[1. Рекурсия и рекурсивные алгоритмы](#_Toc5051329)

[1.1. Рекуррентные соотношения. Рекурсия и итерация](#_Toc5051330)

[1.2. Сущность рекурсии](#_Toc5051331)

[1.3. Виды рекурсии](#_Toc5051332)

[1.3.1. Прямая рекурсия](#_Toc5051333)

[1.3.2. Косвенная рекурсия.](#_Toc5051334)

[1.3.3. Линейная рекурсия](#_Toc5051335)

[1.3.4. Ветвящаяся рекурсия](#_Toc5051336)

[1.3.5. Бесконечная рекурсия](#_Toc5051337)

[1.3.6. Сложная рекурсия](#_Toc5051338)

[1.3.6.1. Опережающее описание](#_Toc5051339)

[1.3.7. Типовой пример.](#_Toc5051340)

[1.4. Имитация работы цикла с помощью рекурсии](#_Toc5051341)

[1.4.1. Пример 1. Инкремент счетчика итераций](#_Toc5051342)

[1.4.2. Пример 2. Декремент счетчика итераций](#_Toc5051343)

[1.4.3. Пример 3. Двойной цикл](#_Toc5051344)

[1.4.4. Пример 4: Перевод числа в двоичную систему.](#_Toc5051345)

[1.5. Произвольное количество вложенных циклов](#_Toc5051346)

[1.6. Графы](#_Toc5051347)

[1.6.1. Понятие и представление графа: матрица смежности, список смежности](#_Toc5051348)

[1.6.2. Способы представления графов](#_Toc5051349)

[1.6.2.1. Матрица смежности](#_Toc5051350)

[1.6.2.2. Список смежности](#_Toc5051351)

[1.6.3. Использование графа с использованием STL для конкурентного программирования](#_Toc5051352)

[1.7. Фракталы](#_Toc5051353)

[1.8. Деревья](#_Toc5051354)

[1.8.1. Основные определения. Способы изображения деревьев](#_Toc5051355)

[1.8.2. Прохождение деревьев](#_Toc5051356)

[1.8.2.1. Алгоритм обхода в прямом порядке](#_Toc5051357)

[1.8.2.2. Алгоритм обхода в обратном порядке](#_Toc5051358)

[1.8.2.3. Алгоритм обхода в концевом порядке](#_Toc5051359)

[1.8.3. Представление дерева в памяти компьютера](#_Toc5051360)

[1.8.3.1. Рекурсивные структуры данных](#_Toc5051361)

[1.8.3.1.1. Пример 1. Простой пример связанного списка](#_Toc5051362)

[1.8.3.1.2. Пример 2. Связанный список (динамический список)](#_Toc5051363)

[1.8.3.1.3. Пример 3. Односвязный однонаправленный список](#_Toc5051364)

[1.8.3.1.4. Пример 4. Однонаправленный связанный список](#_Toc5051365)

[1.9. Примеры рекурсивных алгоритмов](#_Toc5051366)

[1.9.1. Рисование дерева](#_Toc5051367)

[1.9.2. Ханойские башни](#_Toc5051368)

[1.9.2.1. Прмер 1.](#_Toc5051369)

[1.9.2.2. Прмер 2.](#_Toc5051370)

[1.9.2.3. Прмер 3.](#_Toc5051371)

[1.9.3. Синтаксический анализ арифметических выражений](#_Toc5051372)

[1.9.3.1. Пример 1. Вычисление выражения (Pascal)](#_Toc5051373)

[1.9.3.2. Пример 2. Синтаксический анализатор математических выражений](#_Toc5051374)

[1.9.4. Быстрые сортировки](#_Toc5051375)

[1.9.4.1. Алгоритм 1: «Быстрая» сортировка (quicksort).](#_Toc5051376)

[1.9.4.1.1. Quicksort. Пример 1.](#_Toc5051377)

[1.9.4.1.2. Quicksort. Пример 2.](#_Toc5051378)

[1.9.4.1.3. Quicksort. Пример 3.](#_Toc5051379)

[1.9.4.2. Алгоритм 2: Сортировка слиянием (merge sort).](#_Toc5051380)

[1.9.4.2.1. Merge sort. Пример 1.](#_Toc5051381)

[1.9.4.2.2. Merge sort. Пример 2.](#_Toc5051382)

[1.9.4.2.3. Merge sort. Пример 3.](#_Toc5051383)

[1.9.4.3. Алгоритм 3: Сортировка деревом (tree sort).](#_Toc5051384)

[1.9.4.3.1. Tree sort. Пример 1.](#_Toc5051385)

[1.9.4.3.2. Tree sort. Пример 2.](#_Toc5051386)

[1.10. Рекурсия или цикл. Избавление от рекурсии](#_Toc5051387)

[1.10.1. Рекурсия](#_Toc5051388)

[1.10.2. Цикл](#_Toc5051389)

[1.10.3. Контрольные примеры](#_Toc5051390)

[1.10.3.1. Вычисление суммы](#_Toc5051391)

[1.10.3.2. Вычисление факториала](#_Toc5051392)

[1.10.4. Области применения рекурсии](#_Toc5051393)

[1.10.5. Явное использование стека](#_Toc5051394)

[1.10.6. Запоминание последовательности рекурсивных вызовов](#_Toc5051395)

[1.10.7. Определение узла дерева по его номеру](#_Toc5051396)

[1.11. Контрольные вопросы](#_Toc5051397)

[1.12. Задачи](#_Toc5051398)

[1.13. Литература](#_Toc5051399)

[1.14. Другие материалы на этом сайте](#_Toc5051400)

**http://www.tvd-home.ru/recursion**

1. Рекурсия и рекурсивные алгоритмы

Рекурсией называется ситуация, когда адгоритм вызывает сама себя. Впервые сталкиваясь с такой алгоритмической конструкцией, большинство людей испытывает определенные трудности, однако немного практики и рекурсия станет понятным и очень полезным инструментом в вашем программистском арсенале.

## Рекуррентные соотношения. Рекурсия и итерация

Говорят, что последовательность векторов \{\vec{x}_n\}задана **рекуррентным** соотношением, если задан начальный вектор \vec{x}_0=(x_0^1, \ldots, x_0^D) и функциональная зависимость последующего вектора от предыдущего

\vec{x}_n=\vec{f}(\vec{x}_{n-1})~~~~~(1)

Простым примером величины, вычисляемой с помощью рекуррентных соотношений, является факториал

n!=1 \cdot 2 \cdot 3 \cdot \ldots \cdot n

Очередной факториал n! можно вычислить по предыдущему как:

n!=(n-1)! \cdot n~~~~~(2)

Введя обозначение x_n=n!, получим соотношение:

x_n=x_{n-1} \cdot n,~x_0=1~~~~~(3)

Вектора \vec{x}_n из формулы (1) можно интерпретировать как наборы значений переменных. Тогда вычисление требуемого элемента последовательности будет состоять в повторяющемся обновлении их значений. В частности для факториала:

x = 1;

for(i = 2; i <= n; i++)

x = x \* i;

printf("%d", x);

Каждое такое обновление (x := x \* i) называется ***итерацией***, а процесс повторения итераций – ***итерированием***.

Обратим, однако, внимание, что соотношение (1) является чисто рекурсивным определением последовательности и вычисление n-го элемента есть на самом деле многократное взятие функции f от самой себя:

x_n=\displaystyle{\underbrace{f(f(...f(x_0)))}_n}~~~~~(4)

В частности для факториала можно написать:

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

int Factorial(int);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

printf("5! = %d\n", Factorial(5));

system("PAUSE");

}

int Factorial(int n)

{

if(n > 1)

return (n \* Factorial(n-1));

else

return(1);

}



**Следует понимать, что вызов функций влечет за собой некоторые дополнительные накладные расходы, поэтому первый вариант вычисления факториала будет несколько более быстрым. Вообще итерационные решения работают быстрее рекурсивных.**

**Прежде чем переходить к ситуациям, когда рекурсия полезна, обратим внимание еще на один пример, где ее использовать не следует.**

Рассмотрим частный случай рекуррентных соотношений, когда следующее значение в последовательности зависит не от одного, а сразу от нескольких предыдущих значений. Примером может служить известная последовательность Фибоначчи, в которой каждый следующий элемент есть сумма двух предыдущих:

x_n=x_{n-1}+x_{n-2},~x_0=1,~x_1=1~~~~~(5)

При «лобовом» подходе можно написать Fib(int n):

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

int Fib(int);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

int pp = 13;

printf("Для %d число Фибоначчи = %d\n", pp, Fib(pp));

system("PAUSE");

}

int Fib(int n)

{

if(n == 1 || n == 2)

return(n-1);

else

return (Fib(n-1) + Fib(n-2));

}



Каждый вызов Fib создает сразу две копии себя, каждая из копий – еще две и т.д. Количество операций растет с номером *n* экспоненциально, хотя при итерационном решении достаточно линейного по *n* количества операций.

На самом деле, **приведенный пример учит нас НЕ КОГДА рекурсию не следует использовать, а тому КАК ее не следует использовать**. В конце концов, если существует быстрое итерационное (на базе циклов) решение, то тот же цикл можно реализовать с помощью рекурсивной процедуры или функции. Например:

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

int Fib2(int, int, int);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

int pp = 13;

printf("Для %d число Фибоначчи = %d\n", pp, Fib2(1, 1, pp));

system("PAUSE");

}

int Fib2(int x1, int x2, int n)

// x1, x2 – начальные условия (1, 1)

// n – номер требуемого числа Фибоначчи

{

int x3;

if(n > 3){

x3 = x1 + x2;

x1 = x2;

x2 = x3;

return(Fib2(x1, x2, n-1));

}

else

return(x2);

}



Второй вариант решения этой задачи

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

unsigned long Fib3(long);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

for(int count = 1; count <= 15; count++)

printf("Для %2d число Фибоначчи = %d\n", count, Fib3(count));

system("PAUSE");

}

unsigned long Fib3(long n)

// n – номер требуемого числа Фибоначчи

{

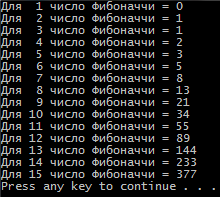
if(n == 1 || n == 2)

return(n-1);

else

return(Fib3(n - 1) + Fib3(n - 2));

}



И все же итерационные решения предпочтительны. Спрашивается, когда же в таком случае, следует пользоваться рекурсией?

**Любые рекурсивные процедуры и функции, содержащие всего один рекурсивный вызов самих себя, легко заменяются итерационными циклами. Чтобы получить что-то, не имеющее простого нерекурсивного аналога, следует обратиться к процедурам и функциям, вызывающим себя два и более раз. В этом случае множество вызываемых процедур образует уже не цепочку, а целое дерево. Существуют широкие классы задач, когда вычислительный процесс должен быть организован именно таким образом. Как раз для них рекурсия будет наиболее простым и естественным способом решения.**

## Сущность рекурсии

Процедура или функция может содержать вызов других процедур или функций. В том числе процедура может вызвать саму себя. Никакого парадокса здесь нет – компьютер лишь последовательно выполняет встретившиеся ему в программе команды и, если встречается вызов процедуры, просто начинает выполнять эту процедуру. Без разницы, какая процедура дала команду это делать.

Пример рекурсивной процедуры:

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

void Rec(int);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

Rec(3);

system("PAUSE");

}

void Rec(int a)

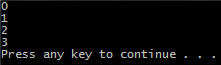
{

if(a > 0)

Rec(a - 1);

printf("%d\n", a);

}



Рассмотрим, что произойдет, если в основной программе поставить вызов, например, вида Rec(3). Ниже представлена блок-схема, показывающая последовательность выполнения операторов.

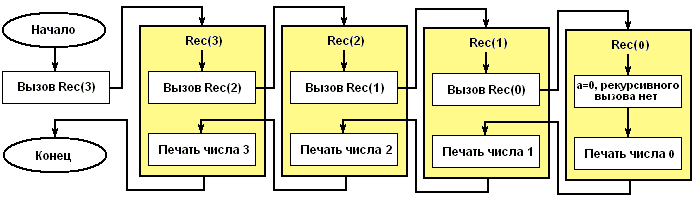


Рис. 1. Блок схема работы рекурсивной процедуры.

Процедура Rec вызывается с параметром a = 3. В ней содержится вызов процедуры Rec с параметром a = 2. Предыдущий вызов еще не завершился, поэтому можете представить себе, что создается еще одна процедура и до окончания ее работы первая свою работу не заканчивает. Процесс вызова заканчивается, когда параметр a = 0. В этот момент одновременно выполняются 4 экземпляра процедуры. Количество одновременно выполняемых процедур называют ***глубиной рекурсии***.

Четвертая вызванная процедура (Rec(0)) напечатает число 0 и закончит свою работу. После этого управление возвращается к процедуре, которая ее вызвала (Rec(1)) и печатается число 1. И так далее пока не завершатся все процедуры. Результатом исходного вызова будет печать четырех чисел: 0, 1, 2, 3.

Еще один визуальный образ происходящего представлен на рис. 2.

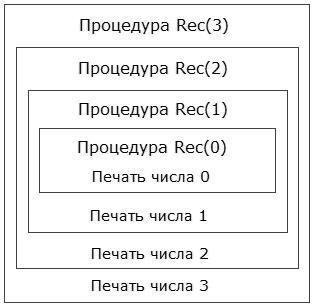
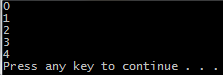


Рис. 2. Выполнение процедуры Rec с параметром 3 состоит из выполнения процедуры Rec с параметром 2 и печати числа 3. В свою очередь выполнение процедуры Rec с параметром 2 состоит из выполнения процедуры Rec с параметром 1 и печати числа 2. и т. д.

В качестве самостоятельного упражнения подумайте, что получится при вызове Rec(4).



Также подумайте, что получится при вызове описанной ниже процедуры Rec2(4), где операторы поменялись местами.

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

void Rec2(int);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

Rec2(4);

system("PAUSE");

}

void Rec2(int a)

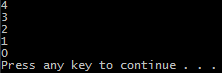
{

printf("%d\n", a);

if(a > 0)

Rec2(a - 1);

}



Обратите внимание, что в приведенных примерах рекурсивный вызов стоит внутри условного оператора. Это необходимое условие для того, чтобы рекурсия когда-нибудь закончилась. Также обратите внимание, что сама себя процедура вызывает с другим параметром, не таким, с каким была вызвана она сама. Если в процедуре не используются глобальные переменные, то это также необходимо, чтобы рекурсия не продолжалась до бесконечности.

## Виды рекурсии

**Рекурсия** - метод определения класса объектов или методов предварительным заданием одного или нескольких (обычно простых) его базовых случаев или методов, а затем заданием на их основе правила построения определяемого класса, ссылающегося прямо или косвенно на эти базовые случаи.

Другими словами, **рекурсия** - способ общего определения объекта или действия через себя, с использованием ранее заданных частных определений. Рекурсия используется, когда можно выделить само подобие задачи.

**Рекурсивный алгоритм (процедура, функция):**

* алгоритм называется рекурсивным, если в его определении содержится прямой или косвенный вызов этого же алгоритма;
* рекурсивная функция - одно из математических уточнений интуитивного понятия вычислимой функции.

Адаптивный рекурсивный алгоритм - алгоритм, который благодаря рекурсивности учитывает те или иные индивидуальные характеристики решаемой задачи из области своего определения.

**Базис рекурсии** - это предложение, определяющее некую начальную ситуацию или ситуацию в момент прекращения. Как правило, в этом предложении записывается некий простейший случай, при котором ответ получается сразу, даже без использования рекурсии.

**Шаг рекурсии** - это правило, в теле которого обязательно содержится, в качестве подцели, вызов определяемого предиката. (Предикат ([лат.](https://ru.wikipedia.org/wiki/%D0%9B%D0%B0%D1%82%D0%B8%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Praedicatum — сказанное) в [логике](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0) и [лингвистике](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B3%D0%B2%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B0) — [сказуемое](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D0%B0%D0%B7%D1%83%D0%B5%D0%BC%D0%BE%D0%B5) [суждения](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%B6%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5), то, что высказывается (утверждается или отрицается) о [субъекте](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Предикат находится с субъектом в [предикативном отношении](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F) и показывает наличие (отсутствие) у предмета некоторого признака). Прмер: «Это животное — кошка», «Авраам — отец Исаака», «Климат на севере суровый»

**Подпрограмма** - все, что находится внутри рекурсивной функции.

**Глубина рекурсии** - количество одновременно выполняемых процедур**.**

Основное правило рекурсии: до рекурсивного вызова должна стоять проверка на возврат из рекурсии. **Существуют следующие виды рекурсии:**

### Прямая рекурсия

**Прямая рекурсия** - непосредственный вызов алгоритма (функции, процедуры, метода) из текста самого метода.

В данном случае функция r1( ) вызывает саму себя.

#include<iostream>

usingnamespace **std;**

void **r1 (**int **a);**

void **r2 (**int **a);**

void **r3 (**int **a);**

void **r1(**int **a)**

{

**cout <<** "#" **<< a <<**" function r1" **<< endl;**

if **(a < 6)**

**r1(a+1);**

}

int **main ( )**

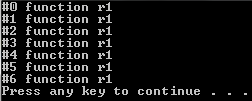
{

**r1 (0);**

**system(**"PAUSE");

return **NULL;**

}



### Косвенная рекурсия.

При косвенной рекурсии имеется циклическая последовательность вызовов нескольких алгоритмов.

В данном случае функция r1( ) вызывает функцию r2( ), которая вызывает r3( ).

Функция r3( ) в свою очередь снова вызывает r1( ).

#include<iostream>

usingnamespace **std;**

void **r1 (**int **a);**

void **r2 (**int **a);**

void **r3 (**int **a);**

void **r1(**int **a)**

{

**cout <<** "#" **<< a <<** " function r1" **<< endl;**

if **(a < 6)**

**r2(a+1);**

}

void **r2(**int **a)**

{

**cout <<** "#" **<< a <<** " function r2" **<< endl;**

if **(a < 6)**

**r3(a+1);**

}

void **r3(**int **a)**

{

**cout <<** "#" **<< a <<** " function r3" **<< endl;**

if **(a < 6)**

**r1(a+1);**

}

int **main ( )**

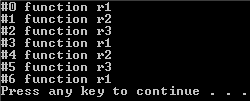
{

**r1 (0);**

**system(**"PAUSE");

return **NULL;**

}



### Линейная рекурсия

**Линейная рекурсия** - если исполнение подпрограммы приводит только к одному вызову этой же самой подпрограммы, то такая рекурсия называется линейной.

#include<iostream>

usingnamespace **std;**

void **function(**int **a);**

void **function (**int **a)**

{

**cout <<** " input a= " **<< a << endl;**

if **(a > 0)**

**function(a - 1);**

**cout <<** "output a= " **<< a << endl;**

}

int **main ( )**

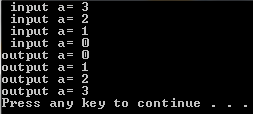
{

**function(3);**

**system(**"PAUSE");

return **NULL;**

}



### Ветвящаяся рекурсия

**Ветвящаяся рекурсия** - если каждый экземпляр подпрограммы может вызвать себя несколько раз, то рекурсия называется нелинейной или "ветвящейся".

#include<iostream>

usingnamespace **std;**

int **function(**int **a);**

int **function (**int **a)**

{

if **(a > 3)**

**a = function (a - 1) \* function(a - 2);**

return **a;**

}

int **main ()**

{

**cout << function(6) << endl;**

**system(**"PAUSE");

return **NULL;**

}



### Бесконечная рекурсия

**Бесконечная рекурсия** (на самом деле это условное обозначение так как при переполнении памяти компьютера программа выдаст ошибку и/или завершит ее в аварийном режиме).

Одна из самых больших опасностей рекурсии - бесконечный вызов функцией самой себя.

Например:

void **function()**

{

**function();**

}

Другой пример:

int **Function (**unsignedint **n)**

// Unsigned int - тип, содержащий неотрицательные значения

{

if **(n > 0)**

**{**

**Function(n++);**

return **n;**

**}**

else

return **0;**

}

При использовании подобных алгоритмов появляется ошибка, предупреждающая о переполнении стека. Причиной такой проблемы чаще всего является отсутствие базиса, либо других точек останова, так же неправильно заданные точки прерывания рекурсии

### Сложная рекурсия

Возможна чуть более сложная схема: функция A вызывает функцию B, а та в свою очередь вызывает A. Это называется *сложной рекурсией*. При этом оказывается, что описываемая первой процедура должна вызывать еще не описанную. Чтобы это было возможно, требуется использовать [опережающее описание](http://tvd-home.ru/prog/11_9).

Опережающее описание

В теле каждой процедуры или функции может содержаться вызов других процедур или функций при условии, что они описаны раньше, чем процедура или функция их вызывающая. Однако есть возможность вызывать и те процедуры, которые описаны после вызывающей. Для этого надо скопировать заголовок вызываемой подпрограммы и разместить его выше всех описаний. Например:

//Опережающее описание функции F1 - заголовок без тела функции

float F1(float x);

float F6()

{

float x;

//Какие-то действия...

//Вызов функции F1 возможен благодаря опережающему описанию

x = F1(x);

//Какие-то действия...

}

//Описание функции F1, теперь тело функции присутствует

float F1(float x)

{

//Какие-то действия...

return(F1); //Что-то

}

Это может понадобиться, когда есть две процедуры, вызывающие другу друга, например:

#include<stdio.h>

#include<iostream>

#include<conio.h>

#include<windows.h>

usingnamespace **std;**

void **A(**int **n);** //Опережающее описание первой процедуры

void **B(**int **n);** //Опережающее описание второй процедуры

void **main()**

{

//FONT: Lucida Console

**SetConsoleCP(1251);** // установка кодовой страницы win-cp 1251 в поток ввода

**SetConsoleOutputCP(1251);** //установка кодовой страницы win-cp 1251 в поток вывода

**setlocale(LC\_ALL,** "Rus");

**A(3);**

**system(**"PAUSE");

}

void **A(**int **n){** //Полное описание процедуры A

**printf(**"A>>> %2d\n", n);

**B(n-1);**

}

void **B(**int **n)** //Полное описание процедуры B

{

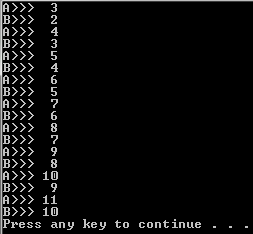
**printf(**"B>>> %2d\n", n);

if(n < 10)

**A(n+2);**

}

Процедура A вызывает процедуру B, та в свою очередь вызывает A, а та снова B и т.д. Данная цепочка вызовов закончится, поскольку B вызывается каждый раз с на единицу большим значением параметра n. Когда n перестанет быть меньше 10, процедура B завершится, не вызвав A, что позволит завершиться и прочим процедурам в цепочке.



Опережающее описание процедуры B позволяет вызывать ее из процедуры A. Опережающее описание процедуры A в данном примере не требуется и добавлено из эстетических соображений.

Если обычную рекурсию можно уподобить уроборосу (рис. 3), то образ сложной рекурсии можно почерпнуть из известного детского стихотворения, где «Волки с перепуга, скушали друг друга». Представьте себе двух съевших друг друга волков, и вы поймете сложную рекурсию.

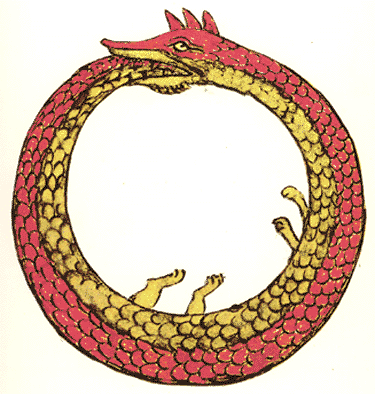


Рис. 3. Уроборос – змей, пожирающий свой хвост. Рисунок из алхимического трактата «Synosius» Теодора Пелеканоса (1478г).

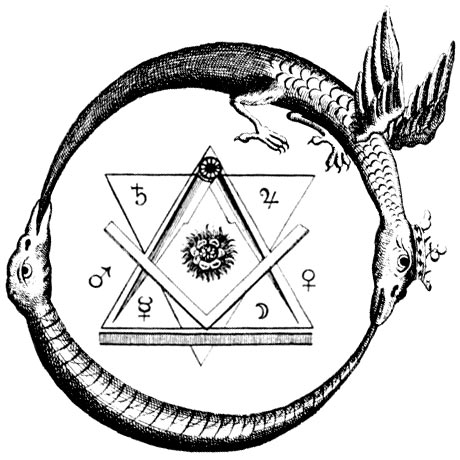


Рис. 4. Сложная рекурсия.

### Типовой пример.

Одной и наиболее востребованной операцией в экономической сфере является расчет процентов по вкладу**. Пример задачи**: вкладчик положил в банк сумму в **sum** денежных единиц под **pr** процентов за один период времени (год, месяц, неделя и т.д.). Составить программу, возвращающую величину вклада по истечении **m** периодов времени (**m** = 24)

#include<iostream>

#include<windows.h>

usingnamespace **std;**

double **rec\_fun(**double **sum,** double **pr,** int **m,** int **i)**

{

if(m==i)

return **sum;**

**sum+=(sum\*(pr/100));**

return **rec\_fun(sum, pr, m, i+1);**

}

int **main ()**

{

int **m;**

double **sum, pr;**

**setlocale(LC\_ALL,** "Rus");

**cout <<** "Введите начальную сумму вклада: "**;**

**cin >> sum;**

**cout <<** "Ежемесячный процент по вкладу: ";

**cin >> pr;**

**cout <<** "Количество месяцев вклада: ";

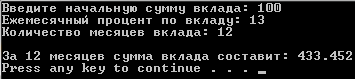
**cin >> m;**

**cout << endl <<** "За " **<< m <<** " месяцев сумма вклада составит: " **<< rec\_fun(sum, pr, m, 0) << endl;**

**system(**"PAUSE");

return **0;**

}



## Имитация работы цикла с помощью рекурсии

Если процедура вызывает сама себя, то, по сути, это приводит к повторному выполнению содержащихся в ней инструкций, что аналогично работе цикла. Некоторые языки программирования не содержат циклических конструкций вовсе, предоставляя программистам организовывать повторения с помощью рекурсии (например, **Пролог**, где рекурсия - основной прием программирования).

Для примера сымитируем работу цикла for. Для этого нам потребуется переменная счетчик шагов, которую можно реализовать, например, как параметр процедуры.

### Пример 1. Инкремент счетчика итераций

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

void LoopImitation(int, int);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

LoopImitation(1, 10);

system("PAUSE");

}

void LoopImitation(int i, int n)

{

//Первый параметр – счетчик шагов, второй параметр – общее количество шагов

//Здесь любые инструкции, которые будут повторятся

printf("Hello N %d\n", i);

if(i < n) //Пока счетчик цикла не станет равным максимальному

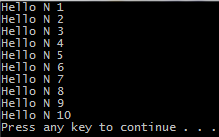
//значению n, повторяем инструкции путем вызова

//нового экземпляра процедуры

LoopImitation(i+1, n);

}

Результатом вызова вида LoopImitation(1, 10) станет десятикратное выполнение инструкций с изменением счетчика от 1 до 10. В данном случае будет напечатано:



Вообще, не трудно видеть, что параметры процедуры это пределы изменения значений счетчика.

Можно поменять местами рекурсивный вызов и подлежащие повторению инструкции, как в следующем примере.

### Пример 2. Декремент счетчика итераций

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

void LoopImitation2(int, int);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

LoopImitation2(1, 10);

system("PAUSE");

}

void LoopImitation2(int i, int n)

{

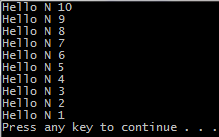
if(i < n)

LoopImitation2(i+1, n);

printf("Hello N %d\n", i);

}

В этом случае, прежде чем начнут выполняться инструкции, произойдет рекурсивный вызов процедуры. Новый экземпляр процедуры также, прежде всего, вызовет еще один экземпляр и так далее, пока не дойдем до максимального значения счетчика. Только после этого последняя из вызванных процедур выполнит свои инструкции, затем выполнит свои инструкции предпоследняя и т.д. Результатом вызова LoopImitation2(1, 10) будет печать приветствий в обратном порядке:



### Пример 3. Двойной цикл

Если представить себе цепочку из рекурсивно вызванных процедур, то в примере 1 мы проходим ее от раньше вызванных процедур к более поздним. В примере 2 наоборот от более поздних к ранним.

Наконец, рекурсивный вызов можно расположить между двумя блоками инструкций. Например:

#include<stdio.h>

#include<iostream>

#include<conio.h>

#include<windows.h>

usingnamespace **std;**

void **LoopImitation3(**int, int);

void **main()**

{

//FONT: Lucida Console

**SetConsoleCP(1251);** // установка кодовой страницы win-cp 1251 в поток ввода

**SetConsoleOutputCP(1251);** //установка кодовой страницы win-cp 1251 в поток вывода

**setlocale(LC\_ALL,** "Rus");

**LoopImitation3(1, 10);**

**system(**"PAUSE");

}

void **LoopImitation3(**int **i,** int **n)**

{

//Здесь может располагаться первый блок инструкций

**printf(**"Hello N %2d\n", i);

if(i < n)

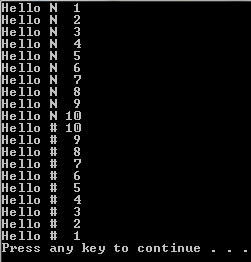
**LoopImitation3(i+1, n);**

//Здесь может располагаться второй блок инструкций

**printf(**"Hello # %2d\n", i);

}

Здесь сначала последовательно выполнятся инструкции из первого блока затем в обратном порядке инструкции второго блока. При вызове LoopImitation3(1, 10) получим:



Потребуется сразу два цикла, чтобы сделать то же самое без рекурсии.

Тем, что выполнение частей одной и той же процедуры разнесено по времени можно воспользоваться. Например:

### Пример 4: Перевод числа в двоичную систему.

Получение цифр двоичного числа, как известно, происходит с помощью деления с остатком на основание системы счисления 2. Если есть число x, то его последняя цифра в его двоичном представлении равна

c_1=x~\mathrm{mod}~2.

Взяв же целую часть от деления на 2:

x_2=x~\mathrm{div}~2,

получим число, имеющее то же двоичное представление, но без последней цифры. Таким образом, достаточно повторять приведенные две операции пока поле очередного деления не получим целую часть равную 0**. Без рекурсии это будет выглядеть так:**

while(x > 0) do

{

c = x % 2; //c := x mod 2;

x = x / 2; //x := x div 2;

printf("%d", c);

}

Проблема здесь в том, что цифры двоичного представления вычисляются в обратном порядке (сначала последние). Чтобы напечатать число в нормальном виде придется запомнить все цифры в элементах массива и выводить в отдельном цикле.

**С помощью рекурсии нетрудно добиться вывода в правильном порядке без массива и второго цикла.** А именно:

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

void BinaryRepresentation(int);

void main()

{

//FONT: Lucida Console

SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода

SetConsoleOutputCP(1251); //установка кодовой страницы win-cp 1251 в поток вывода

setlocale(LC\_ALL, "Rus");

printf("10 ==> ");

BinaryRepresentation(10);

printf("\n");

system("PAUSE");

}

void BinaryRepresentation(int x)

{

int c;

//Первый блок. Выполняется в порядке вызова процедур

c = x % 2; //c := x mod 2;

x = x / 2; //x := x div 2;

//Рекурсивный вызов

if(x > 0)

BinaryRepresentation(x);

//Второй блок. Выполняется в обратном порядке

printf("%d", c);

}

Вообще говоря, никакого выигрыша мы не получили. Цифры двоичного представления хранятся в локальных переменных, которые свои для каждого работающего экземпляра рекурсивной процедуры. То есть, память сэкономить не удалось. Даже наоборот, тратим лишнюю память на хранение многих локальных переменных x. Тем не менее, такое решение кажется мне красивым.



## Произвольное количество вложенных циклов

**Разместив рекурсивные вызовы внутри цикла, по сути, получим вложенные циклы, где уровень вложенности равен глубине рекурсии.**

Для примера напишем процедуру, печатающую все возможные сочетания из k чисел от 1 до n (\mathrm{C}_n^k). Числа, входящие в каждое сочетание, будем печатать в порядке возрастания. Сочетания из двух чисел (k=2) печатаются так:

#include <iostream>

using namespace std;

int main()

{

// переходим на русский язык в консоли

setlocale(LC\_ALL, "Rus");

int n = 5;

for (int i1 = 1; i1 <= n; i1++)

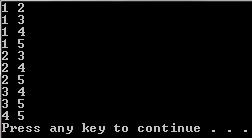
for (int i2 = i1+1; i2 <= n; i2++)

cout << i1 << " " << i2 << "\n";

system("PAUSE");

return 0;

}



Сочетания из трех чисел (k=3) так:

#include <iostream>

using namespace std;

int main()

{

// переходим на русский язык в консоли

setlocale(LC\_ALL, "Rus");

int n = 5;

for (int i1 = 1; i1 <= n; i1++)

for (int i2 = i1+1; i2 <= n; i2++)

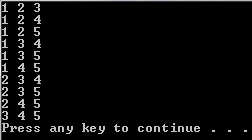
for (int i3 = i2+1; i3 <= n; i3++)

cout << i1 << " " << i2 << " " << i3 << "\n";

system("PAUSE");

return 0;

}



Однако, если количество чисел в сочетании задается переменной, то придется прибегнуть к рекурсии.

// все комбинации цифр от 1 до k в n-значном числе

#include <iostream>

using namespace std;

void Combinations( int, int, int x[], int);

int main()

{

int t[10000];

int n=7;

int k=4;

int d=0;

// переходим на русский язык в консоли

setlocale(LC\_ALL, "Rus");

cout << "Значность числа n= " << n << "\n";

cout << "Комбинации цифр k= " << k << "\n";

cout << "Счетчик глубины рекурсии d= " << d << "\n\n";

Combinations( n, k, t, d);

system("PAUSE");

return 0;

}

void Combinations( int n, int k, int Indexes[], int d)

{

//Indexes Массив, в котором будем формировать сочетания

//d Счетчик глубины рекурсии

int i, i\_min;

if(d < k){

if(d == 0)

i\_min = 1;

else

i\_min = Indexes[d-1] + 1;

for (i = i\_min; i <= n; i++){

Indexes[d] = i;

Combinations(n, k, Indexes, d+1);

}

}

else{

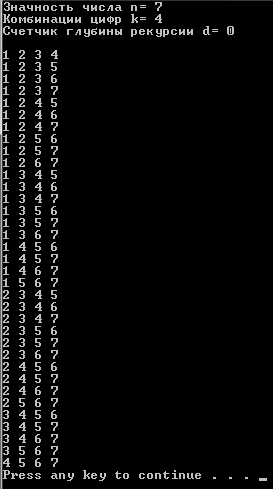
for (i = 0; i <= k-1; i++)

cout << Indexes[i] << " ";

cout << "\n";

}

}



// все комбинации цифр от 1 до k в n-значном числе

#include <iostream>

using namespace std;

void Combinations( int, int, int Indexes[], int);

int main()

{

int Indexes[10000] = {0};

int n=4;

int k=3;

int d=0;

// переходим на русский язык в консоли

setlocale(LC\_ALL, "Rus");

Combinations( n, k, Indexes, d);

system("PAUSE");

return 0;

}

void Combinations( int n, int k, int Indexes[], int d)

{

//Indexes Массив, в котором будем формировать сочетания

//d Счетчик глубины рекурсии

int i, i\_min;

if(d < k){

if(d = 0)

i\_min = 1;

else

i\_min = Indexes[d-1] + 1;

for (i = i\_min; i <= n; i++){

Indexes[d] = i;

Combinations(n, k, Indexes, d+1);

}

}

else{

for (i = 0; i <= k-1; i++)

cout << Indexes[i] << " ";

cout << "\n";

}

}

**Второй вариант этой же задачи.**

// все комбинации цифр от 1 до k в n-значном числе

#include <iostream>

using namespace std;

int c=1;

int ind[100];

int i=1;

int asd=1;

void comb(int n, int k);

int main(){

for (int j=0; j<100; j++)

ind[j]=1;

int n,k;

//cin >> n >> k;

n = 4;

k = 3;

cout << "n= " << n << " k= " << k << "\n";

for (int j=1; j<=n; j++)

asd = asd\*k;

for (i=1; i<=n; i++)

cout << ind[i-1] << ' ';

cout << endl;

c++;

comb(n,k);

system("pause");

return 0;

}

void comb(int n, int k)

{

for (i=n-1; i>=1; i--)

if((ind[i]<k) && (ind[i-1]<k)){

ind[i]++;

break;

}

else if(ind[i-1] < k){

ind[i]=1;

ind[i-1]++;

break;

}

else if((ind[i-1] == k) && (ind[i] == k)){

if(i-1 != 0){

ind[i-2]++;

ind[i-1]=1;

ind[i]=1;

break;

}

}

else if(ind[i-1] == k){

ind[i]++;

break;

}

for(i=1; i<=n; i++)

cout << ind[i-1] << ' ';

cout << endl;

c++;

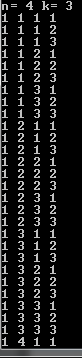
if(c == asd+1)

return;

else

comb(n,k);

}



**Третий вариант этой же задачи.**

// все комбинации цифр от 1 до k в n-значном числе

#include <iostream>

using namespace std;

int ind[100];

void comb(int n, int k, int t);

int main()

{

int n=4, k=3;

//cin>>n>>k;

cout << "n= " << n << " k= " << k << "\n";

comb(n, k, 0);

system("pause");

return 0;

}

void comb(int n, int k, int t)

{

if(t == n){

for(int i=0; i<n; i++)

cout << ind[i] << " ";

cout << endl;

return;

}

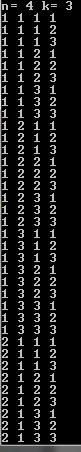
for(int i=1; i<=k; i++){

ind[t] = i;

comb(n, k, t+1);

}

}



## Графы

### Понятие и представление графа: матрица смежности, список смежности

***Графом*** называют графическое изображение, состоящее из ***вершин* (*узлов*)** и соединяющих некоторые пары вершин *ребер* (рис. 11а).

Более строго: **граф** – совокупность множества вершин и множества ребер. Множество ребер – подмножество евклидова квадрата множества вершин (то есть ребро соединяет ровно две вершины).

Ребрам можно также присвоить **направление**. Граф в этом случае называется **ориетированным** (рис. 11б).

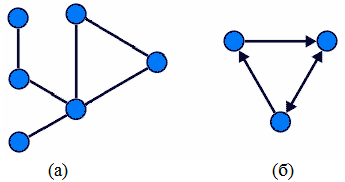
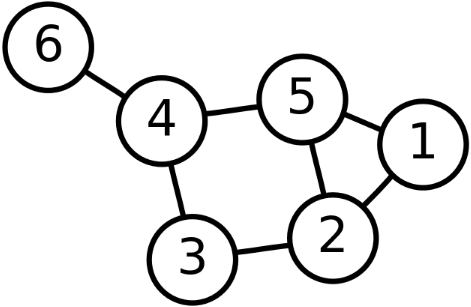


Рис. 11. (а) Граф. (б) Ориентированный граф.

**Графы** – фундаментальное понятие как в математике, так и в информатике. Проще всего объяснить его с помощью аналогии с дорожной системой. Существует определённый набор городов, некоторые из которых связаны дорогами, которые могут быть как односторонними, так и двухсторонними. Вся эта структура и называется графом.

Ну а более формально, граф – комбинация набора вершин и набора рёбер. Вершины – это города, а рёбра – дороги. Визуально граф можно представить так:

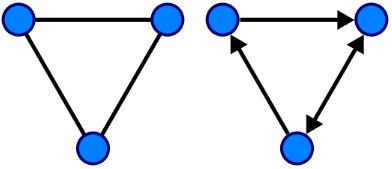


Этот граф состоит из 6 вершин, пронумерованных начиная с единицы, и 7 двухсторонних рёбер. Рёбра обычно записывают в виде пар вершин, которые они соединяют: **1-2, 1-5, 2-3, 2-5, 3-4, 4-5, 4-6**.

**Ориентированные и неориентированные графы**

Мы уже упоминали, что "дороги" в графе могут быть как односторонними, так и двухсторонними. Для этого свойства существует отдельный термин: односторонние "дороги" называются **ориентированными** рёбрами (или дугами), а двухсторонние – **неориентированными**.

**Граф, в котором все рёбра неориентированные, также называют неориентированным, а граф с ориентированными рёбрами, соответственно, ориентированным.**



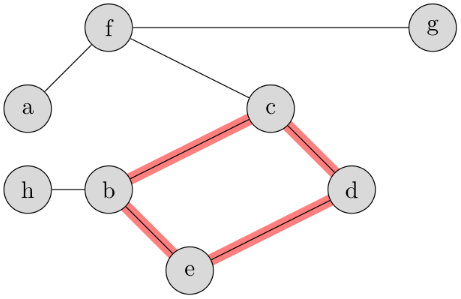
Слева изображён неориентированный граф, а справа – ориентированный. Как несложно догадаться, левый граф можно обходить как по часовой стрелке, так и против, а правый можно полностью обойти только по часовой, хотя одно из ребёр в нём также неориентированное (считается, что это два противоположных ориентированных ребра).

**Пути и циклы**

**Путём в графе** называется последовательность вершин, каждая из которых соединена со следующей ребром. Чаще всего под "путём" подразумевают простой путь, все вершины которого различны. **Путь, который проходит через какую-либо вершину более одного раза называют сложным путём.**

**Если первая вершина пути совпадает с последней, то такой путь называют циклом.**

Приведём примеры на этом графе:

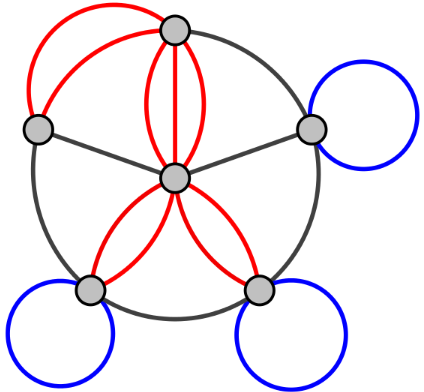


Из множества возможных простых путей самый длинный: **a−f−c−d−e−b−h**(существуют и другие пути с такой же длиной).

Циклом является путь **b−c−d−e−b**(выделен цветом). Можно начать и с любой другой вершины, например, **c−d−e−b−c**.

**Кратные рёбра и петли**

Существует множество разновидностей графов, и среди них встречаются довольно специфические. В частности, так называемые **мультиграфы** разрешают наличие между двумя вершинами нескольких рёбер (называемых **кратными рёбрами**), а также наличие петель. **Петля** – ребро, входящее в ту же вершину, из которой исходит. Выглядят они следующим образом:

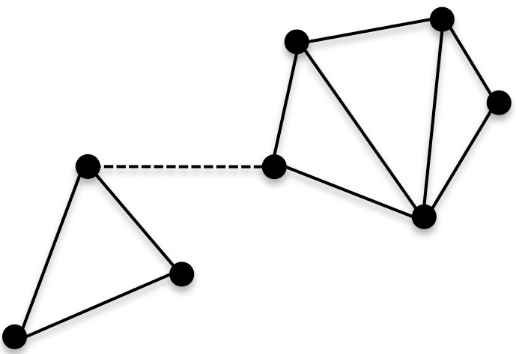


Красным выделены **кратные** рёбра, а синим – петли.

Мультиграфы встречаются в задачах реже чем обычные графы (называемые простыми), но всё же встречаются, поэтому стоит иметь о них элементарное представление.

**Связные графы**

Граф называется связным, если между любой парой вершин существует хотя бы один путь. Как пример рассмотрим следующий граф:



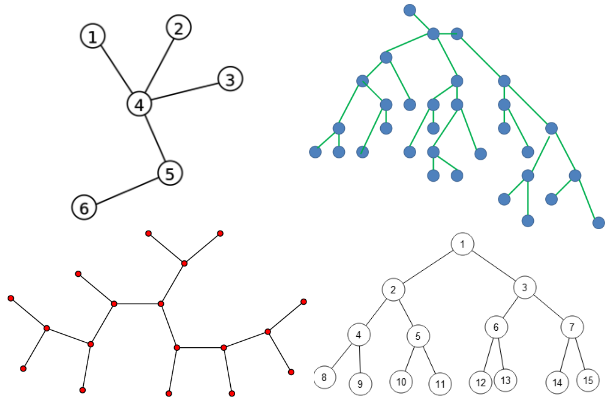
Одно из рёбер проведено штрихами. Если это ребро присутствует, то граф является **связным**. Если же его убрать, то связность теряется, граф разбивается на две части, друг с другом не связанные. Такие части называются компонентами связности.

**Определение дерева**

**Дерево** – вид графа, который можно назвать самым простым, но они обладают множеством особых свойств и встречаются в задачах чуть ли не чаще остальных графов.

**Дерево – это связный граф без циклов, петель и кратных рёбер.**

Все изображённые графы являются деревьями:



Среди множества свойств деревьев можно выделить два самых известных:

* Количество рёбер связано с количеством вершин формулой E=V−1.
* Между любой парой вершин существует ровно один путь.

### Способы представления графов

Существует **два основных способа представления графов в программировании**. Один из них, **список смежности** и **матрица смежности**, используется гораздо реже, но очень просто реализуется.

Матрица смежности

Граф из **N** вершин задаётся матрицей (двумерным массивом) **N\*N**, в которой **g[i][j]** – логическое значение, **true** или **false**, обозначающее, существует ли ребро из вершины **i** в вершину **j**.

В качестве примера решим простую задачу: для каждой вершины графа выведем количество рёбер, смежных с ней.

#include <iostream>

using namespace std;

bool graph[1000][1000];

int main() {

int n, m; //количество вершин и рёбер соответственно

setlocale(LC\_ALL, "Rus");

//cin >> n >> m;

cout << "количество вершин n= ";

cin >> n;

cout << "количество рёбер m= ";

cin >> m;

cout << "\n";

for (int i = 0; i < m; i++) {

int u, v; //номера вершин, соединённых очередным ребром

cout << "номера вершин, соединённых очередным ребром (u v) ";

cin >> u >> v;

u--, v--; //Здесь стоит остановиться и вдуматься.

//Чаще всего в задачах вершины будут нумероваться с 1 до N,

//в отличие от индексации массивов в C++.

//У этой проблемы есть два решения.

//Первое: работать с номерами "как есть": создавать массивы размером N + 1,

//использовать циклы от 1 до N, и т.д.

//Второе: уменьшать номера вершин на единицу при вводе, и увеличивать обратно при выводе

//Какое из них использовать - ваш личный выбор.

//Для меня 1-индексация в С++ выглядит очень чужеродно, поэтому я использую второе решение.

graph[u][v] = graph[v][u] = true; //Если бы граф был ориентированным, то обратное ребро мы бы не создавали.

}

cout << "\n";

for (int i = 0; i < n; i++) {

int c = 0;

for (int j = 0; j < n; j++) {

if (graph[i][j]) {

c++;

}

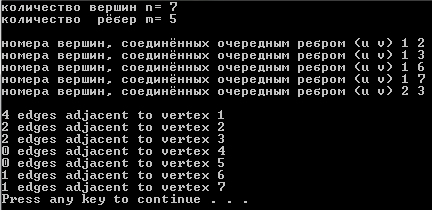
}

cout << c << " edges adjacent to vertex " << i + 1 << endl;

}

system("PAUSE");

}



Преимущества матрицы смежности:

* Сложность проверки наличия ребра между двумя вершинами: O(1)

Недостатки матрицы смежности:

* Занимает N2N2 памяти, что неприемлемо для достаточно больших графов.
* Сложность перебора всех вершин, смежных с данной: O(N)

Список смежности

Гораздо чаще для представления графов используется список смежности. Его идея заключается в хранении для каждой вершины расширяемого массива (вектора), содержащего всех её соседей.

Решим ту же задачу с использованием списка смежности (и С++11 для for-each):

**https://brestprog.neocities.org/lections/graphs.html**

//#include <iostream>

//#include <cliext/vector>

////#include <bits/stdc++.h>

//using namespace std;

#include <iostream>

#include <vector>

#include <algorithm>

#include <utility>

using namespace std;

vector<int> graph[100000]; //массив из 100000 векторов.

int main()

{

int n, m; //количество вершин и рёбер соответственно

setlocale(LC\_ALL, "Rus");

//cin >> n >> m;

cout << "количество вершин n= ";

cin >> n;

cout << "количество рёбер m= ";

cin >> m;

cout << "\n";

for (int i = 0; i < m; i++) {

int u, v;

cout << "номера вершин, соединённых ребром (u v) ";

cin >> u >> v;

u--, v--;

graph[u].push\_back(v);

graph[v].push\_back(u);

}

for (int i = 0; i < n; i++) {

int c = 0;

for (int v : graph[i]) { //можно было бы просто записать "int c = graph[i].size();",

c++; //но такая реализация показывает, как можно перебирать

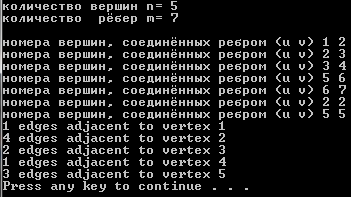
} //соседние вершины.

cout << c << " edges adjacent to vertex " << i + 1 << endl;

}

system("PAUSE");

}



Если требуется также удалять рёбра, то вместо вектора нужно использовать std::set.

Преимущества списка смежности:

* Использует O(M) памяти, что оптимально.
* Позволяет быстро перебирать соседей вершины.
* Позволяет за O(logN) проверять наличие ребра и удалять его (при использовании std::set).

Недостатки списка смежности:

* При работе с насыщенными графами (количество рёбер близко к N2) скорости O(logN) может не хватать (единственный повод использовать матрицу смежности).
* Для взвешенных графов приходится хранить vector<pair<int, int>>, что усложняет код.

Теория графов находит применения в самых разных областях. Несколько примеров:

* Логистика и транспортные системы. Вершинами будут склады с товарами или пункты назначения, а ребра – дороги, их соединяющие.
* Маршрутизация сетей. Вершины – компьютеры, соединенные в сеть, ребра – связи между ними. Решается задача о путях передачи данных с одного компьютера на другой.
* Компьютерная химия. Модели в виде графов используются для описания путей протекания сложных реакций. Вершины – участвующие в реакциях вещества, ребра – пути превращений веществ. Также графом является изображение структур молекул: вершины – атомы, ребра – химические связи.
* Электрические сети.
* Сайты в Интернете можно считать узлами ориентированного графа, ребрами которого будут гиперссылки.
* И т. д.

Современная теория графов представляет собой мощную формальную систему, имеющую необозримое множество применений.

***Путем*** или ***цепью*** в графе называется последовательность вершин, в которой каждая вершина соединена ребром со следующей. Пути, в которых начальная и конечная вершина совпадают, называют ***циклами***. Если для каждой пары вершин существует путь их соединяющих, то такой граф называют ***связным***.

В программировании используются три способа хранения в памяти информации о структуре графов.

**1) *Матрицы смежности***

Квадратная матрица M, где как строки, так и столбцы соответствуют вершинам графа. Если вершины с номерами i и j соединены ребром, то M*ij* = 1, иначе M*ij* = 0. Для неориентированного графа матрица, очевидно, симметрична. Ориентированный граф задается антисимметричной матрицей. Если ребро выходит из узла *i* и приходит в узел *j*, то M*ij* = 1, а симметричный элемент M*ji* = -1.

**2) *Матрица инцидентности***

Столбцы матрицы соответствуют вершинам, а строки ребрам. Если ребро с номером *i* соединяет вершины с номерами *j* и *k*, то элементы матрицы I*ij* = I*ik* = 1. Остальные элементы *i*-й строки равны 0.

**3) *Список ребер***

Просто набор пар номеров вершин, соединенных ребрами.

Рассмотренные выше деревья являются частным случаем графов. Деревом будет любой связный граф, не содержащий циклов.

Задачи, возникающие в теории графов многочисленны и разнообразны. Про них пишутся толстые книги, и нет никакой возможности сколько-нибудь полно их здесь обозреть. Поэтому мы ограничимся замечанием, что многие из этих задач требуют систематического перебора вершин. Если перебирать вершины, связанные ребрами и при этом посещать каждую вершину только один раз, то множество посещаемых алгоритмом вершин будет образовывать дерево, а сам алгоритм естественно сделать рекурсивным.

Например, классической задачей является поиск пути из одной вершины в другую. Алгоритм поиска должен будет построить дерево возможных путей из начальной вершины, концевыми узлами которого будут вершины, из которых нельзя попасть ни в какую вершину, не принадлежащую ранее построенной ветви (не помеченную как уже посещенную). Задача будет решена, когда один из концевых узлов совпадет с конечной вершиной, путь в которую требуется найти.

### Использование графа с использованием STL для конкурентного программирования

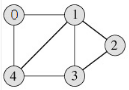
**https://www.geeksforgeeks.org/graph-implementation-using-stl-for-competitive-programming-set-1-dfs-of-unweighted-and-undirected/**

Мы ввели основы графа и его представления. Здесь используется другое представление на основе STL, которое может быть полезно для быстрого внедрения графика с использованием векторов. Реализация предназначена для отображения списка смежности.

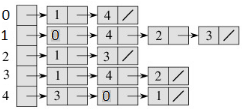
**Замечание**

[Стандартная библиотека шаблонов](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BE%D0%B2) (Standard Template Library, STL)

Ниже приведен пример неориентированного и невзвешенного графа с 5 вершинами.



Ниже представлен список граф смежности.



Используется вектор в STL для реализации графа с использованием представления списка смежности.

**Vector**: контейнер последовательности. Здесь мы используем его для хранения списков смежности всех вершин. Мы используем число вершин в качестве индекса в этом векторе.

Идея состоит в том, чтобы представлять граф как массив векторов, так что каждый вектор представляет собой список смежности вершины. Ниже приведена полная версия программы на C ++ для DFS Traversal.

// A simple representation of graph using STL,

// for the purpose of competitive programming

//#include<bits/stdc++.h>

//using namespace std;

#include <iostream>

#include <vector>

#include <algorithm>

#include <utility>

using namespace std;

// A utility function to add an edge in an

// undirected graph.

void addEdge(vector<int> adj[], int u, int v)

{

adj[u].push\_back(v);

adj[v].push\_back(u);

}

// A utility function to do DFS of graph

// recursively from a given vertex u.

void DFSUtil(int u, vector<int> adj[],

vector<bool> &visited)

{

visited[u] = true;

cout << u << " ";

for (int i = 0; i<adj[u].size(); i++)

if (visited[adj[u][i]] == false)

DFSUtil(adj[u][i], adj, visited);

}

// This function does DFSUtil() for all

// unvisited vertices.

void DFS(vector<int> adj[], int V)

{

vector<bool> visited(V, false);

for (int u = 0; u<V; u++)

if (visited[u] == false)

DFSUtil(u, adj, visited);

}

// Driver code

int main()

{

int const V = 5;

vector<int> adj[V];

addEdge(adj, 0, 1);

addEdge(adj, 0, 4);

addEdge(adj, 1, 2);

addEdge(adj, 1, 3);

addEdge(adj, 1, 4);

addEdge(adj, 2, 3);

addEdge(adj, 3, 4);

DFS(adj, V);

cout << endl;

system("PAUSE");

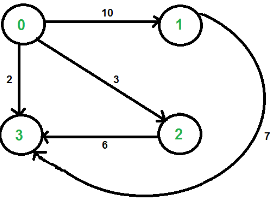
return 0;

}



**https://www.geeksforgeeks.org/graph-implementation-using-stl-for-competitive-programming-set-2-weighted-graph/**

Здесь обсуждается взвешенное графическое представление с использованием STL. Реализация предназначена для представления списка смежности взвешенного графика.



Мы используем два контейнера STL для представления графика:

* vector: контейнер последовательности. Здесь мы используем его для хранения списков смежности всех вершин. Мы используем число вершин в качестве индекса в этом векторе.
* pair: простой контейнер для хранения пары элементов. Здесь мы используем его для хранения соседнего числа вершин и веса края, соединяющегося с соседним.

Идея состоит в том, чтобы использовать вектор парных векторов. Ниже код реализует то же самое.

// C++ program to represent undirected and weighted graph

// using STL. The program basically prints adjacency list

// representation of graph

#include <iostream>

#include <vector>

#include <algorithm>

#include <utility>

using namespace std;

// To add an edge

void addEdge(vector <pair<int, int> > adj[], int u,

int v, int wt)

{

adj[u].push\_back(make\_pair(v, wt));

adj[v].push\_back(make\_pair(u, wt));

}

// Print adjacency list representaion ot graph

void printGraph(vector<pair<int, int> > adj[], int V)

{

int v, w;

for (int u = 0; u < V; u++)

{

cout << "Node " << u << " makes an edge with \n";

for (auto it = adj[u].begin(); it != adj[u].end(); it++)

{

v = it->first;

w = it->second;

cout << "\tNode " << v << " with edge weight ="

<< w << "\n";

}

cout << "\n";

}

}

// Driver code

int main()

{

int const V = 5;

vector<pair<int, int> > adj[V];

addEdge(adj, 0, 1, 10);

addEdge(adj, 0, 4, 20);

addEdge(adj, 1, 2, 30);

addEdge(adj, 1, 3, 40);

addEdge(adj, 1, 4, 50);

addEdge(adj, 2, 3, 60);

addEdge(adj, 3, 4, 70);

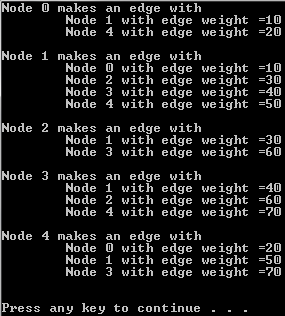
printGraph(adj, V);

cout << endl;

system("PAUSE");

return 0;

}



## Фракталы

***Фракталами*** называют геометрические фигуры, обладающие свойством самоподобия, то есть состоящие из частей, подобных всей фигуре.

Классическим примером является кривая Коха, построение которой показано на рис. 12. Изначально берется отрезок прямой (рис. 12а). Он делится на три части, средняя часть изымается и вместо нее строится угол (рис. 12б), стороны которого равны длине изъятого отрезка (то есть 1/3 от длины исходного отрезка). Такая операция повторяется с каждым из получившихся 4-х отрезков (рис. 12в). И так далее (рис. 12г). Кривая Коха получается после бесконечного числа таких итераций. На практике построение можно прекратить, когда размер деталей окажется меньше разрешения экрана (рис. 12д).

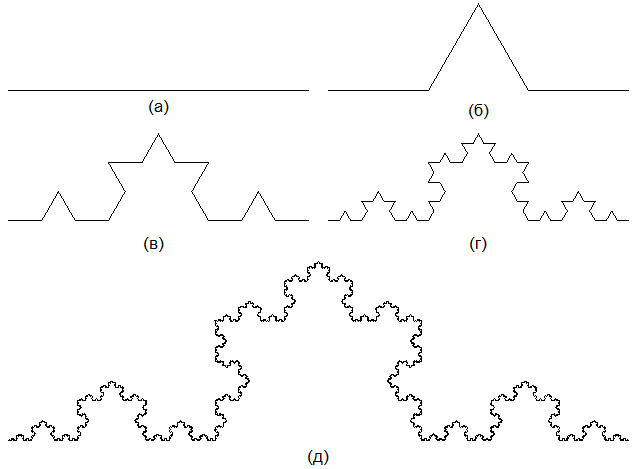


Рис. 12. Процесс построения кривой Коха.

Фракталы, по сути, рекурсивные структуры и их построение естественно производить с помощью рекурсивных процедур.

## Деревья

Теоретической базой для рекурсивных функций, вызывающих себя более одного раза, служит раздел дискретной математики, изучающий деревья.

### Основные определения. Способы изображения деревьев

***Определение****:* *Деревом* будем называть конечное множество *T*, состоящее из одного или более узлов, таких что:

* Имеется один специальный узел, называемый корнем данного дерева.
* Остальные узлы (исключая корень) содержатся в m \geq 0 попарно непересекающихся подмножествах T_1, T_2, \ldots, T_m, каждое из которых в свою очередь является деревом. Деревья T_1, T_2, \ldots, T_mназываются *поддеревьями* данного дерева.

Это определение является рекурсивным. Если коротко, то дерево это множество, состоящее из корня и присоединенных к нему поддеревьев, которые тоже являются деревьями. Дерево определяется через само себя. Однако данное определение осмысленно, так как рекурсия конечна. Каждое поддерево содержит меньше узлов, чем содержащее его дерево. В конце концов, мы приходим к поддеревьям, содержащим всего один узел, а это уже понятно, что такое.

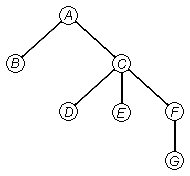


Рис. 3. Дерево.

На рис. 3 показано дерево с семью узлами. Хотя обычные деревья растут снизу вверх, рисовать их принято наоборот. При рисовании схемы от руки такой способ, очевидно, удобнее. Из-за данной несогласованности иногда возникает путаница, когда говорят о том, что один из узлов находится над или под другим. По этой причине удобнее пользоваться терминологией, употребляемой при описании генеалогических деревьев, называя **более близкие к корню узлы предками**, а **более далекие потомками**.

Узлы, не содержащие поддеревьев, называются ***концевыми узлами* или *листьями***. Множество не пересекающихся деревьев называется ***лесом***. Например, лес образуют поддеревья, исходящие из одного узла.

Графически дерево можно изобразить и некоторыми другими способами. Некоторые из них представлены на рис. 4. Согласно определению дерево представляет собой систему вложенных множеств, где эти множества или не пересекаются или полностью содержатся одно в другом. Такие множества можно изобразить как области на плоскости (рис. 4а). На рис. 4б вложенные множества располагаются не на плоскости, а вытянуты в одну линию. Рис. 4б также можно рассматривать как схему некоторой алгебраической формулы, содержащей вложенные скобки. Рис. 4в дает еще один популярный способ изображения древовидной структуры в виде уступчатого списка.

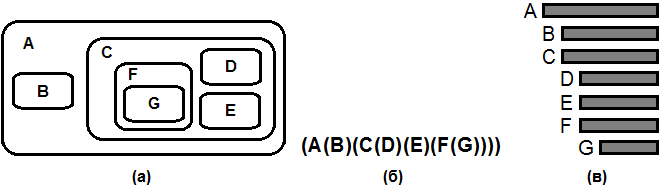


Рис. 4. Другие способы изображения древовидных структур: (а) вложенные множества; (б) вложенные скобки; (в) уступчатый список.

Уступчатый список имеет очевидное сходство со способом форматирования программного кода. Действительно, программа, написанная в рамках парадигмы структурного программирования, может быть представлена как дерево, состоящее из вложенных друг в друга конструкций.

Также можно провести аналогию между уступчатым списком и внешним видом оглавлений в книгах, где разделы содержат подразделы, те в свою очередь поподразделы и т.д. Традиционный способ нумерации таких разделов (раздел 1, подразделы 1.1 и 1.2, подподраздел 1.1.2 и т.п.) называется десятичной системой Дьюи. В применении к дереву на рис. 3 и 4 эта система даст:

1. A;

1.1 B;

1.2 C;

1.2.1 D;

1.2.2 E;

1.2.3 F;

1.2.3.1 G;

### Прохождение деревьев

Во всех алгоритмах, связанных с древовидными структурами неизменно встречается одна и та же идея, а именно идея ***прохождения* или *обхода дерева*.** Это – такой способ посещения узлов дерева, при котором каждый узел проходится точно один раз. При этом получается линейная расстановка узлов дерева. В частности существует **три способа**: можно проходить узлы в **прямом**, **обратном** и **концевом** порядке.

Алгоритм обхода в прямом порядке

Алгоритм обхода в прямом порядке сводится к:

* Попасть в корень,
* Пройти все поддеревья слева направо в прямом порядке.

Данный алгоритм рекурсивен, так как прохождение дерева содержит прохождение поддеревьев, а они в свою очередь проходятся по тому же алгоритму.

В частности для дерева на рис. 3 и 4 прямой обход дает последовательность узлов: A, B, C, D, E, F, G.

Получающаяся последовательность соответствует последовательному слева направо перечислению узлов при представлении дерева с помощью вложенных скобок и в десятичной системе Дьюи, а также проходу сверху вниз при представлении в виде уступчатого списка.

При реализации этого алгоритма на языке программирования попадание в корень соответствует выполнение процедурой или функцией некоторых действий, а прохождение поддеревьев – рекурсивным вызовам самой себя. В частности для бинарного дерева (где из каждого узла исходит не более двух поддеревьев) соответствующая процедура будет выглядеть так:

// Preorder Traversal – английское название для прямого порядка

void PreorderTraversal({Аргументы})

{

//Прохождение корня

DoSomething({Аргументы});

//Прохождение левого поддерева

if {Существует левое поддерево}

PreorderTransversal({Аргументы 2});

//Прохождение правого поддерева

if {Существует правое поддерево}

PreorderTransversal({Аргументы 3});

}

То есть сначала процедура производит все действия, а только затем происходят все рекурсивные вызовы.

Алгоритм обхода в обратном порядке

Алгоритм обхода в обратном порядке сводится к:

* Пройти левое поддерево,
* Попасть в корень,
* Пройти следующее за левым поддерево.
* Попасть в корень,
* и т.д пока не будет пройдено крайнее правое поддерево.

То есть проходятся все поддеревья слева на право, а возвращение в корень располагается между этими прохождениями. Для дерева на рис. 3 и 4 это дает последовательность узлов: B, A, D, C, E, G, F.

В соответствующей рекурсивной процедуре действия будут располагаться в промежутках между рекурсивными вызовами. В частности для бинарного дерева:

// Inorder Traversal – английское название для обратного порядка

void InorderTraversal({Аргументы})

{

//Прохождение левого поддерева

if {Существует левое поддерево}

InorderTraversal({Аргументы 2});

//Прохождение корня

DoSomething({Аргументы});

//Прохождение правого поддерева

if {Существует правое поддерево}

InorderTraversal({Аргументы 3});

}

Алгоритм обхода в концевом порядке

Алгоритм обхода в концевом порядке сводится к:

* Пройти все поддеревья слева на право,
* Попасть в корень.

Для дерева на рис. 3 и 4 это даст последовательность узлов: B, D, E, G, F, C, A.

В соответствующей рекурсивной процедуре действия будут располагаться после рекурсивных вызовов. В частности для бинарного дерева:

// Postorder Traversal – английское название для концевого порядка

void PostorderTraversal({Аргументы})

{

//Прохождение левого поддерева

if {Существует левое поддерево}

PostorderTraversal({Аргументы 2});

//Прохождение правого поддерева

if {Существует правое поддерево}

PostorderTraversal({Аргументы 3});

//Прохождение корня

DoSomething({Аргументы});

}

### Представление дерева в памяти компьютера

Если некоторая информация располагается в узлах дерева, то для ее хранения можно использовать соответствующую динамическую структуру данных. На Паскале это делается с помощью переменной типа запись (record), содержащей указатели на поддеревья того же типа. Например, бинарное дерево, где в каждом узле содержится целое число можно сохранить с помощью переменной типа PTree, который описан ниже:

**type**

PTree = ^TTree;

TTree = **record**

Inf: integer;

LeftSubTree, RightSubTree: PTree;

**end**;

В языке С++

struct TTree

{

int Inf;

struct TTree \*LeftSubTree, \*RightSubTree;

} PTree;

Каждый узел имеет тип PTree. Это указатель, то есть каждый узел необходимо создавать, вызывая для него процедуру New. Если узел является концевым, то его полям LeftSubTree и RightSubTree присваивается значение **null(nil)**. В противном случае узлы LeftSubTree и RightSubTree также создаются процедурой New.

Схематично одна такая запись изображена на рис. 5.

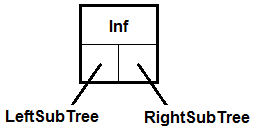


Рис. 5. Схематичное изображение записи типа TTree. Запись имеет три поля: Inf – некоторое число, LeftSubTree и RightSubTree – указатели на записи того же типа TTree.

Пример дерева, составленного из таких записей, показан на рисунке 6.

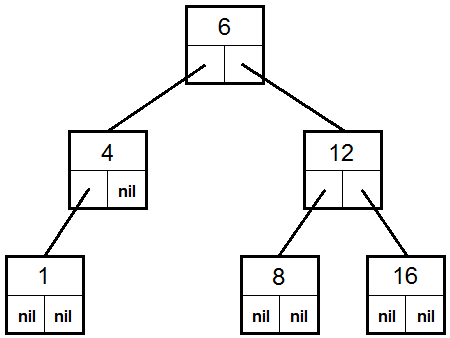


Рис. 6. Дерево, составленное из записей типа TTree. Каждая запись хранит число и два указателя, которые могут содержать либо **nil**, либо адреса других записей того же типа.

Если вы ранее не работали со структурами состоящими из записей, содержащих ссылки на записи того же типа, то рекомендуем ознакомиться с материалом о [рекурсивных структурах данных](http://www.tvd-home.ru/prog/16_4).

Рекурсивные структуры данных

Пусть описан тип-запись, и одним из полей этой записи является указатель. В этот указатель можно записать адрес, по которому данная запись располагается, либо, что более интересно адрес другой записи того же типа. Это позволяет при помощи указателей создать структуру данных, называемую ***связанным списком*.**

Пример описания такой структуры данных для языка Pascal:

**type**

PListElement = ^TListElement;

TListElement = **record**

<Произвольные поля записи>

NextElement: PListElement;

**end**;

Пример описания такой структуры данных для языка C++:

struct TListElement

{

struct TListElement ^TListElement;

//<Произвольные поля записи>

struct TListElement \*NextElement;

} PListElement;

Здесь сначала описывается тип – указатель PListElement на запись типа TListElement. Затем идет описание самого типа записи, одно из полей которой имеет тип PListElement. Получается, что мы используем идентификатор TListElement еще до того, как его описали. Обычно такие вещи запрещено делать, но в данном случае, когда описывается ссылочный тип, из этого правила сделано исключение.

Если имеется запись этого типа, то можно с помощью указателя динамически создать еще один элемент этого типа. По указателю в этом новом элементе еще один элемент и т.д., сколько потребуется. Такая структура данных называется однонаправленным связанным списком. Ее схематическое изображение показано на рис. 1.

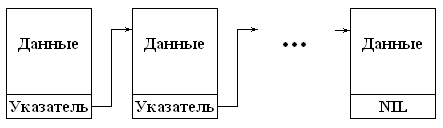


Рис. 1. Связанный список.

#### Пример 1. Простой пример связанного списка

// Программа на С++ - простой пример связанного списка

#include <stdio.h>

#include <strstream>

#include <iostream>

using namespace std;

struct sSS

{

int x;

struct sSS \*nextSS;

} varSS, \*firstSS, \*currentSS;

void SSlocation(struct sSS \*ptr1);

void output\_data(struct sSS \*ptr2);

int data\_in(struct sSS \*ptr3);

int main(void)

{

firstSS = (struct sSS \*) new (struct sSS);

if(firstSS == NULL) exit(1);

if(data\_in(&varSS) != 0) exit(1);

\*firstSS = varSS;

currentSS = firstSS;

while(data\_in(&varSS) == 0)

{

currentSS -> nextSS = (struct sSS \*) new (struct sSS);

if(currentSS -> nextSS == NULL) return(1);

currentSS = currentSS -> nextSS;

\*currentSS = varSS;

}

currentSS -> nextSS = NULL; // указывает на конец списка

SSlocation(firstSS);

cout << "\n";

system("PAUSE");

return(0);

}

void SSlocation(struct sSS \*node)

{

do{

output\_data(node);

} while ((node = node -> nextSS) != NULL);

}

void output\_data(struct sSS \*ptrSS)

{

cout << "\n";

cout << "x: " << ptrSS -> x ;

}

int data\_in(struct sSS \*ptrSS)

{

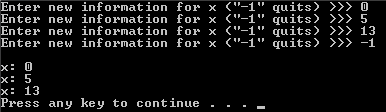
cout << "Enter new information for x (\"-1\" quits) >>> ";

cin >> ptrSS -> x;

if(ptrSS -> x == -1) return (1);

return(0);

}



#### Пример 2. Связанный список (динамический список)

//Связанный список (динамический список)

#include <iostream>

using namespace std;

//стуктура, описывающая узел связанного списка

struct Node

{

int data;

Node \*next;

};

//класс, описывающих объект "связанный список"

class List

{

private:

Node \*head; //"голова" связанного списка

public:

List() //конструктор класса без параметров

{

head = NULL; //первого элемента пока нет

}

//метод, добавляющий новый узел в список

void addNode(int d)

{

Node \*nd = new Node; //динамически создаем новый узел

nd->data = d; //задаем узлу данные

nd->next = NULL; //новый узел в конце, поэтому NULL

if(head == NULL) //если создаем первый узел

head = nd;

else //если узел уже не первый

{

Node \*current = head;

//ищем в цикле предшествующий последнему узел

while(current->next != NULL)

current = current->next;

//предшествующий указывает на последний

current->next = nd;

}

}

//метод, выводящий связанный список на экран

void printList()

{

Node \*current = head;

while(current != NULL)

{

cout << current->data << endl;

current = current->next;

}

}

};

int main()

{

List myList;

myList.addNode(5);

myList.addNode(11);

myList.addNode(27);

myList.addNode(35);

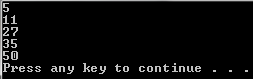
myList.addNode(50);

myList.printList();

system("PAUSE");

return 0;

}



#### Пример 3. Односвязный однонаправленный список

Под односвязным однонаправленным списком подразумевается реализация последовательного расположения набора элементов. Начиная с некоторого узла, мы считаем его первым, в этом узле есть ссылки на следующий, и так далее. В большинстве случаев первый узел делают пустым, потому что тогда со списком легче работать. В односвязном списке каждый узел имеет ссылки только на следующий элемент, то есть мы не можем двигаться с конца списка до его начала. Однонаправленный список заканчивается пустым ссылкой NULL.

Давайте подумаем, какие функции могут нам понадобиться для работы со списком. Наверное, это - организация самого списка, вывод списка, сортировка списка, вставка нового элемента в отсортированный список. Так что сначала надо **организовать сам узел**, из совокупности которых будет формироваться список.

struct List

{

int item; //полезная информация узла списка

List \*next;//ссылка на следующий элемент списка.

};

Теперь нам нужно **создать переменную start типа List**, которая будет указывать на первый элемент списка.

List \*start = new List; //выделяем память для первого элемента

start->next = NULL;// на данный момент первый элемент одновременно является и последним том next ссылается на NULL

Так же как мы выделяли память для первого элемента, так же нужно будет делать и для всех последующих.

Теперь рассмотрим часть кода, которая будет **формировать список**.

List \*pList = start;

for (int i=0; i<20; i+=2){

List \*tmp = new List;//Выделяем память для

tmp->item = i;// здесь все понятно

tmp->next = NULL;//мы вставляем узел в конец списка и поэтому next = NULL

pList->next = tmp;//теперь pList-> next указывает на созданный узел

pList = pList->next;//Передвигаем указатель на последний элемент.

}

здесь мы используем рабочий указатель для того, чтобы не менять start, который всегда указывать на первый узел списка.

Давайте **выведем в консоль созданный список**. Для этого напишем отдельную функцию:

void show (List \*start)//параметр - указатель на начало списка.

{

List \*pList; //рабочий указатель

pList = start->next;//в начале он ссылается на второй узел списка.

while (pList != NULL){

cout << pList->item << endl;//Выводим данные

pList = pList->next;//передвигаем указатель на один узел.

}

}

Теперь напишем функцию, которая будет **вставлять элемент в отсортированный список**.

void insert (List \*start, int Node)//указатель на начало списка, число которое нужно вставить в список.

{

List \*pList;//рабочий указатель.

pList = start->next;//в начале он ссылается на второй узел списка.

while (pList->next != NULL){

if (pList == start->next && pList->item >Node){ //ета проверка нужна для того, чтоб проверить не надо ли вставить елемент в начало списка.

List \*temp = new List; //временная переменная, в которую буедм записывать узел, на который ссылается pList

\*temp = \*pList;//запоминаем \*pList в \*tmp. Здесь мы запоминаем не адрес, а именно значение

pList->item = Node;//теперь в item записываем елемент который нужно было вставить в список.

pList->next = temp;//и в next записываем адрес tmp

break;

}

if (pList->item <= Node && pList->next->item >= Node){ //Если элемент надо всативить не на начало и не в конец списка.

List \*tmp = new List;//Выделяем память для нового узла

tmp->item = Node;

tmp->next = pList->next;

pList->next = tmp; //Вставляем узел после узла, на который указывает pList

break;

}

pList = pList->next; //передвигаем указатель на следующий елемент.

if (pList->next == NULL && Node>=pList->item){ //Проверяем не нужно вставить узел в конец списка.

List \*tmp = new List;

tmp->item = Node;

tmp->next = NULL; //Так как вставляем узел в конец списка то его элемент next должен ссылаться на NULL

pList->next = tmp;//вставляем узел в список.

break;

}

}

}

Теперь напишем **функцию, которая будет сортировать список**. Будем использовать один из простейших алгоритмов сортировки - **сортировка пузырьком**.

void sort(List \*start) //параметр - указатель на начало списка.

{

List \*pList; // указатель для внешнего цыкла for

List \*ppList; // для внутреннего

for (pList = start->next; pList != NULL; pList = pList->next) //пробегаемся по всех узлах списка

for (ppList = start->next; ppList->next != NULL; ppList = ppList->next) //а здесь только до узла, next которого указывает на NULL

//обычный обмен значениями

if (ppList->item > ppList->next->item){

int tmp;

tmp = ppList->item; //

ppList->item = ppList->next->item;

ppList->next->item = tmp;

}

}

И последняя функция - **удаление элемента из списка**:

void dell(List \*start, int Node) //указатель на начало списка, и значение которое мы будем искать в списке, если найдем значит нужно удалить.

{

List \*pList = start->next; //рабочий указатель, который будем использовать для пробека по списку.

List \*prev = start; //указатель, который всегда будет указывать на елемент предедущий до pList.

while (pList != NULL){ //пробегаемся по всему списку

//проверка - нужно ли удаление

if (pList->item == Node){

// временней узел, в который будем записывать значение того узла который нужно удалить.

List \*tmp = new List;

\*tmp = \*pList;

//теперь нужно сделать так, чтоб елемент next предыдущего узла указывал на узел, который идет после pList,то есть на pList->next

prev->next = pList->next;

delete tmp;

}

prev = pList; // перемещаем указатель prev на один елемент

pList = pList->next; //так же перемещаем next.

}

}

**Вот как это примерно выглядит:**

#include <iostream>

using namespace std;

struct List

{

int item; //полезная информация узла списка

List \*next;//ссылка на следующий элемент списка.

};

void create\_List(List \*start)

{

//так как \*start - указатель на начало списка, нам нужен рабочий указатель с помощью которого мы будем работать с целым списком.

List \*pList = start->next; //так как первый элемент будет пустым, присваиваем рабочем указателе ссылка на следующий элемент, который нужно заполнить.

//теперь нужно создать наш список. Заполним его елементы числами.

for (int i=0; i<20; i+=2){

List \*tmp = new List; //Выделяем память, для следующего узла.

tmp->item = i; //заполняем елемент узла, который хранить полезные данные

tmp->next = NULL; //Мы вставляем элементы в конец списка. Это означает, что каждый элемент который мы добавляем имеет ссылаться на NULL

pList->next = tmp; //Добавляем созданный узел в список.

pList = pList->next; //Сдвигаем указатель на один елемент. Теперь он указывает на последний елемент который мы добавили.

}

}

void show (List \*start)//параметр - указатель на начало списка.

{

List \*pList; //рабочий указатель

pList = start->next;//в начале он ссылается на второй узел списка.

while (pList != NULL){

cout << pList->item << endl;//Выводим данные

pList = pList->next;//передвигаем указатель на один узел.

}

}

void insert (List \*start, int Node)//указатель на начало списка, число которое нужно вставить в список.

{

List \*pList;//рабочий указатель.

pList = start->next;//в начале он ссылается на второй узел списка.

while (pList->next != NULL){

if (pList == start->next && pList->item >Node){ //ета проверка нужна для того, чтоб проверить не надо ли вставить елемент в начало списка.

List \*temp = new List; //временная переменная, в которую буедм записывать узел, на который ссылается pList

\*temp = \*pList;//запоминаем \*pList в \*tmp. Здесь мы запоминаем не адрес, а именно значение

pList->item = Node;//теперь в item записываем елемент который нужно было вставить в список.

pList->next = temp;//и в next записываем адрес tmp

break;

}

if (pList->item <= Node && pList->next->item >= Node){ //Если элемент надо всативить не на начало и не в конец списка.

List \*tmp = new List;//Выделяем память для нового узла

tmp->item = Node;

tmp->next = pList->next;

pList->next = tmp; //Вставляем узел после узла, на который указывает pList

break;

}

pList = pList->next; //передвигаем указатель на следующий елемент.

if (pList->next == NULL && Node>=pList->item){ //Проверяем не нужно вставить узел в конец списка.

List \*tmp = new List;

tmp->item = Node;

tmp->next = NULL; //Так как вставляем узел в конец списка то его элемент next должен ссылаться на NULL

pList->next = tmp;//вставляем узел в список.

break;

}

}

}

void sort(List \*start) //параметр - указатель на начало списка.

{

List \*pList; // указатель для внешнего цыкла for

List \*ppList; // для внутреннего

for (pList = start->next; pList != NULL; pList = pList->next) //пробегаемся по всех узлах списка

for (ppList = start->next; ppList->next != NULL; ppList = ppList->next) //а здесь только до узла, next которого указывает на NULL

//обычный обмен значениями

if (ppList->item > ppList->next->item){

int tmp;

tmp = ppList->item; //

ppList->item = ppList->next->item;

ppList->next->item = tmp;

}

}

void dell(List \*start, int Node) //указатель на начало списка, и значение которое мы будем искать в списке, если найдем значит нужно удалить.

{

List \*pList = start->next; //рабочий указатель, который будем использовать для пробека по списку.

List \*prev = start; //указатель, который всегда будет указывать на елемент предедущий до pList.

while (pList != NULL){ //пробегаемся по всему списку

//проверка - нужно ли удаление

if (pList->item == Node){

// временней узел, в который будем записывать значение того узла который нужно удалить.

List \*tmp = new List;

\*tmp = \*pList;

//теперь нужно сделать так, чтоб елемент next предыдущего узла указывал на узел, который идет после pList,то есть на pList->next

prev->next = pList->next;

delete tmp;

}

prev = pList; // перемещаем указатель prev на один елемент

pList = pList->next; //так же перемещаем next.

}

}

int main()

{

List \*start = new List; //выделяем память для первого элемента

start->next = NULL;// на данный момент первый элемент одновременно является и последним том next ссылается на NULL

List \*pList = start;

for (int i=0; i<20; i+=2){

List \*tmp = new List;//Выделяем память для

tmp->item = i;// здесь все понятно

tmp->next = NULL;//мы вставляем узел в конец списка и поэтому next = NULL

pList->next = tmp;//теперь pList-> next указывает на созданный узел

pList = pList->next;//Передвигаем указатель на последний элемент.

}

show (start);

insert(start,2);

insert(start,5);

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

show (start);

dell(start,2);

dell(start,2);

dell (start,18);

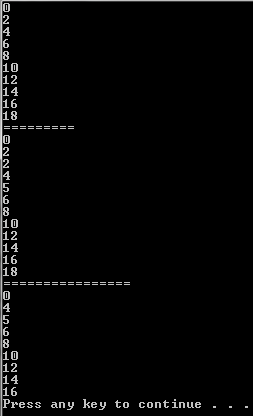
cout << "================\n";

show(start);

system("PAUSE");

return 0;

}



Преимуществами динамических списков является то, что мы можем достаточно легко вставить элемент или удалить его из списка. В массиве это несколько сложнее. Также преимуществом является то, что размер списка можно изменить в любой момент выполнения программы.

Но в списках есть и недостатки. Одним из важнейших является то, что мы не можем получить доступ к k-му узлу так же как в массиве, в списках мы вынуждены пройтись по всем элементам, которые идут до k. Также узлы в списке не размещаются последовательно в памяти и программа работает немного медленнее, чем при работе с массивом, где все элементы размещены последовательно.

#### Пример 4. Однонаправленный связанный список

**Пусть в виде однонаправленного связанного списка хранятся целые числа. Создать процедуры, которые:**

**а) Создают список, динамически выделяя из кучи память под заданное число элементов N.**

**б) Заполняют информационную часть элементов списка числами 10, 20, …, N.**

**в) Печатают в столбик информационную часть списка.**

**б) Заменяют информационную часть элементов списка числами 1, 2, …, N.**

**г) Уничтожают элементы(узлы) списка, возвращая в кучу выделенную под них память.**

Решение:

#include <stdio.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

using namespace std;

struct List{

int item; //Информационное поле

List \*next; //Ссылка на следующий элемент списка

};

void CreateList(List \*start, int Num)

{

//Процедура создает список, содержащий Num элементов

//начинающийся с элемента start

cout << "===CreateList()===" << endl;//

start->next = NULL; //первый элемент одновременно является и последним

//поэтому next ссылается на NULL

List \*pList = start;

//При работе с однонаправленными списками принципиально

//важно не потерять ссылку на первый элемент. Поэтому

//дальнейшие операции будем проводить не с start, а со

//вспомогательной переменной tmp

for(int n=1; n<=Num; n++){

List \*tmp = new List; //Выделяем память, для следующего узла

tmp->item = n \* 10; //елемент узла, который хранить полезные данные

tmp->next = NULL; //вставляем элементы в конец списка.

//Это означает, что каждый элемент

//который мы добавляем имеет ссылаться на NULL

cout << "Created Object: " << n << " item= " << tmp->item << "\n";

pList->next = tmp; //Добавляем созданный узел в список

pList = pList->next; //Сдвигаем указатель на один елемент.

//Теперь он указывает на последний элемент, который мы добавили

}

}

void PrintList(List \*start)//параметр - указатель на начало списка

{

//Процедура печатает информационную часть элементов списка

cout << "===PrintList()===" << endl;//

List \*pList; //рабочий указатель

pList = start->next;//в начале он ссылается на второй узел списка

while(pList != NULL){

cout << pList->item << endl;//Выводим данные

pList = pList->next;//передвигаем указатель на один узел

}

}

void FillList(List \*start)

{

//Процедура заполняет информационную часть элементов списка

//числами 1, 2, 3, …

cout << "===FillList()===" << endl;//

List \*pList; //рабочий указатель

pList = start->next;//в начале он ссылается на второй узел списка

int n = 1;

while(pList != NULL){

pList->item = n;

cout << "Filled Object: " << n << " item = " << pList->item << "\n";

n = n + 1;

pList = pList->next;

}

}

void DeleteNode(List \*start, int Node)

{

//Процедура освобождающая память, выделенную под список

List \*pList = start->next;//рабочий указатель, который будем использовать для пробега по списку

List \*prev = start; //указатель, который всегда будет указывать на елемент предедущий до pList

//Поскольку список все равно уничтожается,

//можем спокойно изменять Head

while(pList != NULL){//пробегаемся по всему списку

if (pList->item == Node){//проверка - нужно ли удаление

// временней узел, в который будем записывать значение того узла который нужно удалить

List \*tmp = new List;

\*tmp = \*pList;

//теперь нужно сделать так, чтоб елемент next предыдущего узла указывал на узел,

//который идет после pList,то есть на pList->next

prev->next = pList->next;

delete tmp;

}

prev = pList; //перемещаем указатель prev на один елемент

pList = pList->next; //так же перемещаем next.

}

}

void main()

{

struct List \*start = new List;//выделяем память для первого элемента

CreateList(start, 5); //Создаем список из 5-ти элементов

cout << "\n";

PrintList(start); //Печатаем информационные части элементов списка

cout << "\n";

FillList(start); //В информационные части пишем числа 1, 2, 3, 4, 5

cout << "\n";

PrintList(start); //Печатаем информационные части элементов списка

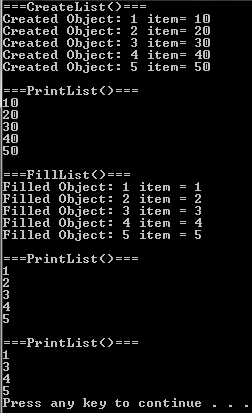
cout << "\n";

DeleteNode(start, 2); //удаление узла

PrintList(start); //Печатаем информационные части элементов списка

system("PAUSE");

}



## Примеры рекурсивных алгоритмов

### Рисование дерева

Рассмотрим алгоритм рисования деревца, изображенного на рис. 6. Если каждую линию считать узлом, то данное изображение вполне удовлетворяет определению дерева, данному в предыдущем разделе.

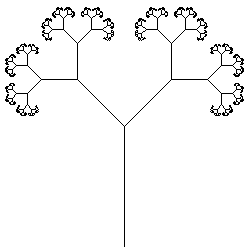


Рис. 6. Деревце.

Рекурсивная процедура, очевидно должна рисовать одну линию (ствол до первого разветвления), а затем вызывать сама себя для рисования двух поддеревьев. Поддеревья отличаются от содержащего их дерева координатами начальной точки, углом поворота, длиной ствола и количеством содержащихся в них разветвлений (на одно меньше). Все эти отличия следует сделать параметрами рекурсивной процедуры.

Пример такой процедуры, написанный на Delphi, представлен ниже:

procedure Tree(

Canvas: TCanvas; //Canvas, на котором будет рисоваться дерево

x,y: extended; //Координаты корня

Angle: extended; //Угол, под которым растет дерево

TrunkLength: extended; //Длина ствола

n: integer //Количество разветвлений (сколько еще предстоит

//рекурсивных вызовов)

);

var

x2, y2: extended; //Конец ствола (точка разветвления)

begin

x2 := x + TrunkLength \* cos(Angle);

y2 := y - TrunkLength \* sin(Angle);

Canvas.MoveTo(round(x), round(y));

Canvas.LineTo(round(x2), round(y2));

if n > 1 then

begin

Tree(Canvas, x2, y2, Angle+Pi/4, 0.55\*TrunkLength, n-1);

Tree(Canvas, x2, y2, Angle-Pi/4, 0.55\*TrunkLength, n-1);

end;

end;

Для получения рис. 6 эта процедура была вызвана со следующими параметрами:

Tree(Image1.Canvas, 175, 325, Pi/2, 120, 15);

Заметим, что рисование осуществляется до рекурсивных вызовов, то есть дерево рисуется в прямом порядке.

### Ханойские башни

Согласно легенде в Великом храме города Бенарас, под собором, отмечающим середину мира, находится бронзовый диск, на котором укреплены 3 алмазных стержня, высотой в один локоть и толщиной с пчелу. Давным-давно, в самом начале времен монахи этого монастыря провинились перед богом Брамой. Разгневанный, Брама воздвиг три высоких стержня и на один из них поместил 64 диска из чистого золота, причем так, что каждый меньший диск лежит на большем. Как только все 64 диска будут переложены со стержня, на который Бог Брама сложил их при создании мира, на другой стержень, башня вместе с храмом обратятся в пыль и под громовые раскаты погибнет мир.  
В процессе требуется, чтобы больший диск ни разу не оказывался над меньшим. Монахи в затруднении, в какой же последовательности стоит делать перекладывания? Требуется снабдить их софтом для расчета этой последовательности.

Независимо от Брамы данную головоломку в конце 19 века предложил французский математик Эдуард Люка. В продаваемом варианте обычно использовалось 7-8 дисков (рис. 7).

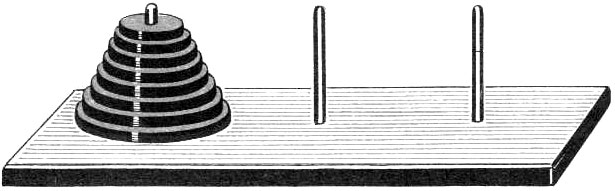


Рис. 7. Головоломка «Ханойские башни».

Предположим, что существует решение для *n*-1 диска. Тогда для перекладывания *n* дисков надо действовать следующим образом:

* Перекладываем *n*-1 диск.
* Перекладываем *n*-й диск на оставшийся свободным штырь.
* Перекладываем стопку из *n*-1 диска, полученную в пункте (1) поверх *n*-го диска.

Поскольку для случая *n* = 1 алгоритм перекладывания очевиден, то по индукции с помощью выполнения действий (1) – (3) можем переложить произвольное количество дисков.

Создадим рекурсивную процедуру, печатающую всю последовательность перекладываний для заданного количества дисков. Такая процедура при каждом своем вызове должна печатать информацию об одном перекладывании (из пункта 2 алгоритма). Для перекладываний из пунктов (1) и (3) процедура вызовет сама себя с уменьшенным на единицу количеством дисков.

//n – количество дисков

//a, b, c – номера штырьков. Перекладывание производится со штырька a,

//на штырек b при вспомогательном штырьке c.

procedure Hanoi(n, a, b, c: integer);

begin

if n > 1 then

begin

Hanoi(n-1, a, c, b);

writeln(a, ' -> ', b);

Hanoi(n-1, c, b, a);

end else

writeln(a, ' -> ', b);

end;

Заметим, что множество рекурсивно вызванных процедур в данном случае образует дерево, проходимое в обратном порядке.

Прмер 1.

// Ханойские башни

#include <iostream>

#include <Windows.h>

using namespace std;

void Towers(int number, int from, int to, int free)

{

//number - это колличество дисков

//from - это стержень с которого переносим все диски.

//to - это стержень на который переносим все диски.

//free - это третий стержень.

SetConsoleOutputCP(1251);

if(number!=0){

Towers(number-1, from, free, to);

cout << "\n Снимаем " << number << "-й диск с " << from

<< "-го стержня и одеваем его на " << to << "-й стержень";

Towers(number-1, free, to, from);

}

}

void main()

{

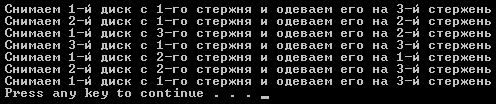
setlocale(LC\_ALL,"rus");

Towers(3, 1, 3, 2);

cout<<"\n ";

system("PAUSE");

}



Прмер 2.

// Ханойские башни

#include <iostream>

using namespace std;

void HanoyTown(int nLevel, char from, char to, char mid)

{

if (nLevel > 0)

{

HanoyTown(nLevel-1, from, mid, to);

printf(("%c ==> %c\n"), from, to);

HanoyTown(nLevel-1, mid, to, from);

}

}

int main()

{

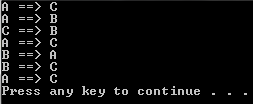
setlocale(LC\_ALL,"rus");

HanoyTown(3, 'A', 'C', 'B');

system("PAUSE");

return 0;

}



Прмер 3.

// Ханойские башни

#include <iostream>

using namespace std;

void hanoi\_towers(int quantity, int from, int to, int buf\_peg)

//quantity-число колец, from-начальное положение колец(1-3),to-конечное положение колец(1-3)

//buf\_peg - промежуточный колышек(1-3)

{

if (quantity != 0){

hanoi\_towers(quantity-1, from, buf\_peg, to);

cout << from << " -> " << to << endl;

hanoi\_towers(quantity-1, buf\_peg, to, from);

}

}

int main()

{

setlocale(LC\_ALL,"rus");

int start\_peg, destination\_peg, buffer\_peg, plate\_quantity;

cout << "Номер первого столбика: ";

cin >> start\_peg;

cout << "Номер конечного столбика: ";

cin >> destination\_peg;

cout << "Номер промежуточного столбика: ";

cin >> buffer\_peg;

cout << "Количество дисков: ";

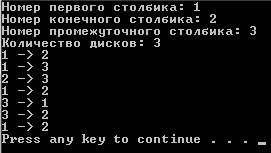
cin >> plate\_quantity;

hanoi\_towers(plate\_quantity, start\_peg, destination\_peg, buffer\_peg);

system("PAUSE");

return 0;

}



Полезно:

https://habrahabr.ru/post/200758/

### Синтаксический анализ арифметических выражений

Задача синтаксического анализа заключается в том, чтобы по имеющейся строке, содержащей арифметическое выражение, и известным значениям, входящих в нее переменных, вычислить значение выражения.

Процесс вычисления арифметических выражений можно представить в виде бинарного дерева. Действительно, каждый из арифметических операторов (+, –, \*, /) требует двух операндов, которые также будут являться арифметическими выражениями и, соответственно могут рассматриваться как поддеревья. Рис. 8 показывает пример дерева, соответствующего выражению:

x-2*(1/x+x/3)~~~~~(6)

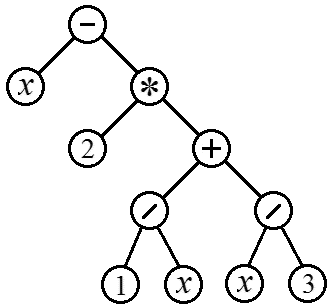


Рис. 8. Синтаксическое дерево, соответствующее арифметическому выражению (6).

В таком дереве концевыми узлами всегда будут переменные (здесь x) или числовые константы, а все внутренние узлы будут содержать арифметические операторы. Чтобы выполнить оператор, надо сначала вычислить его операнды. Таким образом, дерево на рисунке следует обходить в концевом порядке. Соответствующая последовательность узлов

x~2~1~x~/~x~3~/~+~*~-~~~~~(7)

**называется *обратной польской записью* арифметического выражения**.

При построении синтаксического дерева следует обратить внимание на следующую особенность. Если есть, например, выражение

a-b+c~~~~~(8)

и операции сложения и вычитания мы будем считывать слева на право, то правильное синтаксическое дерево будет содержать минус вместо плюса (рис. 9а). По сути, это дерево соответствует выражению a-(b-c). Облегчить составление дерева можно, если анализировать выражение (8) наоборот, справа налево. В этом случае получается дерево с рис. 9б, эквивалентное дереву 8а, но не требующее замены знаков.

Аналогично справа налево нужно анализировать выражения, содержащие операторы умножения и деления.

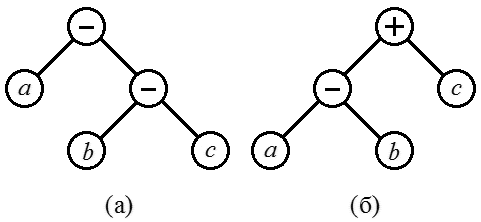


Рис. 9. Синтаксические деревья для выражения a – b + c при чтении слева направо (а) и справа налево (б).

Пример 1. Вычисление выражения (Pascal)

В файле [SynAn.pas](http://www.tvd-home.ru/files/SynAn.zip) приведен пример функции, вычисляющей значения выражений, содержащих только одну переменную *x*. Дадим краткое описание реализованного там алгоритма:

**unit** SynAn;

**interface**

//Вычисляет выражение s при значении переменной x

**function** CalcExpression(s: string; x: extended): extended;

//Вычисляет выражение s, не содержащее знаков + и - вне скобок при значении

//переменной x

**function** CalcMultDiv(s: string; x: extended): extended;

//Вычисляет выражения в скобках, вызывая для них CalcExpression или, если

//выражение простое (константа или просто x), возвращает его значение.

**function** CalcValuesOrOpenParentheses(s: string; x: extended): extended;

//Входное n - номер закрывающей скобки. Процедура находит и записывает в n

//номер соответствующей открывающей скобки.

**procedure** SkipParentheses(s: string; **var** n: integer);

//Находит и записывает в n номер символа, содержащего "+" или "-"

**procedure** FindPlusMinus(s: string; **var** n: integer);

//Находит и записывает в n номер символа, содержащего "\*" или "/"

**procedure** FindMultDiv(s: string; **var** n: integer);

**implementation**

//Вычисляет выражение s при значении переменной x

**function** CalcExpression(s: string; x: extended): extended;

**var**

n: integer;

s2: string;

Res: extended;

**begin**

n := length(s);

Res:=0;

**repeat**

FindPlusMinus(s, n);

s2 := Copy(s, n+1, length(s));

**if** s[n] = '-' **then**

Res := Res - CalcMultDiv(s2, x)

**else**

Res := Res + CalcMultDiv(s2, x);

Delete(s, n, length(s));

n := n - 1;

**until** n < 1;

CalcExpression := Res;

**end**;

//Вычисляет выражение s, не содержащее знаков + и - вне скобок при значении

//переменной x

**function** CalcMultDiv(s: string; x: extended): extended;

**var**

n: integer;

s2: string;

Res: extended;

**begin**

n := length(s);

Res:=1;

**repeat**

FindMultDiv(s, n);

s2 := Copy(s, n+1, length(s));

**if** s[n] = '/' **then**

Res := Res / CalcValuesOrOpenParentheses(s2, x)

**else**

Res := Res \* CalcValuesOrOpenParentheses(s2, x);

Delete(s, n, length(s));

n := n - 1;

**until** n < 1;

CalcMultDiv := Res;

**end**;

//Вычисляет выражения в скобках, вызывая для них CalcExpression или, если

//выражение простое (константа или просто x), возвращает его значение.

**function** CalcValuesOrOpenParentheses(s: string; x: extended): extended;

**var**

c: integer;

Res: extended;

**begin**

**if** s[1] = '(' **then**

Res := CalcExpression(Copy(s, 2, length(s)-2), x)

**else**

**if** s = 'x' **then**

Res := x

**else**

Val(s, Res, c);

CalcValuesOrOpenParentheses := Res;

**end**;

//Находит и записывает в n номер символа, содержащего "+" или "-"

**procedure** FindPlusMinus(s: string; **var** n: integer);

**begin**

**repeat**

**if** s[n] = ')' **then**

SkipParentheses(s, n);

n := n - 1;

**until** (n < 1)**or**(s[n] = '+')**or**(s[n] = '-');

**end**;

//Находит и записывает в n номер символа, содержащего "\*" или "/"

**procedure** FindMultDiv(s: string; **var** n: integer);

**begin**

**repeat**

**if** s[n] = ')' **then**

SkipParentheses(s, n);

n := n - 1;

**until** (n < 1)**or**(s[n] = '\*')**or**(s[n] = '/');

**end**;

//Входное n - номер закрывающей скобки. Процедура находит и записывает в n

//номер соответствующей открывающей скобки.

**procedure** SkipParentheses(s: string; **var** n: integer);

**var**

pn: integer; //Счетчик скобок

**begin**

pn := 1;

**repeat**

n := n - 1;

**if** s[n] = ')' **then**

pn := pn + 1;

**if** s[n] = '(' **then**

pn := pn - 1;

**until** pn = 0;

**end**;

**end**.

* Вычисляющая выражение функция (CalcExpression) находит в строке все знаки «+» и «–», не заключенные в скобки. Эти знаки разбивают выражение на части, содержащие (вне скобок) только операции умножения и деления. Для вычисления значений этих частей вызывается функция CalcMultDiv.
* Функция CalcMultDiv находит в строке все знаки «\*» и «/», не заключенные в скобки. Эти знаки разбивают выражение на части, содержащие числовые константы, переменную *x* или выражения в скобках. Для вычисления значений этих частей вызывается функция CalcValuesOrOpenParentheses.
* Функция CalcValuesOrOpenParentheses определяет тип попавшего ей на вход выражения. Если это числовая константа или переменная *x*, то она возвращает их значение. Если это выражение в скобках, то для его вычисления рекурсивно вызывается процедура CalcExpression.

Заметим, что в данном примере вычисления производятся одновременно с анализом строкового выражения. Это приводит к тому, что для некоторых выражений вычисления могут происходить в 100 – 1000 раз медленнее, чем, если бы эти выражения были скомпилированы как часть программы. Если одно и то же выражение требуется вычислить много раз при различных значения переменных, то следует разделить анализ строки и вычисления. Такой подход может позволить ускорить вычисления в сотни раз.

Результатом анализа строки должна быть последовательность узлов дерева в концевом порядке. Каждый узел должен хранить информацию о подузлах и о той операции, которая в нем совершается. Например, узлы можно реализовать в виде записей, одно из полей который имеет процедурный тип. Другой вариант – каждый узел это объект, где операция реализована как виртуальный метод.

Пример 2. Синтаксический анализатор математических выражений

В данном примере рассматривается программа, которая будет принимать на входе строку с математическим выражением (например 2+8\*(9/4-1,5)^2) и выдавать соответствующий результат. Для анализа выражения будем использовать алгоритм рекурсивного спуска. Преимущество этого алгоритма заключается в том, что мы можем обеспечить выполнение арифметических операций в нужном порядке, в соответствии с законами математики. Наш анализатор будет состоять из нескольких функций, которые будут вызываться одна из другой. В каждой функции будут выполняться определенные арифметические действия, нам лишь необходимо вызывать функции в нужной последовательности, которую диктуют нам правила математики. Это и есть принцип алгоритма.

**Разбиение строки на лексемы.**

Прежде чем обрабатывать выражение, необходимо разбить его на составляющие части (лексемы). **Лексема** — минимальная, не делимая часть выражения. Например выражение 25+2\*3 состоит из следующих лексем: 25, +, 2, \*, 3.

Для разбиения выражения на лексемы напишем функцию getToken(). Данная функция будет возвращать очередную лексему из строки с указанием ее типа. В нашем случаи нам понадобится три типа: число, оператор и переменная. Оператор - это знаки +, -, \*, /, %, (, ). Переменные — это латинские буквы от A до Z (Примечание: в данном примере анализатор не может обрабатывать переменные. Их обработку я рассмотрю в следующем примере, чуть позже)

И так, приступим к написанию функции getToken, вот ее код:

char \*expr; //Указатель на обрабатываемую строку

char token[80]; //Лексема

enum {Empty, Operator, Variable, Number} type; //Тип лексемы

int\* getToken(char \*expr)

{

static int i=0;

type=Empty;

if(expr[i]=='\0'){ //Если конец выражения

i=0;

return 0;

}

while(isspace(expr[i])) i++; //Пропустить разделительные символы

if(strchr("+-\*/%^=()", expr[i])){

\*token = expr[i];

\*(token+1) = '\0';

type = Operator;

}

else if(isalpha(expr[i])){

\*token = expr[i];

\*(token+1) = '\0';

type = Variable;

}

else if(isdigit(expr[i])){

int j=0;

token[j]=expr[i];

while(isdigit(expr[i+1])||expr[i+1]=='.')

token[++j]=expr[++i];

token[j+1]='\0';

type=Number;

}

i++;

return &i;

}

Функция принимает указатель на обрабатываемую строку \*expr. В начале функции объявлена статическая переменная i, которая хранит номер текущий позиции в анализируемой строке, далее устанавливается пустой тип лексемы (данный тип будет свидетельствовать о том, что лексем больше нет). Далее происходит проверка, не достигнут ли конец строки, и если он достигнут функция завершает свою работу. Потом с помощью библиотечной функции isspace пропускаются все разделительные символы (такие как пробел, символ табуляции и другие). Далее с помощь библиотечных функций strchr (поиск подстроки в строке), isalpha и isdigit происходит определение типа лексемы. Лексема записывается в глобальную переменную token[80], а ее тип определяется глобальным константным перечисление type. После этого функция возвращает адрес i, это необходимо, что бы обеспечить возможность обнулить эту переменную вне функции getToken().

После того, как мы получили возможность разбивать выражение на лексемы, мы можем приступить к его анализу. Ниже приведен полный код анализатора. Файл **Parser.h**

#ifndef PARSER\_H\_INCLUDED

#define PARSER\_H\_INCLUDED

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include <math.h>

int\* getToken(char\*); //Получает лексему из строки

void pars(char\*); //Точка входа анализатора

int fSum(double\*); //Обрабатывает сложение и вычитание

int fMulti(double\*); //Обрабатывает умножение и деление

int fExp(double\*); //Возведение в степень

int fUnary(double\*); //Обработка унарных операторов

int fBrack(double\*); //Обрабатывает выражение в скобках

int fAtom(double\*); //Получает значение числа

char \*expr; //Указатель на обрабатываемую строку

char token[80]; //Лексема

enum {Empty, Operator, Variable, Number} type; //Тип лексемы

enum {No, Syntax, Zero} error; //Значение ошибки

void pars(char \*line)

{

int \*pointer;

double result;

error=No;

expr=line;

pointer=getToken(expr);

fSum(&result);

\*pointer=0;

switch(error)

{

case No:

sprintf(expr, "%f", result);

break;

case Syntax:

strcpy(expr, "Syntax error!");

break;

case Zero:

strcpy(expr, "Divide by zero!");

break;

}

}

int\* getToken(char \*expr)

{

static int i=0;

type=Empty;

if(expr[i]=='\0'){ //Если конец выражения

i=0;

return 0;

}

while(isspace(expr[i])) i++; //Пропустить разделительные символы

if(strchr("+-\*/%^=()", expr[i])){

\*token = expr[i];

\*(token+1) = '\0';

type = Operator;

}

else if(isalpha(expr[i])){

\*token = expr[i];

\*(token+1) = '\0';

type = Variable;

}

else if(isdigit(expr[i])){

int j=0;

token[j]=expr[i];

while(isdigit(expr[i+1])||expr[i+1]=='.')

token[++j]=expr[++i];

token[j+1]='\0';

type=Number;

}

i++;

return &i;

}

int fSum(double \*anw)

{

char op;

double temp;

if(fMulti(anw)) return 1;

while((op = \*token) == '+' || op == '-'){

getToken(expr);

fMulti(&temp);

switch(op)

{

case '+':

\*anw += temp;

break;

case '-':

\*anw -= temp;

break;

}

}

return 0;

}

int fMulti(double \*anw)

{

char op;

double temp;

if(fExp(anw)) return 1; //Ошибка

while((op = \*token) == '\*' || op == '/' || op == '%'){

getToken(expr);

if(fExp(&temp)) return 1; //Ошибка

switch(op)

{

case '\*':

\*anw \*= temp;

break;

case '/':

if(temp == 0.0)

{

error=Zero;

return 1;

}

\*anw /= temp;

break;

case '%':

\*anw = (int)\*anw % (int)temp;

break;

}

}

return 0;

}

int fExp(double \*anw)

{

double temp;

if(fUnary(anw)) return 1; //Ошибка

while(\*token == '^'){

getToken(expr);

if(fUnary(&temp)) return 1; //Ошибка

\*anw = pow(\*anw, temp);

}

return 0;

}

int fUnary(double \*anw)

{

char op=0;

if(\*token == '+' || \*token == '-'){

op = \*token;

getToken(expr);

}

if(fBrack(anw)) return 1; //Ошибка

if(op == '-') \*anw = -(\*anw);

return 0;

}

int fBrack(double \*anw)

{

if(\*token == '('){

getToken(expr);

fSum(anw);

if(\*token != ')'){

error=Syntax;

return 1;

}

getToken(expr);

}

else

if(fAtom(anw)) return 1; //Ошибка

return 0;

}

int fAtom(double \*anw)

{

if(type == Number){

\*anw = atof(token);

getToken(expr);

}

else{

error=Syntax;

return 1;

}

return 0;

}

#endif // PARSER\_H\_INCLUDED

Обратите внимание, что весь код в файле Parser.h находится между условными выражениями #ifndef и #endif, а так же определена константа #define PARSER\_H\_INCLUDED, это позволяет исключить повторное включение данного файла в ваш проект.

В начале подключаются необходимые библиотечные функции и объявляются прототипы всех функций. Точкой входа анализатора является функция pars(), она принимает указатель на обрабатываемую строку и присваивает его значение глобальной переменной \*expr, что бы все функции могли обращаться к этой строке. Далее функция вызывает getToken() и в переменную token[80] попадает первая лексема. Потом вызывается первая из арифметических функций — fSum() и ей передается адрес переменной result, в которой в итоге будет содержаться результат вычислений.

Функция fSum() вызывает следующую — fMulti(), та в свою очередь вызывает fExp(). Когда управление переходит к функции fUnary(), она проверяет, не является ли лексема унарным плюсом или унарным минусом, если это так, функция записывает лексему (т.е. знак + или -) в переменную op и получает следующую лексему. Далее вызывается функция fBrack(), она проверяет, не является ли лексема знаком открывающийся скобки, и если это так функция рекурсивно вызывает fSum() и начинается вычисление выражения в скобках. Самой последний в цепочки вызывается функция fAtom(), она с помощью библиотечной функции atof() преобразует лексему в число типа double и записывает ее по адресу \*anw, т.е. в переменную result.

Далее управление передается функциям в обратном порядке. После завершения fAtom() мы попадаем в функцию fBrack(), потом в fUnary(). Функция fUnary() проверяет, содержится ли в переменной op унарный минус, и если это необходимо, она изменяет переменную result. Далее в функции fExp() происходит обработка возведения в степени, после чего мы попадаем в функцию fMulti(), которая проверяет, является ли текущая лексема знаком умножения, деления или знаком %(остаток от деления), если необходимо, функция производит нужные подсчеты. Реализуется это следующим образом: вначале функция fMulti() записывает текущий оператор в переменную op, затем вызывается функция fExp(), которой передается адрес временной переменной temp и вся цепочка начинается сначала. После завершения цепочки управление возвращается в функцию fMulti(), а переменная temp содержит значения второго операнда для оператора, который сохранен в переменной op. После выполнения арифметических действий управление переходит к функции fSum(), которая действует аналогично функции fMulti().

После всего этого функция переменная result содержит результат вычислений. Функция pars() копирует этот результат в исходную строку, на которую указывает \*expr, и работа анализатора завершается.

Теперь необходимо рассмотреть что будет, если в одной из функция произойдет ошибка, например не верное исходное выражение (20-8\*\*3). В данном выражении два раза подряд встречается оператор умножения, следовательно анализатор не сможет обработать его. Все арифметические функции возвращает 0, если работа завершена без ошибки, и 1 если она обнаружена. При вызове каждой функции проверяется ее возвращенное значение, например

if(fMulti(anw)) return 1;

Если какая-либо функция обнаружила ошибку, она устанавливает соответствующее значение ошибки в перечислении error, и тут же завершает свою работу вернув 1. Далее по цепочки завершаются все функции, а функция pars() вместо результата копирует в обрабатываемую строку соответствующее сообщение об ошибке.

**Использование анализатора.**

Следующая небольшая программа демонстрирует использование анализатора. Файл **parser\_console.cpp**

char \*expr; //Указатель на обрабатываемую строку

char token[80]; //Лексема

enum {Empty, Operator, Variable, Number} type; //Тип лексемы

int\* getToken(char \*expr)

{

static int i=0;

type=Empty;

if(expr[i]=='\0'){ //Если конец выражения

i=0;

return 0;

}

while(isspace(expr[i])) i++; //Пропустить разделительные символы

if(strchr("+-\*/%^=()", expr[i])){

\*token = expr[i];

\*(token+1) = '\0';

type = Operator;

}

else if(isalpha(expr[i])){

\*token = expr[i];

\*(token+1) = '\0';

type = Variable;

}

else if(isdigit(expr[i])){

int j=0;

token[j]=expr[i];

while(isdigit(expr[i+1])||expr[i+1]=='.')

token[++j]=expr[++i];

token[j+1]='\0';

type=Number;

}

i++;

return &i;

}

В конечном представлении анализатор оформлен в виде программы

#include<iostream>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<ctype.h>

#include<math.h>

usingnamespace **std;**

int\* getToken(char\*); //Получает лексему из строки

void **pars(**char\*); //Точка входа анализатора

int **fSum(**double\*); //Обрабатывает сложение и вычитание

int **fMulti(**double\*); //Обрабатывает умножение и деление

int **fExp(**double\*); //Возведение в степень

int **fUnary(**double\*); //Обработка унарных операторов

int **fBrack(**double\*); //Обрабатывает выражение в скобках

int **fAtom(**double\*); //Получает значение числа

char **\*expr;** //Указатель на обрабатываемую строку

char **token[80];** //Лексема

enum **{Empty, Operator, Variable, Number} type;** //Тип лексемы

enum **{No, Syntax, Zero} error;** //Значение ошибки

void **pars(**char **\*line)**//Точка входа анализатора

{

int **\*pointer;**

double **result;**

**error=No;**

//принимает указатель (line) на обрабатываемую строку и

//присваивает его значение глобальной переменной \*expr,

// что бы все функции могли обращаться к этой строке

**expr=line;**

**pointer=getToken(expr);**//в переменную token[80] попадает первая лексема

**fSum(&result);**//вызывается первая из арифметических функций

//ей передается адрес переменной result

**\*pointer=0;**

switch(error)

**{**

case **No:**

**sprintf(expr,** "%f", result);//копирует результат в исходную строку, на которую указывает \*expr

break;

case **Syntax:**

**strcpy(expr,** "Syntax error!");

break;

case **Zero:**

**strcpy(expr,** "Divide by zero!");

break;

**}**

}

int\* getToken(char **\*expr)**//Получает лексему из строки

{

staticint **i=0;**//номер текущий позиции в анализируемой строке

**type=Empty;**//пустой тип лексемы (данный тип будет свидетельствовать о том, что лексем больше нет)

if(expr[i]=='\0') //Если конец выражения. Конец строки?

**{**

**i=0;**

return **0;**

**}**

while(isspace(expr[i])) i++; //Пропустить разделительные символы.

//Пропускаются все разделительные символы (пробел, табуляции и другие)

// token[80] глобальная переменная

// ее тип определяется глобальным константным перечисление type

if(strchr("+-\*/%^=()", expr[i])) //поиск подстроки в строке

**{**

**\*token = expr[i];**

**\*(token+1) =** '\0';

**type = Operator;**

**}**

elseif(isalpha(expr[i])) //определение типа лексемы

**{**

**\*token = expr[i];**

**\*(token+1) =** '\0';

**type = Variable;**

**}**

elseif(isdigit(expr[i])) //определение типа лексемы

**{**

int **j=0;**

**token[j]=expr[i];**

while(isdigit(expr[i+1])||expr[i+1]=='.')

**token[++j]=expr[++i];**

**token[j+1]=**'\0';

**type=Number;**

**}**

**i++;**

return **&i;** //необходимо, что бы обеспечить возможность обнулить эту переменную вне функции getToken()

}

int **fSum(**double **\*anw)**//Обрабатывает сложение и вычитание

{

char **op;**

double **temp;**

if(fMulti(anw)) return **1;**//вызывается следующая арифметическая функция

while((op = \*token) == '+' **|| op ==** '-')

**{**

**getToken(expr);**

**fMulti(&temp);**

switch(op)

**{**

case'+':

**\*anw += temp;**

break;

case'-':

**\*anw -= temp;**

break;

**}**

**}**

return **0;**

}

int **fMulti(**double **\*anw)**//Обрабатывает умножение и деление

{

char **op;**

double **temp;**

if(fExp(anw)) return **1;** //Ошибка

////вызывается следующая арифметическая функция

while((op = \*token) == '\*' **|| op ==** '/' **|| op ==** '%')//запись текущего оператора в переменную op

**{**

**getToken(expr);**

//передается адрес временной переменной temp и вся цепочка начинается сначала

if(fExp(&temp)) return **1;** //Ошибка

switch(op)

**{**

case'\*':

**\*anw \*= temp;**

break;

case'/':

if(temp == 0.0)

**{**

**error=Zero;**

return **1;**

**}**

**\*anw /= temp;**

break;

case'%':

**\*anw = (**int)\*anw % (int)temp;

break;

**}**

**}**

return **0;**

}

int **fExp(**double **\*anw)**//Возведение в степень

//обработка возведения в степени

{

double **temp;**

if(fUnary(anw)) return **1;** //Ошибка

while(\*token == '^')

**{**

**getToken(expr);**

if(fUnary(&temp)) return **1;** //Ошибка

**\*anw = pow(\*anw, temp);**

**}**

return **0;**

}

int **fUnary(**double **\*anw)**//Обработка унарных операторов

//проверяет, не является ли лексема унарным плюсом или унарным минусом,

//если это так, функция записывает лексему (т.е. знак + или -) в переменную op и

//получает следующую лексему

{

char **op=0;**

if(\*token == '+' **|| \*token ==** '-')

**{**

**op = \*token;**

**getToken(expr);**

**}**

if(fBrack(anw)) return **1;** //Ошибка

//содержится ли в переменной op унарный минус, и

//если это необходимо, изменяет переменную result

if(op == '-') \*anw = -(\*anw);

return **0;**

}

int **fBrack(**double **\*anw)**//Обрабатывает выражение в скобках

//проверяет, не является ли лексема знаком открывающийся скобки, и если

//это так функция рекурсивно вызывает fSum() и начинается вычисление выражения в скобках

{

if(\*token == '(')

**{**

**getToken(expr);**

**fSum(anw);**

if(\*token != ')')

**{**

**error=Syntax;**

return **1;**

**}**

**getToken(expr);**

**}**

else

if(fAtom(anw)) return **1;** //Ошибка

return **0;**

}

int **fAtom(**double **\*anw)**//Получает значение числа

//с помощью библиотечной функции atof() преобразует лексему

//в число типа double и записывает ее по адресу \*anw,

//т.е. в переменную result

{

if(type == Number)

**{**

**\*anw = atof(token);**

**getToken(expr);**

**}**

else

**{**

**error=Syntax;**

return **1;**

**}**

return **0;**

}

int **main(**void)

{

char **expr[255];** //Содержит вычисляемое выражение

while(1)

**{**

**printf(**"expration>> ");

**gets(expr);**

if(!\*expr) break; //Если введена пустая строка - завершить программу

**pars(expr);** //Точка входа анализатора. Вычислить выражение

**printf(**" Result: %s\n\n", expr);

**}**

return **0;**

}

Обработка выражения происходит в бесконечном цикле до тех пор, пока не будет введена пустая строка



### Быстрые сортировки

Простые методы сортировки вроде метода выбора или метода пузырька сортируют массив из *n* элементов за O(*n*2) операций. Однако с помощью принципа «разделяй и властвуй» удается построить более быстрые, работающие за O(*n* log2 *n*) алгоритмы. Суть этого принципа в том, что решение получается путем рекурсивного разделения задачи на несколько простые подзадачи того же типа до тех пор, пока они не станут элементарными. Приведем в качестве примеров несколько быстрых алгоритмов такого рода.

Алгоритм 1: «Быстрая» сортировка (quicksort).

* Выбирается опорный элемент (например, первый или случайный).
* Реорганизуем массив так, чтобы сначала шли элементы меньшие опорного, потом равные ему, затем большие. Для этого достаточно помнить, сколько было найдено меньших (*m*1) и больших (*m*2), чем опорный и ставить очередной элемент на место с индексом *m*1, а очередной больший на место с индексом *n*-1-*m*2.

После выполнения такой операции опорный элемент и равные ему стоят на своем месте, их переставлять больше не придется. Между «меньшей» и «большей» часть массива перестановок также быть не может. То есть эти части можно сортировать независимо друг от друга.

* Если «меньшая» или «большая» часть состоит из одного элемента, то она уже отсортирована и делать ничего не надо. Иначе сортируем эти части с помощью алгоритма быстрой сортировки (то есть, выполняем для нее шаги 1-3).

Как видите, быстрая сортировка состоит из выполнения шагов 1 и 2 и рекурсивного вызова алгоритма для получившихся частей массива.

#### Quicksort. Пример 1.

***Быстрая сортировка*** представляет собой усовершенствованный метод сортировки, основанный на принципе обмена. Пузырьковая сортировка является самой неэффективной из всех алгоритмов прямой сортировки. Однако усовершенствованный алгоритм является лучшим из известных методом сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель Ч. Хоар назвал его быстрой сортировкой.

Для достижения наибольшей эффективности желательно производить обмен элементов на больших расстояниях. В массиве выбирается некоторый элемент, называемый **разрешающим**. Затем он помещается в то место массива, где ему полагается быть после упорядочивания всех элементов. В процессе отыскания подходящего места для разрешающего элемента производятся перестановки элементов так, что слева от них находятся элементы, меньшие разрешающего, и справа — большие (предполагается, что массив сортируется по возрастанию).

Тем самым массив разбивается на две части:

* не отсортированные элементы слева от разрешающего элемента;
* не отсортированные элементы справа от разрешающего элемента.

Чтобы отсортировать эти два меньших подмассива, алгоритм рекурсивно вызывает сам себя.

Если требуется сортировать больше одного элемента, то нужно

* выбрать в массиве разрешающий элемент;
* переупорядочить массив, помещая элемент на его окончательное место;
* отсортировать рекурсивно элементы слева от разрешающего;
* отсортировать рекурсивно элементы справа от разрешающего.

Ключевым элементом быстрой сортировки является **алгоритм переупорядочения**.

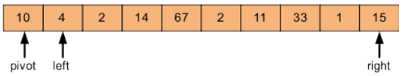
Рассмотрим сортировку на примере массива:

Рассмотрим сортировку на примере массива:

**10, 4, 2, 14, 67, 2, 11, 33, 1, 15.**

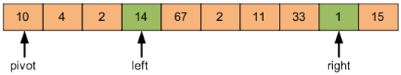
Для реализации алгоритма переупорядочения используем указатель left на крайний левый элемент массива. Указатель движется вправо, пока элементы, на которые он показывает, остаются меньше разрешающего. Указатель right поставим на крайний правый элемент массива, и он движется влево, пока элементы, на которые он показывает, остаются больше разрешающего.

Пусть крайний левый элемент — разрешающий pivot. Установим указатель left на следующий за ним элемент; right — на последний. Алгоритм должен определить правильное положение элемента 10 и по ходу дела поменять местами неправильно расположенные элементы.



Движение указателей останавливается, как только встречаются элементы, порядок расположения которых относительно разрешающего элемента неправильный.

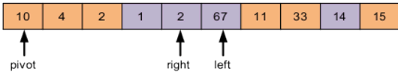
**Указатель left перемещается до тех пор, пока не покажет элемент больше 10; right движется, пока не покажет элемент меньше 10**.



**Эти элементы меняются местами и движение указателей возобновляется**.

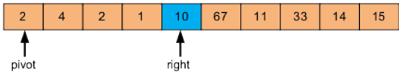


**Процесс продолжается до тех пор, пока right не окажется слева от left**.



**Тем самым будет определено правильное место разрешающего элемента.**

**Осуществляется перестановка разрешающего элемента с элементом, на который указывает right**.



**Разрешающий элемент находится в нужном месте: элементы слева от него имеют меньшие значения; справа — большие. Алгоритм рекурсивно вызывается для сортировки подмассивов слева от разрешающего и справа от него**.

Реализация алгоритма быстрой сортировки

#include <stdio.h>

#include <stdlib.h>

// Функция быстрой сортировки

void quickSort(int \*numbers, int left, int right)

{

int pivot; // разрешающий элемент

int l\_hold = left; //левая граница

int r\_hold = right; // правая граница

pivot = numbers[left];

while (left < right) // пока границы не сомкнутся

{

while ((numbers[right] >= pivot) && (left < right))

right--; // сдвигаем правую границу пока элемент [right] больше [pivot]

if (left != right) // если границы не сомкнулись

{

numbers[left] = numbers[right]; // перемещаем элемент [right] на место разрешающего

left++; // сдвигаем левую границу вправо

}

while ((numbers[left] <= pivot) && (left < right))

left++; // сдвигаем левую границу пока элемент [left] меньше [pivot]

if (left != right) // если границы не сомкнулись

{

numbers[right] = numbers[left]; // перемещаем элемент [left] на место [right]

right--; // сдвигаем правую границу вправо

}

}

numbers[left] = pivot; // ставим разрешающий элемент на место

pivot = left;

left = l\_hold;

right = r\_hold;

if (left < pivot) // Рекурсивно вызываем сортировку для левой и правой части массива

quickSort(numbers, left, pivot - 1);

if (right > pivot)

quickSort(numbers, pivot + 1, right);

}

int main()

{

int a[10];

// Заполнение массива случайными числами

for (int i = 0; i<10; i++)

a[i] = rand() % 20 - 10;

// Вывод элементов массива до сортировки

for (int i = 0; i<10; i++)

printf("%d ", a[i]);

printf("\n");

quickSort(a, 0, 9); // вызов функции сортировки

// Вывод элементов массива после сортировки

for (int i = 0; i<10; i++)

printf("%d ", a[i]);

printf("\n");

getchar();

return 0;

}



#### Quicksort. Пример 2.

Представляется рекурсивный пример реализации быстрой сортировки. Массив, который нужно отсортировать будет глобальным, если же вы захотите сделать его локальным, то просто передавайте его как параметр в функцию. Конечно, после того, как вы сделаете это вам нужно будет использовать template для того, чтобы ваша функция работала со всеми типами данных. Приведем код рекурсивной сортировки quicksort:

#include <iostream>

using namespace std;

int a[100];

void quickSort(int l, int r)

{

int x = a[l + (r - l) / 2];

//запись эквивалентна (l+r)/2,

//но не вызввает переполнения на больших данных

int i = l;

int j = r;

//код в while обычно выносят в процедуру particle

while(i <= j){

while(a[i] < x) i++;

while(a[j] > x) j--;

if(i <= j){

swap(a[i], a[j]);//взаимозамена элементов

i++;

j--;

}

}

if (i<r)

quickSort(i, r);

if (l<j)

quickSort(l, j);

}

int main()

{

int n;//количество элементов в массиве

setlocale(LC\_ALL, "Rus");

cout << "Количество элементов в массиве: ";

cin >> n;

for(int i = 0; i < n; i++){

cout << "array a[" << i << "]= ";

cin >> a[i];

}

quickSort(0, n-1);

cout << "\n";

for(int i = 0; i < n; i++){

cout << a[i] << " ";

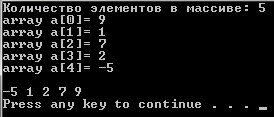
}

cout << "\n";

system("PAUSE");

return 0;

}



#### Quicksort. Пример 3.

#include<stdio.h>

#include<stdlib.h>

#include<iostream>

usingnamespace **std;**

void **quickSort(**int **arr[],** int **left,** int **right) {**

int **i = left, j = right;**

int **tmp;**

int **pivot = arr[(left + right) / 2];**

/\* partition \*/

while **(i <= j) {**

while **(arr[i] < pivot)**

**i++;**

while **(arr[j] > pivot)**

**j--;**

if **(i <= j) {**

**tmp = arr[i];**

**arr[i] = arr[j];**

**arr[j] = tmp;**

**i++;**

**j--;**

**}**

**};**

/\* recursion \*/

if **(left < j)**

**quickSort(arr, left, j);**

if **(i < right)**

**quickSort(arr, i, right);**

}

int **main()**

{

int **a[10];**

// Заполнение массива случайными числами

for **(**int **i = 0; i<10; i++)**

**a[i] = rand() % 20 - 10;**

// Вывод элементов массива до сортировки

for **(**int **i = 0; i<10; i++)**

**printf(**"%d ", a[i]);

**printf(**"\n");

**quickSort(a, 0, 9);** // вызов функции сортировки

// Вывод элементов массива после сортировки

**printf(**"\n");

for **(**int **i = 0; i<10; i++)**

**printf(**"%d ", a[i]);

**printf(**"\n");

**getchar();**

return **0;**

}



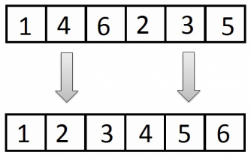
Алгоритм 2: Сортировка слиянием (merge sort).

* Делим массив на две части примерно одинакового размера и, если получившаяся половина массива содержит больше одного элемента, то сортируем ее с помощью сортировки слиянием. Как видите, этот пункт содержит рекурсивное обращение ко всему алгоритму в целом.
* Соединяем две отсортированные половины так, чтобы получился один отсортированный массив. Для этого помещаем во вспомогательный массив элементы из первой половины, пока они не превосходят очередного элемента из второй половины. Затем начинаем помещать туда элементы второй половины, пока они не превосходят очередного элемента из первой половины. Затем снова берем элементы первой половины и т.д. Эта операция называется слиянием и требует столько шагов, сколько элементов в обоих соединяемых массивах.

#### Merge sort. Пример 1.

**http://programmado.ru/59-sortirovka-sliyaniem-na-s.html**

**Сортировка слиянием на С++**



Сортировка слиянием - это довольно быстрая сортировка, время работы которой О(n \* log n). Однако ее недостатком является тот факт, что она требует относительно много памяти.

Итак, пусть дан некоторый неупорядоченный массив int a[maxn].

Основная идея состоит в том, что на каждом шаге мы разбиваем массив на 2 равные части, сортируем их, а потом сливаем два отсортированных куска. То есть получается рекурсивная сортировка, т.к. каждую из этих 2 частей мы будем сортировать аналогично. Выход из рекурсии будет происходить тогда, когда у нас остается меньше 3 элементов. Если их остается всего 2, то меняем их между собой по мере надобности. Если остается только 1 элемент, то оставляем его в покое.

Пример. Пусть дан исходный массив из 7 элементов:

#include<iostream>

#include<conio.h>

usingnamespace **std;**

#define **maxn 100**

//int n;

//int a[maxn];

int **n = 7;**

int **a[7] = {7, 4, 2, 1, 0, 5, 3};**

void **merge(**int **l,** int **r) {**

if **(r == l)**

return; //остался один элемент

if **(r - l == 1) {** //осталось два элемента

if **(a[r] < a[l])**

**swap(a[r], a[l]);** //перестановка элементов местами

return;

**}**

int **m = (r + l) / 2;** //деление массива на части

**merge(l, m);**//рекурсия для левой части массива обрабатывается впервую очередь

**merge(m + 1, r);**//рекурсия для правой части массива выполняется второй

//начинает работать на 3 шаге примера(выполняет слияние)

int **buf[maxn];**

int **xl = l;**//

int **xr = m + 1;**//

int **cur = 0;**

//формируем сортированный буффер (сливание)

while **(r - l + 1 != cur) {**

if **(xl > m)**

**buf[cur++] = a[xr++];**

elseif **(xr > r)**

**buf[cur++] = a[xl++];**

elseif **(a[xl] > a[xr])**

**buf[cur++] = a[xr++];**

else **buf[cur++] = a[xl++];**

**}**

//перазаписываем массив из буфера

for **(**int **i = 0; i < cur; i++)**

**a[i + l] = buf[i];**

}

int **main()**

{

**setlocale(LC\_ALL,** "Rus");

//cout << "Количество элементов в массиве: ";

//cin >> n;

//for (int i = 0; i < n; i++){

// cout << "array a[" << i << "]= ";

// cin >> a[i];

//}

//n=7;

//a[0]=7; a[1]=4; a[2]=2; a[3]=1; a[4]=0; a[5]=5; a[6]=3;

**merge(0, n - 1);**

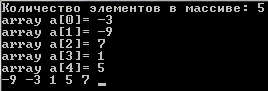
for **(**int **i = 0; i < n; i++)**

**cout << a[i] <<** " ";

**getch();**

return **0;**

}



#### Merge sort. Пример 2.

**https://prog-cpp.ru/sort-merge/**

Алгоритмы сортировки массивов не всегда применимы, если сортируемые данные расположены в структуре с последовательным доступом, которая характеризуется тем, что в каждый момент имеется непосредственный доступ к одному и только одному компоненту.

Основной применяемый метод для сортировки файлов — ***сортировка слиянием***.

***Сортировка слиянием*** — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке.

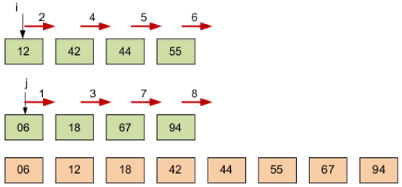
***Слияние*** означает объединение двух (или более) последовательностей в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент.

Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Затем их решения комбинируются, и получается решение исходной задачи.

Операция, которая однократно обрабатывает множество данных, называется ***фазой***.

Наименьший подпроцесс, который, повторяясь, образует процесс сортировки, называется ***проходом*** или ***этапом***.

Процедура слияния предполагает объединение двух предварительно упорядоченных подпоследовательностей размерности n/2 в единую последовательность размерности n. Начальные элементы предварительно упорядоченных последовательностей сравниваются между собой, и из них выбирается наименьший. Соответствующий указатель перемещается на следующий элемент. Процедура повторяется до тех пор, пока не достигнут конец одной из подпоследовательностей. Оставшиеся элементы другой подпоследовательности при этом передаются в результирующую последовательность в неизменном виде.



Сортировка слиянием во многом похожа на метод быстрой сортировки. Производительность сортировки слиянием лежит между производительностью пирамидальной и быстрой сортировки. Но в отличие от пирамидальной и быстрой сортировок, метод сортировки слиянием ведет себя стабильно, поскольку он не зависит от перестановок элементов в массиве.

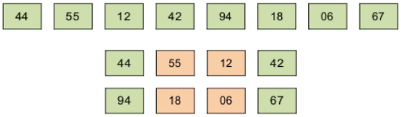
**Еще одним достоинством сортировки слиянием является то, что он удобен для структур с последовательным доступом к элементам, таким как файлы на внешнем устройстве или связные списки. Этот метод, прежде всего, используется для внешней сортировки.**

**Недостатки метода заключаются в том, что он требует дополнительной памяти по объему равной объему сортируемого файла. Поэтому для больших файлов проблематично организовать сортировку слиянием в оперативной памяти.**

В случаях, когда гарантированное время сортировки важно и размещение в оперативной памяти, возможно, следует предпочесть метод сортировки слиянием.

**Алгоритм двухпутевого слияния**

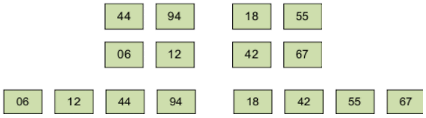
Исходная последовательность разбивается на две подпоследовательности:



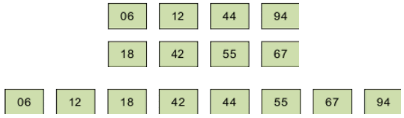
Эти две подпоследовательности объединяются в одну, содержащую упорядоченные пары.



Полученная последовательность снова разбивается на две, и пары объединяются в упорядоченные четверки:



Полученная последовательность снова разбивается на две и собирается в упорядоченные восьмерки.



Данная операция повторяется до тех пор, пока полученная упорядоченная последовательность не будет иметь такой же размер, как у сортируемой.

Основной операцией является слияние. При слиянии требуется дополнительная память для размещения файла, образующегося при слиянии. Операция линейно зависит от количества элементов в объединяемых массивах.

#include <stdio.h>

#include <stdlib.h>

// Функция сортировки двухпутевым слиянием

void merge(int \*a, int n)

{

int mid = n / 2; // находим середину сортируемой последовательности

if (n % 2 == 1)

mid++;

int h = 1; // шаг

// выделяем память под формируемую последовательность

int \*c = (int\*)malloc(n \* sizeof(int));

int step;

while (h < n){

step = h;

int i = 0; // индекс первого пути

int j = mid; // индекс второго пути

int k = 0; // индекс элемента в результирующей последовательности

while (step <= mid){

while ((i < step) && (j < n) && (j < (mid + step))){

// пока не дошли до конца пути

// заполняем следующий элемент формируемой последовательности

// меньшим из двух просматриваемых

if (a[i] < a[j]){

c[k] = a[i];

i++; k++;

}

else {

c[k] = a[j];

j++; k++;

}

}

while (i < step){

// переписываем оставшиеся элементы первого пути (если второй кончился раньше)

c[k] = a[i];

i++; k++;

}

while ((j < (mid + step)) && (j<n)){

// переписываем оставшиеся элементы второго пути (если первый кончился раньше)

c[k] = a[j];

j++; k++;

}

step = step + h; // переходим к следующему этапу

}

h = h \* 2;

// Переносим упорядоченную последовательность (промежуточный вариант) в исходный массив

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

a[i] = c[i];

}

}

int main()

{

int a[8];

// Заполнение массива случайными числами

for (int i = 0; i<8; i++)

a[i] = rand() % 20 - 10;

// Вывод элементов массива до сортировки

for (int i = 0; i<8; i++)

printf("%d ", a[i]);

printf("\n");

merge(a, 8); // вызов функции сортировки

// Вывод элементов массива после сортировки

for (int i = 0; i<8; i++)

printf("%d ", a[i]);

printf("\n");

getchar();

return 0;

}



#### Merge sort. Пример 3.

**http://cybern.ru/mergesortcpp.html**

Основной недостаток алгоритма сортировки слиянием — это использование дополнительной памяти. Для того, чтобы удобно работать с ней в качестве контейнера для массива используется вектор. Для демонстрации алгоритма сортировки слиянием изначальный вектор заполняется случайными значениями.

#include <vector>

#include <iostream>

using namespace std;

vector <int> merge(vector <int> a, vector <int> b)

//функция сливает два вектора

{

vector<int> c(a.size()+b.size());//результирующий вектор

//количество использованных вершин

int kol1 = 0;//в векторе а

int kol2 = 0;//в векторе b

//заполним массив c

for(int i = 0; i < c.size(); i++){

//если мы уже полностью использовали вектора а,

//то нам осталось дописать элементы вектора b

if(kol1==a.size()){

c[i] = b[kol2];

kol2++;

continue;

}

//наоборот

if(kol2==b.size()){

c[i] = a[kol1];

kol1++;

continue;

}

//так как наши вектора еще не полностью

//использованы, то нужно сравнить какое

//значение ставить на следующее место

if(a[kol1]<=b[kol2]){

c[i] = a[kol1];

kol1++;

}

else{

c[i] = b[kol2];

kol2++;

}

}

return c;

}

//функция сортирует вектор

vector<int> merge\_sort(vector <int> a)

{

//если размер вектора меньше 1,

//то он отсортирован

if(a.size()<=1)

return a;

vector<int> b,c;

b.assign(a.begin(), a.end()-(a.size()/2));

c.assign(a.end()-(a.size()/2), a.end());

return merge(merge\_sort(b),merge\_sort(c));

}

int main()

{

vector<int> a;

for(int i = 0; i < 10; i++){

a.push\_back(rand());

}

a = merge\_sort(a);

for(int i = 0;i < a.size();i++){

cout<<a[i]<<" ";

}

getchar();

return 0;

}



Алгоритм 3: Сортировка деревом (tree sort).

Прежде чем переходить к объяснению сути алгоритма введем одно понятие. ***Двоичным деревом поиска*** называется бинарное дерево, в узлах которого располагаются числа таким образом, что в левом поддереве каждого узла находятся числа меньшие, чем в этом узле, а в правом поддереве больше или равные тому, что в этом узле. На рис. 10 показано два примера деревьев поиска, составленных из одних и тех же чисел.

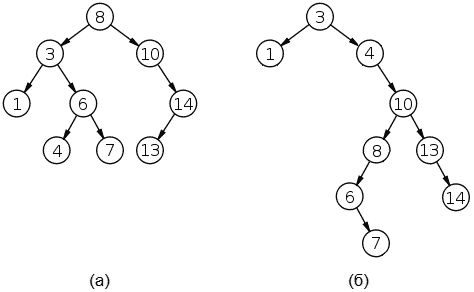


Рис. 10. Двоичные деревья поиска, составленные из чисел 1, 3, 4, 6, 7, 8, 10, 13, 14.

Если для каждой вершины высота поддеревьев различается не более чем на единицу, то дерево называется *сбалансированным*. Сбалансированные деревья поиска также называются *АВЛ-деревьями* (по первым буквам фамилий изобретателей Г. М. Адельсона-Вельского и Е. М. Ландиса). Как видно на рис. 10а показано сбалансированное дерево, на рис. 10б несбалансированное.

Заметим, что расположение чисел по возрастанию получится, если обходить эти деревья в обратном прядке.

Сортировка деревом получится, если мы сначала последовательно будем добавлять числа из массива в двоичное дерево поиска, а затем обойдем его в обратном порядке.

Если дерево будет близко к сбалансированному, то сортировка потребует примерно *n* log2 *n* операций. Если не повезет и дерево окажется максимально несбалансированным, то сортировка займет *n*2 операций.

#### Tree sort. Пример 1.

**https://prog-cpp.ru/sort-tree/**

Сортировка с помощью дерева осуществляется на основе бинарного дерева поиска.

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

* оба поддерева – левое и правое, являются двоичными деревьями поиска;
* у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
* у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

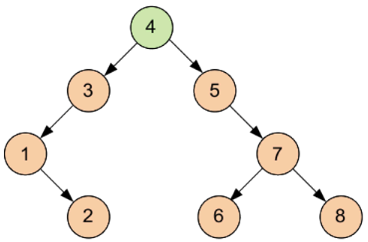
Для сортировки с помощью дерева исходная сортируемая последовательность представляется в виде структуры данных "дерево".

Например, исходная последовательность имеет вид:

**4, 3, 5, 1, 7, 8, 6, 2**

Корнем дерева будет начальный элемент последовательности. Далее все элементы, меньшие корневого, располагаются в левом поддереве, все элементы, большие корневого, располагаются в правом поддереве. Причем это правило должно соблюдаться на каждом уровне.

После того, как все элементы размещены в структуре "дерево", необходимо вывести их, используя [**инфиксную форму обхода**](https://prog-cpp.ru/data-tree/)**.**



#include <iostream>

using namespace std;

// Структура - узел дерева

struct tnode

{

int field; // поле данных

struct tnode \*left; // левый потомок

struct tnode \*right; // правый потомок

};

// Вывод узлов дерева (обход в инфиксной форме)

void treeprint(tnode \*tree)

{

if (tree != NULL) { //Пока не встретится пустой узел

treeprint(tree->left); //Рекурсивная функция вывода левого поддерева

cout << tree->field << " "; //Отображаем корень дерева

treeprint(tree->right); //Рекурсивная функция вывода правого поддерева

}

}

// Добавление узлов в дерево

struct tnode \* addnode(int x, tnode \*tree) {

if (tree == NULL) // Если дерева нет, то формируем корень

{

tree = new tnode; //память под узел

tree->field = x; //поле данных

tree->left = NULL;

tree->right = NULL; //ветви инициализируем пустотой

}

else // иначе

if (x < tree->field) //Если элемент x меньше корневого, уходим влево

tree->left = addnode(x, tree->left); //Рекурсивно добавляем элемент

else //иначе уходим вправо

tree->right = addnode(x, tree->right); //Рекурсивно добавляем элемент

return(tree);

}

//Освобождение памяти дерева

void freemem(tnode \*tree)

{

if (tree != NULL){ // если дерево не пустое

freemem(tree->left); // рекурсивно удаляем левую ветку

freemem(tree->right); // рекурсивно удаляем правую ветку

delete tree; // удаляем корень

}

}

// Тестирование работы

int main()

{

struct tnode \*root = 0; // Объявляем структуру дерева

setlocale(LC\_ALL, "Rus"); // переходим на русский язык в консоли

//system("cls");

int a; // текущее значение узла

// В цикле вводим 8 узлов дерева

cout << "Введите 8 узлов\n";

for (int i = 0; i< 8; i++){

cout << "Введите узел " << i + 1 << ": ";

cin >> a;

root = addnode(a, root); // размещаем введенный узел на дереве

}

cout << "\n\n";

treeprint(root); // выводим элементы дерева, получаем отсортированный массив

freemem(root); // удаляем выделенную память

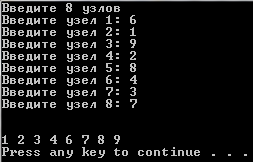
cout << "\n";

system("PAUSE");

//cin.get(); cin.get();

return 0;

}



#### Tree sort. Пример 2.

**http://algolist.manual.ru/sort/faq/q7.php**

Двоичным(бинарным) деревом назовем упорядоченную структуру данных, в которой каждому элементу - предшественнику или корню (под)дерева - поставлены в соответствие по крайней мере два других элемента (преемника).

Причем для каждого предшественника выполнено следующее правило: левый преемник всегда меньше, а правый преемник всегда больше или равен предшественнику.

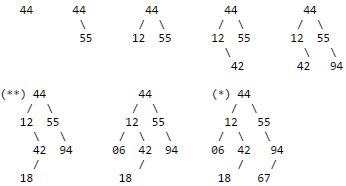
Вместо 'предшественник' и 'преемник' также употребляют термины 'родитель' и 'сын'. Все элементы дерева также называют 'узлами'.

При добавлении в дерево нового элемента его последовательно сравнивают с нижестоящими узлами, таким образом вставляя на место.

Если элемент >= корня - он идет в правое поддерево, сравниваем его уже с правым сыном, иначе - он идет в левое поддерево, сравниваем с левым, и так далее, пока есть сыновья, с которыми можно сравнить.

Вот процесс построения дерева из последовательности

**44 55 12 42 94 18 06 67**



Дерево может быть и более-менее ровным, как на (\*), может и иметь всего две основные ветви (\*\*), а если входная последовательность уже отсортирована, то дерево выродится в **линейный список**.

Если мы будем рекурсивно обходить дерево по правилу "левый сын - родитель - правый сын", то, записывая все встречающиеся элементы в массив, мы получим упорядоченное в порядке возрастания множество. Это и есть основана идея сортировки деревом.

Более подробно правило обхода можно сформулировать как обойти левое поддерево - вывести корень - обойти правое поддерево, где рекурсивная процедура 'обойти' вызывает себя еще раз, если сталкивается с узлом-родителем и выдает очередной элемент, если у узла нет сыновей.

/\*\*\*\*\*\*\*\*\*\*\* сортировка с помощью двоичного дерева \*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

typedef struct tree

{

int a; // данные

struct tree \*left; // левый сын

struct tree \*right; // правый сын

} TREE;

TREE \*add\_to\_tree(TREE \*root, int new\_value)

{

if (root==NULL) // если нет сыновей - создаем новый элемент

{

root = (TREE\*)malloc(sizeof(TREE));

root->a = new\_value;

root->left = root->right = 0;

return root;

}

if (root->a < new\_value) // добавлем ветвь

root->right = add\_to\_tree(root->right, new\_value);

else

root->left = add\_to\_tree(root->left, new\_value);

return root;

}

void tree\_to\_array(TREE \*root, int a[]) // процедура заполнения массива

{

static int max2=0; // счетчик элементов нового массива

if (root==NULL) return; // условие окончания - нет сыновей

tree\_to\_array(root->left, a); // обход левого поддерева

a[max2++] = root->a;

tree\_to\_array(root->right, a); // обход правого поддерева

free(root);

}

void sort\_tree(int a[], int elem\_total) // собственно сортировка

{

TREE \*root;

int i;

root = NULL;

for (i=0; i<elem\_total; i++) // проход массива и заполнение дерева

root = add\_to\_tree(root, a[i]);

tree\_to\_array(root, a); // заполнение массива

}

/\* тестовая программа \*/

void main() {

int i;

/\* Это будем сортировать \*/

int a[14]={ 0,7,8,3,52,14,16,18,15,13,42,30,35,26 };

printf("Initial array:\n");

for (i=0; i<14; i++) printf("%d ",a[i]);

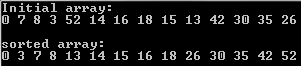
sort\_tree(a, 14);

printf("\n\nsorted array:\n");

for (i=0;i <14; i++) printf("%d ",a[i]);

getchar();

}



Общее быстродействие метода O(nlogn). Поведение неестественно, устойчивости, вообще говоря, нет.

Основной недостаток этого метода - большие требования к памяти под дерево. Очевидно, нужно n места под ключи и, кроме того, память на 2 указателя для каждого из них.

Поэтому TreeSort обычно применяют там, где

* построенное дерево можно с успехом применить для других задач.
* данные уже построены в 'дерево'. } не тратится
* данные можно считывать непосредственно в дерево. } лишняя
* например, при потоковом вводе с консоли или из файла. } память

Кроме того, ее элементы иногда используются в смежных задачах. Другое описание метода 'древесной сортровки' можно найти, например, у H. Вирта в книге 'Алгоритмы + Структуры Данных = Программы'.

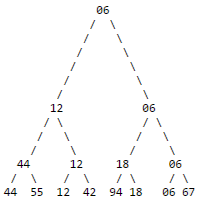
Основное различие - строится не представление данных в виде дерева, а 'дерево выбора'. С помощью n/2 сравнений можно определить наименьший элемент из каждой пары, при помощи следующих n/4 - наименьший из каждой пары таких наименьших ключей и т.д

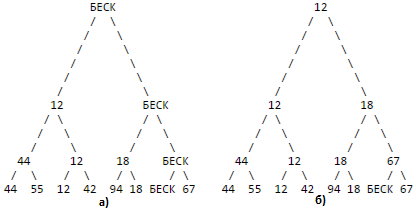
При этом за n-1 сравнений мы можем построить дерево выбора, как показано для чисел

**44 55 12 42 94 18 06 67:**

Это HЕ двоичное дерево в смысле, определенном выше. И это HЕ пирамида, используемая в HeapSort.

Hа втором шаге мы спускаемся по пути, указанному наименьшим ключом и исключаем его, последовательно заменяя либо на 'дыру' (или ключ 'бесконечность'(БЕСК), либо на элемент, находящийся на противоположной ветви промежуточного узла:





а) взяли ключ 06 сверху

б) опять выбрали наименьший с учетом, что любой ключ < БЕСК.

Очевидно, требуется n операций на создание первоначального дерева, а затем n шагов, каждый по log n сравнений для выбора нового наименьшего.

Такой вариант TreeSort довольно сильно отличается от изложенного выше. Этот алгоритм еще называют 'выбор с замещением' и его можно неплохо применять в тех же случаях, что и выше.

При этом он может быть даже выгоднее предыдущего метода, хотя бы потому, что не нужно запоминать позицию, на которой мы остановились при обходе дерева, т.е можно взять верхний элемент -> восстановить дерево -> проделать некоторые операции -> взять следующий элемент и т.п.

Обходить же дерево удобно сразу целиком, либо какую-то его большую часть.Кроме того, самой структура дерева выбора также может быть полезна.

Однако, нужно учесть, что памяти для дерева выбора нужно на 2n-1 элементов (элемент = ключ + указатели), в отличие от n элементов для простого дерева.

Пример использования выбора с замещением можно увидеть в многофазной сортировке (см соответствующий вопрос). Элементы обоих 'древесных' сортировок также используются в смежных

## Рекурсия или цикл. Избавление от рекурсии

Программисты находятся в постоянном поиске наиболее продуктивных способов повышения производительности приложений. И хотя большинству программистов известны концепции циклов и рекурсии, многие используют лишь один из этих методов, не задумываясь о том, насколько можно повысить производительность, выбрав оптимальный подход. На самом деле даже небольшое изменение может оказать значительное влияние на производительность приложений.

### Рекурсия

Рекурсией в программировании называется *метод, вызывающий сам себя*. Несмотря на простоту определения, этот подход достаточно сложен для понимания, поскольку человеку не свойственен подобный стиль мышления. Более того, существуют рекурсии разных типов, что вносит еще большую путаницу. Чаще всего применяются рекурсии следующих двух типов:

**Головная рекурсия**— рекурсивный вызов выполняется ближе к началу метода и является одним из первых обрабатываемых объектов. Поскольку он вызывает сам себя, ему приходится полагаться на результаты предыдущей операции, помещенной в стек вызовов. Из-за использования стека вызовов существует вероятность переполнения стека, если стек вызовов недостаточно велик.

**Концевая рекурсия**— рекурсивный вызов выполняется в конце и является последней строкой обрабатываемого кода. Этот метод не использует стек вызовов независимо от глубины рекурсии.

В математике рекурсия распространена достаточно широко, поэтому проще будет объяснить ее на примере рекурсивного вызова. Выражение 5! (факториал числа 5) можно записать следующим способом:

5!

5 \* 4!

5 \* 4 \* 3!

5 \* 4 \* 3 \* 2!

5 \* 4 \* 3 \* 2 \* 1!

5 \* 4 \* 3 \* 2 \* 1

В листинге 1 приведен пример вычисления 5! с помощью головной рекурсии.

public long getFactorial(long currNum) {

if (currNum == 1)

return 1;

return currNum \* getFactorial(currNum - 1);

}

В этом случае рекурсивные вызовы не используют стек вызовов для хранения информации, необходимой для расчета факториала. Вместо этого они передают результат предыдущему вызову.

getFactorial(5,0)

getFactorial(4,5)

getFactorial(3,20)

getFactorial(2,60)

getFactorial(1,120)

getFactorial(0,120)

120

### Цикл

Языки программирования предоставляют циклы нескольких разных типов, очень хорошо знакомых большинству программистов. В языке программирования Java, C имеются циклы **for, do** и **while**. Цикл — это многократное исполнение нескольких операторов. Циклы не заносят данные в стек вызовов независимо от числа исполнений цикла. Важным отличием циклов от рекурсивных функций является тот факт, что циклы используют для подсчета числа исполнений итератор, а рекурсивные функции для определения момента выхода должны выполнять сравнение результатов. Другим важным отличием является возможность применения в циклах фильтров и прочих селекторов. Примером такой ситуации может служить цикл **foreach**.

### Контрольные примеры

Приведенные ниже контрольные примеры запускались в 64-разрядной среде исполнения IBM Java Runtime Environment (JRE) 7.0.4.0 (с аргументом командной строки -Xms256m -Xmx256m -Dcom.ibm.tools.attach.enable=no). Чтобы среда исполнения не тратила время на расширение и сжатие кучи, JRE запускалась с фиксированным размером кучи 256 МБ. Отключение API Attach не позволяет JRE запускать приложения-агенты (обычно используемые для мониторинга), что нормализует производительность в каждом тесте. При увеличении стека вызовов для инициализации стека и поддержания его на уровне 3 МБ использовался аргумент командной строки -Xss3m -Xssi3m.

Вычисление суммы

При суммировании чисел цикл показал значительно более высокую производительность, а концевая рекурсия оказалась быстрее головной. При увеличении стека вызовов Java до 3 МБ головная рекурсия сравнялась по скорости с концевой, но все же не смогла догнать цикл.

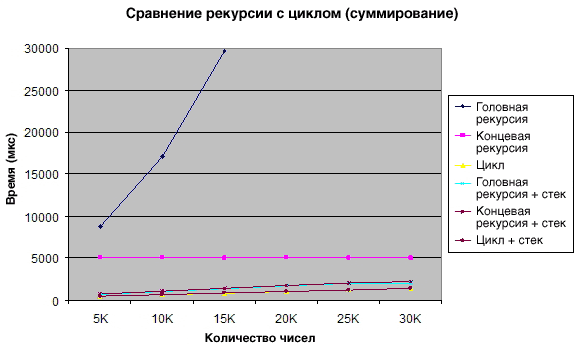


Рисунок 1. Вычисление суммы

Вычисление факториала

Этот примечательный пример иллюстрирует зависимость результатов от используемых операторов. При использовании простого типа данных **int** лучшие результаты во всех случаях получились для цикла. Применение типа int ограничивает величину результата до 32-разрядного целого числа со знаком. Для больших факториалов можно использовать тип данных BigInteger, но такая конструкция будет более затратной. Результаты применения **BigInteger** показали, что использование головной рекурсии в паре с концевой обеспечивает лучшее быстродействие, чем чисто концевая рекурсия или цикл.

Неправильный русунок

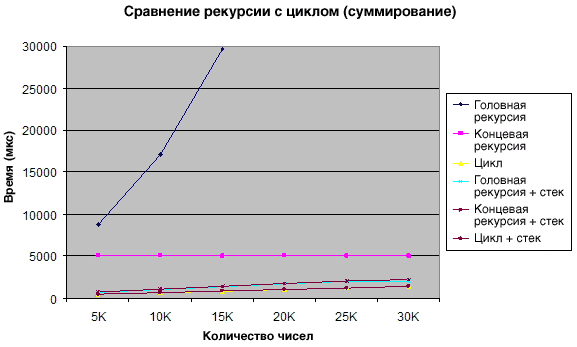


Рисунок 2. Вычисление факториала

### Области применения рекурсии

Рекурсия является очень мощным инструментом программирования. Ее можно использовать для решения задач, которые более эффективно представляются рекурсией и могут использовать стек вызовов. Как правило, это относится к задачам сортировки, которые эффективно решаются с помощью рекурсии, обеспечивающей более высокую скорость, чем циклы.

В задаче, получившей название [**Ханойская башня**](http://www.skorks.com/2010/03/solving-the-towers-of-hanoi-mathematically-and-programmatically-the-value-of-recursion/), даны три стрежня и диски разного размера, которые в исходном состоянии надеты на первый стержень в виде башни. Задача состоит в том, чтобы перенести башню на другой стержень, при этом запрещается класть большой диск на маленький. Эту замечательную задачу можно легко решить с помощью рекурсии за 2n - 1 ходов, где n — число дисков.

Например, возьмем четыре диска и попытаемся перенести их со стержня A на стержень C, используя стержень B для временного хранения. С помощью описанной ниже рекурсивной функции это может быть выполнено за 15 ходов. Процесс решения можно визуализировать[этим апплетом](http://www.mazeworks.com/hanoi/). Функция вызывается (2n \* 2) – 1, или 31 раз. Причина, по которой число вызовов функции не равно числу ходов, кроется в том, что для обработки ходов необходимо установить стек вызовов. В этом примере используется головная рекурсия (листинг 4).

Листинг 4.

private static void solveTower(int num, int fromPeg, int toPeg,

int tempPeg) {

if (num > 0) {

// move a disc from the fromPeg to the tempPeg

solveTower(num - 1, fromPeg, tempPeg, toPeg);

System.out.println("Disc moved from " + fromPeg + " to " + toPeg);

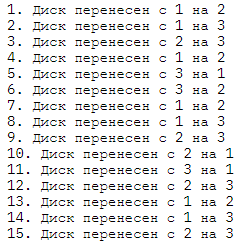
// move disc from the tempPeg to the toPeg

solveTower(num - 1, tempPeg, toPeg, fromPeg);

}

}

Результат для четырех дисков показан ниже



Ханойская башня — широко распространённая задача, которую можно решить с помощью рекурсии. Однако не все задачи столь очевидны, многие из них требуют времени на анализ и решение. Ханойскую башню можно решить и с помощью циклов, однако это потребует использования нескольких различных циклов и итераторов. Это решение будет работать дольше и потребует большего объема кода.

Ниже показан итерационный способ решения Ханойской башни. В этом примере использованы операции с битами, которые работают быстрей математических операций. За основу была взята [эта программа на языке C](http://www.waset.org/journals/ijmcs/v7/v7-1-2.pdf).

Листинг 6.

int numMoves = (1 << numDiscs) - 1;

int[] pegs = { 1, 2, 3, 1, 3, 2 };

int count = 0;

for (int currMove = 1; currMove <= numMoves; currMove++) {

int disc = 0;

while ((currMove >> disc & 1) == 0) {

disc++;

}

int level = (numDiscs - disc) & 1;

int fromPeg = (currMove >> ++disc) % 3;

fromPeg = pegs[fromPeg + (level \* 3)];

int toPeg = (fromPeg + level) % 3 + 1;

System.out.println(++count + ". Disc moved from " + fromPeg + " to " + toPeg);

}

**Заключение**

Хотя многим программистам привычнее работать с циклами, рекурсию также нельзя сбрасывать со счетов, особенно если она дает прирост производительности или позволяет выгодно использовать стек вызовов. Не всегда легко предсказать, какой метод даст лучшие результаты, поэтому обязательно нужно проводить тесты производительности. При использовании головной рекурсии также необходимо принимать во внимание размер стека.

***Что предпочтительнее цикл или рекурсия?***

На этот вопрос ответить непросто. Зачастую циклы дают лучшую производительность, чем рекурсивные вызовы, поскольку вызовы методов потребляют больше ресурсов, чем исполнение обычных операторов. В случае головной рекурсии стек вызовов разрастается, и его необходимо просматривать для получения конечного ответа. Тем не менее, это утверждение справедливо не всегда и зависит от типа решаемой задачи.

*Аргументы для ответа*

* рекурсия - это правильный способ обработки данных, например, в некоторых функциях сортировки, в бинарных деревьях и т.д. Для задач (например, обход вложенных каталогов), когда при этом у каждого имеется ряд своих отдельных переменных (например, количество файлов в данном каталоге), или асинхронных потоков, то поддерживать легче будет рекурсию. Да и рекурсия в данном случае будет удобнее, потому что обход одного каталога совсем не зависит от результатов обхода другого соседнего каталога, и они могут работать параллельно, независимо друг от друга. А затем в конце просто объединяют все свои результаты.
* рекурсия будет эффективна, если рекурсивная функция кешируемая, например, она запоминает результат и при следующем запросе просто возвращается кешированный вариант с циклом работать привычнее (проще), хотя кто-то владеет рекурсией лучше, чем циклом
* создание и уничтожение кадров стека более дорого, чем простой переход.
* вы всегда сможете усовершенствовать цикл и заставить его работать быстрее, чем рекурсивная функция, потому, что в цикле отсутствует постоянная настройка новых кадров стека
* в цикле решать задачи, где результат следующего полностью зависит от результата предыдущего (например, факториал)
* если мыслишь решение задачи как функциональную зависимость (пусть для того же факториала), то тебе поможет рекурсия. Она позволит отделить тебе одно вычисление от другого, которое опирается только на результат первого. Впрочем, если ты четко видишь, что именно следует делать с переменными предыдущей итерации, чтоб получить следующий результат, то цикл будет организовать проще.
* Используй то, что более удобно. Если реализация очевидна в терминах цикла, не следует использовать рекурсию. И наоборот.
* Если бы что-нибудь одно было предпочтительнее другого во всех случаях, этого другого бы не было.

Любой рекурсивный алгоритм может быть переписан без использования рекурсии. Заметим, что **быстродействие алгоритмов при избавлении от рекурсии, как правило, повышается. Еще одной причиной чтобы избавиться от рекурсии является ограничение на объем хранимых программой локальных переменных и значений параметров одновременно выполняющихся процедур.** При очень глубокой рекурсии этот объем возрастает, и программа перестает работать, выдавая ошибку «Stack overflow» (переполнение стека).

Так почему же люди продолжают пользоваться рекурсивными алгоритмами? Очевидно, потому что это проще и естественнее, чем соответствующие нерекурсивные решения. Тем не менее, знание о способах обойтись без рекурсии необходимо.

Ниже представлено несколько вариантов того, как это можно сделать.

### Явное использование стека

*Стеком* называется структура данных, в которой добавление и извлечение данных происходит с одного конца, называемого вершиной стека (рис. 13). Наглядным образом стека может служить стопка тарелок – добавлять или забрать тарелки можно только сверху. Каждая тарелка соответствует элементу данных.

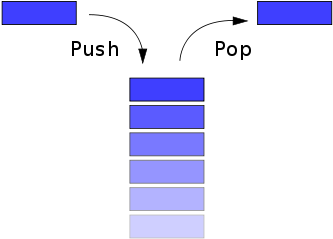


Рис. 13. Наглядное представление стека. Push (проталкивание) – традиционное название для операции добавления данных в стек, Pop (выталкивание) – традиционное название для операции извлечения данных из стека.

Когда одна процедура или функция вызывает другую, то параметры первой процедуры, а также место, с которого ее выполнение должно продолжиться после того как отработает вызванная процедура (точка возврата), запоминаются в так называемом *стеке вызовов*. Если вызванная процедура в свою очередь чего-нибудь вызывает, то ее параметры и точка возврата также добавляются в стек.

При рекурсивных вызовах стек вызовов хранит цепочку из данных об одновременно работающих процедурах. Во всех продвинутых средах разработки эту цепочку вместе с запомненными параметрами процедур можно просмотреть во время отладки. Соответствующая команда обычно называется “Call Stack” (в Delphi ей соответствует сочетание клавиш Ctrl – Alt – S).

**Универсальный способ избавиться от рекурсии – это самостоятельно запрограммировать те действия со стеком, которые фактически происходят, когда вы используете рекурсивные вызовы.** Покажем, как это можно сделать, на примере дважды вызывающей себя рекурсивной процедуры.

Для начала реализуем в виде класса стек, хранящий параметры процедуры:

type

//Запись для хранения параметров процедур

Parameters = record

//Список параметров

end;

//Стек удобно реализовать с помощью связанных списков

//(<http://www.tvd-home.ru/prog/16_4>)

PList = ^List;

List = record

Data: Parameters;

Next: PList;

end;

//Описанный одновсязанный список соединим с методами

//добавления и удаления элементов и получим стек.

Stack = class

private

StackTop: PList;

public

//Добавление данных

procedure Push(NewData: Parameters);

//Извлечение данных

function Pop: Parameters;

//Проверка наличия данных

function Empty: boolean;

end;

implementation

//Добавление данных

procedure Stack.Push(NewData: Parameters);

var

NewElement: PList;

begin

New(NewElement);

NewElement^.Data := NewData;

NewElement^.Next := StackTop;

StackTop := NewElement;

end;

//Извлечение данных

function Stack.Pop: Parameters;

var

PopedElement: PList;

begin

PopedElement := StackTop;

StackTop := StackTop^.Next;

Pop := PopedElement^.Data;

Dispose(PopedElement);

end;

//Проверка наличия данных

function Stack.Empty: boolean;

begin

Empty := StackTop = nil;

end;

Рассмотрим обобщенную рекурсивную процедуру с двумя вызовами самой себя.

procedure Recurs(P1: Parameters);

begin

DoSomething(P1);

if <условие> then

begin

P2 := F(P1);

Recurs(P2);

P3 := G(P1);

Recurs(P3);

end;

end;

В данной процедуре некоторые действия (DoSomething) выполняются много раз при разных значениях параметров. Нерекурсивный аналог должен хранить эти параметры в стеке. Каждый рекурсивный вызов будет соответствовать добавлению очередных параметров в стек. Вместо рекурсии появляется цикл, который выполняется, пока в стеке есть необработанные параметры.

procedure NonRecurs(P1: Parameters);

var

S: Stack;

P: Parameters;

begin

S := Stack.Create;

S.Push(P1);

while not S.Empty do

begin

P1 := S.Pop;

DoSomething(P1);

if <условие> then

begin

P3 := G(P1);

S.Push(P3);

P2 := F(P1);

S.Push(P2);

end;

end;

end;

Обратите внимание, что рекурсивные вызовы шли сначала для параметров P2, потом для P3. В нерекурсивной процедуре в стек отправляются сначала параметры P3, а только потом P2. Это связано с тем, что при рекурсивных вызовах в стек, по сути, отправляется недовыполненная часть процедуры, которая в нашем случае содержит вызов Recurs(P3).

Упомянутой выше перестановки можно избежать, если вместо стека использовать очередь – структуру данных, где добавление и извлечение элементов происходит с разных концов. Это будет некоторым отступлением от точной имитации процессов при рекурсивных вызовах. Однако в данном примере это кажется более удобным: каждый рекурсивный вызов будет прямо заменяться добавлением параметров в очередь.

### Запоминание последовательности рекурсивных вызовов

Как говорилось выше, рекурсивные вызовы образуют дерево, где каждый узел соответствует вызову одной процедуры. Последовательность выполнения этих процедур соответствует тому или иному алгоритму обхода узлов. Если требуется много раз обойти узлы одного и того же дерева, то можно один раз обойти их рекурсивно, запомнить количество и последовательность узлов, а затем, пользуясь этой информацией, обходить узлы уже нерекурсивно.

Например, обсуждалась задача вычисления арифметических выражений, заданных строкой. Может возникнуть ситуация, когда одно и то же выражение потребуется вычислить много раз при различных значениях переменной *x*. Синтаксическое дерево, которое требуется обходить при таких вычислениях, не зависит от *x*. Можно обойти его один раз, построив при этом массив, где каждый элемент будет соответствовать узлу дерева, а их последовательность – порядку обхода. Повторные вычисления при новом *x* потребуют только нерекурсивного перебора элементов массива.

Еще один пример такого запоминания в задаче о вычислении значений многомерных полиномов смотрите тут: [**http://tvd-home.ru/numerical/polynom**](http://tvd-home.ru/numerical/polynom).

Такой подход не избавляет нас от рекурсии полностью. Однако он позволяет ограничиться только одним обращением к рекурсивной процедуре, что может быть достаточно, если мотивом является забота о максимальной производительности.

### Определение узла дерева по его номеру

Идея данного подхода в том, чтобы заменить рекурсивные вызовы простым циклом, который выполнится столько раз, сколько узлов в дереве, образованном рекурсивными процедурами. Что именно будет делаться на каждом шаге, следует определить по номеру шага. Сопоставить номер шага и необходимые действия – задача не тривиальная и в каждом случае ее придется решать отдельно.

Например, пусть требуется выполнить k вложенных циклов по n шагов в каждом:

for i1 := 0 to n-1 do

for i2 := 0 to n-1 do

for i3 := 0 to n-1 do

…

for **(**int **i1 = 0; i1 <= n-1; i1++)**

for **(**int **i2 = 0; i2 <= n-1; i2++)**

for **(**int **i3 = 0; i3 <= n-1; i3++)**

**…**

Если k заранее неизвестно, то написать их явным образом, как показано выше невозможно. Используя прием, продемонстрированный в разделе 6.5 можно получить требуемое количество вложенных циклов с помощью рекурсивной процедуры:

procedure NestedCycles(Indexes: array of integer; n, k, depth: integer);

var

i: integer;

begin

if depth <= k then

for i:=0 to n-1 do

begin

Indexes[depth] := i;

NestedCycles(Indexes, n, k, depth + 1);

end

else

DoSomething(Indexes);

end;

void **NestedCycles(**intarray, int **Indexes,** int **n,** int **k,** int **depth)**

{

int **i;**

if(depth <= k)

for(i=0; i<-=n-1; i++){

**Indexes[depth] = i;**

**NestedCycles(Indexes, n, k, depth + 1);**

**}**

**DoSomething(Indexes);**

}

Чтобы избавиться от рекурсии и свести все к одному циклу, обратим внимание, что если нумеровать шаги в системе счисления с основанием n, то каждый шаг имеет номер, состоящий из цифр i1, i2, i3, … или соответствующих значений из массива Indexes. То есть цифры соответствуют значениям счетчиков циклов. Номер шага в обычной десятичной системе счисления:

i=i_1 n^{k-1}+i_2 n^{k-2}+\ldots+i_k~~~~~(9)

Всего шагов будет nk. Перебрав их номера в десятичной системе счисления и переведя каждый из них в систему с основанием n, получим значения индексов:

M := round(IntPower(n, k));

for i := 0 to M-1 do

begin

Number := i;

for p := 0 to k-1 do

begin

Indexes[k – p] := Number mod n;

Number := Number div n;

end;

DoSomething(Indexes);

end;

M = round(IntPower(n, k));

for(i=0; i<-=M-1; i++){

**Number = i;**

for(p=0; i<-=k-1; p++){

**Indexes[k – p] = Number % n;**

**Number = Number / n;**

**}**

**DoSomething(Indexes);**

}

Еще раз отметим, что метод не универсален и под каждую задачу придется придумывать что-то свое.

Еще один замечательный пример - вычисление по номеру шага перекладываний в задаче о Ханойских башнях смотрите тут: [**http://algolist.manual.ru/maths/combinat/hanoi.php**](http://algolist.manual.ru/maths/combinat/hanoi.php)

## Контрольные вопросы

1. Определите, что сделают приведенные ниже рекурсивные процедуры и функции.

(а) Что напечатает приведенная ниже процедура при вызове Rec(4)?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | procedure Rec(a: integer);  begin  writeln(a);  if a>0 then  Rec(a-1);  writeln(a);  end; |

(б) Чему будет равно значение функции Nod(78, 26)?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | function Nod(a, b: integer): integer;  begin  if a > b then  Nod := Nod(a – b, b)  else  if b > a then  Nod := Nod(a, b – a)  else  Nod := a;  end; |

(в) Что будет напечатано приведенными ниже процедурами при вызове A(1)?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | procedure A(n: integer);  procedure B(n: integer);  procedure A(n: integer);  begin  writeln(n);  B(n-1);  end;  procedure B(n: integer);  begin  writeln(n);  if n < 5 then  A(n+2);  end; |

(г) Что напечатает нижеприведенная процедура при вызове BT(0, 1, 3)?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | procedure BT(x: real; D, MaxD: integer);  begin  if D = MaxD then  writeln(x)  else  begin  BT(x – 1, D + 1, MaxD);  BT(x + 1, D + 1, MaxD);  end;  end; |

2. Уроборос – змей, пожирающий собственный хвост (рис. 14) в развернутом виде имеет длину L, диаметр около головы D, толщину брюшной стенки d. Определите, сколько хвоста он сможет в себя впихнуть и в сколько слоев после этого будет уложен хвост?

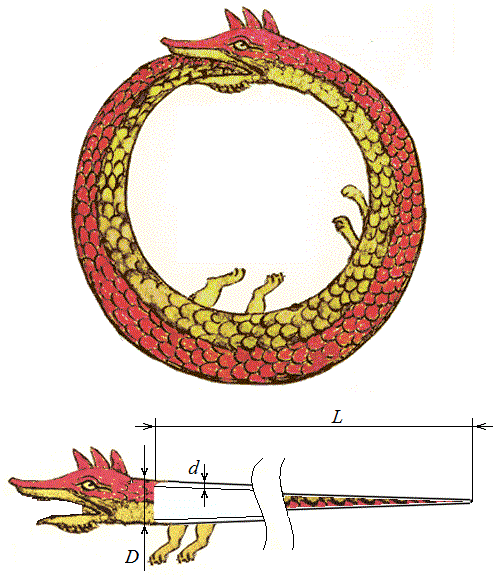


Рис. 14. Развернутый уроборос.

3. Для дерева на рис. 10а укажите последовательности посещения узлов при прямом, обратном и концевом порядке обхода.

4. Изобразите графически дерево, заданное с помощью вложенных скобок: (A(B(C, D), E), F, G).

5. Изобразите графически синтаксическое дерево для следующего арифметического выражения:

2x(x-1)+1/x

Запишите это выражение в обратной польской записи.

6. Для приведенного ниже графа (рис. 15) запишите матрицу смежности и матрицу инцидентности.

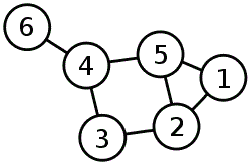


Рис. 15.

## Задачи

1. Вычислив факториал достаточно большое количество раз (миллион или больше), сравните эффективность рекурсивного и итерационного алгоритмов. Во сколько раз будет отличаться время выполнения и как это отношение будет зависеть от числа, факториал которого рассчитывается?

2. Напишите рекурсивную функцию, проверяющую правильность расстановки скобок в строке. При правильной расстановке выполняются условия:

(а) количество открывающих и закрывающих скобок равно.  
 (б) внутри любой пары открывающая – соответствующая закрывающая скобка, скобки расставлены правильно.

Примеры неправильной расстановки: )(, ())(, ())(() и т.п.

3. В строке могут присутствовать скобки как круглые, так и квадратные скобки. Каждой открывающей скобке соответствует закрывающая того же типа (круглой – круглая, квадратной- квадратная). Напишите рекурсивную функцию, проверяющую правильность расстановки скобок в этом случае.

Пример неправильной расстановки: ( [ ) ].

4. Число правильных скобочных структур длины 6 равно 5: ()()(), (())(), ()(()), ((())), (()()).  
Напишите рекурсивную программу генерации всех правильных скобочных структур длины 2n.

Указание: Правильная скобочная структура минимальной длины «()». Структуры большей длины получаются из структур меньшей длины, двумя способами:

(а) если меньшую структуру взять в скобки,  
 (б) если две меньших структуры записать последовательно.

5. Создайте процедуру, печатающую все возможные перестановки для целых чисел от 1 до N.

6. Создайте процедуру, печатающую все подмножества множества {1, 2, …, N}.

7. Создайте процедуру, печатающую все возможные представления натурального числа N в виде суммы других натуральных чисел.

8. Создайте функцию, подсчитывающую сумму элементов массива по следующему алгоритму: массив делится пополам, подсчитываются и складываются суммы элементов в каждой половине. Сумма элементов в половине массива подсчитывается по тому же алгоритму, то есть снова путем деления пополам. Деления происходят, пока в получившихся кусках массива не окажется по одному элементу и вычисление суммы, соответственно, не станет тривиальным.

Замечание: Данный алгоритм является альтернативой [приему накопления суммы](http://www.tvd-home.ru/prog/3_2). В случае вещественнозначных массивов он, обычно, позволяет получать меньшие погрешности округления.

9. Запрограммируйте быстрые методы сортировки массивов, описанные в разделе 6.4.

10. Создайте процедуру, рисующую кривую Коха (рис. 12).

11. Воспроизведите рис. 16. На рисунке на каждой следующей итерации окружности в 2.5 раза меньше (этот коэффициент можно сделать параметром).

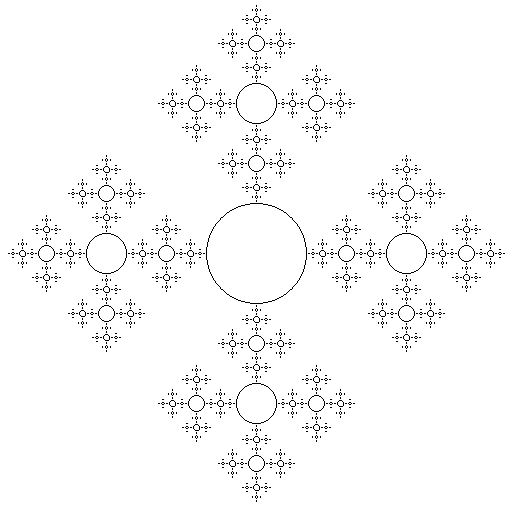


Рис. 16.

## Литература

1. Д. Кнут. Искусство программирования на ЭВМ. т. 1. (раздел 2.3. «Деревья»).  
2. Н. Вирт. Алгоритмы и структуры данных.

## Другие материалы на этом сайте

Близкие разделы [учебника по программированию](http://tvd-home.ru/prog):

[Рекуррентные соотношения](http://tvd-home.ru/prog/4_1)  
 [Рекурсивные структуры данных](http://tvd-home.ru/prog/16_4)

[Вычисление полиномов от нескольких переменных](http://tvd-home.ru/numerical/polynom) - еще один пример рекурсивного алгоритма.