абсолютным значением, если он имеет отрицательное значение, его номер поместить в начало списка.
20. Частное от деления первых 2-х элементов поместить на 3-е место, если первый элемент больше, иначе в начало списка вставить их разность.
21. Вставить в начало списка остаток от деления первых 2-х элементов, если первое число больше второго, иначе поменять эти элементы местами.
22. Если среди первых трех элементов есть два равных, то удалить элемент, не равный двум другим, иначе вставить в начало списка максимальный из них.
23. Если среди первых трех элементов есть один нечетный, то его удалить, иначе в начало списка вставить количество четных элементов списка.
24. Если все три элемента не равны между собой, то на второе место поместить меньший элемент, иначе в начало списка вставить номера равных элементов.

Тема 4: Представление и обработка линейных динамических структур.

ЦЕЛЬ: Изучение методов работы с динамическими структурами список, стек и очередь.

ТРЕБОВАНИЯ:

1. Программа должна быть реализована с применением методов ООП.
2. Данные получают из текстового файла, дополнительные параметры могут вводиться либо с клавиатуры в диалоговом режиме, либо из раздела констант.
3. Программа в процессе работы должна вывести на экран в текстовом или графическом виде содержимое исходных списков и содержимое полученных списков, а также значения тех результатов, которые предусмотрены заданием.
4. Элементы исходных и полученных списков должны печататься в порядке их следования.
5. По окончании работы программы память должна быть освобождена от динамических структур.
6. Для отчета необходимо представить текст программы, а также продемонстрировать ее работу.

Варианты.

1. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в список L1a. Используя структуры «Очередь», перенести в начало списка только нечетные числа, а в конец – четные. Результат записать в выходной файл.

2. Во входном файле дан текст. Используя структуры “Queue” («Очередь»), удалить из текста слова с чередованием гласных и согласных букв, а затем переписать текст с сохранением строчной структуры в выходной файл.

3. Дан текст во входном файле. Используя структуры “Stack”, исключить из текста слова, не являющиеся палиндромами и записать результат в выходной файл.

4. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в стек St1. Используя структуры «Очередь», в выходной файл записать сначала простые числа, затем полные квадраты, а потом остальные, сохраняя их взаимный порядок.

5. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в очередь Qu1. В очередь Qu2 записать самую длинную неубывающую подпоследовательность из очереди Qu1.

Результат записать в выходной файл.
Пример: **Qu1 = '34 67 12 15 17 19 21 27 19 11 67 8',**
преобразуется в **Qu2 = '12 15 17 19 21 27'.**

6. Из текстового файла строки символов переписать в структуру «Очередь». Используя структуру «Stack», в выходной файл переписать все слова каждой строки входного файла в обратном порядке, не меняя порядок символов в слове. Символы знаков, кроме пробела, исключить из строк.

7. Из текстового файла строки символов переписать в структуру «Очередь». Используя структуру «Stack», в выходной файл переписать все слова входного файла, причем в словах, в которых встречаются символы цифр, нецифровые символы вывести в обратном порядке. Цифры должны остаться на своих местах.

8. Даны два текстовых файла, Fin1 и Fin2 с изображениями целых чисел, которые переписываются в списки L1A и L1B. В выходной файл вывести только те числа, которые содержатся в обоих списках. Например: L1A содержит
"**12 27 12 43 78 67 14 19 14 89**", а **L1B** содержит
"**18 27 43 67 89 19 189".**
В этом случае результат будет следующий: "**27 43 67 19 89".**

9. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в список L1. Используя структуру «Stack», найти цепочку чисел длиной M с наибольшей суммой элементов и перенести числа этой цепочки в выходной файл.

10. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в список L1a. Количество отрицательных и положительных чисел в этом файле одинаково. Используя структуры «Stack», в выходной файл записать последовательность с чередованием отрицательных и положительных чисел. Пример: список
"34 -9 85 71 7 -2 -127 -1 93 -2" преобразуется в
"34 -9 85 -2 71 -127 7 -1 93 -2".

11. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в очередь Qu1. Удалить из очереди элементы так, чтобы оставшиеся образовывали неубывающую последовательность, которую записать в выходной файл.

12. Дан текстовый файл, в котором слова разделены пробелом. Текст из файла необходимо переписать в список L1a. В список L1b переписать сначала слова, состоящие из одной буквы, затем из двух, далее из трех и т.д. Результат поместить в выходной файл. Так, входной текст:
"begin for i:=1 to N do if i = N div 2 then exit end"
преобразуется в следующий:
"N i = N 2 to do if for div end i:=1 then exit begin".

13. Дан текстовый файл с изображением целых чисел. Используя структуру 'Stack', вывести в выходной файл сначала только числа, которые являются полными квадратами, затем числа, являющиеся полными кубами, в обратном порядке их следования. Например: последовательность: **'144 81 345 121 718 49 27 53 64 78 256'** преобразуется в следующий вид: **'256 49 121 81 144 64 27'.**

14. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в очередь Qu1. Удалить из очереди Qu1 непростые числа, используя стек St1, и записать результат в выходной файл.

15. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в очередь Qu1. Используя структуру «Stack», создать в очереди Qu1 изображения этих чисел в двоичной системе счисления и переписать их в выходной файл. Пример: "**56 13 1234 8 19"** преобразуется в "**111000 1101 10011010010 1000 10011"**

16. Дан текстовый файл с изображением целых чисел, которые необходимо переписать в список L1a. Используя структуру «Stack», в выходной файл вывести все числа, но те их них, которые находятся между максимальным и минимальным значением, в обратном порядке следования. Пример:**"45 78 32 19 58 37 11 23 68"**преобразуется в **"45 78 37 58 19 32 11 23 68".**

17. Квадратная матрица целых чисел из текстового файла записывается построчно в линейный список. Из матрицы удаляются строка и столбец, в которых находится минимальный элемент матрицы (он в матрице единственный). Полученная матрица выводится в выходной файл построчно.

Тема 5: Бинарные деревья.

Задание:

Создать объект – дерево поиска или сбалансированное дерево Tree1. Добавить в стандартный набор объекта ваш метод согласно варианту. Этот метод должен выполняться при условии, что в дереве имеется достаточное количество элементов для выполнения операции.

Выводить дерево на экран до и после выполнения заданной операции.

Искомые числа пометить отличающимся цветом.

Проверить объемы свободной памяти до выполнения программы и после функцией **`_memavl()`**, эти объемы должны совпадать.

Возможный вид метода: **`int Method(<входные параметры>);`**

Функция возвращает 1, если выполнение метода прошло успешно, иначе 0.

Требования: Данные целого типа **`int`** для построения дерева брать из секции констант.

Варианты.

1. Найти все листья в дереве, которые содержат простые числа. Напечатать их количество.
2. Определить, какое простое число находится всех левее в сбалансированном дереве.
3. Для дерева поиска найти уровень, на котором расположено максимальное простое число.
4. Для дерева поиска определить, сколько простых чисел находится на уровне U.
5. Для дерева поиска определить, в каком поддереве (левом или правом) больше элементов на уровне U.
6. На каком уровне сбалансированного дерева расположено минимальное простое число.
7. Для дерева поиска найти элементы, наиболее часто встречающиеся в дереве.
8. Найти, сколько в дереве поиска листьев с нечетными числами.
9. Определить, находится ли максимальное число сбалансированного дерева глубже, чем минимальное.
10. Определить, какое простое число находится наиболее глубоко в дереве поиска.
11. В дереве поиска найдите уровни, на которых имеются одинаковые числа.
12. Найти разность между уровнями, на которых расположены максимальное и минимальное числа.
13. Для сбалансированного дерева найти уровень, на котором сумма элементов равна заданному числу M.

14. Определить минимальный уровень числа, которое является полным квадратом.
15. Определить, являются ли все числа на уровне U простыми.
16. Определить, на каком уровне количество чисел, имеющих только нечетные цифры, максимально.
17. Определить уровень дерева поиска, на котором больше всего листьев с четными числами.
18. Определить, сколько чисел-палиндромов расположено на уровне U дерева поиска.
19. Определить, сколько в дереве поиска листьев, вершин с одним потомком и вершин с двумя потомками.
20. На заданном уровне U дерева поиска определить, имеется ли чередование четных и нечетных чисел.
21. Определить количество простых чисел на самом нижнем уровне сбалансированного дерева.

5.1. Пример реализации структуры "дерево" с минимальным набором методов.

```
// Объект "Дерево" содержит методы, позволяющие
// добавлять элементы в дерево, печатать дерево,
// удалять дерево, определять максимальный уровень.
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>

class btree
 { protected:
     int Dn;
     int B;
     btree* Ln;
     btree* Rn;
   public:
   void Init(){B=1;};
   void Add(int D);
   void Pri(int k);
   void Done();
   int  MLevel(int &mL, int u);
 };
```

```
 void btree::Add(int D)   // Добавить в дерево
   { if (B)
      {B=0; Dn=D; Ln=new(btree); Ln->Init();
             Rn=new(btree); Rn->Init();}
     else {if (D<Dn) Ln->Add(D); else Rn->Add(D);};
   };

 void btree::Pri(int k)   // Распечатка дерева
   { if(!B)
     { Ln->Pri(k+4);
       for(int i=1;i<=k;i++)
           cout<<" "; cout<<Dn<<" "<<endl;
       Rn->Pri(k+4);
     };
   };
 void btree::Done()       // Удаление дерева
   {   if(!B){Ln->Done();   delete(Ln);   Rn->Done();   de-
lete(Rn); }; };

 int btree::MLevel(int &mL, int u)    // Уровень
   { if(!B)
     { Ln->MLevel(mL,u+1);
       Rn->MLevel(mL,u+1);
       if (u>mL)mL=u;
     };
     return mL;
   };
//----- Main Program -----------
void main()
 { clrscr(); randomize(); int mL=0;
   cout<<_memavl()<<"\n";
   btree T1;  T1.Init();  T1.Add(50);
   for (int i=1; i<22; i++) T1.Add(random(99));
   T1.Pri(1);
   cout<<T1.MLevel(mL,0)<<" - ";
   T1.Done();
   cout<<_memavl()<<"\n";
   getch();
 }
```