Конспект лекций
по курсу

Алгоритмы и алгоритмические языки

Доцент каф. Вычислительной математики
Механико-Математического ф-та
МГУ им. М.В.Ломоносова
Староверов В.М.
2018г.

Лекция 1
Представление чисел в ЭВМ

Целые

Беззнаковые целые.

Используется двоичное представление чисел.

$\mathbf{x} = a_{n-1}*2^{n-1} + a_{n-2}*2^{n-2} + \dots a_1*2^1 + a_0*2^0$

здесь каждый член a_i представляет собой 0 или 1 или, иначе говоря, одну цифру в двоичном представлении данных.

n задается количеством бит (в одном бите хранится одна двоичная цифра), отводящихся под данное число.

Принято считать, что в представлении байта старший бит всегда находится слева.

Внутри целого числа байты могут располагаться по-разному. Существует два основных правила расположения байт внутри целого числа:

- Big-endian - от старшего байта к младшему (IBM-формат)
- Little-endian - от младшего байта к старшему (Intel-формат)

В первом случае удобством является сквозная нумерация байт. Во втором – если взять небольшое целое число, например, в 4-байтовом целом и обрезать его до первых двух байт, то мы получим то же самое значение числа.

Целые со знаком.

Существует три основных формата:

- прямой (старший бит служит признаком знака: 1='-'; 0='+')
- обратный (для представления отрицательного числа берется представление его модуля, а потом все биты инвертируются)
- дополнительный (для представления отрицательного числа берется представление его модуля, все биты инвертируются, к результату прибавляется 1)

Реально, в основном, используется только дополнительный код.

Утверждение 1. *Представление чисел в дополнительном коде эквивалентно представлению чисел в кольце вычетов по модулю 2^n, где n – количество бит в двоичном представлении числа.*

Для доказательства Утверждения достаточно доказать, что вышеприведенное определение отрицательного числа в дополнительном коде эквивалентно следующему: пусть $x>0$, то $-x$ получается с помощью операции 2^n-x. В исходном определении число $-x$ получается из $x>0$ следующим образом: сначала производится инверсия числа, что эквивалентно операции 2^n-x-1. Это следует из того, что $(2^n-x-1)+x=2^n-1$=числу, состоящему из n единиц, а это возможно только если на месте единичных бит числа 2^n-x-1 стоят нули в представлении числа x и

наоборот (на месте нулевых бит числа 2^n-x-1 стоят единицы в представлении числа x). Осталось прибавить 1 и мы получим требуемые 2^n-x.

Следствие. *Операции сложения, вычитания и умножения корректно производятся независимо от способа интерпретации целых чисел – дополнительного кода (для знакового целого) или беззнакового целого.*

Вещественные

Вещественные числа с фиксированной запятой.

VB тип *Currency* - восьмибайтовая переменная, которая содержит вещественное число в формате с фиксированной десятичной запятой (четыре десятичных знака после запятой).

C# тип *decimal*

Вещественные числа с плавающей запятой.

Формат закреплен IEEE: Institute of Electrical and Electronics Engineers.
Число представляется в виде

$\mathbf{x} = s * m * 2^d$

где
s – знак (отводится один бит)
m – мантисса в виде 1.xxxxx (где x – двоичная цифра; 1 не хранится)
d – степень (оставшиеся биты)

Согласно формату IEEE вместо степени хранится характеристика = $d-d_{min}$, где d_{min} – минимально возможное значение мантиссы. Таким образом, минимально возможное значение характеристики = 0.
На самом деле, по стандарту IEEE представление более сложное. Пока характеристика положительна поддерживается вышеуказанное представление. Для нулевой характеристики число хранится уже как число с фиксированное запятой в виде

$\mathbf{x} = p * 2^{-dmin}$
где p – число с фиксированной точкой в виде x.xxxxx (где x – двоичная цифра; хранятся все указанные биты!).

Таким образом, отрицательные степени двойки будут иметь битовое представление (степень выделена пробелами):

2^{-122}: 1.88079e-037: 0 00000101 00000000000000000000000
2^{-123}: 9.40395e-038: 0 00000100 00000000000000000000000
2^{-124}: 4.70198e-038: 0 00000011 00000000000000000000000
2^{-125}: 2.35099e-038: 0 00000010 00000000000000000000000
2^{-126}: 1.17549e-038: 0 00000001 00000000000000000000000
2^{-127}: 5.87747e-039: 0 00000000 10000000000000000000000
2^{-128}: 2.93874e-039: 0 00000000 01000000000000000000000
2^{-129}: 1.46937e-039: 0 00000000 00100000000000000000000
2^{-130}: 7.34684e-040: 0 00000000 00010000000000000000000
2^{-131}: 3.67342e-040: 0 00000000 00001000000000000000000
2^{-132}: 1.83671e-040: 0 00000000 00000100000000000000000
2^{-133}: 9.18355e-041: 0 00000000 00000010000000000000000
2^{-134}: 4.59177e-041: 0 00000000 00000001000000000000000
2^{-135}: 2.29589e-041: 0 00000000 00000000100000000000000
2^{-136}: 1.14794e-041: 0 00000000 00000000010000000000000
…
2^{-148}: 5.73972e-042: 0 00000000 00000000000000000000010
2^{-149}: 2.86986e-042: 0 00000000 00000000000000000000001
2^{-150}: 1.43493e-042: 0 00000000 00000000000000000000000

Здесь следует отметить, что в Intel-архитектуре байты переставлены в обратном порядке и в вышеописанном представлении байты выводятся в порядке: 3, 2, 1, 0.

Т.о. число 2^{-150} уже неотличимо от 0.Если вышеуказанные число получались каждый раз путем деления предыдущего числа на 2, то в результате получения последнего число из 2^{-149} мы получили ситуацию *underflow* – нижнее переполнение. Как правило, процессоры могут отслеживать underflow, но нижнее переполнение можно расценивать как ошибку, а можно и не расценивать, ведь при многих вычислениях слишком маленькие числа можно расценивать как нулевые. Поэтому, обычно никаких сообщений в ситуации underflow не производится.

Переполнение при увеличении чисел называется *overflow*. При увеличении чисел с плавающей точкой степень увеличивается до максимально возможной. Когда степень достигает максимально возможного значения, то подобное число уже числом не считается, независимо от значения бит мантиссы:

2^{118}: 3.32307e+035: 0 11110101 00000000000000000000000
2^{119}: 6.64614e+035: 0 11110110 00000000000000000000000
2^{120}: 1.32923e+036: 0 11110111 00000000000000000000000
2^{121}: 2.65846e+036: 0 11111000 00000000000000000000000
2^{122}: 5.31691e+036: 0 11111001 00000000000000000000000
2^{123}: 1.06338e+037: 0 11111010 00000000000000000000000

2^{124}: 2.12676e+037: 0 11111011 00000000000000000000000
2^{125}: 4.25353e+037: 0 11111100 00000000000000000000000
2^{126}: 8.50706e+037: 0 11111101 00000000000000000000000
2^{127}: 1.70141e+038: 0 11111110 00000000000000000000000
2^{128}: 1.#INF : 0 11111111 00000000000000000000000
2^{129}: 1.#INF : 0 11111111 00000000000000000000000
2^{130}: 1.#INF : 0 11111111 00000000000000000000000

Будем далее называть представление вещественных чисел с плавающей точкой в диапазоне $[2^{-dmin}, 2^{dmin}]$ *стандартным представлением вещественных чисел с плавающей точкой.*

Итак, стандартом IEEE закреплено наличие дополнительных констант:

NAN – not a number. Не число = 0/0 [(0111 1111)(1100 0000) 0…0]
+INF – плюс бесконечность [(0111 1111)(1000 0000) 0…0]
-INF – минус бесконечность [(1111 1111)(1000 0000) 0…0]
+0 = 0x00 00 00 00
-0 = 0x80 00 00 00

+INF, -INF,+0, -0 можно корректно сравнивать. +0 = = -0 = = 0.
NAN при сравнении возвращает *ложь* всегда кроме !=. В этом случае возвращается *истина*.

Для IBM PC:

float (32bit)
1bit – знак
8bits – степень+127 ($127=2^7-1$)
23bits – символы xxxxx из мантиссы (мантисса в виде 1.xxxxxx)

пример 1.f=

0 = знак
01111111 = степень+127 (занимает 8 бит)
0000…0 = символы xxxxx из мантиссы (мантисса в виде 1.xxxxxx)
Итого: 00111111 10000000 00000000 00000000
Осталось заметить, что байты идут в обратном порядке (также и в последующих примерах).

doudle (64bit)
1bit – знак
11bits – степень+2^{10}-1
52bits – символы xxxxx из мантиссы (мантисса в виде 1.xxxxxx)

пример 1.f=
00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000

long double (80bit)
1bit – знак
15bits – степень+2^{14}-1
64bits – символы Xxxxxx из мантиссы (мантисса в виде X.xxxxxx)

В некоторых версиях языка C тип **long double** присутствует, но является всего лишь синонимом типа **double** (например, в текущей версии Microsoft C это именно так). В gcc разрядность типа **long double** зависит от ключей компилятора и, конечно же, от архитектуры. По умолчанию в 64-битных системах используется 128-битная реализация **long double** (64 бита мантиссы, 15 бит степени, 1 бит знака, остальные младшие 6 байт биты не используются). Отметим, что данная реализация удовлетворяет IEEE стандарту лишь частично (поведение вблизи нуля аналогично стандартному, но старшая единичка мантиссы в представлении числа присутствует). На самом деле, легко увидеть, что данная реализация основана на 80-битной реализации вещественных чисел с дополнением пустых байт для правильной четности. Разрядность (здесь надо понимать, что 128-битная и 80-битная реализации **long double** отличаются, фактически, только четностью выравнивания) типа **long double** зависит от ключей компилятора *–mlong-double-64, –mlong-double-80, –mlong-double-128*. Ясно, что использование 80-битного представления вещественных чисел бывает редко оправданным. Примером, когда это имеет смысл, служит блестящая задача из online-олимпиады Физтеха: Вычислить $log_x (x^4-64x+4)$ если известно: $x^9-4x^5+16x-1=0$. Задача достаточно просто делается с помощью чистой математики (на что она и рассчитана), однако если попытаться ее сделать с помощью компьютера, то выяснится, что точности **double** для ее решения недостаточно. Т.е. в компиляторе Microsoft получить решение задачи не получится. С помощью некоторой возни решение задачи можно получить на gcc, используя **long double**. Отмечу, что задача была первой в олимпиаде и с помощью нее великолепно отсеивались люди, которые пытались получить решения задач на компьютере. В качестве конкретного наезда ☺ можно отметить, что в Ломоносовской олимпиаде подобных задач до сих пор не было. Надо всегда иметь в виду, что выше описан стандарт представления вещественных чисел с плавающей точкой, который поддерживается на архитектуре используемых персональных компьютеров (Intel/AMD), но

существуют и другие (нестандартные) реализации вещественных чисел с плавающей точкой.
Например, **IBM-формат** вещественных чисел с плавающей точкой предполагает, что число представляется в виде: **x** = s * m * 16^d. Подобный подход расширяет диапазон возможных значений числа, но уменьшает точность представления чисел. Кроме того, данный подход требует хранения всех бит мантиссы (в стандартном представлении старший бит всегда равен 1 и поэтому он не хранится в машинном представлении числа). Числа в данном формате не меняют своего представления для очень маленьких и очень больших по модулю чисел, что упрощает реализацию алгоритмов работы с такими числами. При этом нет проблем с представлением нуля: нулевая мантисса всегда соответствует числу, равному нулю. Данный формат представления чисел до сих пор используется, например, при представлении сейсмических данных в геологической разведке.
Даже в Intel-архитектуре представления вещественных чисел могут присутствовать отклонения от стандарта. Так, 80-битные вещественные числа обычно используются для представления вещественных чисел в родном Intel-формате чисел, с которыми, собственно, работает процессор (тут следует отметить, что в зависимости от установок процессора 80-битные вещественные числа, вообще говоря, могут интерпретироваться процессором по-разному). В этом формате старший бит мантиссы может не изыматься из машинного представления вещественного числа, что несколько уменьшает возможную точность представления вещественного число, в обмен на более простые алгоритмы работы с такими числами.

BCD - Binary Coded Decimal.
Различают упакованный формат BCD (одна десятичная цифр хранится в 4 битах байта, т.е. в одном байте хранится две десятичные цифры) и неупакованный BCD (по одной десятичной цифре в каждом байте). Применяются, как правило, в приложениях, в которых требуются операции, обеспечивающие большую точность чисел. Также данный формат используется в телефонии. Отметим, что неупакованный BCD формат весьма удобен для программной реализации арифметических операций.

Формат NUMBER из СУБД Oracle.
Данный формат похож на упакованный BCD формат. В нем в одном байте хранятся две цифры десятичного представления в т.н. *стоичной системе счисления*.
Число в данном формате хранится как число с плавающей точкой, но его длина не фиксирована.
Данный формат обладает несомненным преимуществом – возможностью точного представления в ЭВМ дробных десятичных чисел. Другим преимуществом данного формата является возможность хранения (передачи) чисел в таком формате в текстовой строке, в которой конец строки помечается символом с нулевым кодом (во внутреннем представлении чисел в формате NUMBER нет байт с нулевым значением). Наконец, числа в данном представлении можно сравнивать лексикографически (как обычные строки).
Основной недостаток – использование программной реализации для работы с числами в данном формате, т.е. отсутствие поддержки данного формата на уровне процессора. Такой формат является весьма удобным при полном отсутствии встроенных в железо операций с вещественными числами, но при их наличии для других форматов формат NUMBER им сильно проигрывает.
Разберем подробно формат представления числа в формате NUMBER из СУБД Oracle. Число в таком формате может занимать различное количество байт, в зависимости от величины числа, поэтому во внутреннем представлении числа сначала хранится байт с длиной числа в байтах, а потом само число. Далее разберем формат хранения самого числа (без байта с его длиной).
Ноль хранится в виде одного байта 0x80.
Для хранения любого другого числа x оно представляется в виде $x=s*y*100^n$, где $s=1$ *или* -1, $1 \le y < 100$ называется *мантиссой*, n – целая *степень* числа. Все незначащие нули в конце y отбрасываются. Например: $1234.56=1*12.3456*100^1$, $1000.=1*10.*100^1$. Если количество цифр до/после точки нечетное, то дополним цифры нулем до/после числа. Например: $234.5=1*02.3450*100^1$. Будем далее работать с этим *нормализованным* представлением числа.
В начале представления числа хранится байт со степенью в виде: если x положительно, то байт имеет вид $n+65+128=n+0x41+0x80=(n+0x41)|0x80=n+0xC1$ т.е. берется $n+65$ и в старший бит прописывается 1. Если x отрицательно, то делается все то же самое, но потом число еще инвертируется. Т.о. старший бит первого байта числа будет хранить знак числа: если бит = 1, то число положительно, иначе – отрицательно.
Отметим, что нулевая степень для положительного числа хранится как 65+128=193=1100 0001b, а для отрицательно – как ~1100 0001b=0011 1110b=62, и от этих чисел (193 и 62) можно вести отсчет при расчете внутреннего представления степени.

Например, для $x>0, n=1$ имеем внутреннее представление степени 1+65+128=1+193=194. Для $x<0, n=1$ имеем внутреннее представление степени ~(1+65+128)=~(1+193)=~194=~1100 0010b=0011 1101b=61=62-1. Для $x<0, n=-2$ имеем внутреннее представление степени 62+2=64. И т.д.

Отметим, что если у двух чисел представления степеней не равны, то большему представлению степени соответствует большее число, что упрощает процесс сравнения таких чисел.

Далее каждая пара цифр (назовем число из этих двух цифр t) мантиссы нормализованного представления числа записывается в соответствующий байт представления числа в виде $t+1$ для положительного x и в виде $101-t$ для отрицательного x. Для отрицательного x в конец представления числа

дописывается байт 102. Утверждается, что это облегчает процесс сравнения отрицательных чисел.

Отметим, что если у двух чисел представления степеней равны, то сравнивать такие числа можно последовательно сравнивая байты мантиссы.

Приведем примеры представления чисел в формате NUMBER (первые два числа=два байта – длина и представление степени; далее две цифры мантиссы = один байт).

110=01.10*100¹: 3 194 02 11

1100=11.*100¹: 2 194 12 (заметим, что число больше предыдущего!)

1101=11.01*100¹: 3 194 12 02

-1101=-11.01*100¹: 4 61 90 100 102

-1=-01.*100⁰ : 3 62 100 102

-1.01=-01.01*100⁰: 4 62 100 100 102

Здесь интересно заметить, что при сравнении двух последних чисел их можно сравнивать с помощью последовательного сравнения байт представления числа, начиная с мантиссы. И это будет корректно!

Легко увидеть, что если из каждого байта представления мантиссы числа вычесть 1, то получится число, очень сильно напоминающее число в дополнительном коде в 100-ичной системе счисления. Приведем примеры алгоритма сложения чисел в формате NUMBER с одинаковыми степенями и с вычтенной 1 из каждого байта.

1)Положительное+положительное. Выполняется обычное сложение байт столбиком с переносом переполнения в следующий байт. Пример (первый столбец: слагаемые и сумма в десятичной системе; столбцы далее: представление байт слагаемых и суммы в формате NUMBER с вычтенной 1).

```
 8989     89  89
 1212     12 12
1 02  01
10201
```

2)Отрицательное+положительное. Выполняется обычное сложение байт столбиком с переносом переполнения в следующий байт. Знак результата определяется по значению переполнения при сложении старших байт: 0 = минус, 1 = плюс.

Если результат положительный, то изо всех байт, кроме младшего, вычитается 1. Примеры.

```
-1111     89  89
 1110     11 10
      0 100 99
   -1
```

```
-111211       89   88  89
 101112        10   11  12
             0  99 100  01
  -10099

-1111   89  89
 1212   12 12
      1   2 01
  101

-111160       89 89  40
 121212        12 12 12
             1 02 01  52
10052

-1              100 100 99
 121212          12   12 12
             1   13   13 11
121211
```

3)Отрицательное+ Отрицательное. Выполняется обычное сложение байт столбиком с переносом переполнения в следующий байт. Изо всех байт, включая байт переполнения, кроме младшего вычитается 1 (если байт переполнения =0, то в него записывается 99).

Примеры.

```
-1111          89 89
-1111          89 89
             1 79 78
-2222

 -6666         34 34
 -6666         34 34
             0 68 68
-13332
```

Все эти операции можно интерпретировать по-другому. На самом деле, аналогом инверсии в 100-ичной системе является операция *99-t* для каждого байта *t*. Тогда операция перевода положительного числа *x* в число *–x* будет в 100-ичной системе выглядеть следующим образом: 1)для всех байт делаем инверсию (операцию *99-t*); 2)к числу прибавляем 1. Или, что то же самое: 1)из числа вычитаем 1; 2) для всех байт делаем инверсию (операцию *99-t*).

3)Для байта степени делаем побитовую инверсию (эквивалентно $t=255-t$)

Ошибки вычислений

Для экономии времени мы не будем давать точного определения понятий `много больше' (>>) и `много меньше' (<<).

Определение 1. $e_1 = \mathbf{argmin\{v>0|1+v != v\}}$
Определение 2. $e_2 = \mathbf{argmax\{v>0|1+v = = v\}}$

Легко видеть, что e_1 =[(0011 1111) (1 000 0000)(0000 0000)(0000 0001)]-1=2^{-23} и что e_2 отличается от него на очень мало, поэтому можно говорить об одном числе $e = e_1$.

Определение 3. Назовем абсолютной ошибкой приближения числа $\mathbf{x_0}$ с помощью числа **x** такое D, что
$| \mathbf{x} - \mathbf{x_0}| < D$

Определение 4. Назовем относительной ошибкой приближения числа $\mathbf{x_0}$ с помощью числа **x** такое d, что
$| \mathbf{x} - \mathbf{x_0}| / | \mathbf{x_0} | < d$

Часто более удобно пользоваться другим определением числа d:
$| \mathbf{x} - \mathbf{x_0}| / | \mathbf{x} | < d$
Это связано с тем, что, как правило, нам известно лишь приближение исследуемого числа (**x**), а не оно само ($\mathbf{x_0}$). В ситуации, когда $| \mathbf{x} - \mathbf{x_0}| << | \mathbf{x_0} |$ и $| \mathbf{x} - \mathbf{x_0}| << | \mathbf{x} |$ эти два Определения отличаются несущественно. Кроме данных допущений, мы также будем предполагать, что все относительные ошибки много меньше 1.

Утверждение 2. *При сложении/вычитании чисел их абсолютные ошибки складываются, т.е.*
$| \mathbf{(x+y) - (x_0+y_0)} | \ £ D_x+D_y$

$| \mathbf{(x - y) - (x_0 - y_0)} | \ £ D_x+D_y$
где $| \mathbf{x - x_0} | < D_x$, $| \mathbf{y - y_0} | < D_y$.

Доказательство элементарно.

Утверждение 3. *При умножении/делении чисел их относительные ошибки складываются (с точностью до пренебрежимо малых членов), т.е.*
$| \mathbf{(x*y - x_0*y_0)} / \mathbf{(x*y)}| \ £» d_x+d_y$

$| \mathbf{(x/y - x_0/y_0)} / \mathbf{(x/y)}| \ £» d_x+d_y$

где $| \mathbf{x - x_0}| / | \mathbf{x} | < d_x$, $| \mathbf{y - y_0} | / | \mathbf{y} | < d_y$.

Доказательство.
1.
Итак, рассмотрим сумму относительных ошибок:

$$\mathbf{(x - x_0) / x + (y - y_0) / y = (x* y - x_0* y + y* x - x* y_0) / (x* y) =}$$

$$\mathbf{= (x* y - x_0* y + y* x - x* y_0 - x* y + x_0* y_0) / (x* y) + (x* y + x_0* y_0) / (x* y)}$$
$$= d_x * d_y + \mathbf{(x* y - x_0* y_0) / (x* y)}$$

получаем:

$$|\mathbf{(x* y - x_0* y_0) / (x* y)}| \ £» d_x + d_y$$

2.

Перепишем относительную ошибку частного:
$$\mathbf{(x/y - x_0/y_0) / (x/y) = (x* y_0 - x_0 y) / (x*y_0)}$$

Итак, рассмотрим сумму относительных ошибок (незначительно отличающихся от основного определения):

$$\mathbf{(x - x_0) / x + (y_0 - y) / y_0 = (x* y_0 - x_0* y_0 + y_0* x - x* y) / (x* y_0) =}$$

$$\mathbf{= (x* y_0 - x_0* y_0 + y_0* x - x* y - x* y_0 + x_0 y) / (x* y_0) + (x* y_0 - x_0 y) / (x*y_0)}$$
$$»= d_x * d_y + \mathbf{(x* y_0 - x_0 y) / (x*y_0)}$$

получаем:
$| \mathbf{(x/y - x_0/y_0)} / \mathbf{(x/y)}| \ £» d_x+d_y$
Ч.Т.Д.

Легко видеть, что число e представляет собой абсолютную ошибку представления числа 1. Здесь имеется в виду, что все числа, отличающиеся от 1 менее, чем на e, будут равны 1. Отсюда сразу же вытекает, что e также является и относительной погрешностью представления числа 1 в ЭВМ.
Из представления вещественного числа
$\mathbf{x} = s * m * 2^d$ ($1 \le m < 2$)

сразу вытекает, что значение вещественного числа по порядку задается только его степенью. Здесь имеется в виду, что при одной и той же степени, изменение мантиссы числа может изменить значение числа не более, чем вдвое:

$$1 \le \mathbf{x} / (s * 2^d) < 2 \text{ (где } \mathbf{x} = s * m * 2^d)$$

т.е.

$\mathbf{x} = s * m * 2^d \Rightarrow s * 2^d$

Но легко видеть, число $\mathbf{x} = s * m * 2^d$ имеет абсолютную погрешность представления, равную $e * 2^d$, т.к. эта величина равна значению последнего бита мантиссы при заданном значении степени. Отсюда сразу вытекает, что относительная ошибка представления числа $\mathbf{x} = s * m * 2^d$, равная

$$e * 2^d / (m * 2^d) = e / m$$

не более, чем вдвое отличается от e. Т.о. мы получили следующую важную теорему

Теорема. *Любое вещественное число **x** в стандартном представлении числа с плавающей точкой (т.е. не являющееся слишком большим или слишком маленьким по модулю в вышеописанном смысле) имеет относительную ошибку представления порядка* e *(точнее, не более, чем вдвое отличающуюся от* e*) и имеет абсолютную ошибку представления порядка **x***e (точнее, не более, чем вдвое отличающуюся от **x***e).*

Данная теорема и вышеприведенные утверждения дают нам возможность явно выписывать абсолютные и относительные ошибки, возникающие при сложении/ вычитании/умножении/делении вещественных чисел с плавающей точкой в стандартном представлении.

Лекция 2

Алгоритмы. Сведение алгоритмов. Нижние и верхние оценки.

Д.Кнут. Искусство программирования для ЭВМ. тт 1-3. Москва. Мир. 1996-1998
Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы. Построение и анализ. Москва. МЦНМО. 1999.
Препарата Ф., Шеймос М. Вычислительная геометрия. Москва. Мир. 1989

Мы не будем давать строгого определения понятия *алгоритма*, однако основные свойства понятия *алгоритма*, задаваемые при его определении, мы все же опишем.

*Алгоритмом **m*** называется формально описанная процедура, имеющая некоторый набор *входных данных **In(m)*** и *выходных данных **Out(m)***. Вводится некоторый параметр, оценивающий объем входных данных ***N=N(In(m)).*** Будем называть этот параметр *размером* входных данных**.** Заметим, что, на самом деле, алгоритм решает некоторую формально поставленную задачу *z*, имеющую те же самые входные и выходные данные.

При создании алгоритма закрепляется некоторый конечный набор *допустимых операций*, в терминах которых формулируется алгоритм. От алгоритма требуется его *конечность*, т.е. мы говорим, что задача решается с помощью некоторого алгоритма, только если для каждого набора входных данных соответствующий набор выходных данных получается с помощью заданного алгоритма за конечное количество допустимых операций.

Для оценки времени работы алгоритма с каждой допустимой операцией ассоциируется *время ее выполнения*. *Временем выполнения реализации алгоритма **T(m)*** для определенных начальных данных называется сумма времен выполнения всех операций алгоритма при выполнении данной реализации.

Верхней оценкой времени выполнения алгоритма ***m*** называется такая функция $F(N)$, что для любого набора входных данных ***In(m)*** размером не более ***N*** время выполнения алгоритма не будет превосходить $F(N)$. Будем говорить, что задача ***z*** *имеет верхнюю оценку времени решения* $F(N)$**,** если существует алгоритм ***m*** с верхней оценкой времени выполнения $F(N)$**.**

Разумно задаться вопросом: а для любого ли алгоритма существует его верхняя оценка времени работы? Элементарно привести пример, для которого верхней оценки времени работы не существует (например, можно рассмотреть задачу выписывания всех десятичных знаков обычного целого числа). Однако, если количество различных вариантов входных данных объема *N* алгоритма для любого *N* конечно, то легко показать, что верхняя оценка времени работы алгоритма существует. Нас интересуют алгоритмы, которые можно реализовать на компьютере. Но у компьютера конечное количество ячеек памяти, поэтому и количество их комбинаций конечно. В таком случае на компьютере можно задать только конечное количество вариантов входных данных для любой задачи.

Последний факт очевиден, но, все же, требует доказательства. Доказательство строится от противного: допустим, количество различных вариантов наборов значений ячеек памяти компьютера N, а задача имеет большее количество наборов исходных данных. В таком случае найдется хотя бы два варианта исходных данных задачи, соответствующих одному и тому же набору значений ячеек памяти компьютера, что противоречит тому, что мы можем задать на компьютере все варианты исходных данных задачи.
Итак, мы получили, что для всех алгоритмов, которые можно реализовать на компьютере существует верхняя оценка времени работы. На самом деле, этот факт не стоит ничего, т.к., с одной стороны, функция $F(N)$ может очень быстро расти, а с другой, конечность количества ячеек компьютера всегда вступает в противоречие с бесконечностью мира. Например, конечность количества ячеек делает бессмысленным рассмотрение функции $F(N)$ для любого натурального N.

Нижней оценкой времени выполнения алгоритма m называется такая функция $j(N)$, что для любого N найдется такой набор входных данных алгоритма размером N, что время работы данного алгоритма на указанных данных будет не меньше $j(N)$.

Нижней оценкой времени решения задачи z называется такая функция $j(N)$, что для любого алгоритма, решающего данную задачу, $j(N)$ будет нижней оценкой времени работы данного алгоритма.

Будем говорить, что *задача* z_1 *сводится к задаче* z_2 *за время* $g(N)$, если

- входные данные задачи z_1, имеющие объем N, могут быть приведены к входным данным задачи z_2, при этом входные данные задачи z_2 тоже имеют объем N;
- выходные данные задачи z_2 могут быть приведены к выходным данным задачи z_1,

и все это за суммарное время $g(N)$ (т.е. суммарное время выполнения алгоритмов приведения = $g(N)$). По аналогии, можно говорить, что *алгоритм* m_1 *сводится к алгоритму* m_2 *за время* $g(N)$ (определение аналогично).

Теорема 1. *Если задача* z_1 *сводится к задаче* z_2 *за время* $g(N)$ и задача z_2 *имеет верхнюю оценку времени решения* $F_2(N)$, *то задача* z_1 *имеет верхнюю оценку времени решения* $F_1(N) = F_2(N) + g(N)$.

Доказательство теоремы тривиально.

Теорема 2. *Если задача* z_1 *сводится к задаче* z_2 *за время* $g(N)$ и задача z_1 *имеет нижнюю оценку времени решения* $j_1(N)$, *то задача* z_2 *имеет нижнюю оценку времени решения* $j_2(N) = j_1(N) - g(N)$.

Доказательство. Выпишем аккуратно условие того, что $j_2(N)$ является нижней оценкой времени решения задачи z_2:

" алгоритма m_2 : $j_2(N)$ - нижняя оценка времени работы алгоритма m_2.

или:

" алгоритма m_2 и "$N>0$ $ $In(m_2)$ с размером, равным N: $T(m_2(In(m_2))) \geq j_2(N) = j_1(N) - g(N)$.

Поведем доказательство от противного. Допустим это не так, т.е. выполняется условие:

$ алгоритм m_2 и $N>0$: " $In(m_2)$: $T(m_2(In(m_2))) < j_2(N) = j_1(N) - g(N)$.

Тогда существует алгоритм (заключающийся в сведении задачи m_1 к задаче m_2 за время $g(N)$), для которого на всех исходных данных размера N задача решается за время, меньшее $j_1(N) - g(N) + g(N) = j_1(N)$, что противоречит условию теоремы.

■

Введем обозначения.

$g(n)=O(f(n))$ *если* $ $N_0>0$ *и* $C>0$, *такие что* " $n>N_0$:| $g(n)$ | £ C| $f(n)$ |

$g(n)=o(f(n))$ *если* " $C>0$ $ $N_0>0$, *такое что* " $n>N_0$: |$g(n)$ | £ C |$f(n)$ |

$g(n)=\Omega(f(n))$ *если* $ $N_0>0$ *и* $C_1, C_2>0$, *такие что* " $n>N_0$:
C_1 |$f(n)$ | £ |$g(n)$ | £ C_2 |$f(n)$ |

$g(n)=\Theta(f(n))$ *если* $ $N_0>0$ *и* $C>0$, *такие что* " $n>N_0$: |$g(n)$ | ³ C |$f(n)$ |

Сортировки

Постановка задачи

Для элементов некоторого множества **P** введены соотношения сравнения. Под этим будем подразумевать следующее: для каждых двух элементов **a,b Î P** верно ровно одно из трех соотношений: **a<b, a>b, a=b.** Эти соотношения должны обладать свойствами транзитивности:

a<b, b<c Þ a<c
a>b, b>c Þ a>c

a=b, b=c Þ a=c
и аналогом свойства симметричности:
a<b Û b>a

Пусть дано некоторое упорядоченное подмножество (последовательность) элементов из **P : {a_1, …, a_N}, a_iÎ P.** Требуется найти такую перестановку **(x_1, …,x_N)**, что **a_{x1}, …, a_{xN}** – будет неубывающей последовательностью, т.е. **$a_{xi} < a_{x(i+1)}$** или **$a_{xi} = a_{x(i+1)}$** . Напомним, что перестановкой ***n*** элементов мы называем некоторое взаимно-однозначное соответствие множества чисел *{1,…,N}* с таким же множеством чисел *{1,…,N}*, т.е. такую функцию ***s*****:** *{1,…,N} -> {1,…,N}*, для которой если i^1j , то $s(i)^1s(j)$.

Здесь, конечно, надо сразу задаться вопросом: а возможно ли это сделать при данных ограничениях на приведенные операции сравнения? Другим разумным вопросом будет: а если это можно сделать, то единственным ли (с точностью до перестановок подряд идущих элементов, между которыми выполняется соотношение =) способом? Ответы на оба вопросы положительны.

Доказательства утверждения, кроющегося в первом вопросе (о существовании перестановки), легко провести по индукции по **n**.

Для доказательства утверждения, кроющегося во втором вопросе (о единственности перестановки), можно сначала показать, что в упорядоченном множестве элементы, между которыми выполняется соотношение = должны идти подряд, что дает возможность заменить их одним элементом. Далее можно ввести функцию ***M(i)*** – количество элементов из **{a_1, …, a_N},** меньших **a_i**. Отметим, что для любых ***i≠j*** выполняется: ***M(i)≠M(j)*** (действительно: выполняется либо **$a_i < a_j$**, либо **$a_i > a_j$**, откуда легко вывести, что, соответственно, ***M(i)<M(j)*** , либо ***M(i)>M(j)***), из чего сразу вытекает (учитывая, что 0≤***M(i)*****<n**), что функция ***M(i)*** принимает все возможные значения от 0 до **n-1**. Легко показать, что эта функция однозначно определяет положение элемента **a_i** в упорядоченном множестве.

Часто, когда исходные данные сами по себе не допускают использования операции сравнения, вводят понятие *ключа сортировки.* Ключом называют некоторую величину, сопоставляемую каждому экземпляру рассматриваемого множества, которая допускает операцию сравнения. Часто в качестве ключа используют целые или вещественные числа.

Будем говорить, что алгоритм сортировки *основан на операциях сравнения*, если алгоритм может быть записан в виде бинарного дерева (*дерева решения*), каждая вершина которого либо является завершающей (т.е. при попадании в нее исходная последовательность данных оказывается отсортированной), либо:

- вычисляется некоторая функция от входных данных алгоритма,
- производится сравнение полученной величины с 0 (одной из операций: <, > или =)

- от каждой вершины дерева, в зависимости от полученного результата, происходит переход к левой или правой ветви дерева
- на каждой ветви дерева происходит одна определенная для данной ветви транспозиция элементов входных данных (обмен местами двух определенных элементов последовательности).

Отметим, что часто в литературе завершающие вершины дерева называются *листьями*, но мы далее дадим для них другое определение, поэтому пока этим понятием пользоваться не будем.

Будем говорить, что алгоритм сортировки *основан на операциях* ***простого*** *сравнения*, если алгоритм основан на операциях сравнения и в нем допускаются только попарные сравнения элементов исходного массива данных.

Если исходные данные задачи принадлежат k-мерному Евклидову пространству и если вычисляемая в узлах функция является многочленом степени ***n***, то говорят, что *алгоритм представим в виде алгебраического дерева степени* ***n***.

Сортировка пузырьком.
Алгоритм:

N-1 раз выполняется следующая процедура:
Для всех ***i*** от ***1*** до ***N-1*** с шагом ***1***:
если **$a_{xi} > a_{x(i+1)}$** *то* поменять местами **a_{xi}** и **$a_{x(i+1)}$**

Легко видеть, что алгоритм требует порядка ***O(N^2)*** арифметических операций.

Теорема. *Алгоритм сортировки пузырьком является оптимальным по порядку времени выполнения среди алгоритмов, основанных на операции сравнения, если обмен местами двух элементов последовательности требует* ***O(N)*** *времени.*

Ситуация, отраженная в теореме, возможна, например, при сортировке данных записанных на ленте, в какой-то мере, для данных, записанных на жестком диске. В быту данная ситуация реализуется в случае, когда у нас есть большой набор карточек с какими-то записями, выложенными в длинный ряд.

Доказательство. Рассмотрим следующую последовательность:
N, N-1, N-2, … 2, 1
Для любой сортировки этой последовательности следует первый в ней элемент поставить на последнее место (=порядка ***O(N-1)*** операций), второй элемент поставить на предпоследнее место (=порядка ***O(N-2)*** операций) и т.д. Итого, только перестановки займут не менее ***O(N^2)*** времени, откуда мы получаем

нижнюю оценку времени выполнения алгоритмов данного класса. Данная оценка достигается на предложенном алгоритме.

■

Далее мы будем рассматривать алгоритмы, в которых операции сравнения и перестановки двух элементов занимают единичное время.

Теорема. *Нижней оценкой времени решения задачи сортировки в рамках алгоритмов, основанных на операции сравнения, является* Q ***(N log₂ N)****. Т.е. существует функция* ***g(N)***=Q ***(N log₂ N)****, являющаяся нижней оценкой решения задачи сортировки в рамках алгоритмов, основанных на операции сравнения.*

Замечание. На самом деле, не обязательно ограничивать операции, допустимые после сравнения элементов в дереве решения, лишь одной транспозицией. Можно разрешить выполнять произвольную перестановку всех ***N*** элементов за 1 единицу времени. Теорема останется верной.

Доказательство. Рассмотрим решение задачи о сортировке набора из ***N*** целых чисел от ***1*** до ***N***. Решение можно представить как дерево решения. Исключим из этого дерева все ветки, начиная с элемента, в который нельзя попасть и до завершающего элемента дерева. Удаление данных веток никак не повлияет на реальный алгоритм.

Теперь каждой перестановке ***s(1,…,N)*** (здесь под ***s*** подразумевается перестановка элементов множетва ***{1,…,N}***) соответствует своя концевая вершина в дереве решения (соответствующая только этой перестановке), такая что ветка дерева решения от корня до данной вершины задает перестановку ***p***, обратную ***s***.: ***p(s(i))=i*** "***i***Î***{1,…,N}***. Т.е. данная ветка задает решение задачи сортировки для последовательности исходных данных ***{s(1), s(2,)… s(N)}***.

Действительно, в силу определения дерева решений, для каждой последовательности исходных данных ***{s(1), s(2,)… s(N)}*** мы имеем ровно одну ветвь дерева решений (от корня до завершающей дерево вершины), сортирующую данную последовательность. Причем, каждая завершающая дерево вершина является концевой ровно для одной перестановки ***s(1,…,N)***. Действительно, мы исключили вершины, до которых нельзя в принципе добраться, поэтому осталось исключить ситуацию, когда вершина соответствует сразу двум различным перестановкам ***s(1,…,N)*** и ***r(1,…,N)***. Но в последнем случае мы имеем: ***p(s(i))=i*** и ***p(r(i))=i*** "***i***Î***{1,…,N}***, где перестановка ***p*** описана выше. Из чего сразу получаем, что перестановки ***s*** и ***r*** совпадают.

Таким образом, мы доказали, что количество завершающих вершин дерева решений равно количеству перестановок множества ***{1,…,N}***, равно ***n!***.

Будем называть *глубиной дерева* количество вершин в его самой длинной ветке. Для дерева глубины ***h*** мы имеем, что хотя бы в одной ветви дерева количество сравнений равно ***h-1***, из чего сразу получаем, что ***h-1*** является нижней оценкой времени работы всех алгоритмов, описываемых деревьями сравнения глубины ***h*** (мы задаем время сравнения равное 1).

Дерево глубины ***h*** не может иметь количество концевых вершин ***K*** более чем
2^{h-1}: $K \le 2^{h-1}$, откуда получаем: $h \ge \log_2 K + 1$.
Итого, в нашем случае:

$$h \ge \log_2 K + 1 = \log_2 N! + 1 = \Omega(N \log_2 N).$$

Здесь мы использовали известную формулу Стирлинга:

$$n! = n^n e^{-n} \sqrt{2\pi N}\,(1+o(1))$$

из которой сразу следует, что

$$\log_2 N! = (N \log_2 N - N \log_2 e + \log_2 \sqrt{2\pi N})(1+o(1)) = (N \log_2 N)(1+o(1))$$

■

Приведем примеры, показывающие, что приведенная нижняя оценка времени решения задачи сортировки достижима.

Сортировка слиянием с рекурсией.
Слиянием двух упорядоченных множеств называется процесс упорядочения объединения данных множеств.

Теорема. *Пусть даны два упорядоченных множества* ***{A₁,…,A_N}*** *и* ***{B₁, …,B_N}.*** *В рамках алгоритмов, основанных на простых сравнениях, данные множества нельзя слить быстрее, чем за* ***2N-1*** *сравнение в худшем случае. Т.е.* ***2N-1*** *является нижней оценкой времени работы алгоритма, если учитывать только время, расходуемой на сравнения элементов множеств, и если положить время одного сравнения равным 1.*

Доказательство. Пусть для конкретных заданных множеств выполняются соотношения $A_i < B_i$ и $A_{i+1} > B_i$. Тогда отсортированное объединение множеств выглядит следующим образом: ***{A₁, B₁, A₂, B₂ ,…, A_N,B_N}.*** Если хотя бы одно из приведенных ***2N-1*** соотношений не будет проверено, то найдется еще хотя бы одна перестановка элементов множества, удовлетворяющая всем приведенным соотношениям. Например, если не будет проверено соотношение $A_2 > B_1$, то следующая последовательность будет удовлетворять всем остальным соотношениям:
{A₁, A₂, B₁, B₂ ,…, A_N,B_N}.

Более того, отношения между всеми остальными элементами останутся неизменными. Т.о. мы доказали необходимость всех приведенных сравнений для правильного упорядочивания указанных данных, из чего непосредственно вытекает требуемое.

■

Дословно так же доказывается следующая теорема

Теорема. *Пусть даны два упорядоченных множества* $\{A_1,\ldots,A_{N+1}\}$ *и* $\{B_1,\ldots,B_N\}$. *В рамках алгоритмов, основанных на простых сравнениях, данные множества нельзя слить быстрее, чем за* $2N$ *сравнений элементов множества в худшем случае.*

Алгоритм слияния. *Пусть даны два упорядоченных множества* $\{A_1,\ldots,A_M\}$ *и* $\{B_1,\ldots,B_N\}$. *Введем индексы* i, j *и* k. *Изначально* $i=1, j=1$ *и* $k=1$.

Пока $i \le M$ *и* $j \le N$***:***
  Если $A_i < B_j$ ***то***
    $C_{k++} = A_{i++}$
  иначе
    $C_{k++} = B_{i++}$
  Конец Если
Конец Цикла
Пока $I \le M$***:***
  $C_{k++} = A_{i++}$
Конец Цикла
Пока $j \le N$***:***
  $C_{k++} = B_{i++}$
Конец Цикла

Легко увидеть, что в данном алгоритме элементы множества сравниваются не более $M+N-1$ раз. Т.о. данный алгоритм оказывается строго оптимальным по числу сравнений элементов сортируемого множества (по крайней мере в алгоритмах, основанных на простых сравнениях).

Вопрос на понимание*: можно ли два упорядоченных множества* $\{A_1,\ldots,A_N\}$ *и* $\{B_1,\ldots,B_N\}$ *слить быстрее чем за* $2N-1$ *операций сравнения в каком либо алгоритме, основанном операциях сравнения? … на операциях простого сравнения?*

Алгоритм сортировки слиянием. *Обозначим данный алгоритм* $Z(A_1,\ldots,A_M)$, *где* $\{A_1,\ldots,A_N\}$ *– сортируемое множество элементов. Алгоритм имеет следующий вид*

Если *число обрабатываемых элементов* ≤ 1 ***то ВЫЙТИ***
$M_1 = [\, M/2 \,]$; $M_2 = M-M_1$; *// размеры половин массива*
$Z(A_1,\ldots,A_{M1})$
$Z(AM1+1,\ldots,AM)$
Слить *упорядоченные множества* $\{A_1,\ldots,A_{M1}\}$ *и* $\{A_{M1+1},\ldots,A_M\}$ *в массив* B.
Скопировать *массив* B *в массив* $\{A_1,\ldots,A_N\}$.

Легко видеть, что данный алгоритм решает задачу за время $O(N \log_2 N)$, где N – количество элементов в сортируемом массиве.
Недостатком алгоритма является необходимость использования дополнительного массива с размером, равным размеру исходного массива.

Сортировка слиянием без рекурсии.

Предыдущий алгоритм можно модифицировать так, что он уже не будет использовать рекурсию. Действительно. Рассмотрим последовательно все пары элементов в сортируемом массиве. Каждый из элементов в паре представляет собой уже отсортированный массив длины 1, поэтому эти массивы (пока длины 1) можно слить в упорядоченные куски длины 2. Далее мы рассматриваем уже пары упорядоченных массивов длины 2 и сливаем их в массивы длины 4. И т.д.
Отметим, что при этих операциях на k-том проходе по упорядочиваемому массиву на правом конце массива мы будем получать либо ситуацию, когда у правого оставшегося куска (длины $\le 2^k$) вообще нет парного куска для слияния, либо кусок есть и его длина $\le 2^k$. В первом случае делать вообще ничего не нужно, а во втором следует стандартным способом сливать куски, возможно, существенно различной длины.
Легко видеть, что данный алгоритм решает задачу за время $O(N \log_2 N)$, где N – количество элементов в сортируемом массиве.

Лекция 3

Алгоритмы. Сведение алгоритмов. Сортировки и связанные с ними задачи.

Д.Кнут. Искусство программирования для ЭВМ. тт 1-3. Москва. Мир. 1996-1998

Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы. Построение и анализ. Москва. МЦНМО. 1999.

Препарата Ф., Шеймос М. Вычислительная геометрия. Москва. Мир. 1989

QuickSort.

Определение. *Медианой множества* $A = \{a_1, \ldots, a_N\}$ *называется элемент с индексом* $(N+1)/2$ *в отсортированном по возрастанию множестве* A.

Пусть, для определенности, мы сортируем массив вещественных чисел. Идея алгоритма заключается в следующем. Выберем некоторое число, желательно близкое, в каком-то смысле, к медиане сортируемого множества. Разобьем наше множество на две половины – в одной (левой половине) должны быть элементы меньше или равные выбранного элемента, а в другой (в правой половине) – больше или равные. Из построения этих подмножеств следует, что их расположение совпадает с их расположением в отсортированном множестве чисел (расположение – в смысле множеств), т.е. после сортировки элементы из этих подмножеств останутся на месте этих же подмножеств. Т.о., применив рекурсивно эту же процедуру для каждого из подмножеств, мы, в конечном итоге, получим отсортированное множество.

Для реализации этой идеи рассмотрим алгоритм, который не предполагает хранения медианы в отдельной ячейке памяти. В процессе работы алгоритма мы будем следить за тем, в какой ячейке памяти располагается элемент исходного множества, который мы выбрали в качестве медианы. Подобная реализация в реальности чуть более медленная, но доказательство работы алгоритма намного проще, чем в случае хранения медианы в отдельной ячейке.

В следующей реализации в комментариях показаны соотношения на значения элементов, которые выполняются после каждого шага алгоритма. Эти соотношения доказывают, что каждый раз массив разбивается на части, левая из которых не превосходит медианы, а правая – не меньше медианы. Здесь для простоты множество элементов $\{A_s, A_{s+1}, \ldots, A_t\}$ будем обозначать $\{s,t\}$. Медиану будем обозначать M.

```
QuickSort(A,p,q)
Если  q-p < 1 то ВЫЙТИ
Вечный цикл
   i=p; j=q; // пусть M=A_j
//цикл 1:
   Пока A_i < A_j :  i + +;
//{p,i-1}<=M<={j,q}, A_i>=M
   поменять местами A_i и A_j ;//M -> A_i
//{p,i}<=M<={j,q}
   j --;
//{p,i}<=M<={j+1,q}
   Если  i >= j то
//либо i==j                 то {p, j}<=M<={ j+1,q}
// либо i==j+1              то M== A_{j+1}  => {p, j}<=M<={ j+1,q}
    { QuickSort(A, p, j ); QuickSort(A, j+1, q );ВЫЙТИ }
//цикл 2:
   Пока A_j > A_i :  j - -;
//{p,i}<=M<={j+1,q}, A_j<=M
   поменять местами A_i и A_j ;//x -> A_j
//{p,i}<=M<={j,q}
   i + +;
//{p,i-1}<=M<={j,q}
   Если  i >= j то
//либо i==j                 то M== A_j  => {p, j}<=M<={ j+1,q}
// либо i==j+1              то {p, j}<=M<={ j+1,q}
    { QuickSort(A, p, j ); QuickSort(A, j+1, q );ВЫЙТИ }
```

Конец вечного цикла

В силу построения алгоритма j не может стать меньше 0 и не может быть больше или равным q, поэтому гарантируется, что мы не попадем в бесконечную рекурсию и границы рассмотрения массива корректны.

Отметим, что после первого цикла также имеем:

```
   Если  i >= j то
//либо i==j                 то {p, i}<=M<={ i+1,q}
// либо i==j+1              то M== A_{j+1}  => {p, i}<=M<={ i+1,q}
```

т.е. рекурсию можно было бы организовать в виде:

```
{ QuickSort(A, p, i ); QuickSort(A, i+1, q );ВЫЙТИ }
```

но в этом случае мы можем попасть в бесконечную рекурсию, т.к. в цикле i может дойти вплоть до q.

После второго цикла также имеем:

```
   Если  i >= j то
//либо i==j                 то {p, i-1}<=M<={ i,q}
// либо i==j+1              то {p, i-1}<=M<={ i,q}
```

т.е. рекурсию можно было бы организовать в виде:

```
{ QuickSort(A, p, i-1 ); QuickSort(A, i, q );ВЫЙТИ }
```

В этом случае ***i*** не может стать меньше 1 и не может быть больше ***q***, поэтому такой вариант алгоритма также возможен.

В более стандартной реализацией основной идеи данного алгоритма выбирается произвольный элемент *x* сортируемого множества в качестве среднего элемента и помещается в отдельную ячейку памяти. Далее, один шаг алгоритма заключается в следующем. Мы двигаемся слева направо, пока элементы множества меньше *x*. Затем мы движемся справа налево, пока элементы множества больше *x*. Если мы еще не встретились, то следует поменять местами элементы, на которых мы стоим при движении в каждую сторону и повторить шаг алгоритма. Если же мы встретились, то алгоритм вызывается для каждой из полученных половин множества.

```
QuickSort(A,p,q)
Если  q-p < 1 то ВЫЙТИ
 i=p; j=q; x=Ai
Вечный цикл
  Пока A i  < x :  i + +;//{p,i-1}<=x, Ai >=x
  Пока A j  > x :  j - -;//{j+1,q}>=x, Aj <=x

  Если  i < j то
    поменять местами A i  и A j ; //{p,i}<=x, {j,q}>=x
  иначе
    {
//либо i==j      то Ai ==x => {p,j}<=x, {j+1,q}>=x
//либо i==j+1 то {p,j}<=x, {j+1,q}>=x
    QuickSort(A, p, j ); QuickSort(A, j+1, q );ВЫЙТИ
    }
  i + +; j - -;//{p,i-1}<=x, {j+1,q}>=x
```

Конец вечного цикла

Замечание 1. *При работе алгоритм индексы массива i и j никогда не выйдут за его границы* ***p*** *и* ***q***.

Замечание 2. В алгоритме никогда не произойдет вечной рекурсии, т.е. при рекурсивном вызове

$$p \le j < q$$

Замечание 3. Алгоритм гарантирует корректное разбиение массива, т.е. после разбиения массива выполняются соотношения

$A_k \le x$ для всех $k: \ p \le k \le j$

$A_l \ge x$ для всех k: $j+1 \le k \le q$

Тонкость алгоритма характеризует следующее наблюдение. Давайте попробуем ``обезопасить алгоритм'' и включим обеспечение условия $i \le j$ в циклы алгоритма ***Пока… .*** Т.е. приведем алгоритм к следующему виду:

```
QuickSort*(A,p,q)
Если  q-p < 1 то ВЫЙТИ
 i=p; j=q; x=Ai
Вечный цикл
  Пока A i  < x  и  i < j :  i + +;
  Пока A j  > x  и  i < j :  j - -;
  Если  i < j то
    поменять местами A i  и A j ;
  иначе
   { QuickSort(A, p, j ); QuickSort(A, j+1, q );ВЫЙТИ }
  i + +; j - -;
```

Конец вечного цикла

Алгоритм ***QuickSort**** оказывается неверным !!! Это легко увидеть на простейшем примере: ***{3,4,2,5}***.

Доказательство корректности работы алгоритма.

Доказательство корректности работы алгоритма сводится к доказательству **Замечаний 1-3** для каждого тела вечного цикла алгоритма.

Рассмотрим два принципиально разных случая.

1. Случай $A_p = min\{ A_p , A_{p+1} , ..., A_q \}$. Все остальные элементы массива больше A_p.

В конце первого выполнения тела вечного цикла алгоритма ***i=p, j=p***. Все элементы в первой половине множества (она, в данном случае, состоит из одного элемента) оказываются меньше элементов из второй половины. Массив разбивается на половины размерами ***1:N-1***, где ***N=q-p+1*** – количество элементов в массиве.

2. Все остальные случаи.

Доказательство Замечания 1. При работе алгоритм индексы массива ***i*** и ***j*** никогда не выйдут за его границы ***p*** и ***q***.

Выход за границы массива, потенциально, может произойти только в результате выполнения циклов ***Пока***... или при выполнении операций ***i + +; j - -;*** в конце тела вечного цикла алгоритма.

Изначально на первом месте массива стоит $A_p = x$. В конце выполнения первого тела вечного цикла алгоритма A_p может поменяться местами с элементом меньше либо равным x и в дальнейшем элемент A_p не изменится. Т.о. на первом месте массива всегда будет стоять элемент $\le x$ и в процессе выполнения цикла ***Пока… j*** не сможет оказаться меньше p. В результате выполнения операций $i++; j--;$ выхода за левую границу массива также не может произойти, т.к. если бы это произошло, то это означало бы, что перед этой операцией выполнялось бы $j=p$. Но в этом случае оказывается неверным соотношение $i < j$ (т.к. $i^3 p$) и, следовательно, попасть в данную точку алгоритма при этих условиях оказывается невозможным.

Разберемся с правой границей. Если в первый момент $A_q \,^3\, x$, то j сразу же уменьшится на 1 и впоследствии A_q не изменится. Это гарантирует, что в процессе выполнения цикла ***Пока… i*** не сможет оказаться больше q. Если же $A_q < x$, то после циклов ***Пока…*** i и j не изменятся и A_p и A_q сразу же поменяются местами. Далее A_q не изменится и i не сможет превысить q. В результате выполнения операций $i++; j--;$ выхода за правую границу массива не сможет произойти по причинам аналогичным обсужденным ранее. Замечание доказано.

Доказательство Замечания 2. *В алгоритме никогда не произойдет вечной рекурсии, т.е. при рекурсивном вызове*

$$p \le j < q$$

Первое из требуемых неравенств доказано выше. Второе также легко доказать. Действительно, если при первом входе в тело вечного цикла алгоритма $A_q \,^3\, x$, то j сразу же уменьшится, и мы получим требуемое. Иначе, при первом выполнении тела вечного цикла алгоритма в циклах ***Пока…*** , i и j не изменятся, поэтому после этих циклов $i < j$ и j обязано уменьшится в конце тела цикла. Замечание доказано.

Доказательство Замечания 3. *Алгоритм гарантирует корректное разбиение массива, т.е. после разбиения массива выполняются соотношения*

$$A_k \le x \text{ для всех } k:\ p \le k \le j$$
$$A_l \,^3\, x \text{ для всех } k:\ j+1 \le l \le q$$

Согласно построению алгоритма в конце выполнения тела вечного цикла алгоритма гарантируется, что

$$A_k \le x \text{ для всех } k < i$$
$$A_l \,^3\, x \text{ для всех } l > j$$

Т.о. мы сразу же получаем выполнение второго из неравенств **Замечания 3**. Первое неравенство оказывается более хитрым (именно оно не выполняется при работе алгоритма ***QuickSort****). В рассматриваемом случае среди элементов правее первого найдется элемент меньше первого, из чего следует, что после первого выполнения циклов ***Пока…*** в тела вечного цикла алгоритма i останется строго меньше j, после чего A_i поменяется местом с A_j и выполнится $i++; j--$. Т.о. элемент со значением x далее останется в правой половине массива (этот факт мы используем далее в доказательстве теоремы о среднем времени работы алгоритма ***QuickSort***).

Если после выполнения циклов ***Пока..*** в некотором теле вечного цикла алгоритма окажется, что $i > j$, то из приведенных соотношений сразу следует первое неравенство **Замечания 3**. Случай $i < j$ говорит о незавершенности алгоритма. Осталось рассмотреть случай $i = j$. Этот вариант может реализоваться только в случае, когда $A_j = x$, тогда получаем, что $A_k \le x$ для всех $k < j$ и $A_k = x$ для всех $k = j$. Итого, получаем первое неравенство **Замечания 3** (в алгоритме ***QuickSort****) этот случай создается искусственно и поэтому первое неравенство из **Замечания 3** остается невыполненным).

■

Оценки времени работы алгоритма.

Оценим временя работы приведенного алгоритма в худшем случае.

Теорема. *Время работы алгоритма* ***QuickSort*** *равно* $O(N^2)$*, где* N *– количество элементов в сортируемом массиве.*

Доказательство. После каждого разбиения массива на две части длина самой большой из двух образовавшихся половин оказывается меньше либо равной длине разбиваемого массива -1. Поэтому на каждой ветви алгоритма будет не более N узлов (разбиений массива). На каждом уровне дерева разбиений присутствуют не более N сортируемых элементов, поэтому время, затрачиваемое на слияние их подмножеств равно $O(N)$. Итого, суммарное время работы алгоритма равно $O(N) * N = O(N^2)$.

Данная оценка достижима на массиве $\{N, N-1, \ldots, 1\}$.

■

Оказывается, что число ``неприятных'' случаев, т.е. таких расположений массивов чисел, при которых время работы алгоритма ***QuickSort*** велико, оказывается, относительно, небольшим. Вообще, верна теорема

Теорема. *Среднее время работы алгоритма* ***QuickSort*** *равно* $Q(N \log_2 N)$*, где* N *– количество элементов в сортируемом массиве. Под средним временем подразумевается среднее время по всем перестановкам любого массива входных данных длины N, состоящего из различных элементов.*

Данная теорема объясняет, в каком смысле данный алгоритм является оптимальным. В то же время, в реальной жизни, часто поток входных данных не является случайным, поэтому в качестве медианы следует брать случайно выбранный элемент. Для этого внесем в алгоритм ***QuickSort*** следующее дополнение. Перед присваиванием $x = A_i$ поменяем местами i-ый элемент массива со случайно выбранным элементом среди элементов с индексами от p до q. Назовем получившийся алгоритм ***QuickSortP.*** Приведенная теорема верна также

и для алгоритма ***QuickSortP.*** Докажем ее именно для последнего алгоритма. Будем, кроме того, предполагать, что все элементы входной последовательности различны, или, что то же самое, на входе подается последовательность различных элементов из множества ***{1,…,N}.***
В рассматриваемом случае если ***x=1***, то перед входом в рекурсию алгоритма *QuickSort* множество разобьется на части размером ***1*** и ***N-1***. В любом другом случае, как отмечалось выше в доказательстве Замечания 3, элемент ***x*** останется в правой половине массива и размер левой половины массива, поэтому, будет равен ***x-1***.
Выпишем рекуррентное соотношение на среднее время работы алгоритма

$$T(N) = [\,(T(1) + T(N-1)) + \sum_{i=2}^{i \le N} (T(i-1) + T(N-i+1))\,]/N + Q(N) =$$
$$= [\,(T(1) + T(N-1)) + \sum_{i=1}^{i<N} (T(i) + T(N-i))\,]/N + Q(N) =$$

$$= [\,(T(1) + T(N-1)\,]/N + [\,\sum_{i=1}^{i<N} (T(i) + T(N-i))\,]/N + Q(N) =$$
$$= [\,\sum_{i=1}^{i<N} (T(i) + T(N-i))\,]/N + Q(N)$$

Предположим, для $i<N$ верно:

$T(i) < a\, i \log i + c$ для некоторых $a>0,\ c>0$,

тогда задача сводится к нахождению таких $a>0,\ c>0$, что для них всегда бы выполнялось соотношение

$$[\,\sum_{i=1}^{i<N} (T(i) + T(N-i))\,]/N + Q(N) < a\, N \log N + c$$

Итак

$$[\,\sum_{i=1}^{i<N} (T(i) + T(N-i))\,]/N < [\,\sum_{i=1}^{i<N} (a\, i \log i + c + a\,(N-i) \log (N-i) + c))\,]/N =$$
$$= [\,\sum_{i=1}^{i<N} (a\, i \log i + c))\,]2/N < a\,[\,\sum_{i=1}^{i<N} i \log i\,]2/N + 2c$$

Оценим сумму из соотношения:

$$\sum_{i=1}^{i<N} i \log i = \sum_{i=1}^{i<N} i \log N + \sum_{i=1}^{i<N} i \log (i/N) = N^2 \log N / 2 - \sum_{i=1}^{i<N} i \log (N/i) \le$$
$$N^2 \log N / 2 - \sum_{i=1}^{i<N/4} i \log (N/i) \le N^2 \log N / 2 - \sum_{i=1}^{i<N/4} i\, 2 \le N^2 \log N / 2 - N^2/8$$

Т.о. имеем

$$[\,\sum_{i=1}^{i<N} (T(i) + T(N-i))\,]/N + Q(N) < a\,(N \log N - N/4) + 2c + Q(N) =$$
$$= a\, N \log N + c + (Q(N) + c - a\, N/4)$$

Осталось взять такое большое ***a***, что $(Q(N) + c - a\, N/4) < 0$, после чего мы получаем требуемое соотношение.
■

К сожалению, обе приведенные реализации алгоритма *QuickSort* не являются жизнеспособными. Это связано с тем, что в обоих алгоритмах максимально возможная глубина рекурсии равна *N*. В этом случае требуется порядка *O(N)* байт дополнительной памяти, что не фатально, но проблема в том, что эта память будет выделяться в стеке задачи, а стек задачи всегда имеет маленький (относительно) размер. Поэтому, например, в *Microsoft Visual Studio* мы может ожидать ситуации *stack overflow* при размерах целочисленного массива порядка 100000 (размер стека здесь по умолчанию равен 1M).
Выходом из положения является следующее решение. Будем рекурсивно решать задачу сортировки только для меньшей половины массива. А большую половину будем сортировать в этой же процедуре. В таком случае глубина погружения в рекурсию не будет превосходить $]\log_2 N[$, что является приемлемым.
В реальности для данной реализации требуется внести минимальные добавки в исходный алгоритм быстрой сортировки. Например, алгоритм ***QuickSort*** может быть модифицирован следующим образом:

```
QuickSortR1(A,p,q)
Если  q-p < 1 то ВЫЙТИ
Вечный цикл
Метка:
  i=p; j=q; x=A_i
  Пока A_i < x :  i + +;
  Пока A_j > x :  j - -;
  Если  i < j то
    поменять местами A_i и A_j ;
  иначе
   {
    if(j-p<q-(j+1))
    {
     QuickSort(A, p, j );
      p=j+1;q=q;goto Метка;
    }
    иначе
    {
     QuickSort(A, j+1, q );
      p=p;q=j;goto Метка;
    }
    ВЫЙТИ
   }
  i + +; j - -;
```

Конец вечного цикла

Здесь добавлены:
- метка (может быть заменена еще одним вечным циклом),
- проверка, какая часть массива больше,
- переназначение ***p,q*** в каждом случае.

Некоторые задачи, сводящиеся к сортировке.
К задачам сортировки могут быть за линейное время сведены следующие классические задачи

Задача 1. *Найти все различные элементы семейства* $\{A_1,\ldots,A_N\}$, *где –* A_i *, например, целые или вещественные числа.*

Задача 2. *Определить, все ли элементы в семействе* $A=\{A_1,\ldots,A_N\}$ *различаются, где –* A_i, *например, целые или вещественные числа.*

Задача 3. *Определить, совпадают ли два семейства* $\{A_1,\ldots,A_N\}$ *и* $\{B_1,\ldots,B_N\}$ *с учетом количества одинаковых элементов в каждом семействе, где* A_i, B_i *, например, – целые или вещественные числа.*

Слово *семейство* здесь используется вместо слова *множество* в силу возможности повторения элементов.

Т.о., мы получаем, что для всех трех приведенных задач верхняя оценка времени решения равна $\Theta(N \log N)$.

Можно задаться вопросом: а можно ли решить эти задачи быстрее? Заметим, что Задача 2 может быть за линейное время сведена к Задаче 1, поэтому если мы докажем, что нижняя оценка времени решения Задачи 2 есть $\Omega(N \log N)$, то тем мы докажем неулучшаемость полученной оценки для Задачи 1.

Задача 2 относится к классу задач *о принятии решения.* Это значит, что на выходе таких задач выдается всего один бит информации, т.е. ответ `*да*' или `*нет*'. Мы будем рассматривать алгоритмы решения задач о принятии решений, которые сводятся к *бинарному дереву принятия решений.* Под *деревом принятия решений* имеется в виду следующее. Пусть на входе нашего алгоритма находится вектор входных данных $a \in IR^N$. Рассмотрим бинарное дерево, в каждой вершине которого вычисляется некоторая функция от вектора a и в зависимости от знака этой функции происходит переход на правую или левую ветвь дерева. Каждая конечная вершина дерева будет называться либо *принимающей* либо *отвергающей*, в зависимости от приписанного ей соответствующего атрибута. Достижение принимающей вершины означает выдачу алгоритмом ответа `*да*', а отвергающей, соответственно, - `*нет*'. Далее будем предполагать, что до всех конечных вершин дерева принятия решения можно добраться при каких-то значениях входных данных.

Дерево принятия решений называется *алгебраическим деревом принятия решений степени* n, если функции в вершинах дерева являются многочленами степени не выше n.

Далее мы рассмотрим лишь алгоритмы, представимые в виде алгебраического дерева принятия решений степени 1. Мы будем предполагать, что время вычисления каждой функции в вершине дерева есть $\Theta(1)$. Приведенная далее теорема верна и для деревьев высшей степени, но для ее доказательства потребуются весьма серьезные факты из теории функций, которые мы здесь привести не сможем.

Введем два определения.

Разделимые множества. *Множества* $A, B \subset IR^N$ *называются* ***разделимыми*** *(****линейно-разделимыми****) если* $\forall\, a \in A, b \in B$ *найдется* $c \in [a,b]$, *такое что* $c \notin A$, $c \notin B$.

Будем говорить, что **множество** X **состоит из разделимых связных компонент** $\{A_i\}$, если $X=\cup\{A_i\}$, A_i *связны и не пересекаются и* $\forall\, a \in A_i, b \in A_j$ $(i \neq j)$ *найдется* $c \in [a,b]$, *такое что* $c \notin X$.

Связное множество. ***Связным множеством*** *называется такое множество, что при любом его разбиении на два непересекающихся непустых подмножества одно из них содержит точку, предельную для другого.* В евклидовом пространстве открытое множество связно тогда и только тогда, когда любые две его точки можно соединить целиком лежащей в нём ломаной.

Теорема. *Пусть* $W \subset IR^N$ *– множество точек, на которых решением задачи будет `да'. Пусть* $\#W$ *– количество разделимых связных компонент множества* W. *Тогда, в рамках алгоритмов, описывающихся алгебраическими деревьями степени 1, существует нижняя оценка времени решения задачи, равная* $\Omega(\log_2 \#W)$.

Доказательство. Докажем, что количество принимающих вершин дерева принятия решений не меньше $\#W$. Для этого докажем, что все элементы R^N, относящиеся к одной принимающей вершине дерева решений находятся внутри одной разделимой связной компоненты W.

Предположим противное: существует принимающая вершина дерева принятия решений, в которую попадает алгоритм при использовании двух различных $a,b \in W$, таких что a и b принадлежат различным разделимым связным компонентам $V_a, V_b \subset W$, соответственно. Рассмотрим $[a,b]$. Линейные функции от N переменных обладают свойством монотонности на отрезке, поэтому все функции, вычисляющиеся в вершинах дерева принятия решений, сохраняют свой знак на $[a,b]$. Т.о. весь отрезок $[a,b] \subset W$ (т.к. все точки из отрезка

обязаны попасть в одно и ту же принимающую вершину). Это противоречит предположению о принадлежности ***a*** и ***b*** различным разделимым компонентам.

Итак, мы получили, что количество концевых вершин бинарного дерева принятия решений не меньше ***#W***, из чего сразу следует, что высота дерева не меньше ***[log 2 #W]***.

■

Бен-Ором было доказано более сильное утверждение, которое мы будем называть *Теоремой Бен-Ора*:

Теорема. *Пусть* ***W*** Ì ***IR*** ***N*** *– множество точек, на которых решением задачи будет `да'. Пусть* ***#W*** *– количество разделимых связных компонент множества* ***W****. Тогда, в рамках алгоритмов, описывающихся алгебраическими деревьями фиксированной степени* ***d****, существует нижняя оценка времени решения задачи, равная* Q ***(log 2 (#W) - N)***.

Вернемся к решению Задачи 2. Рассмотрим множество ***W*** для Задачи 2. Легко увидеть, что ***W*** – открыто. Для точки ***P*** Î ***IR*** ***N*** будем называть *связной компонентой, содержащей* ***P****,* множество W_P Ì ***W***, состоящее из таких точек ***R*** Î ***W***, для которых существует ломаная, соединяющая ***P*** и ***R*** , содержащаяся в ***W***. В силу открытости ***W*** каждая такая точка ***R*** имеет окрестность O_R Ì ***W***, а значит O_R Ì W_P . Но тогда W_P можно представить как объединение всех описанных окрестностей O_R , а объединение любого количества открытых множеств открыто. Т.о. множество W_P открыто, а значит (в силу определения множества) и связно.

Рассмотрим в качестве различных вариантов входных данных задачи все перестановки последовательности ***{1,2,…,N}***. Т.е.

A_p***= {p(1), p(2),…, p(N)}***, где ***p –*** некоторая перестановка.

Докажем, что каждая последовательность A_p принадлежит своей связной компоненте W_{AP} множества ***W*** и что все W_{AP} линейно разделимы. Тогда мы докажем, что каждая A_p принадлежит своей разделимой связной компоненте. Действительно, допустим противное, т.е. пусть две различные последовательности A_{p1} и A_{p2} принадлежат одной связной компоненте ***W.*** Тогда найдутся ***1 £ i, j £ N***, такие что $(p_1(i)- p_1(j))\ (p_2(i)- p_2(j))<0$, т.е. $(p_1(i)- p_1(j))$ и $(p_2(i)- p_2(j))$ имеют разные знаки. Но тогда для любой ломаной ***S***, соединяющей точки A_{p1} и A_{p2} Î ***IR*** ***N*** **,** найдется ***x*** Î ***S*** такая, что $x_i = x_j$, что противоречит принадлежности A_{p1} и A_{p2} одной связной компоненте множества ***W***.

Аналогично доказывается, что компоненты W_{P1} и W_{P2} линейно разделимы. Действительно, рассмотрим произвольные точки p_1' Î W_{P1} и p_2' Î W_{P2}. Для тех же самых индексов ***i, j*** получим, что $(p'_1(i)- p'_1(j))\ (p'_2(i)- p'_2(j))<0$ (здесь мы использовали следующее утверждение: для любой пары ***i, j*** внутри одной открытой связной компоненты ***W*** знак ***p(i)- p(j)*** для всех точек одинаков), из чего сразу получаем, что на отрезке, соединяющем p'_1 и p'_2, найдется точка, не принадлежащая ***W***.

В приведенном доказательстве требует объяснения следующий факт: для двух различных перестановок множества ***{1,2,…,N}*** всегда найдутся ***1 £ i, j £ N***, такие что $(p_1(i)- p_1(j))$ и $(p_2(i)- p_2(j))$ имеют разные знаки. Пусть для каждой перестановки ***p:*** t_k – количество чисел ***p(i)*** больших ***p(k)*** для ***i>k.*** Легко увидеть, что ***p*** и ***t*** однозначно задают друг друга.

Действительно, для ***p(i)=N*** имеем ***t(i)=0***, причем для ***j<i*** выполняется: ***t(j)>0***. (отметим, что ***t(i)=0*** может выполняться и для других ***i***) Поэтому если мы найдем минимальное ***i*** такое**,** что ***t(i)=0***, то сразу получим, что ***p(i)=N***; далее положим ***p(i)=-1***, что потребует уменьшить на ***1*** все ***t(j)*** для ***j<i***, а ***t(i)*** тоже положим равной ***-1***, чтобы исключить из рассмотрения. Для нахождения ***i*** такого, что ***p(i)=N-1***, мы опять ищем минимальное ***i*** такое**,** что ***t(i)=0***. После исключения из рассмотрения найденного ***i*** и уменьшения на ***1*** всех ***t(j)*** для ***j<i*** мы можем перейти к поиску ***i*** такого, что ***p(i)=N-2***, и т.д.

Пример:

p={4,5,3,1,2} => t={1,0,0,1,0}

Ищем ***p*** по заданной ***t***:

1)Ищем минимальное ***i***, для которого ***t(i)=0***: ***i=2 => p(2)=5***

Для всех ***j<2*** уменьшаем ***t*** на ***1*** и кладем ***t(i)=-1***:

t={0,-1,0,1,0}

2) Ищем минимальное ***i***, для которого ***t(i)=0***: ***i=1 => p(1)=4***

Для всех ***j<1*** уменьшаем ***t*** на ***1*** (таковых нет) и кладем ***t(i)=-1***:

t={-1,-1,0,1,0}

3) Ищем минимальное ***i***, для которого ***t(i)=0: i=3 => p(3)=3***

Для всех ***j<3*** уменьшаем ***t*** на ***1*** и кладем ***t(i)=-1***:

t={-2,-3,-1,1,0}

4) Ищем минимальное ***i***, для которого ***t(i)=0***: ***i=5 => p(5)=2***

Для всех ***j<5*** уменьшаем ***t*** на ***1*** и кладем ***t(i)=-1***:

t={-3,-4,-2,0,-1}

5) Ищем минимальное ***i***, для которого ***t(i)=0***: ***i=4 => p(4)=1***

Т.о. для двух различных перестановок p_1 и p_2 мы получим различные соответствующие t_1 и t_2 . Выберем ***i:*** $t_1(i) \neq t_2(i)$**.** Пусть, например, $t_1(i) > t_2(i)$, но тогда найдется ***j>i***, такое что $p_1(j) > p_1(i)$, но $p_2(j) < p_2(i)$**.** Получили требуемое.

■

Таким образом, воспользовавшись формулой Стирлинга, мы сразу докажем следующую теорему:

Теорема. *Задачи 1 и 2 имеют нижнюю оценку времени решения* W ***(NlogN)*** *на алгоритмах, основанных на алгебраическом дереве принятия решения первой степени.*

Для Задачи 3 также можно доказать аналогичную теорему:

Теорема. *Задача 3 имеет нижнюю оценку времени решения* W ***(NlogN)*** *на алгоритмах, основанных на алгебраическом дереве принятия решения первой степени.*

Для доказательства рассмотрим все пары последовательностей *{1,2,…,N}* и $\{s_1,s_2,\ldots,s_N\}$ для всех перестановок *s*. Назовем эти последовательности *A* и B_s. Покажем, что количество принимающих вершин любого алгебраического дерева принятия решения первой степени, решающего заданную задачу, не меньше, чем количество всевозможных пар $\{A, B_s\}$. В этом случае мы сразу получим, что высота дерева принятия решения будет не меньше Q ***(log(N!))***= Q ***(N log N)***, что докажет нашу теорему.

Доказательство требуемого факта тривиально. Нам достаточно доказать, что никакие две различные пары $\{A, B_s\}$ и $\{A, B_r\}$ не могут попасть в одну принимающую вершину данного дерева принятия решений. Докажем данный факт от противного.

Пусть есть две различные пары $\{A, B_s\}$ и $\{A, B_r\}$, на которых данный алгоритм попадает в одну принимающую вершину алгебраического дерева принятий решений первой степени. Тогда найдется индекс *i* для которого $r_i \neq s_i$. Рассмотрим пары последовательностей $\{A, B_{st}\}$, для которых $B_{st}=tB_s+(1-t)B_r$, где $t \hat{I} (0,1)$. Значения B_{st} для $t \hat{I} [0,1]$ образуют собой отрезок в пространстве $\boldsymbol{IR}^{2N}$. В данных парах последовательностей элемент B_{st} с индексом *i* должен принимать все возможное значения в интервале $(Min(r_i,s_i), Max(r_i,s_i))$. Тогда, с одной стороны любая пара $\{A, B_{st}\}$ должна попасть в ту же принимающую вершину (в силу проверки линейных соотношений в каждой вершине дерева и сохранения знака линейной функции на отрезке), а с другой, среди возможных значений *t* обязательно найдется значение, не встречающееся в последовательности *A*. Получаемое противоречие завершает доказательство.

■

Заметим, что Задача 3 также, как и Задача 2, может быть сведена к Задаче 1.

Лекция 4

Алгоритмы. Сведение алгоритмов. Сортировки и связанные с ними задачи.

Д.Кнут. Искусство программирования для ЭВМ. тт 1-3. Москва. Мир. 1996-1998
Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы. Построение и анализ. Москва. МЦНМО. 1999.
Препарата Ф., Шеймос М. Вычислительная геометрия. Москва. Мир. 1989

К вопросу о понимании предыдущих лекций. Найти ошибку.

``Доказательство'' того, что любое натуральное число можно однозначно получить с помощью алгоритма, задаваемого не более, чем 20-тью словами (имеется в виду, можно использовать только существующие в языке слова, а не что-нибудь вроде "ШестьсотШестьдесятШесть").
Пусть это не так. Тогда существует множество, являющееся подмножеством натуральных чисел, каждый элемент которого невозможно получить с помощью алгоритма, задаваемого не более, чем 30 словами. У всякого подмножества натуральных чисел есть наименьший элемент.
Получаем: "Наименьшее натуральное число, которое нельзя получить с помощью алгоритма, задаваемого не более, чем 30 словами, имеющимися в русском языке" - итого 20 слов потребовалось, дабы назвать данное число, которое принадлежит этому великолепному множеству => противоречие.

***HeapSort* или сортировка с помощью *пирамиды*.**

Алгоритм основан на промежуточном упорядочивании массива входных данных $\{A_1,\ldots, A_N\}$. Мы докажем, что промежуточно-упорядоченный массив (мы будем его называть *пирамидально-упорядоченным*) обладает свойством максимальности своего первого элемента. Тогда мы отрезаем от массива первый элемент и восстанавливаем утраченное свойство пирамидально-упорядоченности у оставшегося куска. Так, отрезая по одному (максимальному из оставшихся) элементу, мы можем `набрать' полный упорядоченный массив.

Определение. *Массив* $\{A_1,\ldots, A_N\}$ *называется* ***пирамидально-упорядоченным,*** *если для всех допустимых* ***i****:* $A_{[i/2]} \geq A_i$.

Иначе данное соотношение можно выписать следующим образом:

$A_i \ge A_{2i}$ и $A_i \ge A_{2i+1}$ **(*)**

Легко видеть, что данные соотношения задают древообразную структуру, в вершине которой находится первый элемент дерева. Его потомками являются элементы с номерами **2** и **3**, и т.д. В получившемся дереве все слои заполнены, кроме, быть может, последнего. Поэтому глубина дерева равна $[\log N]+1$, где N – количество элементов в множестве.

Пусть для некоторого поддерева пирамиды, начинающегося с элемента с индексом i и заканчивающегося элементом с индексом N, выполнено свойство **(*)** для всех элементов поддерева, кроме вершины поддерева. Т.е. свойство выполняется для всех элементов, имеющих индексы больше i (здесь имеется в виду возможное невыполнения условий $A_k \ge A_{2k}$ и $A_k \ge A_{2k+1}$ для $k=i$ и его выполнение при $k>i$).

Определим процедуру ***Heapify(A,i,N)***, которая в данном случае подправляет элементы поддерева до полной пирамидально-упорядоченности элементов с индексами от i до N. Здесь A – рассматриваемый массив, i – индекс массива с которого начинается рассматриваемое поддерево, N – количество элементов во всем дереве.

Процедура ***Heapify(A,i,N)*** осуществляет следующие действия. Она проверяет условия

$A_i \ge A_{2i}$ в случае $2i \le N$

$A_i \ge A_{2i+1}$ в случае $2i+1 \le N$.

Если они выполняются (случаи $2i \ge N$, $2i+1 \ge N$ легко рассмотреть отдельно), то дальше ничего делать не надо, происходит выход из процедуры. Иначе, выбирается максимальный из элементов A_i, A_{2i}, A_{2i+1} и выбранный элемент меняется местами с A_i. Не ограничивая общности рассуждений, допустим, что максимальным оказался элемент A_{2i}, тогда после перестановки имеем $A_i \ge A_{2i+1}$, при этом элемент A_{2i+1} не изменился, поэтому свойство пирамидально-упорядоченности будет выполняться и дальше в данном (правом) поддереве. Далее рекурсивно вызываем процедуру ***Heapify(A,2i,N).***

Исходя из построения процедуры ***Heapify,*** имеем следующее утверждение

Утверждение 1. *Процедура* ***Heapify(A,i,N)*** *выполняется за время* $O(h(i,N))$*, где* $h(i,N)$ *– глубина поддерева в пирамиде из* N *элементов, начинающегося с элемента с индексом* i.

Алгоритм ***Heapsort(A,N)*** выглядит следующим образом

Heapsort(A,N)

Для всех i *от* ***N-1*** *до* ***1*** *с шагом* ***–1 выполнить: Heapify(A,i,N)***

Для всех i *от* ***1*** *до* ***N-1*** *с шагом* ***1 выполнить***

Поменять местами *элементы* A_1 *и* A_{N-i+1}

Heapify(A,1,N-i)

Первый цикл в алгоритме создает пирамиду, а второй, используя ее свойство максимальности первого элемента, создает упорядоченный массив. Согласно Утверждению 1, каждый цикл состоит из N процедур, каждая из которых выполняется за время $O(\log_2 N)$, из чего вытекает теорема

Теорема. *Время работы алгоритма* ***Heapsort(A,N)*** *равно* $O(N \log_2 N)$.

На самом деле, оказывается, что время работы первого из двух циклов алгоритма равно $O(N)$. Действительно, процедура ***Heapify(A,i)*** для каждого i из последнего уровня дерева выполняется за время $O(1)$ (а в этом уровне содержится половина всех элементов!). Для следующего уровня время выполнения процедуры равно уже $O(2)$. И т.д.

Т.о. суммарное время работы алгоритма вычисляется по формуле

$$T(N) = O(\sum_{i=0}^{i \le h} (h-i+1)\, 2^i)$$

где высота дерева равна $h+1$ (т.е. дерево имеет уровни с номерами от 0 до h).

Докажем соотношение $T(N)/2^h = \Theta(1)$. Отсюда и из того, что количество элементов в дереве высотой $h+1$ находится между 2^{h-1} и 2^h, мы сразу получим, что время работы алгоритма равно $O(N)$.

Рассмотрим следующие равенства для некоторого $x \ne 1$

$1 + x + x^2 + \ldots + x^N = (1 - x^{N+1})/(1-x)$, тогда, взяв производную по x, получим

$1 + 2x + 3x^2 + \ldots + Nx^{N-1} = (-(N+1)x^N(1-x) + (1 - x^{N+1}))/(1-x)^2 = \Theta(1)$, если $x = 1/2$.

С другой стороны, положим $j = h-i+1$, тогда

$$T(N)/2^h = O(\sum_{i=0}^{i \le h} (h-i+1)\, 2^{i-h}) = O(\sum_{j=1}^{j \le h+1} j\, 2^{-j+1}) = O(\sum_{j=1}^{j \le h+1} j\, 2^{-j}) = \Theta(1).$$

■

Из вышесказанного вытекает, что мы имеем возможность получить упорядоченный подмассив, состоящий из довольно большого количества самых больших элементов исходного массива, за время $O(N)$, где N – количество элементов в массиве. Более строго, верна теорема

Теорема. *Получить упорядоченный массив из* $N / \log_2 N$ *самых больших элементов массива можно за время* $O(N)$.

Алгоритмы сортировки за время O(N)

Итак, мы рассмотрели алгоритмы, основанные на операциях сравнения, и для них получили нижнюю оценку времени выполнения. Возникает вопрос, а можно ли на ЭВМ выполнять операцию сортировки быстрее? Здесь следует отметить, что на ЭВМ есть операция, которая принципиально не вписывается в множество рассмотренных операций. Это – операция индексации массива с использованием в качестве индекса функций, вычисляемых от упорядочиваемых элементов. Все алгоритмы, выполняющиеся за время ***O(N)*** используют эту операцию.

Сортировка подсчетом

Пусть мы хотим отсортировать ***N*** целых чисел $A=\{A_1,\ldots, A_N\}$, каждое из которых не превосходит ***K***, при этом ***K=O(N)***. Тогда мы можем создать временный массив ***B*** размером ***K***, в который можно поместить для каждого ***i*** количество чисел в массиве ***A,*** не превосходящих ***i***. Тогда для каждого $1 \le i \le N$: в отсортированном массиве в элементе с индексом $B_{A\,i}$ лежит элемент, равный A_i.

Итак, приведем реализацию данного алгоритма. Результат будем помещать в третий массив ***C***

CountingSort (A,C, N,K, B)

Для всех i *от* ***1*** *до* ***K*** *с шагом* ***1 выполнить: B[i]=0***
Для всех i *от* ***1*** *до* ***N*** *с шагом* ***1 выполнить: B[A[i]] ++***
Для всех i *от* ***1*** *до* ***N*** *с шагом* ***1 выполнить: B[A[i]]= B[A[i]]+ B[A[i-1]]***
Для всех i *от* ***N*** *до* ***1*** *с шагом* ***-1 выполнить: C[B[A[i]]] = A[i]; B[A[i]]- -***

Единственным дополнением к вышеприведенному описанию в этом алгоритме является добавка в его конец `***B[A[i]]- -***' . Эта добавка гарантирует, что если в массиве ***A*** есть элементы с равными значениями, то они будут положены в различные ячейки массива ***C***. Более того, каждый следующий элемент со значением, равным некоторому ***x*** (при обратном проходе!), будет помещаться в ячейку левее предыдущей. Поэтому данная сортировка сохраняет взаимное расположение равных элементов. Этой свойство сортировки называется *устойчивостью.* Это свойство имеет смысл, когда равенство элементов в смысле сравнения не влечет тождественного равенства элементов. Например, это происходит если сортировка идет по ключу.

Цифровая сортировка

Идея весьма проста. Пусть требуется отсортировать массив целых чисел, записанных в некоторой позиционной системе исчисления. Сначала мы сортируем массив устойчивым методом по младшей цифре. Потом – по второй, и т.д. Очередная сортировка не меняет порядок уже отсортированных элементов, поэтому в конце мы получим отсортированный массив. Прямой проверкой доказывается следующая

Теорема. *Алгоритм цифровой сортировки требует* ***O(nd)*** *операций, где* ***n*** *– максимальное количество операций для одной внутренней сортировки,* ***d*** *– количество цифр.*

Этот алгоритм облегчает использование сортировки подсчетом. Действительно, если есть большой массив 32-битных целых чисел без приемлемых ограничений на их величину, то можно разбить их на 2 либо 4 части и рассмотреть каждую часть как одну цифру в алгоритме цифровой сортировки.

Сортировка вычерпыванием

Пусть требуется отсортировать массив из ***N*** вещественных чисел $A=\{A_1,\ldots, A_N\}$, равномерно распределенных на интервале ***[0,1).*** Идея алгоритма заключается в следующем. Разобьем интервал ***[0,1)*** на ***N*** равных частей и каждой части сопоставим свой контейнер элементов (например, в самом простом случае, массив вещественных чисел длины ***N***). Каждое число ***x*** положим в контейнер с номером ***[x*N]***. После этого отсортируем элементы в каждом контейнере и соберем по порядку элементы из всех контейнеров вместе.

Более конкретно, для реализации контейнеров мы сначала посчитаем, сколько элементов попадет в каждый контейнер, а потом для распределения элементов по контейнерам нам достаточно будет иметь один массив вещественных чисел длины ***N***. Итак, для сортировки массива ***A***, состоящего из ***N*** элементов, мы должны завести массивы целых чисел ***M, I*** длины ***N*** и массив вещественных чисел ***B*** длины ***N.*** Пусть функция ***Sort(B,i0,n)*** выполняет сортировку пузырьком части массива ***B*** , начинающейся с элемента с индексом ***i0***, состоящей из ***n*** элементов. Тогда алгоритм имеет следующий вид

SortB (A, N, M, B)

Для всех i *от* ***1*** *до* ***N*** *с шагом* ***1 выполнить: M[i]=0; I[i]=0; B[i]=0***
Для всех i *от* ***1*** *до* ***N*** *с шагом* ***1 выполнить: M[A[i]*N+1] ++***
Для всех i *от* ***2*** *до* ***N*** *с шагом* ***1 выполнить: M[i] = M[i]+ M[i-1]***
Для всех i *от* ***N*** *до* ***2*** *с шагом* ***-1 выполнить: M[i] = M[i-1]***
M[0]=0
Для всех i *от* ***1*** *до* ***N*** *с шагом* ***1 выполнить: B[M[i]+I[i]+A[i]*N+1]= A[i]; I[i]++***
Для всех i *от* ***1*** *до* ***N*** *с шагом* ***1 выполнить: Sort(B,M[i],I[i])***
Для всех i *от* ***1*** *до* ***N*** *с шагом* ***1 выполнить:***
  Для всех j *от* ***1*** *до* ***I[i]*** *с шагом* ***1 выполнить: A[k]= B[M[i]+j]; k++***

Во втором цикле алгоритма мы подсчитываем количество элементов, попавших в ***i***-ый интервал. В третьем и четвертом циклах мы помещаем в ***M[i]*** индекс

первого элемента части массива ***B***, относящейся к контейнеру с номером ***i***. В пятом цикле мы помещаем элементы в соответствующие контейнеры. В шестом цикле происходит сортировка элементов в контейнерах. Далее мы последовательно выбираем элементы в результирующий массив ***A***.

Теорема. *Алгоритм **SortB** работает за время **O(N)** в среднем, где **N** – количество сортируемых элементов.*
Доказательство. Пусть ***p=1/N.*** Вероятность попадания в один контейнер ***k*** элементов равна $p_k=C_N^k p^k (1-p)^{N-k}$ (биноминальное распределение)**.** Время работы алгоритма сортировки в одном контейнере равно $\sum O(k^2)$, где ***k*** – количество элементов, попавших в ***i-***ый контейнер.

Согласно свойствам биномиального распределения, среднее (математическое ожидание) количество элементов в контейнере равно $M(k)=\sum_k p_k k=Np=1$. Средне-квадратичное отклонение от среднего значения (дисперсия) количества элементов в контейнере равно $D(k)=\sum_k p_k(k-M(k))^2=\sum_k p_k(k-1)^2=Np(1-p)=1-1/N$.

$D(k)=M(k^2)-(M(k))^2$ из чего сразу следует $M(k^2)=D(k)+(M(k))^2=2-1/N$. Итого, среднее время сортировки одного контейнера равно ***O(1)***, а среднее время сортировок ***N*** контейнеров равно ***O(N).***
■

Лекция 5

Алгоритмы. Сведение алгоритмов.

Д.Кнут. Искусство программирования для ЭВМ. тт 1-3. Москва. Мир. 1996-1998
Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы. Построение и анализ. Москва. МЦНМО. 1999.
Препарата Ф., Шеймос М. Вычислительная геометрия. Москва. Мир. 1989

Примеры задач, к которым может быть сведена сортировка.

Напомним важную теорему 2 из Лекции 2:

*Если задача z_1 сводится к задаче z_2 за время **g(N)*** и задача z_1 *имеет нижнюю оценку времени решения* $\jmath_1(N)$***,*** *то задача z_2 имеет нижнюю оценку времени решения* $\jmath_2(N)=\jmath_1(N)-g(N)$***.***

Напомним также, что при создании алгоритмов, решающих рассматриваемую задачу, изначально фиксируется набор допустимых операций, используемых в алгоритмах, и задаются времена выполнения этих операций. Для исследования нижних оценок времени решения задачи сортировки мы рассматривали алгоритмы на основе дерева решений. Это задавало нам набор допустимых операций с возможностью задания времени работы каждой операции. Далее в данной лекции мы будем использовать тот же самый набор допустимых операций. Т.е. все рассматриваемые задачи будут решаться на основе алгоритмов, основанных на сравнениях.

Таким образом, если мы докажем, что решение задачи сортировки может быть сведено за линейное время к решению некоторой другой задачи ***z***, то мы сразу докажем, что у задачи ***z*** в классе алгоритмов, основанных на сравнениях, имеется нижняя оценка времени решения задачи $\jmath(N)=O(N \log N)$.

Данная лекция будет, по сути, состоять из довольно большого количества чисто геометрических доказательств. В лекции будет использоваться довольно много чисто геометрических утверждений. Если эти факты очевидны и их доказательство элементарно делается на уровне школьной геометрии, то для сокращения объема лекции их доказательства производиться не будут.

Построение выпуклой оболочки

Определение. *Множество точек S на плоскости называется **выпуклой оболочкой** множества точек $\{A_1,\ldots,A_N\}$ на плоскости, если S является наименьшим выпуклым множеством, содержащим точки $\{A_1,\ldots,A_N\}$.*

Отметим, что вместо стандартного определения *выпуклого множества* (***M** – выпукло, если для любых **A,B**$\in$**M**: **[A,B]** $\in$**M***) в школе дают другое определение *выпуклого многоугольника*: *многоугольник **M** – выпуклый, если для*

любой прямой ***l****, проходящей через ребро* ***M****,* ***M*** *лежит в одной полуплоскости с границей* ***l***. Доказательство эквивалентности этих определений в одну сторону (***M*** – выпуклый многоугольник Þ ***M*** – выпуклое множество) элементарно (полуплоскость – выпукла, а пересечение выпуклых множеств выпукло), а в другую все же требует одного шага (докажем, что если многоугольник лежит по две стороны от прямой, лежащей на ребре, то он не выпуклый; для доказательства рассмотрим такую окрестность некоторой точки ***X*** на этом ребре, что с одной стороны ребра точки окрестности принадлежат ***M****,* а с другой – нет; осталось соединить ***X*** с точкой многоугольника на второй стороне прямой и мы сразу получим, что ***M*** не выпуклый).

Существование такого множества доказывается элементарно: рассмотрим все выпуклые множества, содержащие $\{A_1,\ldots, A_N\}$. Пересечение всех этих множеств является выпуклым множеством и (по построению) содержится в любом выпуклом множестве, содержащем $\{A_1,\ldots, A_N\}$. Т.о. полученное множество точек и дает требуемую выпуклую оболочку точек $\{A_1, \ldots, A_N\}$.

Приведем без доказательства довольно очевидную (здесь не сказано, что *тривиальную*) теорему:

Теорема 1. *Выпуклой оболочкой конечного множества точек на плоскости* $\{A_1,\ldots, A_N\}$ *является некоторый выпуклый многоугольник с вершинами, принадлежащими множеству* $\{A_1,\ldots, A_N\}$**.**

Везде далее фраза *множество точек лежит по одну сторону от прямой* ***l*** будет синонимом (сокращением) фразы *множество точек лежит в одной полуплоскости с границей, лежащей на прямой* ***l***.

Для каждой пары точек A_i, A_j, принадлежащих $\{A_1,\ldots, A_N\}$, верно одно из двух утверждений: либо все остальные точки множества $\{A_1,\ldots, A_N\}$ лежат в одной полуплоскости с границей (A_i, A_j), либо точки из множества $\{A_1,\ldots, A_N\}$ присутствуют с двух сторон относительно (A_i, A_j). При этом выполняется следующая

Теорема 2. *Если для* (A_i, A_j) *все точки множества* $\{A_1,\ldots, A_N\}$ *лежат в одной полуплоскости с границей* (A_i, A_j)*, то* $[A_i, A_j]$ *принадлежит границе многоугольника выпуклой оболочки* $\{A_1,\ldots, A_N\}$*, т.е. является, как минимум, частью некоторого ребра многоугольника выпуклой оболочки.*

Теорема легко доказывается от противного. Назовем многоугольник выпуклой оболочки ***M***. $[A_i, A_j]$ принадлежит ***M***, т.к. выпуклая оболочка – выпуклое множество и точки A_i и A_j принадлежат этому множеству. Здесь A_i, A_j – такие точки из множества $\{A_1,\ldots, A_N\}$, что все точки множества $\{A_1,\ldots, A_N\}$ лежат в одной полуплоскости с границей (A_i, A_j).

Допустим, некоторая точка ***P*** отрезка $[A_i, A_j]$ не принадлежит границе выпуклой оболочки. Тогда ***P*** – внутренняя точка многоугольника ***M***. Но поскольку многоугольник – замкнутое множество точек, то получаем, что ***P*** принадлежит ***M*** вместе с некоторой своей окрестностью. Т.о. мы получаем, что по обе стороны от отрезка $[A_i, A_j]$ присутствуют точки из ***M***. Рассмотрим полуплоскость π с границей (A_i, A_j), внутри которой находятся все точки из $\{A_1, \ldots, A_N\}$. Рассмотрим многоугольник $M' = M \cap \pi$. M' является выпуклым многоугольником (т.к. M' есть пересечение выпуклого многоугольника и полуплоскости). По построению M' содержит все точки $\{A_1,\ldots, A_N\}$. ***M*** содержит точки, не принадлежащие M'. Но это противоречит тому, что ***M*** является наименьшим выпуклым множеством, содержащим $\{A_1,\ldots, A_N\}$. Из полученного противоречия следует, что весь $[A_i, A_j]$ принадлежит границе ***M***.

■

Следствие из Теоремы 2. *Объединение всех отрезков* $[A_i, A_j]$ *таких, что все точки множества* $\{A_1,\ldots, A_N\}$ *лежат в одной полуплоскости с границей* (A_i, A_j)*, представляет собой границу многоугольника выпуклой оболочки* $\{A_1,\ldots, A_N\}$.

Доказательство следствия элементарно: с одной стороны, если отрезок $[A_i, A_j]$ обладает указанным в следствии свойством, то по тереме 2 он принадлежит границе выпуклой оболочки. С другой стороны, если отрезок $[C,D]$ является ребром многоугольника выпуклой оболочки $\{A_1,\ldots, A_N\}$, то по теореме 1 и определению выпуклого множества он является некоторым отрезком $[A_i, A_j]$ и все точки множества $\{A_1,\ldots, A_N\}$ лежат в одной полуплоскости с границей (A_i, A_j).

■

Данное следствие сразу дает алгоритм построения выпуклой оболочки ***M*** конечного множества точек $\{A_1,\ldots, A_N\}$. Несмотря на очевидную идею, алгоритм состоит из нескольких нетривиальных этапов:

1. Рассмотрим все пары точек из данного множества $\{A_i, A_j\}$. Прямым перебором всех оставшихся точек определяем, лежат ли остальные точки по одну сторону от (A_i, A_j), включая саму прямую. Если лежат по одну сторону, то $[A_i, A_j]$ является частью границы выпуклой оболочки и мы добавляем точки A_i, A_j к списку ***Q*** точек, принадлежащих выпуклой оболочке (при этом, мы за линейное время проверяем, есть ли эти точки уже в списке, и если есть, то добавление не происходит), а отрезок $[A_i, A_j]$ к списку отрезков ***R***.
2. Далее за кубическое время выбрасываем из полученного списка ***Q*** все такие точки ***A***, что в списке ***Q*** есть точки ***B*** и ***C*** такие, что ***A, B, C*** лежат на одной прямой и ***A*** находится между ***B*** и ***C***. Т.о. в списке ***Q*** останутся только вершины многоугольника выпуклой оболочки.

3. Далее в списке R оставляем только отрезки с вершинами из Q (это можно сделать за кубическое время, исходя из того, что в списке отрезков $O(N^2)$ элементов, а в списке вершин $O(N)$ элементов).
4. Теперь для каждой вершины T многоугольника M в списке R находится всего два отрезка с концом в T и мы имеем возможность за квадратичное время восстановить последовательность вершин и ребер многоугольника M.

По построению данный алгоритм будет работать за время $O(N^3)$.

Следует отметить, что выбранные в алгоритме отрезки $[A_i, A_j]$ не обязаны быть ребрами многоугольника выпуклой оболочки, но обязаны им (ребрам) принадлежать. В обосновании алгоритма также используется следующий факт, который элементарно доказать: каждое ребро многоугольника выпуклой оболочки является одной из выбранных выше пар точек A_i, A_j.

Теорема 3. *В рамках алгоритмов, основанных на сравнениях,* $\Omega(N)=O(N \log N)$ *является нижней оценкой времени решения задачи нахождения многоугольника выпуклой оболочки множества точек* $\{A_1,\ldots, A_N\}$.

Для доказательство теоремы докажем, что задача сортировки сводится к задаче нахождения выпуклой оболочки за время $O(N)$. Рассмотрим множества вещественных чисел $\{x_1,\ldots, x_N\}$. На числовой плоскости построим множество точек $\{(x_1, x_1^2), (x_2, x_2^2)\ldots, (x_N, x_N^2)\}$. Т.е. каждой точке x_i сопоставим точку на стандартной параболе (x_i, x_i^2). Для построенного множества точек решим задачу построения выпуклой оболочки. Найдем точку (x_i, x_i^2) с минимальным значением x_i. Последовательный перебор вершин построенного многоугольника даст набор точек с отсортированными значениями x_j.

■

Теорема 4. *Пусть* M *– выпуклая оболочка множества точек* $\{A_1,\ldots, A_N\}$. P *– внутренняя точка некоторого треугольника* (A_i,A_j,A_k), *то* P *– внутренняя точка* M.

Доказательство теоремы элементарно: если P – внутренняя точка треугольника A_i,A_j,A_k, то она принадлежит данному треугольнику вместе с некоторой своей окрестностью O. $A_i,A_j,A_k \in M$, M – выпукло, то $[A_j,A_k] \in M$. Пусть для каждой точки $X \in O$ точка $S=[A_j,A_k] \cap [A_i,X]$, то $S \in M$, тогда $[A_i,S] \in M$, тогда $X \in M$. Т.о. $O \in M$ и P – внутренняя точка M.

■

Для нахождения верхней оценки времени решения задачи нахождения выпуклой оболочки предъявим алгоритм, который решает данную задачу за время $O(N \log N)$, что докажет точность предъявленных верхней и нижней оценок.

Требуется найти выпуклую оболочку множества точек на плоскости $\{A_1, \ldots, A_N\}$ ($N>2$). За линейное время проверим, все ли точки лежат на одной прямой. Если все точки лежат на одной прямой, то задачу легко решить за линейное время. Иначе, выберем три точки, не лежащие на одной прямой и рассмотрим точку O – точку пересечения медиан треугольника с вершинами в найденных точках. По теореме 4 точка O будет лежать внутри выпуклой оболочки точек $\{A_1,\ldots, A_N\}$.

Упорядочим множество точек $\{A_1,\ldots, A_N\}$ лексикографически по паре значений (полярный угол, расстояние от O до данной точки) (пусть угол отсчитывается по часовой стрелке), отсчитываемому от точки O. Обозначим отсортированное множество точек $\{B_1,\ldots, B_N\}$. Будем считать, что B_1 – самая левая точка в множестве (точка с минимальной абсциссой). Этого легко добиться циклическим сдвигом полученного отсортированного множества точек. Легко увидеть, что B_1 лежит на границе выпуклой оболочки. Для доказательства этого рассмотрим луч, направленный вверх из B, B_1Q и точку B_k с минимальным углом QB_1B_k. Пусть B_1Q' – луч направленный из B_1 вниз. Все остальные точки из рассматриваемого множества будут лежать в угле $Q'B_1B_k$, т.е. все они будут лежать в одной полуплоскости с границей (B_1B_k). Тогда по теореме 2 $[B_1B_k]$ принадлежит границе выпуклой оболочки точек $\{A_1,\ldots, A_N\}$.

Будем последовательно перебирать точки B_i и добавлять их к двусвязному списку L граничных точек выпуклой оболочки. На каждом шаге (после добавления точки B_i к границе M) будем проверять точку B_{i-1} на величину угла $\alpha=(B_{i-2}\ B_{i-1}\ B_i)$ (угол отсчитывается против часовой стрелки). Если $\alpha \geq 180$, то точка B_{i-1} исключается из списка L и процедура проверки выполняется заново до тех пор, пока точка B_{i-1} не останется в списке. После этого переходим на следующий шаг добавления точки к списку. Полученная в конце обхода последовательность точек $\{B_{i1}, B_{i2}, \ldots, B_{ik}\}$ будет последовательностью вершин многоугольника выпуклой оболочки.

Для обоснования этого факта надо заметить, что полученный многоугольник будет выпуклым (т.к. все его углы <180). Пусть $\{B_{m1}, B_{m2}, \ldots, B_{ml}\}$ – последовательность вершин многоугольника выпуклой оболочки $\{B_1, B_2, \ldots, B_N\}$. Доказательство корректности данного алгоритма строится по индукции на основе простого факта: для любого j: $m_k<j<m_{(k+1)}$ верно: $B_j \in \Delta(O,B_{mk},B_{m(k+1)})$, где $\Delta(O,B_{mk},B_{m(k+1)})$ – треугольник с вершинами $O,B_{mk},B_{m(k+1)}$ без отрезка $[O,B_{mk}]$.

$B_{m1}= B_{i1}$ ($i_1= m_1=1$) и эта точка не будет удалена из создаваемого списка точек по построению. Таким образом мы получаем основание индукции.

Пусть в процессе построения списка точек $B_{mk}= B_{ik}$. Из вышеприведенного факта непосредственно следует, что при последующем построении списка точек B_{mk} не будет исключена из списка (отметим, что если бы мы изначально упорядочивали последовательность точек только по полярному углу, то вышеприведенный факт не имел бы место, и данное свойство не было бы верным). При переборе точек в какой-то момент мы дойдем до точки

$B_{m(k+1)}$ и по построению и по вышеприведенному факту ни одна из точек A_j ($m_k<j<m_{(k+1)}$) не останется в списке. Что завершает обоснование шага индукции.

В описанном алгоритме мы добавляем к списку не более N точек и удаляем из списка не более N точек, поэтому процедура обхода полученного множества точек $\{B_1,\ldots, B_N\}$ выполняется за линейное время. Данная процедура обхода носит название ***обхода Грэхема***.

Итак, алгоритм, использующий упорядочивание точек по полярному углу и обход Грэхема получившейся последовательности точек, имеет верхнюю оценку времени работы $F(N)=O(N \log N)$.

Мы получили точные нижнюю и верхнюю оценки времени решения задачи построения выпуклой оболочки, с использованием алгоритмов на основе сравнения.

Для улучшения (неулучшаемой ☺) оценки имеет смысл рассмотреть частный случай: пусть количество ребер выпуклой оболочки малО. Тогда имеет смысл построить алгоритм создания выпуклой оболочки, имеющий другую верхнюю оценку времени работы алгоритма. В худшем случае данный алгоритм будет работать медленнее алгоритма на основе обхода Грэхема, но если количество ребер многоугольника выпуклой оболочки будет существенно меньше $\log N$, то алгоритм окажется более успешным (например, в случае, когда есть уверенность, что многоугольник выпуклой оболочки имеет $O(1)$ ребер). Данный алгоритм носит название *алгоритма на основе* ***обхода Джарвиса*** или *алгоритм* ***заворачивания подарка***. Идея алгоритма проста: на ***k-м*** шаге алгоритма мы имеем k последовательных точек границы выпуклой оболочки $\{p_1, \ldots, p_k\}$ множества точек $\{A_1,\ldots, A_N\}$.Следующая точка p_{k+1} ищется из соображений минимальности величины пары (угол между лучами $[p_{k-1}, p_k)$ и $[p_k, x)$, расстояние от p_k до x) (здесь обход идет по часовой стрелке и угол отсчитывается по часовой стрелке), где в качестве x рассматриваются все оставшиеся точки множества $\{A_1,\ldots, A_N\}$. Пары сравниваются лексикографически (т.е. сначала сравнивается угол, а если углы равны, то сравниваются расстояния). Исходя из построения алгоритма, имеем следующую теорему о времени работы данного алгоритма.

Теорема 5. *Алгоритм построения выпуклой оболочки множества точек* $\{A_1,\ldots, A_N\}$ *на основе обхода Джарвиса имеет верхнюю оценку времени работы алгоритма* $F(N)=O(N M)$, где M – количество ребер границы многоугольника выпуклой оболочки.

Диаграмма Вороного. Триангуляция Делоне.

Определение. *Рассмотрим множество точек* $\{A_1,\ldots, A_N\}$ *на плоскости. Диаграммой Вороного называется разбиение плоскости на множества* $\{P_1,\ldots, P_N\}$, *где* P_i *состоит из точек, более близких к* A_i, *чем к остальным точкам из множества* $\{A_1,\ldots, A_N\}$. *Будем называть* A_i ***центром*** *области* P_i.

Легко увидеть, что для двух точек A_1 и A_2 диаграммой Вороного будет разбиение плоскости на две полуплоскости прямой, являющейся серединным перпендикуляром к $[A_1, A_2]$. В общем случае (для произвольного конечного множества $\{A_1,\ldots, A_N\}$) P_i представляет собой пересечение полуплоскостей, образованных серединным перпендикуляром ко всем отрезкам $[A_i, A_j]$ (i≠j), содержащих A_i. Отсюда непосредственно вытекает, что P_i является выпуклым многоугольником.

Будем считать диаграмму Вороного построенной, если нам известны:

1)массив вершин всех многоугольников Вороного,

2)массив ребер всех многоугольников Вороного (для каждого ребра задаются номера вершин)

3)массив многоугольников Вороного (для каждого многоугольника задается номер вершины его центра и массив номеров ребер),

4)для каждого ребра многоугольника Вороного номера многоугольников, для которых данное ребро является общим.

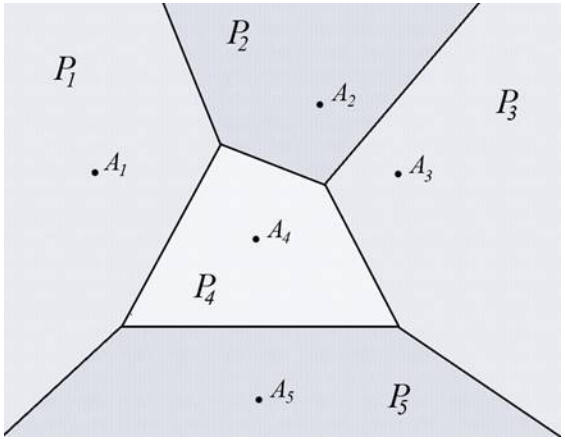


Рис 1. Многоугольники $\{P_1,\ldots,P_5\}$ диаграммы Вороного точек $\{A_1,\ldots,A_5\}$

Теорема 6. *В рамках алгоритмов, основанных на сравнениях,* $j(N)=O(N \log N)$ *является нижней оценкой времени решения задачи диаграммы Вороного множества точек* $\{A_1,\ldots, A_N\}$.

Для доказательства данной теоремы покажем, что задача сортировки сводится за линейное время к задаче построения диаграммы Вороного. Рассмотрим множество чисел $\{x_1,\ldots, x_N\}$. Для решения задачи сортировки данного множества чисел разместим эти числа на оси (OX) числовой плоскости. Решим задачу построения диаграммы Вороного полученного множества точек $\{(x_1,0)\ldots,(x_N,0)\}$. Ребрами диаграммы Вороного будут прямые, являющиеся

серединными перпендикулярами отрезков, соединяющих соседние числа на оси (OX) в отсортированном множестве чисел $\{x_1, \ldots, x_N\}$. За линейное время мы можем найти минимальное x_i. Для сортировки множества чисел каждое следующее x_j находится по предыдущему x_i очевидным образом: ищется многоугольник Вороного P_j (=полоса), смежный к многоугольнику P_i. Мы можем это сделать за постоянное время, поскольку у каждого многоугольника есть всего два ребра, а для каждого ребра мы знаем номера двух многоугольников, для которых данное ребро является общим.

Т.о. мы показали, что задача сортировки за линейное время сводится к задаче построения диаграммы Вороного.

■

Везде далее мы будем предполагать, что наши рассматриваемые точки $\{A_1, \ldots, A_N\}$ имеют ***общее положение***, имея под этим в виду, что никакие четыре точки из рассматриваемого множества не лежат на одной окружности. Данное предположение избавляет нас от рассмотрения большого количества частных случаев (несомненно, в реальных алгоритмах эти случаи рассматривать приходится).

Докажем несколько свойств Диаграммы Вороного (напомним, что везде далее мы предполагаем, что точки имеют общее положение).

Теорема 8. *Каждое ребро e многоугольника Вороного P_i лежит на серединном перпендикуляре к отрезку, соединяющему A_i и центр второго многоугольника Вороного, для которого e также является ребром.*

Из построения многоугольников Вороного сразу вытекает, что каждое ребро e многоугольника Вороного P_i лежит на серединном перпендикуляре к отрезку, соединяющему A_i и некоторую другую точку A_j из множества $\{A_1, \ldots, A_N\}$. Рассмотрим A_k – центр второго многоугольника Вороного, для которого e также является ребром. Опять же по построению, ребро e лежит на серединном перпендикуляре к отрезку, соединяющему A_k и некоторую другую точку A_l (см. Рис.2).

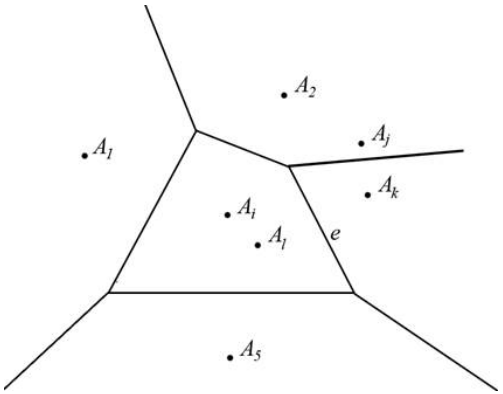


Рис 2. e – *серединный перпендикуляр к отрезкам* $[A_i, A_j]$ *и* $[A_k, A_l]$

Если предположить, что A_j не совпадает с A_k, то мы сразу получаем что все четыре точки A_i, A_j, A_k, A_l лежат на одной окружности, что противоречит общему расположению точек. Стоит отметить, что использование в доказательстве требования общего положения точек не честно, т.к. все можно доказать без него.

■

Теорема 9. *В каждую вершину многоугольника Вороного приходит ровно три ребра.*

По теореме 8 получаем, что каждый отрезок диаграммы Вороного лежит на серединном перпендикуляре к отрезку, соединяющему центры многоугольников Вороного, для которых данный отрезок является ребром. Отсюда сразу вытекает, что центры данных многоугольников равноудалены от точки пересечения ребер X. Т.о. если в точку X сходится более трех ребер, то сразу получаем противоречие с общим положением точек. Если же ребер только два, то сразу получаем, что это ребра соответствуют одним и тем же двум многоугольникам Вороного, но тогда эти ребра должны лежать на одной прямой. Это утверждение завершает доказательство.

■

Теорема 10. *Каждая вершина многоугольника Вороного является центром окружности, проходящей через три точки исходного множества $\{A_1, \ldots, A_N\}$, при этом ни одна из точек множества $\{A_1, \ldots, A_N\}$ не лежит внутри данной окружности.*

То, что каждая вершина X многоугольника Вороного является центром окружности, проходящей через три точки исходного множества, следует из теоремы 9 и ее доказательства. Допустим, внутрь данной окружности попала некоторая точка из $\{A_1, \ldots, A_N\}$. Это не центр многоугольников Вороного $A_{i1}, A_{i2}, A_{i3},$, для которых ребра, сходящиеся в X, являются ребрами (центры этих многоугольников лежат НА окружности). Но если это некоторая другая точка A_j, то получается, что она ближе к X, чем центры A_{i1}, A_{i2}, A_{i3}. Но это противоречит тому, что точка X ближе к центрам A_{i1}, A_{i2}, A_{i3}, чем к остальным точкам.

■

Теорема 11. *Многоугольники Вороного с центрами, лежащими на выпуклой оболочке исходного множества точек, бесконечны. Других бесконечных многоугольников Вороного нет.*

Пусть точка X является вершиной многоугольника выпуклой оболочки. Проведем через нее прямую l такую, что вся выпуклая оболочка лежит по одну сторону прямой. Построим луч r, выходящий из X, перпендикулярный l, лежащий в другой полуплоскости, относительно l, чем точки исходного

множества (см. Рис. 3). Не представляет труда доказать, что для всех точек этого луча R точка X находится ближе, чем все остальные точки из $\{A_1,\ldots, A_N\}$.

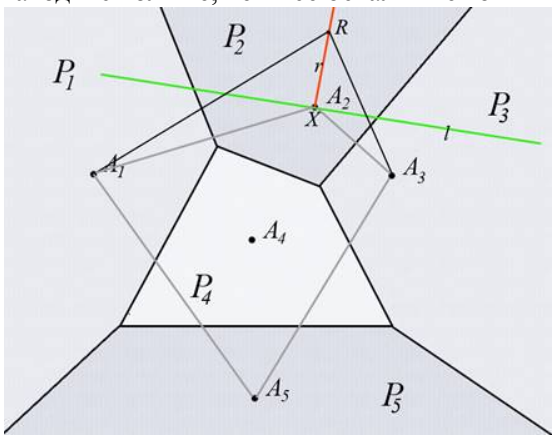


Рис 3. X является ближайшей точкой к R среди точек $\{A_1,\ldots,A_5\}$

Отсюда вытекает, что весь луч принадлежит многоугольнику Вороного точки X, следовательно, этот многоугольник бесконечный.

Пусть точка X находится внутри выпуклой оболочки точек $\{A_1,\ldots, A_N\}$. Рассмотрим последовательность вершин выпуклой оболочки исходного множества точек $\{B_1,\ldots, B_k\}$ и последовательность лучей $\{(X,B_1),\ldots, (X,B_k)\}$. Углы между соседними лучами в этой последовательности <180. Рассмотрим многоугольник T = пересечению полуплоскостей, образованных серединными перпендикулярами к отрезкам $\{[X,B_1],\ldots, [X,B_k]\}$, содержащими точку X. Многоугольник Вороного точки X принадлежит T, но T является конечным многоугольником, т.к. все его углы между всеми его ребрами <180 (легко показать, что угол между соседними ребрами многоугольника T = 180-угол между соответствующими лучами $[X,B_k]$ и $[X,B_{k+1}]$). Следовательно, многоугольник Вороного точки X тоже конечный.

■

Определение. ***Триангуляцией*** *по заданному набору точек* $\{A_1,\ldots, A_N\}$ *называется разбиение выпуклой оболочки множества точек* $\{A_1,\ldots, A_N\}$ *на непересекающиеся по внутренности треугольники таким образом, чтобы множество вершин треугольников совпадало бы с множеством точек* $\{A_1,\ldots, A_N\}$.

Теорема 12. *Рассмотрим множество треугольников, образованных тройками точек из* $\{A_1,\ldots, A_N\}$, *являющимися центрами многоугольников Вороного, имеющими одну общую вершину. Данное множество треугольников дает триангуляцию выпуклой оболочки множества точек* $\{A_1,\ldots, A_N\}$ *(Делоне 1934).*

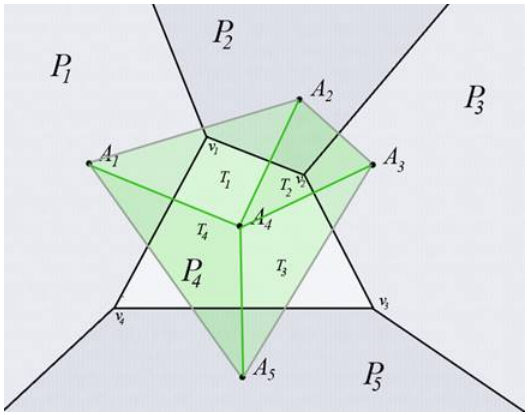


Рис 4. Многоугольники $\{P_1,\ldots,P_5\}$ диаграммы Вороного точек $\{A_1,\ldots,A_5\}$ и треугольники $\{T_1,T_2,T_3,T_4\}$, соответствующие вершинам многоугольников Вороного $\{v_1,v_2,v_3,v_4\}$

Для доказательства теоремы нам надо доказать следующие утверждения: 0)треугольники T_i не вырождены (т.е. точки треугольника не лежат на одной прямой); 1)треугольники T_i не пересекаются по внутренности; 2)для любой точки X, принадлежащей выпуклой оболочке точек $\{A_1,\ldots,A_5\}$, существует треугольник T_i : $X\hat{I}T_i$. Докажем данные утверждения.

0)Если вершины A, B, C некоторого треугольника T_i лежат на одной прямой, то серединные перпендикуляры к ним параллельны, что противоречит наличию точки их пересечения v_i.

1)Докажем, что два любых треугольника T_i,T_j не пересекаются по внутренности. Рассмотрим окружности, описанные вокруг этих треугольников, O_i,O_j с центрами в v_i,v_j. Эти окружности не могут быть одна внутри другой, т.к. это сразу противоречило бы теореме 10. Тогда, если окружности не пересекаются, или пересекаются по одной точке, то треугольники T_i,T_j не пересекаются по внутренности. Осталось рассмотреть случай, когда эти окружности пересекаются по двум точкам Q_1, Q_2.

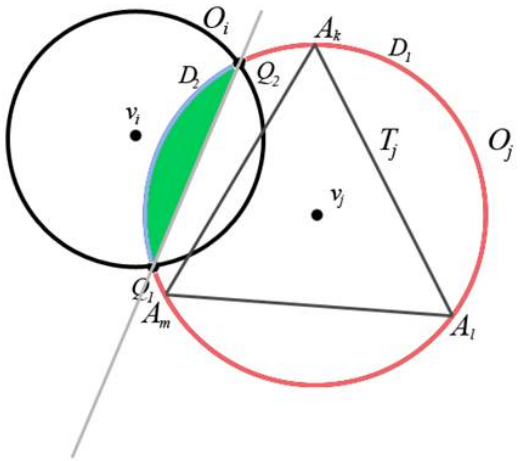


Рис 5. Две вершины диаграммы Вороного v_i,v_j с окружностями O_i,O_j, описанными вокруг вершин соответствующих вершинам v_i,v_j треугольников T_i,T_j.

Покажем, что треугольник T_j (=D(A_k,A_l,A_m)), соответствующий вершине диаграммы Вороного v_j, может лежать только по одну сторону от прямой (Q_1,Q_2). Из этого и аналогичного факта для треугольника T_i сразу вытекает, что треугольники T_i, T_j лежат по разные стороны от (Q_1,Q_2) и, следовательно, они не имеют общих внутренних точек. Пусть окружность O_j состоит из двух дуг D_1,D_2, разделенных точками Q_1,Q_2 (см. Рис. 5). Вершины треугольника D(A_k,A_l,A_m) не могут лежать на открытой дуге D_2, т.к. это противоречило бы теореме 10. Но тогда A_k,A_l,A_m лежат по одну сторону от прямой (Q_1,Q_2) и, следовательно, весь треугольник D(A_k,A_l,A_m) лежит по одну сторону от прямой (Q_1,Q_2).

2)Пусть точка x принадлежит внутренности выпуклой оболочки множества точек $\{A_1, ,A_2\ldots\}$ и не принадлежит ни одному треугольнику T_i.

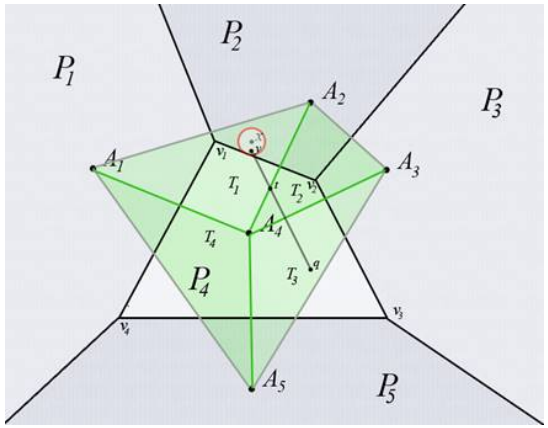


Рис 6. Пусть точка x принадлежит внутренности выпуклой оболочки множества точек $\{A_1,\ldots,A_5\}$ и не принадлежит ни одному треугольнику T_i.

Тогда и некоторая окрестность $O(x)$ точки x принадлежит выпуклой оболочке множества точек $\{A_1,A_2,A_3\ldots\}$ и не пересекается ни с одним треугольником T_i. Рассмотрим произвольный треугольник из множества $\{T_1,T_2,T_3\ldots\}$. Например (см. Рис. 6) это – T_3. Выберем в нем произвольную точку q. Выберем в $O(x)$ такую точку y, что ни одна из точек $\{A_1,A_2,A_3\ldots\}$ не лежит на прямой (q,y).

q принадлежит T_3, y не принадлежит ни одному треугольнику $T_1,T_2,T_3\ldots$. Тогда на $[q,y]$ найдется последняя точка t, принадлежащая одному из $T_1,T_2,T_3\ldots$. Пусть этот последний треугольник называется T_2 , а точка $t\hat{I}[A_2,A_4]$. Исходя из предположения, получаем, что интервал (t,y) не пересекается с множеством треугольников $\{T_1,T_2,T_3\ldots\}$. $[A_2,A_4]$ не лежит на границе выпуклой оболочки множества точек $\{A_1,\ldots,A_5\}$, т.к. по обе его стороны есть точки из выпуклой оболочки (например, y и A_3). Но тогда ребро диаграммы Вороного, выходящее из v_2, перпендикулярное $[A_2,A_4]$ должно быть конечным (по теореме 11). Пусть оно завершается в точке v_1. Но тогда треугольник T_1 опирается на отрезок $[A_2,A_4]$ и начало отрезка $[t,y]$ обязано принадлежать T_1. А это противоречит тому, что интервал (t,y) не пересекается с множеством треугольников $\{T_1,T_2,T_3\ldots\}$.

■

Классическое построение диаграммы Вороного осуществляется методом деления пополам. Основная идея алгоритма сводится к тому, что множество точек $\{A_1,A_2,A_3\ldots\}$, по которому строится диаграмма Вороного, разбивается на две [почти] равные части с помощью разбиения точек по оси абсцисс. Для каждой части запускается тот же алгоритм построения диаграммы Вороного, а потом предъявляется алгоритм объединения этих двух диаграмм Вороного в диаграмму всего множества точек за линейное время. Мы не будем описывать подробности данного алгоритма, но если поверить, что указанные действия можно осуществить, то мы сразу получаем теорему о верхней оценке времени решения задачи построения диаграммы Вороного.

Теорема 13. *В рамках алгоритма, основанных на сравнениях, диаграмму Вороного множества точек $\{A_1,\ldots, A_N\}$ можно построить за время* $O(N \log N)$*, что является верхней оценкой времени решения данной задачи в рамках алгоритмов, основанных на сравнениях.*

Определение. *Триангуляцией Делоне множества точек $\{A_1,\ldots, A_N\}$ называется такая триангуляция множества точек $\{A_1,\ldots, A_N\}$, что для любого треугольника триангуляции* T *описанная вокруг него окружность не содержит внутри никаких точек из множества $\{A_1,\ldots, A_N\}$.*

Теорема 14. *Для любого множества точек $\{A_1,\ldots, A_N\}$ триангуляцией Делоне существует. Для случая общего расположения точек $\{A_1,\ldots, A_N\}$ триангуляцией Делоне единственна.*

Существование триангуляции Делоне мы доказали (на самом деле, существование мы доказали для общего расположения точек, но существование есть и в общем случае). Единственность триангуляции Делоне следует из взаимно-однозначного соответствия между триангуляцией Делоне и диаграммой Вороного. Мы доказали это соответствие только в одну сторону. В обратную сторону доказательство проводиться не будет. Но единственность триангуляции Делоне можно доказать с помощью другого подхода.

Лемма 1. *Рассмотрим произвольную окружность* $(x-a)^2+(y-b)^2=r^2$ *на плоскости* XY*. Вертикальная проекция данной окружности на стандартный параболоид* $z=x^2+y^2$ *лежит в одной плоскости. Все точки внутри окружности проецируются в точки, лежащие ниже данной плоскости, а точки снаружи окружности – в точки выше данной плоскости.*

Распишем уравнение окружности: $x^2+y^2-2ax-2by=r^2-a^2-b^2$. При проектировании этой окружности на параболоид x^2+y^2 можно заменить на z. После чего мы получим уравнение точек, лежащих в одной плоскости: $-2ax-2by+z=r^2-a^2-b^2$.

Если некоторая (x_1,y_1) точка лежит внутри данной окружности, то для нее выполняется соотношение $(x-a)^2+(y-b)^2=p^2$, где $p<r$. Но тогда проекция данной точки на параболоид удовлетворяет соотношению $x_1^2+y_1^2-2ax_1-2by_1=p^2-a^2-b^2$, и следовательно $2ax_1-2by_1+z_1=p^2-a^2-b^2$. Откуда сразу получаем $z_1<z$, где z удовлетворяет соотношению $2ax_1-2by_1+z=r^2-a^2-b^2$. Т.е. проекция точки, лежащей внутри окружности, находится ниже плоскости, в которой лежит окружность. Доказательство для точек вне окружности аналогично. **Ч.Т.Д.**

Рассмотрим триангуляцию Делоне нашего множество точек на плоскости $\{A_1,A_2,A_3\ldots\}$ и вертикальную проекцию вершин этой триангуляции на стандартный параболоид $z=x^2+y^2$. Вершины триангуляции лежат на параболоиде. Ребра триангуляции лежат сверху параболоида. Окружность, описанная вокруг любого треугольника T данной триангуляции, не содержит внутри других точек из множества $\{A_1,A_2,A_3\ldots\}$. Отсюда из леммы 1 сразу следует, что все проекции всех остальных точек $\{A_1,A_2,A_3\ldots\}$ на параболоид лежат с верхней стороны от плоскости, проходящей через проекцию вершин T на параболоид. Но тогда получается, что объединение треугольников, вершины которых являются проекциями вершин треугольников триангуляции Делоне, дает нижнюю половину выпуклой оболочки проекций $\{A_1,A_2,A_3\ldots\}$ на параболоид. А поскольку выпуклая оболочка единственна, то мы сразу получаем единственность триангуляции Делоне.
■

Можно много говорить о значимости триангуляции Делоне и диаграммы Вороного. Например, диаграмма Вороного дает наиболее разумный способ определения кусочно-постоянной интерполяции функции для отображения $R^2\rightarrow R$. Другой хороший пример: если использовать утверждение о том, что для триангуляции количество ребер равно $O(n)$, где n – количество вершин триангуляции (это утверждение мы будем доказывать, когда будем рассматривать алгоритмы на графах), то за линейное время легко свести задачу поиска всех ближайших соседей в множестве точек $\{A_1,\ldots, A_N\}$ к задаче построения диаграммы Вороного множества точек $\{A_1,\ldots, A_N\}$. Триангуляция Делоне дает хорошую формализацию понятия *соседних вершин* в множестве точек $\{A_1,\ldots, A_N\}$. И т.д.

Лекция 6

Алгоритмы. Сведение алгоритмов.

Д.Кнут. Искусство программирования для ЭВМ. тт 1-3. Москва. Мир. 1996-1998

Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы. Построение и анализ. Москва. МЦНМО. 1999.

Препарата Ф., Шеймос М. Вычислительная геометрия. Москва. Мир. 1989

Порядковые статистики.

Определение. *Медианой множества* $A = \{a_1, \ldots, a_N\}$ *называется элемент с индексом* $(N+1)/2$ *в отсортированном по возрастанию множестве* A.

Определение. k*-той порядковой статистикой* множества из N вещественных чисел $A=\{A_1, \ldots, A_N\}$ называется k-тое число в упорядоченном множестве A. Легко увидеть, что $[(N+1)/2]$-ая порядковая статистика является медианой множества. В свою очередь, если бы мы могли эффективно искать медиану множества, то это дало бы хорошую модификацию алгоритма ***QuickSort.***

Из предыдущей главы следует, что верхней оценкой времени поиска k-той порядковой статистикой является $O(N \log_2 N)$. Оказывается, что эту оценку можно улучшить.

Поиск порядковой статистики за время $Q(N)$ **в среднем**

Для поиска порядковых статистик можно, практически один в один, применять алгоритм ***QuickSort*** с единственной модификацией: после каждого деления массива на две части мы можем точно сказать, в какой из них лежит искомая k-тая порядковая статистика, поэтому другую половину можно далее не рассматривать.

```
QFindStatP (A,p,q,k)
Если  q-p < 1 то return Ap
Вечный цикл
  i=p; j=q;
 Поменять местами Ap и случайно выбранный элемент Al , где p ≤ l ≤ q
  x=Ai
  Пока A i  < x :  i + +;
  Пока A j  > x :  j - -;
  Если  i < j то
    поменять местами A i  и A j ;
  иначе
   {Если k ≤ j
     то  return QFindStatP (A, p, j, k)
     иначе return QFindStatP (A, j+1, q, k)
   }
  i + +; j - -;
```

Конец вечного цикла

Теорема. *Время работы алгоритма* ***QFindStatP*** *равно* $O(N^2)$, *где* N – *количество элементов в обрабатываемом массиве.*

Доказательство. После каждого разбиения массива на две части длина самой большой из двух образовавшихся половин оказывается меньше либо равной длине разбиваемого массива **–1**. Поэтому на каждой ветви алгоритма будет не более N узлов (разбиений массива). На каждом уровне дерева разбиений присутствуют не более N элементов, по которым производится поиск, поэтому суммарное время работы на одном уровне дерева равно $O(N)$. Итого, суммарное время работы алгоритма равно $O(N) * N = O(N^2)$.

Данная оценка достижима на массиве $\{1, \ldots, N-1, N\}$ при поиске, например, N-ой порядковой статистики и при том, что в качестве псевдомедианы каждый раз будет выбираться первый элемент в подмассиве.

Теорема. Среднее время работы алгоритма ***QFindStatP*** равно $Q(N)$, где N – количество элементов в обрабатываемом массиве. Под средним временем подразумевается среднее время по всем перестановкам любого массива, состоящего из различных элементов входных данных длины N.

Доказательство. Выпишем рекуррентное соотношение на среднее время работы алгоритма

$$T(N) \le [\, T(N-1) + \sum_{i=2}^{i \le N} MAX(T(i-1), T(N-i+1)) \,]/N + O(N) =$$
$$= [\, T(N-1) + \sum_{i=1}^{i < N} MAX(T(i), T(N-i)) \,]/N + O(N) \le$$
$$\le [\, T(N-1) + 2\sum_{i=N/2}^{i < N} T(i) \,]/N + O(N) \le$$
$$\le [\, 2 \sum_{i=N/2}^{i < N} T(i) \,]/N + O(N)$$

Предположим, для $i<N$ верно:

$T(i) < a\,i + c$ для некоторых $a>0,\ c>0$,

тогда задача сводится к нахождению таких $a>0,\ c>0$, что для них всегда бы выполнялось соотношение

$$[\, 2 \sum_{i=N/2}^{i < N} T(i) \,]/N + O(N) < a\,N + c$$

Итак

$$T(N) \le [\, 2 \sum_{i=N/2}^{i < N} T(i) \,]/N + O(N) \le a\,3/4 * N + c + O(N)$$

Осталось взять такое большое ***a***, что ***a 3/4 * N + O(N) < a N*** , после чего мы получаем ***T(N) = O(N).*** Осталось заметить, что первое же разбиение массива на две части (а в лучшем случае оно же будет и последним) требует времени Q***(N)***, из чего мы получаем, что ***T(N)*** =Q***(N)***.

■

Поиск порядковой статистики массива целых чисел за время Q*(N)* для случая $|a_i|<O(N)$

Очень часто встречаются задачи, когда требуется искать порядковую статистику большого множества целых чисел, значения которых ограничены некоторой небольшой константой. Классическим примером таких задач является поиск порядковой статистики для некоторого подмножества пикселов серого изображения (=значения пикселов от 0 до 255). Данная задача может быть решена по аналогии с алгоритмом сортировки подсчетом.

Итак, пусть есть массив целых чисел $\{A_i\}$ ($i=0,\ldots,N-1$), $0 \le A_i<M$. Требуется вычислить порядковую статистику с номером *q*. Для работы алгоритма требуется дополнительный массив целых чисел ***B*** длины ***M.*** Алгоритм аналогичен сортировке подсчетом:

_____

QFindStatD (A,N, q, B,M)
Для всех i *от* ***0*** *до* ***M-1*** *с шагом* ***1*** ***выполнить: B[i]=0***
Для всех i *от* ***0*** *до* ***N-1*** *с шагом* ***1*** ***выполнить: B[A[i]] ++***
Для всех i *от* ***0*** *до* ***M-1*** *с шагом* ***1*** ***выполнить: ЕСЛИ q<B[i] то ВЕРНУТЬ i ИНАЧЕ q-=B[i]***

_____

Теорема. ***O(N+M)*** *является верхней оценкой времени работы алгоритма* ***QFindStatD*** *для случая* $0 \le A_i<M$, *где* ***N*** *– количество элементов в обрабатываемом массиве. Для случая* ***M=O(N)*** *верхней оценкой времени работы алгоритма является* ***O(N)***.

Поиск порядковой статистики в большой окрестности каждой точки серого изображения

Обычно под серым изображением имеется в виду массив целых чисел $A_{i,j}$ размером $N_x \times N_y$ со значениями $0 \le A_i \le 255$. Требуется для всех *I,J* найти порядковую статистику с номером *q* для множества пикселов с индексами $I-M \le i \le I+M$, $J-M \le j \le J+M$.

Если воспользоваться предыдущим алгоритмом поиска порядковой статистики, то мы получим верхнюю оценку времени работы алгоритма = $O(N_x \times N_y \times M^2)$, что будет ощутимо медленно работать в случае изображения среднего размера (1000×1000) и средних значений ***M*** (например, ***M*** больше 100).

Оказывается, данную задачу можно решить существенно более быстро. При этом окажется, что среднее время подсчета медианы будет по порядку меньше количества элементов последовательности, по которой считается медиана (!).

Для решения данной задачи модифицируем алгоритм ***QFindStatD*** следующим образом. Будем перебирать индексы пикселов, в окрестности которых считается медиана, по строкам.

Для *I=0* медиана будет считаться с помощью алгоритма ***QFindStatD***.

Далее в цикле по *I* при расчете медианы для окрестности пиксела *I,J* отметим, что в рассматриваемое множество пикселов будут добавляться точки с *i=I+M* (кроме *M* последних столбцов) и удаляться точки с *i=I-M* (кроме *M* первых столбцов).

Тогда, если мы будем хранить в процессе вычислений массив ***B*** из алгоритма ***QFindStatD***, то для каждого *I,J* его модификация потребует всего лишь ***O(M)*** операций. Здесь, конечно, следует отметить, что в оценку времени работы алгоритма войдет количество градаций яркости изображения (у нас = 256), но для больших ***M*** (у нас больше 100) это замечание несущественно.

Таким образом, мы получили алгоритм решения данной задачи с верхней оценку времени работы = $O(N_x \times N_y \times M)$, т.е. среднее время вычисления порядковой статистики в данном случае для каждой области равно квадратному корню от количества точек в рассматриваемой области.

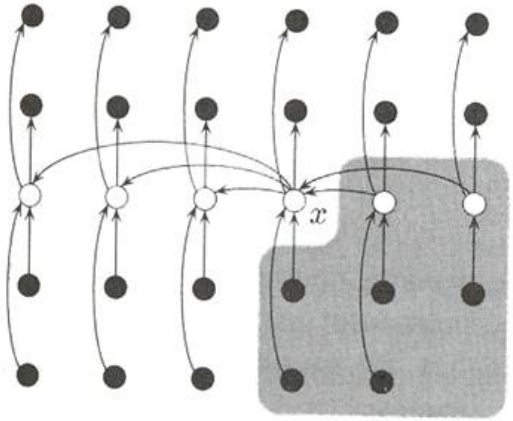
Поиск порядковой статистики за время Q*(N)* в худшем случае

Зададимся целью написать алгоритм нахождения ***k***-ой порядковой статистики, требующий Q***(N)*** операций в худшем случае. Это было бы возможным, если бы в алгоритме ***QFindStatP*** на каждом этапе разбиения множества на две части мы бы получали части размером не менее ***sL***, где ***L*** – длина разбиваемой части множества, ***s<1***. Для этой цели мы построим алгоритм ***QFindStat5,*** который перед разбиением множества на две части разбивает его на пятерки последовательных элементов, в каждой пятерке ищет медиану и на полученном множестве медиан пятерок чисел запускает самого себя для поиска медианы полученного множества. Полученную медиану медиан ***x*** алгоритм использует для разбиения множества на две части, состоящих, соответственно, из элементов меньше или равных ***x***, и из элементов больше или равных ***x***. Далее, в зависимости от ***k,*** следует применить ***QFindStat5*** к одной из полученных половин множества.

Итак, для поиска ***k***-ой статистики массива ***A***, состоящего из N элементов, следующий алгоритм надо вызывать в виде ***QFindStat5(A,1,N,k)*** ***QFindStat5(A,I,M,k)***

Если M -I+1 £ 5 то найти k-ую статистику с номером x любым методом; return x
Разбить массив A[I… M] на пятерки элементов и отсортировать элементы внутри пятерок
x= QFindStat5 (A',1,[((M -I+1)+4)/5], ([((M -I+1)+4)/5]+1)/2), где A' – массив медиан пятерок
Выполнить один шаг QuickSortP для псевдомедианы x
=> массив разбит на куски [I,L] и [L+1, M]
Если k £ L то return QFindStat5 (A,I,L,k)
иначе return QFindStat5 (A,L+1, M,k)

Теорема. *Время работы алгоритма* ***QFindStat5*** *равно* Q***(N),*** *где* ***N*** *– количество элементов в обрабатываемом массиве (****N=M-I+1****).*
Доказательство. В массиве медиан пятерок ***[(N+4)/5***] элементов. Нахождение медианы ***x*** этого множества гарантирует, что в каждой пятерке справа от ***x*** , включая пятерку с ***x*** (см. рисунок ниже), и кроме последней пятерки, ***3*** элемента больше или равны ***x*** (на рисунке эти элементы выделены серой областью). Количество всех указанных пятерок не менее ***[([(N+4)/5]+1)/2],*** последняя пятерка может быть не полной и в ней, в худшем случае, может быть всего один элемент больше или равный ***x***. Если теперь исключить из описанного множества ***x***, то получается, что мы имеем ***3[([(N+4)/5]+1)/2] - 3*** элементов больше или равных ***x***. Легко показать, что такое же количество элементов меньше или равны ***x***. Т.о. после выполнения одного шага ***QuickSortP*** останется не более ***N - 3[([(N+4)/5]+1)/2] + 3 £ N – 3N/10 + 3 = 7N/10 + 3*** элементов.
Оценим время выполнения алгоритма ***QFindStat5 : T(N).*** Оно складывается из времени поиска медианы медиан пятерок элементов ***(T([(N+4)/5])),*** времени поиска медианы в отрезанном куске множества длиной не более ***7N/10 + 3 (T(7N/10 + 3))*** и всего остального ***(O(N)).***



T(N) £ T(N/5+1) + T(7N/10 + 3) + O(N)
Теперь, если предположить, что ***T(i) £ c i***, для ***i<N***, то получим
T(N) £ c(N/5+1) + c(7N/10 + 3) + O(N) £ c(9N/10 + 4) + O(N) = = cN + (O(N) – c(N/10 - 4))
Выбрав достаточно большое с получим, что для ***N>40***
O(N) – c(N/10 - 4) £ 0
Далее, выбрав еще большее с можно получить, что ***для N£40***
T(N) £ cN

Т.о. мы получим, что ***T(N) £ cN*** для любого ***N.***

■

Язык программирования С.

Б.В. Керниган, Д.М. Ричи. Язык программирования С.

Переменные

Основные типы
В языке ***C*** есть две основные разновидности базовых типов: *целые* и *вещественные*. К целым типам относятся типы
char
short
int
long

Перед каждым вышеупомянутым типом может присутствовать слово ***signed*** или ***unsigned.*** Идентификатор ***signed*** указывает на то, что переменная имеет знак, а ***unsigned*** – на отсутствие знака у переменной. По умолчанию, все переменные имеют знак. Исключением является переменная типа ***char***. Полагаться на наличие/отсутствие знака у переменной данного типа нельзя, т.к., как правило, данное свойство может быть изменено с помощью ключей компилятора.
Известно, что в данном списке указанные типы располагаются по неубыванию их размеров. Если требуется узнать конкретный размер переменной в байтах, то он может быть получен с помощью оператора ***sizeof()***: ***sizeof(char).***
К вещественным типам относятся
float
double
Известно, что ***sizeof(float)*** £ ***sizeof(double).***
Кроме базовых типов, существуют производные типы, задающиеся с помощью понятий *массив, указатель, функция, структура, объединение.*

Базовые понятия

- *описание* и *определение*
- *время жизни*
- *область видимости* или *область действия*

Одними из основных понятий программирования являются *описание* и *определение* объектов.

Описание переменной задает ее тип и указывает на то, что переменная где-то определена. Признаком описания переменной служит наличие ключевого слова ***extern*** перед оператором, ее задающим. Отсутствие ключевого слова ***extern*** говорит об определении переменной.

Определение переменной указывает место в программе, задающее время рождения переменной. При определении переменной можно задать инициализирующее ее значение. Определение переменной всегда является ее описанием.

В программе может быть сколько угодно согласующихся описаний одной и той же переменной, но должно быть ровно одно ее определение.

Время жизни переменной задается интервалом между моментом *рождения* переменной и моментом ее *смерти*. Внутри этого интервала гарантируется сохранность значения переменной, т.е. значение переменной может быть изменено только с помощью соответствующих операторов программы (естественно, это верно только в случае правильно написанной программы). С точки зрения времени жизни, в языке ***C*** бывает два вида переменных: статические и автоматические.

Автоматические переменные всегда определяются внутри некоторого *блока*. *Блоком* называется часть *тела* некоторой *функции*, заключенная в фигурные скобки. *Телом функции* называется набор операторов, задающих действия, совершаемые функцией. В частности, *тело* функции также является блоком. Автоматические переменные всегда являются *локальными*, т.е. их область видимости ограничивается их блоком. Автоматические переменные рождаются в момент входа выполнения программы в данный блок и умирают в момент выхода из блока.

Внутри одного блока могут быть другие блоки и если во внутренних блоках определить переменные с именами, совпадающими с именами переменных во внешних блоках, то внутренние имена переменных перекроют видимость для внешних переменных. Перекрытие области видимости не сказывается на времени жизни переменных.

Автоматические переменные разрешается определять только в начале блока, перед *выполняемыми операторами*.

Автоматические переменные используют память, выделяемые в *системном стеке*, что делает процедуру отведения/очистки памяти под них весьма быстрой. Однако, как правило, системный стек имеет ограниченный размер и, поэтому, нельзя создавать автоматические переменные слишком большого размера. Причиной переполнения стека (***stack overflow***) может также служить бесконечная рекурсия, случайно созданная в программе.

Статически созданные переменные рождаются в момент запуска программы и умирают в момент прекращения работы программы. Память под них отводится в *общей куче (heap)*, т.е. в обще-используемой памяти. Ее размер ограничивается размером памяти, доступной программе. Статически созданные переменные это – либо *глобальные* переменные (т.е. переменные, определенные вне всех блоков программы), либо локальные

Осталось упомянуть о *внешних статических* переменных – глобальных переменных, область видимости которых ограничена данным файлом. Как и все глобальные переменные, эти переменные определяются вне всех блоков программы, но перед их определением написано ключевое слово ***static***.

Если есть несколько переменных с одинаковым именем, то внешние статические переменные перекрывают область видимости соответствующих глобальных переменных, а локальные переменные перекрывают область видимости соответствующих внешних статических переменных.

Изначально в языке ***C*** присутствуют *регистровые* переменные. Наличие ключевого слова ***register*** перед определением переменной служит указанием компилятору использовать внутренние машинные регистры для хранения данной переменной. На данный момент, при использовании оптимизирующих компиляторов наличие данного типа, как правило, не может служить для ускорения работы программы. Более того, обязательное использование регистра для хранения данной переменной запрещает его использование для других целей, что может послужить поводом к замедлению работы программы.

Следует упомянуть, что, как правило, использование глобальных переменных является дурным стилем программирования. Их наличие

существенно усложняет дальнейшее развитие программы и слияние написанной программы с другими, отдельно написанными кусками.

Структуры данных.

Структурой данных является реализация понятия *множества* элементов определенного типа. Под реализацией понимается способ хранения данных. Вместе со способом хранения задается набор операций (=алгоритмов) по добавлению, поиску, удалению элементов множества.

Вектор.

Массив в языке ***C*** является точной реализацией структуры данных *вектор. Вектором* называется структура данных, в которой к каждому элементу множества можно обратиться по целочисленному индексу. В структуре данных *вектор* значение индекса может быть ограничено некоторой наперед заданной величиной - *длиной вектора.* В качестве опций в соответствующий исполнитель могут быть добавлены функции проверки невыхода индекса за границу вектора, проверки того, что по данному индексу когда-то был положен элемент.

Создание исполнителя ***вектор*** предполагает наличие следующих функций

- создать вектор длины ***n***
- положить элемент в вектор по индексу ***i***
- взять элемент из вектора по индексу ***i***
- уничтожить вектор

При использовании массивов для реализации структуры данных ***вектор***, создание/уничтожение объекта происходит в соответствующие моменты автоматически. Если же этим процессом надо управлять, то следует использовать функции ***malloc() / free().***

Возможна ситуация, когда размер вектора становится известным уже после написания программы, но до ее компиляции (или мы для одной программы хотим получать ее различные версии для различных длин вектора). В этой ситуации можно задать размер массива константой препроцессора. Значение константы можно передать через ключи компилятора. Например, в программе (в файле ***prog.c***) можно записать следующий набор операторов :

```
#ifndef N
#define N  100
#endif
int Array[N] ;
```

Если константа ***N*** не определена, то ее значение полагается равным ***100.*** Далее создается массив из ***N*** элементов. У большинства компиляторов значение константы препроцессора можно передать через ключ `***D***', например, для компилятора ***gcc*** это будет выглядеть так:

```
gcc –DN=200  prog.c
```

В получившейся программе с именем ***./a.out*** везде вместо идентификатора ***N*** будет подставляться ***200***.

Стек.

Стеком называется структура данных, организованная по принципу ***LIFO –*** *last-in, first-out* , т.е. элемент, попавшим в множество последним, должен первым его покинуть. При практическом использовании часто налагается ограничение на длину стека, т.е. требуется, чтобы количество элементов не превосходило ***N*** для некоторого целого ***N***.

Создание исполнителя ***стек*** предполагает наличие следующих функций

- инициализация
- добавление элемента на вершину стека
- взятие/извлечение элемента с вершины стека
- проверка: пуст ли стек?
- очистка стека

Стек можно реализовать на базе массива или (в языке ***C***) это можно сделать на базе *указателей*.

Стек. Реализация 1.

Для реализации стека целых чисел, состоящего не более чем из ***100*** чисел, на базе массива в языке ***C*** следует определить массив целых, состоящий из ***100*** чисел и целую переменную, указывающую на вершину стека (ее значение будет также равно числу элементов в стеке)

```
int stack[100], i0=0;
```

Также, как было показано выше, максимальный размер стека можно задать после написания программы, но перед ее компиляцией, с помощью передачи соответствующей константы через ключи компилятора.

Стек. Реализация 2.

Для группировки различных переменных в один объект (например, чтобы впоследствии, так или иначе, передавать этот объект в функции за один прием) в языке ***C*** следует использовать *структуры*. Например, все данные, относящиеся к стеку можно поместить в структуру ***struct SStack***:

```
struct SStack
```

```
{
int stack[100];
int i0;
};
```

Здесь создан новый тип с именем ***struct SStack.*** Далее можно создать переменную этого типа:

struct SStack st2;

Структуры нельзя передавать в качестве параметров функций, их нельзя возвращать как результат работы функций, но во всех этих случаях можно использовать указатели на структуры. Например, следующая функция будет инициализировать наш стек:

```
void InitStack(struct SStack *ss){ ss->i0=0 ;}
```

Вызов функции осуществляется следующим способом :

```
InitStack(&st2) ;
```

Стек. Реализация 3.

Если максимальный размер стека можно выяснить только после сборки программы, то память под стек можно выделить динамически. При этом, вершину стека можно указывать не индексом в массиве, а указателем на вершину (отметим, что оба способа указания вершины стека применимы в обоих реализациях стека). Для этого следует определить следующие переменные

int *stack, *head;

Как и ранее, эти переменные можно объединить в структуру:

```
struct SStack3{ int *stack, *head; };
```

Тогда соответствующую переменную ***st3*** можно определить оператором

```
struct SStack3 st3;
```

Стек. Реализация 4.

Однако, можно поступить и по-другому. Т.к. элементы ***stack*** и ***head*** имеют один тип, то их можно объединить в один массив объектов соответствующего типа (т.е. типа ***int******). Массив, естественно, должен быть длины ***2***:

```
int *st4[2];
```

Здесь следует заметить, что при определении/описании переменных квадратные скобки имеют приоритет больший, чем *******, поэтому переменная ***st4*** имеет тип `массив указателей', а не `указатель на массив'.

Функция создания стека не более чем из ***n*** элементов может выглядеть, в простейшем случае, следующим образом

```
void StackCreate4(int n, int *st[2] ) {st[1]= st[0] =
(int*)malloc(n*sizeof(int));}
```

а ее вызов будет выглядеть так: ***StackCreate4(n,st4);***

Простейшая функция добавления элемента к стеку может выглядеть, в простейшем случае (без проверки переполнения), следующим образом

```
void StackAdd4(int v, int * st[2] ) { (*(st[1]++)) = v;}
```

а ее вызов будет выглядеть так: ***StackAdd4 (v, st4);***

Проверка стека на пустоту выглядит следующим образом :

```
int StackIsEmpty4 ( int * st[2] ) { return st[1]<=st[0] ; }
```

Стек. Реализация 5.

У ***Реализации 4*** есть существенный недостаток. Допустим, что стек создан внутри некоторой функции и требуется использовать его вне данной функции. Тогда у нас есть единственная возможность осуществить данную реализацию, это - сделать переменную ***st4*** глобальной или локальной статической. В противном случае, при выходе из данной функции переменная ***st4*** утратит свое существование и указателями ***st4[0], st4[1]*** уже нельзя будет пользоваться. Но, как уже писалось, подобный способ реализации является дурным стилем.

Собственно, вся наша проблема состоит в том, что память под переменную ***st4*** отводится и очищается автоматически. В качестве альтернативы, отведение/очистку памяти под указатели можно взять на себя. Для этого следует использовать указатель на указатель на целую переменную:

```
int **st5;
```

Функция создания стека не более чем из ***n*** элементов может выглядеть, в простейшем случае, следующим образом

int ** StackCreate5(int n)
{int **st; st = (int**)malloc(2*sizeof(int*)); st[1]= st[0] = (int*)malloc(n*sizeof(int));}

а ее вызов будет выглядеть так: ***st5=StackCreate5(n);***
Теперь переменная ***st5*** может быть локальной и если ее вернуть из функции, то содержимое стека не будет потерянным. Очистку стека можно произвести с помощью следующей функции

void StackDelete5(int **st) { free(st[0]); free(st);}
а ее вызов будет выглядеть так: ***StackDelete5 (st5);***

Лекция 7

Структуры данных (+ в языке С: массивы, структуры, оператор *typedef*).

Д.Кнут. Искусство программирования для ЭВМ. тт 1-3. Москва. Мир. 1996-1998
Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы. Построение и анализ. Москва. МЦНМО. 1999.

Структурой данных является реализация понятия *множества* элементов определенного типа. Под реализацией понимается способ хранения данных. Вместе со способом хранения задается набор операций (=алгоритмов) по добавлению, поиску, удалению элементов множества.

Стек.

Стеком называется структура данных, организованная по принципу ***LIFO –*** *last-in, first-out* , т.е. элемент, попавшим в множество последним, должен первым его покинуть. При практическом использовании часто налагается ограничение на длину стека, т.е. требуется, чтобы количество элементов не превосходило ***N*** для некоторого целого ***N***.

Создание исполнителя ***стек*** предполагает наличие следующих функций

- инициализация
- добавление элемента на вершину стека
- взятие/извлечение элемента с вершины стека
- проверка: пуст ли стек?
- очистка стека

Стек можно реализовать на базе массива или (в языке ***C***) это можно сделать на базе *указателей*.

Стек. Реализация 1 (на основе массива).
Для реализации стека целых чисел, состоящего не более чем из ***100*** чисел, на базе массива в языке ***C*** следует определить массив целых, состоящий из ***100*** чисел и целую переменную, указывающую на вершину стека (ее значение будет также равно числу элементов в стеке)

int stack[100], i0=0;

Также, как было показано выше, максимальный размер стека можно задать после написания программы, но перед ее компиляцией, с помощью передачи соответствующей константы через ключи компилятора.

Стек. Реализация 2 (на основе массива с использованием общей структуры).
Для группировки различных переменных в один объект (например, чтобы впоследствии, так или иначе, передавать этот объект в функции за один прием) в языке ***C*** следует использовать *структуры*. Например, все данные, относящиеся к стеку можно поместить в общую структуру ***struct SStack***:

```
struct SStack
{
int stack[100];
int i0;
};
```

Здесь создан новый тип с именем ***struct SStack.*** Далее можно создать переменную этого типа:

struct SStack st2;

Структуры нельзя передавать в качестве параметров функций, их нельзя возвращать как результат работы функций, но во всех этих случаях можно использовать указатели на структуры. Например, следующая функция будет инициализировать наш стек:

void InitStack(struct SStack *ss){ ss->i0=0 ;}

Вызов функции осуществляется следующим способом :

InitStack(&st2) ;

Стек. Реализация 3 (на основе указателей).

Если максимальный размер стека можно выяснить только после сборки программы, то память под стек можно выделить динамически. При этом, вершину стека можно указывать не индексом в массиве, а указателем на вершину (отметим, что оба способа указания вершины стека применимы в обоих реализациях стека). Для этого следует определить следующие переменные

int *stack, *head;

Как и ранее, эти переменные можно объединить в структуру:

struct SStack3{ int *stack, *head; };

Тогда соответствующую переменную ***st3*** можно определить оператором

struct SStack3 st3;

Стек. Реализация 4 (на основе массива из двух указателей).

Однако, можно поступить и по-другому. Т.к. элементы ***stack*** и ***head*** имеют один тип, то их можно объединить в один массив объектов соответствующего типа (т.е. типа ***int****). Массив, естественно, должен быть длины ***2***:

int *st4[2];

Здесь следует заметить, что при определении/описании переменных квадратные скобки имеют приоритет больший, чем *******, поэтому переменная ***st4*** имеет тип `массив указателей', а не `указатель на массив'.

Функция создания стека не более чем из ***n*** элементов может выглядеть, в простейшем случае, следующим образом

void StackCreate4(int n, int *st[2]) {st[1]= st[0] = (int*)malloc(n*sizeof(int));}

а ее вызов будет выглядеть так: ***StackCreate4(n,st4);***

Простейшая функция добавления элемента к стеку может выглядеть, в простейшем случае (без проверки переполнения), следующим образом

void StackAdd4(int v, int * st[2]) { (*(st[1]++)) = v;}

а ее вызов будет выглядеть так: ***StackAdd4 (v, st4);***

Проверка стека на пустоту выглядит следующим образом :

int StackIsEmpty4 (int * st[2]) { return st[1]<=st[0] ; }

Стек. Реализация 5 (на основе указателя на указатель).

У ***Реализации 4*** есть существенный недостаток. Допустим, что стек создан внутри некоторой функции и требуется использовать его вне данной функции. Тогда у нас есть единственная возможность осуществить данную реализацию, это - сделать переменную ***st4*** глобальной или локальной статической. В противном случае, при выходе из данной функции переменная ***st4*** утратит свое существование и указателями ***st4[0], st4[1]*** уже нельзя будет пользоваться. Но, как уже писалось, подобный способ реализации является дурным стилем. Собственно, вся наша проблема состоит в том, что память под переменную ***st4*** отводится и очищается автоматически. В качестве альтернативы, отведение/очистку памяти под указатели можно взять на себя. Для этого следует использовать указатель на указатель на целую переменную:

int **st5;

Функция создания стека не более чем из ***n*** элементов может выглядеть, в простейшем случае, следующим образом

```
int ** StackCreate5(int n )
{int **st; st = (int**)malloc(2*sizeof(int*));
 st[1]= st[0] = (int*)malloc(n*sizeof(int));
}
```

а ее вызов будет выглядеть так: ***st5=StackCreate5(n);***

Теперь переменная ***st5*** может быть локальной и если ее вернуть из функции, то содержимое стека не будет потерянным. Очистку стека можно произвести с помощью следующей функции

void StackDelete5(int **st) { free(st[0]); free(st);}

а ее вызов будет выглядеть так: ***StackDelete5 (st5);***

Стек. Реализация 6 (на основе указателя на указатель с одинарным выделением памяти).

Реализацию 5 можно немного улучшить, избавившись от выделения памяти в два этапа. Мы можем за один раз выделить память под указатель на вершину стека, под указатель на конец отведенной памяти (для контроля за завершением места в стеке) и под данные стека, которые последуют сразу за указателем на вершину стека. При этом в начале выделенного куска будет храниться указатель

на вершину стека, далее – указатель на место после отведенной памяти, а начало стека будет лежать сразу за данными указателями:

```
int **st6=NULL;
```

Функция создания стека не более чем из ***n*** элементов может выглядеть следующим образом:

```
int ** StackCreate6(int n )
{int **st; st = (int**)malloc(sizeof(int*)*2+n*sizeof(int));
 st[0] = (int *)(st+2);
 st[1] = st[0]+n;
}
```

а ее вызов будет выглядеть так: ***st6=StackCreate6(n);***

Как и в предыдущем случае, переменная ***st6*** может быть локальной и если ее вернуть из функции, то содержимое стека не будет потерянным. Очистку стека можно произвести с помощью следующей функции

```
void StackDelete6(int  ***st ) { free(*st);*st=NULL;}
```

а ее вызов будет выглядеть так: ***StackDelete6 (&st6);***

Заметим, что здесь при уничтожении стека мы не просто очищаем память, но и обнуляем указатель на стек. Т.о. у нас будет четко выдержан принцип: либо указатель нулевой, либо он не нулевой и указывает на существующий стек.

Добавлять элемент в стек и извлекать элемент из стека можно следующими функциями:

```
int StackPush6(int **s, int v )
{
 if(s[0]>=s[1])return -1;//стек заполнен
  *(s[0]++)=v;
 return 0;
}
int StackPop6(int **s, int *v )
{
 if(s[0]<=(int*)(s+2))return -1;//стек пуст
  *v=*(--s[0]);
   return 0;
}
```

Очередь.

Очередью называется структура данных, организованная по принципу ***FIFO –*** *first-in, first-out* , т.е. элемент, попавшим в множество первым, должен первым его и покинуть. При практическом использовании часто налагается ограничение на длину очереди, т.е. требуется, чтобы количество элементов не превосходило ***N*** для некоторого целого ***N***.

Создание исполнителя ***очередь*** предполагает наличие следующих функций

- инициализация
- добавление элемента в конец очереди
- взятие/извлечение элемента с начала очереди
- проверка: пуста ли очередь?
- очистка очереди

Обычно, используется реализация *циклической очереди*. Т.е. под очередь отводится некоторый непрерывный кусок памяти (массив) и при подходе хвоста очереди к концу массива хвост автоматически перебрасывается на противоположный конец массива. Например, для реализации стандартной очереди из менее чем 100 целых чисел на базе массива в языке С следует определить следующие данные

```
#define  N  100
int array[N], begin=N-1,end=N-1;
```

Было бы логичным объединить все эти данные в единую структуру:

```
struct SQueue
{
 int Array[N];
 int  Begin, End;
};
```

Тогда функция инициализации очереди может выглядеть

```
void  Init(struct SQueue *queue){ queue->Begin=queue->End=N-1;}
```

Функция добавления элемента может выглядеть следующим образом

```
int  Add(struct SQueue *queue, int value)
{
 if(queue->End - queue->Begin == 1 ||
    queue->Begin - queue->End == N-1)return -1;
  queue->Array[queue->End-- ]=value;
  if(queue->End<0) queue->End=N-1;
return 0;
}
```

Отметим, что при данной реализации один элемент в очереди всегда будет не использован.

Функция извлечения элемента с уничтожением может выглядеть следующим образом

```
int  Get(struct SQueue *queue, int *value)
{
if(queue->Begin== queue->End)return -1;
 *value= queue->Array[queue->Begin];
if((--queue->Begin)<0) queue->Begin=N-1;
 return 0;
}
```

В функции сначала проводится проверка очереди на пустоту, далее из нее извлекается элемент. Если необходимо, индекс начала очереди перебрасывается на конец массива.

Дек.

Деком называется структура данных, представляющих собой упорядоченное множество элементов, в котором элементы можно добавлять в начало и конец множества и извлекать их можно с обеих сторон.

Создание исполнителя ***дек*** предполагает наличие следующих функций

- инициализация
- добавление элемента в начало дека
- добавление элемента в конец дека
- взятие/извлечение элемента из начала дека
- взятие/извлечение элемента с конца дека
- проверка: пуст ли дек?
- очистка дека

Аналогично очереди, обычно используются циклические реализации дека.

Списки

Как правило, рассматриваются *односвязные* и *двусвязные списки*. Данные в списках представляют собой некоторым способом упорядоченное множество. В множестве вводится понятие *текущего элемента*. В каждый момент можно получать данные только о текущем элементе. За одну операцию можно выбрать в качестве текущего элемента первый элемент в списке. Далее за одну операцию можно переместиться к следующему или предыдущему (для двусвязного списка) элементу. Можно стереть текущий элемент или вставить новый элемент вслед за текущим.

Более конкретно, создание исполнителя ***односвязный список*** (***L1-список***) предполагает наличие следующих функций

- инициализация
- установка текущего элемента в начало списка
- перемещение текущего элемента к следующему элементу списка
- взятие значения текущего элемента
- уничтожение текущего элемента с автоматическим перемещением текущего элемента к следующему элементу списка
- вставка нового элемента после текущего
- проверка: пуст ли список?
- проверка: текущий элемент в конце списка?
- очистка списка

Двусвязный список отличается от односвязного наличием дополнительных операций:

- перемещение текущего элемента к предыдущему элементу списка
- вставка нового элемента перед текущим
- проверка: текущий элемент в начале списка?

Иногда используются *циклические списки*. В них ссылки на концах списка циклически замыкаются. Хотя, формально, в данной структуре данных нет начала и конца, тем не менее, понятие *первого элемента списка* следует оставить, т.к. без него сложно, например, реализовать перебор всех элементов списка. Список предписаний несколько модифицируется. Например, для двусвязного списка он может выглядеть следующим образом:

- инициализация
- установка текущего элемента в первый элемент списка
- перемещение текущего элемента к следующему элементу списка
- перемещение текущего элемента к предыдущему элементу списка
- взятие значения текущего элемента
- уничтожение текущего элемента с автоматическим перемещением текущего элемента к следующему элементу списка
- вставка нового элемента после текущего
- вставка нового элемента перед текущим
- проверка: пуст ли список?
- проверка: текущий элемент первый в списке?
- очистка списка

Стандартная ссылочная реализация списков

Обычно, используется ссылочная реализация списков. Для этого следует создать объект, содержащий собственно данные, ссылку на следующий аналогичные объект и, для L2-списков, ссылку на предыдущий объект. Например, для реализации двунаправленного списка целых чисел на языке С следует определить следующий тип

```
struct SList2
{
 int value;
struct SList2 *prev, *next;
};
```

здесь данные будут храниться в члене структуры ***value***, а ссылки, соответственно, на предыдущий/следующий объекты представлены указателями ***prev*** и ***next***. Признаком того, что данный элемент является первым в списке, может служить нулевое значение указателя ***prev***, а признаком конца списка может служить нулевое значение указателя ***next.***

Для сокращения имени типа в языке С можно использовать оператор ***typedef***. Принцип работы данного оператора следующий. Если перед определением некоторой переменной с именем ***NAME*** написать оператор ***typedef***, то ***NAME*** из имени переменной превратится в имя нового типа. Т.о. следующий оператор создает новый тип ***TList2,*** который можно использовать вместо типа ***struct SList2***:

```
typedef struct SList2 TList2;
```

Для работы со списком следует определить два указателя: указатель на головной элемент списка и указатель на текущий элемент списка:

```
TList2 *head=NULL, *current=NULL;
```

Признаком пустоты списка служит нулевое значение указателя ***head.*** Установка текущего элемента на начало списка сводится к присвоению

```
current=head;
```

Вставка элемента со значением ***value*** после текущего может быть произведена следующим образом:

```
TList2 *InsertData(int value, TList2 **head, TList2 **current)
{
 TList2 *New=( TList2 *)malloc(sizeof(TList2));
 New->value = value;
 if(*head == NULL) //Случай пустого списка
 {*current =*head = New; (*head)->next= (*head)->prev=NULL; return New;}
 if(*current==NULL) //Случай вставки в начало списка
 {
  (*current)=New; (*current)->next=*head; (*current)->prev=NULL;
  (*head)->prev=New;
  (*head)=New;
  return New;
 }
 New->next = (*current)->next; New->prev=*current;
 if((*current)->next != NULL) (*current)->next->prev = New;
 (*current) ->next = New;
return New;
}
```

функция возвращает указатель на новый элемент. Текущее положение в списке не изменяется. Для того, чтобы при вызове указатель на текущий элемент перемещался бы на новый элемент, данная функция может быть вызвана следующим образом

```
current = InsertData( value, &head, &current);
```

Здесь следует отметить, что в функции могут изменяться значения указателей на головной и текущий элементы списка. Особенностью языка С является то, что параметры передаются в функции исключительно *по значению*, т.е. в функцию передаются копии переменных. Альтернативой, в других языках программирования (например С++) служит передача параметров *по ссылке.* При передаче параметров по ссылке реально передается не значение переменной, а ее адрес. При этом внешний вид (синтаксис) как *формальных*, так и *фактических* параметров не отличается от случая передачи параметров по значению (*фактическими параметрами* называются параметры, передаваемые снаружи при вызове функции, а *формальными* – параметры, встречающиеся в определении функции). Для указания того, что переменная передается по ссылке, в языке С++ используется символ ***&*** перед именем переменной в описании параметров функции. Поэтому в языке С++ вышеуказанная функция может выглядеть чуть проще

```
TList2 *InsertData(int value, TList2 *&head, TList2 *&current)
{
 TList2 *New=( TList2 *)malloc(sizeof(TList2));
 New->value = value;
 if(head == NULL) //Случай пустого списка
 {current =head = New; head->next= head->prev=NULL; return New;}
 if(current==NULL) //Случай вставки в начало списка
 {
  current=New; current->next=head; current->prev=NULL;
  head->prev=New;
  head=New;
  return New;
 }
 New->next = current->next; New->prev=current;
 if(current->next != NULL) current->next->prev = New;
 current ->next = New;
```

return New;
}
Вызывать эту функцию надо следующим образом
InsertData(value, head, current);

Для полноты описания следует отметить, что в языке C++ структуры можно передавать в функции по значению.

Возможно, в силу невозможности передачи параметров по ссылке, в зыке С нельзя передавать структуры в функции в качестве параметров. Вместо этого, естественно, можно передавать указатели на структуры. Ситуация с массивами выглядит несколько двусмысленно. С одной сторону, массивы, в чистом виде, передаются по ссылке (т.е. при указании массива в качестве параметров функции передается не копия массива а адрес его нулевого элемента), но, с другой стороны, если вспомнить, что массивы, по всем своим свойствам являются постоянными указателями, то все сразу становится ясным: передавая массив в качестве параметра, мы передаем указатель на нулевой элемент массива и, при этом, сам указатель передается по значению.

Для правильного понимания понятия `структура' следует иметь представление о *выравнивании* в структурах. Имеется в виду следующее. При определении структуры гарантируется, что первый член структуры будет располагаться по адресу самой структуры и все члены структуры располагаются в памяти в порядке их расположения в структуре. Однако, размер структуры может не совпадать с суммой размеров членов структуры. Это объясняется тем, что, обычно, адреса всех элементов в структуре должны иметь определенную четность – т.е. они должны быть кратны одному из значений 1, 2, 4, 8 и т.д. Это значение определяет *выравнивание элементов* в структуре (*Structure Alignment*). Например, если создать структуру

struct SS{char a; int b;};

то, с большой вероятностью, по умолчанию, ее размер будет равен 8 (т.е. ***sizeof(struct SS) = = 8).*** Поведение по умолчанию, как правило, можно изменить с помощью соответствующих ключей компилятора. Но это можно сделать не на всех машинах и не на всех компиляторах.

Ссылочная реализация списков с фиктивным элементом

Алгоритмы работы со списком можно упростить, если использовать *фиктивный элемент в списке*. Фиктивный элемент будет играть роль разделителя между началом и концом списка. При этом, в качестве указателя на головной элемент списка будет выступать указатель на следующий элемент за фиктивным. В последнем элементе списка указатель на следующий элемент и в первом элементе указатель на предыдущий элемент должны указывать на фиктивный элемент. При этом, скорость работы алгоритмов повышается, но требуется память под лишний элемент списка.

Для локализации всех данных, относящихся к одному списку, можно завести следующую структуру

```
typedef struct TList_
{
 TList2 temp;
 TList2 *current;
}TList;
```

В данном определении типа мы объединили определение типа ***struct TList_*** и определение нового типа ***TList*** с помощью оператора ***typedef***. При этом в большинстве реализаций языка С (но не во всех!!!) идентификатор ***TList_*** можно вообще не писать.

В качестве инициализации списка можно использовать следующую функцию

```
void ListInit(TList *list)
{list->temp.next=list->temp.prev=&list->temp; list->current=&list->temp;}
```

Вставка элемента со значением ***value*** после текущего может быть произведена следующим образом:

```
TList2 *InsertData(int value, TList *list)
{
 TList2 *New=( TList2 *)malloc(sizeof(TList2));
 New->value = value;
 New->next = list->current->next;  New->prev=list->current;
 list->current->next->prev = New;  list->current ->next = New;
 return New;
}
```

Реализация L2-списка на основе двух стеков

Интересной реализацией L2-списка является его реализация на основе двух стеков, расположенных `вершиной друг к другу'. Первый из стеков представляет начало списка (часть от начала списка до текущего элемента включительно), а второй – конец (часть после текущего элемента). Текущий элемент списка лежит на вершине первого стека. При необходимости переместиться к следующему элементу, значение вершины второго стека извлекается и помещается на вершину первого стека. При необходимости переместиться к предыдущему элементу, значение вершины первого стека извлекается и помещается на вершину второго стека.

Данная реализация активно применялась, например, при создании текстовых редакторов. Текстовый редактор представляет собой двунаправленный список строк. Работа с каждой строкой может быть представлена как работа с простым вектором символов. Операции вставки, редактирования, удаления символов в строке могут быть реализованы за время = ***O(N)***, где ***N*** – количество символов в строке.

Список строк реализуется в виде двух стеков, каждый из которых может быть реализован в виде временного файла. Текущая строка (и, возможно несколько соседних строк) хранятся в оперативной памяти, начало файла – в виде одного временного файла, конец (в обратном порядке) – в виде второго временного файла.

Реализация L2-списка с обеспечением выделения/освобождения памяти

При работе со списками может оказаться, что работа функции ***malloc*** требует слишком много времени. К тому же, эта функция реально выделяет памяти больше, чем запрошено, что делает ее применение слишком дорогим. Возможна реализация списка, в которой память под весь список выделяется сразу как массив элементов списка. На основе данного массива строятся два списка – основной список и список свободного места. В начальный момент список свободного места объединяет все элементы созданного массива.
Теперь для добавления элемента в список требуется взять его из списка свободного места и добавить в основной список. Для извлечения элемента из списка элемент извлекается из основного списка и добавляется в список свободного места.
Если при добавлении элементов в список место в списке свободного места закончилось, то можно либо выдать сообщение об ошибке, либо выделить еще некоторое количество ячеек памяти для хранения дополнительных свободных элементов и добавить их к списку свободного места.
Итак, для хранения всех данных, относящихся к подобному списку можно использовать следующую структуру

```
typedef struct
{
 TList2 *array;
 TList data, empty;
 int NMax;
} SListMem;
```

Функция инициализации может выглядеть следующим образом

```
void SListMemInit( SListMem *list)
{int i;
 list->NMax=1000;
 list->array=(TList2*)malloc(list->NMax*sizeof(TList2));
 ListInit(&(list->data));
 list->empty.current=list->empty.temp.next=list->array+0;
 list->empty.temp.prev=list->array+ list->NMax-1;
 list->array[0].prev=&(list->empty.temp);
 for(i=1;i<list->NMax;i++)
 {
  list->array[i-1].next=list->array+i;
  list->array[i].prev=list->array+i-1;
 }
 list->array[list->NMax-1].next=&(list->empty.temp);
}
```


Лекция 8

Структуры данных. Графы.

Д.Кнут. Искусство программирования для ЭВМ. тт 1-3. Москва. Мир. 1996-1998
Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы. Построение и анализ. Москва. МЦНМО. 1999.

Графы

Определение. ***Графом*** называется пара ***G=(V,E)***, где ***V*** - множество объектов произвольной природы, называемых ***вершинами*** (vertices, nodes), а ***E*** - семейство пар $e_i=(v_{i1}, v_{i2})$, $v_{ij} \hat{I} V$, называемых ***ребрами*** (edges). В общем случае множество ***V*** и/или семейство ***E*** могут содержать бесконечное число элементов, но мы будем рассматривать только *конечные графы*, т.е. графы, у которых как ***V***, так и ***E*** конечны. Отметим, что, вообще говоря, для каждой пары вершин ребра в семействе ***E*** могут быть неуникальными, что не дает возможности назвать ***E*** множеством.

Если порядок элементов, входящих в e_i, имеет значение, то граф называется **ориентированным** (directed graph), сокращенно - орграф (digraph), иначе - **неориентированным** (undirected graph). Ребра орграфа называются **дугами** (arcs).

Некоторая вершина и некоторое ребро графа называются ***инцидентными***, если вершина является одной из вершин ребра. Соответственно, пара вершин инцидентна ребру, если эти вершины являются вершинами одного ребра, а сами вершины называются ***смежными***. Два ребра графа называются ***смежными,*** если они инцидентны одной вершине.

Поиск пути в графе с наименьшим количеством промежуточных вершин

Пусть дан некоторый конечный граф ***G=(V,E)*** и две его вершины ***a,b*** $\hat{I} V$. Требуется найти путь между вершины графа ***a*** и ***b*** с минимальным количеством промежуточных вершин, т.е. требуется найти последовательность $\{a_1,\ldots,a_n\}$, где $a_i \hat{I} V$, $(a_i, a_{i+1}) \hat{I} E$ с минимально возможным ***n***.

Стандартным решением данной задачи является *алгоритм волны*. Базовым понятием в нем является *волна степени* ***n*** – множество вершин графа, до которых можно добраться из вершины ***a*** за ***n*** шагов и нельзя добраться за меньшее количество шагов. Будем называть это множество вершин S_n. Для точки ***a*** волной степени ***0*** является множество, состоящее из одной точки ***a***.

Основная идея алгоритма заключается в том, что волной степени ***n+1***, при известной волне степени ***n***, будет являться множество точек S'_{n+1}, состоящее изо всех точек, смежных к точкам из волны степени ***n,*** которые при этом не состоят в волнах степени меньше или равной ***n.*** Т.о. утверждается, что $S'_{n+1}= S_{n+1}$.

Доказательство этого факта тривиально. Докажем, что $S'_{n+1} \subseteq S_{n+1}$. Действительно, до точек из S'_{n+1} можно добраться за ***n+1*** шаг по построению. За меньшее количество шагов добраться нельзя, т.к. для любого $i \le n$ выбранные точки не принадлежат S_i.

Докажем, что $S_{n+1} \subseteq S'_{n+1}$. Допустим, что найдется точка ***x***, которую мы не включили в созданное множество S'_{n+1}, но $x \in S_{n+1}$. По условию $x \in S_{n+1}$ имеем, что существует путь к данной точке за ***n+1*** шагов, а значит, у данной точки есть смежная вершина из S_n. По определению S_{n+1}, до точки ***x*** нельзя добраться за ***n*** или менее шагов, поэтому ***x*** должна быть включена в создаваемое множество, согласно его построению.

■

Осталось завести два исполнителя *множество* S_0 и S_1 (например, на базе вектора, количество элементов в котором не превосходит количество вершин графа) и добавить в каждую вершину графа ***x*** дополнительную целую переменную l_x, которая будет указывать длину кратчайшего пути от вершины ***a*** до данной вершины. В начальный момент S_0 и S_1 должны быть пусты, а все значения $l=-1$. Первая часть алгоритма состоит в определении для каждой вершины ***x***, до которой можно добраться из ***a*** медленнее, чем до ***b***, значения l_x:

Алгоритм волны. Часть 1.

---

n=0; S_1= {(a,0)}
Вечный цикл
n++
S_0= S_1
S_1=Æ
Для всех x$\in$ S_0
Для всех точек, смежных x: y, таких что l_y==-1
l_y=n; занести y в S_1;
Если b==x то ВЫЙТИ ИЗ ВЕЧНОГО ЦИКЛА
Если S_1 пусто то ВЫЙТИ ИЗ ВЕЧНОГО ЦИКЛА

Если после выхода из алгоритма S_1 пусто, то это говорит о том, что из ***a*** в ***b*** добраться нельзя.

Иначе, во второй части алгоритма следует по результатам построений из первой части алгоритма определить кратчайший (один из кратчайших) путь из ***b*** в ***a***. Для этого следует заполнить искомую последовательность $\{x_1,\ldots,x_n\}$.

$x_n=b;\ x_1=a$

Далее для каждого ***i*** от ***n*** до ***3*** ищется вершина x_{i-1}, смежная к x_i, такая что $l_{x(i-1)}==i-1$. Такая вершина существует из соображений, приведенных выше.

■

Если в постановке задачи рассматривается ориентированный граф, то ничего принципиального в алгоритме не изменяется.

В каждой вершине можно хранить дополнительную информацию о том, откуда мы в нее пришли, тогда вторая часть алгоритма будет выполняться за время ***O(n),*** где ***n*** – длина минимального пути из ***a*** в ***b*** (в случае возможности дойти из ***a*** в ***b***), или за время ***O(N)*** , где ***N*** – количество вершин в графе (в любом случае).

Приведем пример программы на C++ с использованием STL для решения данной задачи. Программа распечатывает найденный путь (номера вершин в обратном порядке) и длину пути.

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
struct SPoint{SPoint(){l=iback=-1;} double x,y; int l,iback;};
struct SEdge{int i0,i1;};

ifstream &operator>>(ifstream &f,SPoint &p){f>>p.x>>p.y; return f;}
ifstream &operator>>(ifstream &f,SEdge &p){f>>p.i0>>p.i1; return f;}

int main(void)
{vector<SPoint> p; vector<SEdge> e; vector<vector<int>> a; int l;
 vector<int> v0,v1;//== Sn S(n+1)
 int i0=0,i1=11;
 {SPoint P; ifstream f("vertex.txt"); while(f>>P)p.push_back(P);}
 {SEdge E; ifstream f("edges.txt"); while(f>>E)e.push_back(E);}
 //--
 a.resize(p.size());
```

```
 for(vector<SEdge>::iterator i=e.begin(); i!=e.end(); i++)
 {a[i->i0].push_back(i->i1);a[i->i1].push_back(i->i0);}
 //--
 for(v0.push_back(i0), p[i0].l=0, l=1; !v0.empty(); v0.swap(v1), l++, v1.clear())
  for(vector<int>::iterator i=v0.begin(); i!=v0.end(); i++)
   for(vector<int>::iterator j=a[*i].begin(); j!=a[*i].end(); j++)
    if(*j==i1)
    {
     cout<<"l="<<l<<endl;
     p[*j].iback=*i;
     for(int k=*j;k>=0;k=p[k].iback)cout<<k<<" ";cout<<endl;
     goto le;
    }
    else if(p[*j].l<0)
    {p[*j].l=l;p[*j].iback=*i;v1.push_back(*j);}
//--
    cout<<"can't find the way"<<endl;
le:
    getchar();
 return 0;
}
```

Можно задаться вопросом о времени работы алгоритма волны. Легко видеть, что без дополнительных предположений это время нельзя оценить с помощью функции от ***n***, т.к. количество ребер, инцидентных вершине не ограничено (здесь имеется в виду, что одна пара вершин может иметь неограниченное количество инцидентных ребер). Тривиальна следующая

Теорема 1. *Верны оценки для графа, состоящего из **N** вершин*

1. *Если в графе отсутствуют петли и кратные ребра, то алгоритм волны работает за время **O(N²).***

2. *Если каждая вершина графа имеет не более M_1 инцидентных ребер, то алгоритм волны работает за время $O(M_1 N)$.*

Отметим, что ребро графа называется ***кратным***, если для пары вершин, инцидентных ребру, существует более одного ребра, инцидентного этой паре вершин. Ребро называется ***петлей***, если вершины, инцидентные ребру, совпадают.

Доказательство теоремы сводится к тому, что время работы алгоритма волны равно ***O(M+N)*** , где ***N*** – количество вершин, ***M*** – количество ребер в графе (точнее, не ребер, а пар смежных вершин, но в условиях теоремы это – одно и тоже). Последний факт следует из того, что каждое ребро при взятии смежной вершины в алгоритме может встретиться не более двух раз.

Чтобы получить более оптимистичную оценку введем несколько новых понятий.

Пусть дан некоторый граф ***G=(V,E)***, пусть каждой его вершине сопоставлена точка в некотором евклидовом пространстве, причем точки, соответствующие различным вершинам не совпадают. Пусть каждому ребру графа сопоставлена некоторая непрерывная кривая, соединяющая соответствующие вершины графа, причем кривые, соединяющие различные пары точек не пересекаются нигде, кроме вершин графа. Такое представление графа называется его ***геометрическим представлением.***

Теорема 2. *Для любого конечного графа существует его геометрическое представление в **R^3**.*

Доказательство. Рассмотрим произвольный отрезок ***[a,b]*** $\in R^3$. Сопоставим всем вершинам графа различные точки на отрезке ***[a,b]***. Сопоставим каждому ребру графа свою плоскость, проходящую через ***[a,b]***. Плоскости, соответствующие ребрам, пересекаются только вдоль отрезка ***[a,b]***, поэтому в каждой плоскости можно нарисовать кривую, соединяющую вершины, инцидентные соответствующему ребру, которая не пересекается с другими построенными кривыми нигде, кроме как в вершинах графа.

■

Граф называется ***планарным***, если существует его геометрическое представление *в* ***R^2. Плоской укладкой*** планарного графа называется такое его геометрическое представление, при котором каждое ребро представляется отрезком на плоскости.

Верна следующая известная теорема, которой мы не будем пользоваться и которую мы представим без доказательства

Теорема 3. *Для любого конечного планарного графа без кратных ребер и петель существует его плоская укладка.*

Иногда в литературе последнее утверждение выступает в виде определения планарного графа, а не его свойства.

Геометрическое представление планарного графа разбивает плоскость на некоторое количество связных областей (одна из них - бесконечна). Эти области называются ***гранями***.

Путем в графе называется такая последовательность $\{x_1,\ldots,x_n\}$ вершин графа, что для всех ***i***: $(x_i,x_{i+1}) \in E$.

Циклом в графе называется такой путь, что крайние вершины в нем совпадают. Т.е. последовательность $\{x_1,\ldots,x_n\}$ вершин графа называется ***циклом***, если для всех ***i***: $(x_i,x_{i+1}) \in E$ и $x_1=x_n$.

Граф называется ***связным***, если для любых двух вершин графа ***a*** и ***b*** существует путь по ребрам графа от ***a*** до ***b***, т.е. существует последовательность $\{x_1,\ldots,x_n\}$ вершин графа, такая что $x_1=a$, $x_n=b$ и для всех ***i:*** пара (x_i, x_{i+1}) инцидентна некоторому ребру графа.

Связный граф без циклов называется ***деревом.*** Вершины, инцидентные не более чем одному ребру дерева называются ***конечными вершинами.*** Иногда в литературе такие вершины называют *листьями*, но нам такое определение листьев неудобно. Отметим, что понятия ориентации ребер в этом определении нет!

Ориентированным деревом называется ориентированный граф, являющийся деревом, для которого ровно одна вершина имеет нулевую степень захода (*степень захода* = количество ребер заходящих в вершину), а все остальные вершины имеют степень захода, равную 1. Для случая ориентированного графа конечной вершиной называется вершина, имеющая нулевую степень исхода (*степень исхода* = количество ребер выходящих из вершины). Это определение отличается от определения для неориентированного дерева!

Лемма 1. *Любое конечное неориентированное дерево имеет плоскую укладку.*

Доказательство. Возьмем произвольную вершину дерева. Поместим ее в точку плоскости с координатами ***(0,0)*** и назовем ***корнем дерева***. Пусть ей инцидентно ***k*** ребер. Поместим вторые вершины, инцидентные данным ребрам в точки ***(-1,i)***. Каждой из вновь размещенных точек сопоставим полосу плоскости со сторонами, параллельными оси ***Y***, не имеющую общих точек с полосами соседних точек.

Пусть для каждой вершины ***x***, помещенной на плоскость в точку с координатой ***y=-h*** найдутся l_x парных ей вершин, которые еще не размещены на плоскости. Будем называть эти вершины ***потомками*** вершины ***x***, а вершину ***x*** – их родителем. Разобьем полосу, соответствующую вершине ***x*** на l_x непересекающихся полос и разместим в них потомков ***x*** на координате ***y=-h-1***.

Для потомков ***x*** рекурсивно выполним процедуру, описанную в предыдущем абзаце.

Условие отсутствия циклов гарантирует, что вершина ***x*** будет соединена с новыми вершинами только одним ребром и что в графе не будет существовать ребер, соединяющих вершину ***x*** с уже размещенными вершинами. Т.о. для каждой вершины, размещенной на плоскости будут реализованы все ребра графа ей инцидентные.

Условие связности гарантирует, что после окончания этого алгоритма все точки будут размещены на плоскости. Действительно, пусть вершина ***b*** принадлежит графу, тогда из связности графа следует, что существует путь по ребрам графа от корня дерева до ***b***. Мы показали, что для вершин, размещенных на плоскости, реализуются все ребра графа, из чего по индукции получаем, что все вершины пути, соединяющего корень дерева и ***b*** будут размещены на плоскости.

■

Доказательство Леммы 1 одновременно служит конструктивным доказательством того, что верна следующая

Лемма 2. *В любом конечном дереве можно ввести ориентацию, т.е. для ребер произвольного дерева можно задать ориентацию так, что дерево станет ориентированным.*

Верна следующая известная теорема

Теорема 4 (формула Эйлера). *Пусть задан связный планарный граф, имеющий в некотором геометрическом представлении* ***p*** *вершин,* ***q*** *ребер,* ***r*** *граней, тогда верна формула*

$$p-q+r=2$$

Лемма 3. *Пусть в некотором графе* ***p*** *вершин и* ***q*** *ребер, то в графе содержится не менее* ***p-q*** *связных компонент. Если в графе нет циклов, то в графе ровно* ***p-q*** *связных компонент. Если в графе есть циклы, то в графе строго больше* ***p-q*** *связных компонент.*

Доказательство леммы. Любой конечный граф ***G=(V,E)*** без циклов может быть получен из графа $G_0=(V,Æ)$ путем последовательного добавления ребер, при этом каждый промежуточный граф не будет содержать циклов. Для графа G_0 данная лемма выполняется (количество связных компонент равно количеству вершин).

Рассмотрим случай отсутствия циклов. При добавлении каждого ребра мы либо соединяем две связные компоненты и при этом лемма остается верной, либо обе соединяемые вершины принадлежат одной связной компоненте. Но в последнем случае перед соединением существовал путь от одной вершины к другой, и добавление ребра образовало цикл, из чего получаем противоречие с отсутствием циклов в подграфах.

Теперь, если разрешить наличие циклов, то в предыдущем доказательстве станет возможным соединение вершин из одной связной компоненты, но тогда количество связных компонент будет строго больше ***p-q***. **Лемма доказана.**

Следствием Леммы 3 является простой критерий наличия циклов в графе:

p-q*=количеству связных компонент в графе* Û *в графе нет циклов.*

В частности, для связного графа мы получаем, что ***p-q=1*** Û *в графе нет циклов.*

Доказательство формулы Эйлера. Исходя из леммы, получаем, что теорема Эйлера верна для деревьев. Действительно, по Лемме 1, каждое дерево с конечным числом вершин имеет плоскую укладку. Количество граней в этой укладке равно1 (иначе существует хотя бы одна конечная грань, а значит существует последовательность ребер на ее границе, образующая цикл). Количество граней в геометрическом представлении дерева равно 1 и в силу Леммы 3

$p-q+r=1+1=2$.

Рассмотрим произвольный связный граф с циклами с q_0 ребрами. По Лемме 3 $p-q_0>1$. Зафиксируем p и предположим, что для всех $q<q_0$ теорема доказана. Заметим, что для случая $p-q=1$ мы уже доказали теорему. По предположению, в графе есть цикл, возьмем одно ребро в этом цикле и исключим из графа. Ребро было в цикле графа, поэтому связность нарушена не была. Т.к. ребро содержалось в цикле, то оно было общим для двух граней, поэтому при исключении ребра количество граней и количество ребер уменьшились на 1. По предположению индукции для урезанного графа теорема выполняется, следовательно, она выполняется и для исходного графа.

■

Рассмотрим планарный граф без кратных ребер и петель, в каждой связной компоненте которого есть хотя бы три ребра. В его плоской укладке для каждой грани i количество ребер на ее границе $q_i \geq 3$. Т.о. имеем

$\sum q_i \geq 3r$

С другой стороны каждое ребро принадлежит не более, чем двум граням, поэтому

$2q \geq \sum q_i$

Итого имеем

$2q \geq 3r$

$$r \leq 2/3\, q.$$

Тогда по формуле Эйлера, примененной для i-ой связной компоненты графа, имеем:

$$q_i=p_i+r_i-2 \leq p_i+2/3\, q_i -2.$$

$$q_i/3 \leq p_i-2$$

$$q_i \leq 3p_i$$

А значит и для всего графа

$$q \leq 3p$$

В силу последнего соотношения верна следующая

Терема 5. *Для случая планарного графа без кратных ребер и петель, состоящего из N вершин, алгоритм волны работает за время O(N).*

Здесь случай количества ребер меньше 3 легко рассмотреть отдельно.

Отметим, что параллельно мы доказали, что в планарном графе количество ребер и граней оценивается через количество вершин, т.е. верна

Терема 6. *Для случая планарного графа без кратных ребер и петель, имеющего в плоском представлении* p *вершин,* q *ребер,* r *граней верны оценки*

$$q \leq 3p$$

$$r \leq 2p$$

Аналогично, здесь случай наличия связных компонент с количеством ребер меньше 3 легко рассмотреть отдельно, т.к.

- количество ребер для каждой такой компоненты строго меньше количества ее вершин;
- при добавлении к графу отдельной связной компоненты с количеством ребер меньше 3 количество граней не увеличится.

Представление графа в памяти ЭВМ

Пусть имеется граф **G=(V,E)**, *имеющий* N вершин и M ребер. Вершинам и ребрам можно сопоставить их номера от 1 до N и от 1 до M, соответственно. Рассмотрим различные варианты хранения данных об этом графе. Отметим, что для работы с графом требуются следующие операции

- перечисление всех ребер, инцидентных вершине i
- перечисление вершин, инцидентных ребру j
- перечисление всех вершин, смежных с вершиной i
- проверка смежности двух вершин
- проверка смежности двух ребер

Массив ребер

Для хранения информации о графе можно использовать ***массив ребер***, в каждом элементе которого хранятся номера вершин, инцидентных ребру

```
#typedef  MMax 100
int edges[MMax][2], n_edges;
```

здесь предполагается, что в графе содержится не более $MMax$ ребер;

Имеем следующие времена выполнения интересующих нас операций

- перечисление всех ребер, инцидентных вершине i — $T=O(M)$
- перечисление вершин, инцидентных ребру j — $T=O(1)$
- перечисление всех вершин, смежных с вершиной i — $T=O(M)$
- проверка смежности двух вершин — $T=O(M)$
- проверка смежности двух ребер — $T=O(1)$

Матрица смежности

Для хранения информации о графе можно использовать ***матрицу смежности*** из N строк и N столбцов, в (i,j) – элементе которой хранится количество ребер, инцидентных паре вершин с номерами i и j.

Имеем следующие времена выполнения интересующих нас операций

- перечисление всех ребер, инцидентных вершине i — -------
- перечисление вершин, инцидентных ребру j — -------
- перечисление всех вершин, смежных с вершиной i — $T=O(N)$
- проверка смежности двух вершин — $T=O(1)$
- проверка смежности двух ребер — -------

Матрица инцидентности

Для хранения информации о графе можно использовать ***матрицу инцидентности*** из ***N*** строк и ***M*** столбцов, в ***(i,j)*** – элементе которой хранится ***1***, если вершина ***i*** инцидентна ребру ***j***, и ***0*** – если нет.

Имеем следующие времена выполнения интересующих нас операций

- перечисление всех ребер, инцидентных вершине ***i*** ***T=O(M)***
- перечисление вершин, инцидентных ребру ***j*** ***T=O(N)***
- перечисление всех вершин, смежных с вершиной ***i*** ***T=O(N M)***
- проверка смежности двух вершин ***T=O(N M)***
- проверка смежности двух ребер ***T=O(N+M)***

Списки смежных вершин

Для хранения информации о графе можно для каждой вершины хранить множество смежных вершин. Множество можно реализовать либо в виде массива:

```
#typedef  NMax 100
          typedef struct CVertex1_
          {
 int AdjacentVertices[NMax];
 int NAdjacentVertices;
} CVertex1;
          CVertex1 vertices[NMax];
```

здесь предполагается, что в графе содержится не более ***NMax*** вершин; либо в виде списка:

```
#typedef  NMax 100
          typedef struct CAdjVertex_
          {
 int i;              //номер текущей вершины
 int i_next;         //номер следующей вершины
} CAdjVertex;
          typedef struct CVertex2_
{
CAdjVertex *AdjacentVerticesList_Head; //голова списка  смежных
          вершин                  int i;          //номер текущей вершины
} CVertex2;
          CVertex2 vertices[NMax];
```

Т.о. в каждой вершине графа будет храниться вектор или список смежных вершин к данной вершине.

Имеем следующие времена выполнения интересующих нас операций

- перечисление всех ребер, инцидентных вершине ***i*** *-------*
- перечисление вершин, инцидентных ребру ***j*** *-------*
- перечисление всех вершин, смежных с вершиной ***i*** ***T=O(N)***
- проверка смежности двух вершин ***T=O(N)***
- проверка смежности двух ребер *-------*

Реберный список с двойными связями (РСДС) (для плоской укладки планарных графов)

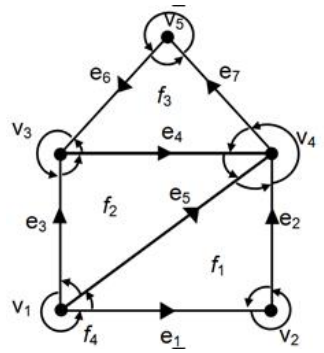
Для плоской укладки планарных графов можно использовать более экономные способы их хранения. Плоская укладка планарных графов представляет собой множество точек на плоскости с отрезками, их соединяющими. Отрезки не пересекаются нигде, кроме вершин графа. Информацию о графе можно хранить в виде информации о каждом ребре графа, или в виде ***реберного узла***. На языке С реберный узел может быть представлен следующей структурой

```
          typedef struct SEdge_
          {
 int vertex0, vertex1;
 int edge0,edge1;
 int face0,face1;
}SEdge;
```

здесь ***vertex0, vertex1*** – номера первой и второй вершины ребра. Порядок вершин задает ориентацию. ***edge0*** – номер первого ребра (в массиве элементов ***SEdge***), выходящего из вершины ***vertex0***, взятого против часовой стрелки (при обходе ребер из вершины ***vertex0***), ***edge1*** – первое ребро, выходящее из вершины ***vertex1***, взятое против часовой стрелки (при обходе ребер из вершины ***vertex1***). ***face0*** – номер грани, находящейся слева от направленного отрезка ***(vertex0, vertex1)***, ***face1*** – номер грани, находящейся справа от направленного отрезка ***(vertex0, vertex1).***

Вместо структуры можно использовать массив из шести целых чисел.

На следующем рисунке показан пример графа. Стрелками на ребрах указан порядок вершин в узлах РСДС, соответствующих ребрам. Стрелками между ребрами указаны ссылки на ребра из соответствующих узлов РСДС.



Для данного графа РСДС будет представлен следующим массивом улов:
{
{1,2, 5,2, 1,4},
{2,4, 1,7, 1,4},
{1,3, 1,4, 4,2},
{3,4, 6,5, 3,2},
{1,4, 3,2, 2,1},
{5,3, 7,3, 3,4},
{4,5, 4,6, 3,4}
}
Легко видеть, что РСДС позволяет за постоянное время находить следующее против часовой стрелки ребро в множестве ребер, инцидентных одной вершине. Также за постоянное время можно находить следующее ребро для данной грани.

Лекция 9

Структуры данных. Графы.

Д.Кнут. Искусство программирования для ЭВМ. тт 1-3. Москва. Мир. 1996-1998
Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы. Построение и анализ. Москва. МЦНМО. 1999.

Поиск кратчайшего пути в графе. Алгоритм Дейкстры (*Dijkstra's algorithm*).
Пусть дан некоторый конечный граф $G=(V,E)$, состоящий из N вершин, и две его вершины $a, b \hat{I} V$. Пусть для каждого ребра графа e задано положительное вещественное число $l(e)$, которое будем называть ***длиной ребра***. Требуется найти путь между вершинами графа a и b минимальной длины, т.е. требуется найти последовательность $\{x_1,\ldots,x_n\}$, где $x_i \hat{I} V, x_1=a, x_n=b, (x_i, x_{i+1}) \hat{I} E$ с минимально возможной $S\, l(x_i,x_{i+1})$.
Стандартным решением данной задачи является *алгоритм Дейкстры*.

Основная идея алгоритма заключается в следующем.

Пусть множество Q_n представляет собой множество n ближайших вершин к вершине a вместе с длинами пути от a до этих вершин. Т.е. в каждом элементе множества $q_i \hat{I} Q_n$ содержится номер соответствующей вершины x_i и длина пути от a до этой вершины l_i: $q_i=(x_i,l_i)$. Можно предполагать, что последовательность $\{l_i\}$ является неубывающей.

Утверждается, что ближайшая вершина графа к a из вершин, не внесенных в Q_n , задается следующим соотношением:

$$argmin(v,w \hat{I} V,\ w \hat{I} Q_n,\ v \ddot{I} Q_n,\ (w,v) \hat{I} E:\ l_w+|(w,v)|) \qquad (*)$$

здесь и далее под $w \hat{I} Q_n$, $v \ddot{I} Q_n$ имеется в виду, что w встречается среди вершин, внесенных в Q_n, а v – нет; длина ребра (w, v) обозначается $|(w,v)|$. Т.о. элемент q_{n+1} складывается из вершины v, на которой достигается указанный минимум и соответствующей длины $l_{n+1} = l_w+|(w,v)|$.

Доказательство. Допустим, что алгоритм $(*)$ не находит ближайшую вершину графа к a из вершин, не внесенных в Q_n . Т.е. существует вершина $s \ddot{I} Q_n$ с минимальной длиной реального пути до a, меньшей величины, найденной в $(*)$. Пусть кратчайший путь от a до s проходит по последовательности вершин графа $\{x_1=a,x_2,\ldots,x_n, x_{n+1}=s\}$. Выберем в этой последовательности элемент x_k такой, что все элементы до него принадлежат Q_n, а он сам не принадлежит Q_n . Такой элемент существует и $k>1$, т.к. x_1 принадлежит Q_n, а x_{n+1} не принадлежит Q_n.

$x_{k-1} \hat{I} Q_n$, $x_k \ddot{I} Q_n$, по допущению длина пути $\{a,x_2,\ldots,x_k\}$ меньше величины, найденной в $(*)$, что сразу приводит к противоречию с алгоритмом вычисления $(*)$.

■

При поиске значения $(*)$, формально, требуется перебрать все вершины $w \hat{I} Q_n$ и для каждой из них требуется перебрать все смежные вершины. Будем далее предполагать, что *в графе отсутствуют кратные ребра и петли*, тогда процедура $(*)$ требует выполнения $O(N^2)$ операций, из чего следует, что суммарное время работы алгоритма есть $O(N^3)$.

На самом деле, для выполнения процедуры $(*)$ можно обойтись $O(N)$ операциями. Для этого заметим, что за один поиск значения $(*)$ к множеству Q_n добавляется всего одна точка и, поэтому, на следующем шаге значения l_x изменятся только у вершин, смежных этой новой точке. Т.о. для пересчета в $(*)$ всех значений l_x требуется пересчитать l_x только для точек, смежных одной точке, полученной при предыдущем выполнении $(*)$. В силу введенных предположений, это можно сделать за $O(N)$ операций. Искать же минимум l_x можно по всем вершинам графа, не принадлежащим Q_n, что тоже требует $O(N)$ операций. Т.о. процедуру $(*)$ можно реализовать за $O(N)$ операций.

Для реализации алгоритма Дейкстры следует добавить в каждую вершину w графа вещественное число l_w, характеризующее длину кратчайшего пути от вершины a до вершины w и логическую переменную s_w, указывающую – посчитана ли на данный момент эта кратчайшая длина пути.

Для упрощения дальнейшей жизни (т.е. для поиска, собственно, кратчайшего пути при определенных значения l_w) можно в каждом элементе также хранить ссылку на вершину, из которой мы в данную вершину пришли. Для текущей вершины w будем эту ссылку называть $back_w$. Т.о. элементы Q_i представляются в виде $q=(w, l_w, s_w, back_w)$.

Будем предполагать, что конкретная техника нам позволяет инициализировать все l_w значением = плюс бесконечность (***+INF***), что мы и сделаем. Инициализируем все s_w значением = ***FALSE***.

Введем переменную ***c***, которая будет хранить последнюю вершину, добавленную в множество вершин с найденной минимальной длиной пути до вершины. Тогда алгоритм будет выглядеть следующим образом

Алгоритм Дейкстры

$c=a$; s_c=TRUE; l_c=0; back$_c$=NULL *//Q_0={(a,0, true,NULL)}*
Вечный цикл
Для всех вершин v, смежных с
Если s_c==FALSE и (l_v==+INF или l_c+\|(c,v)\|< l_v) то
$l_v = l_c$+\|(c,v)\|; back$_v$=c
l=+INF
Для всех вершин графа v
Если s_c==FALSE и l_v< l то l = l_v; z=v

Если l==+INF то ВЫЙТИ *// в графе не осталось элементов*
Если z==b то ВЫЙТИ *// дошли до конца пути*
s_z=TRUE; c= z *// $Q_{n+1}= Q_n$ È {(z,l,true)}*
Конец вечного цикла

Если после завершения первой части алгоритма ***l==+INF*** , то это означает, что от ***a*** до ***b*** дойти по ребрам невозможно. Иначе, мы можем по ссылкам ***back$_w$*** добраться от ***b*** до ***a***.

Можно написать тот же самый алгоритм в виде программы на C++ с использованием STL (будем считать значения не менее 1.e100 бесконечностью, хотя это не является правилом хорошего тона):

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
struct SPoint{SPoint(){l=1.e101;iback=-1;s=0;} double x,y,l;int iback,s;};
struct SEdge{int i0,i1; double d; int operator()(int i){return i==i0?i1:i0;}};
ifstream &operator>>(ifstream &f,SPoint &p){f>>p.x>>p.y; return f;}
ifstream &operator>>(ifstream &f,SEdge &p){f>>p.i0>>p.i1>>p.d; return f;}
int main(void)
{vector<SPoint> p; vector<SEdge> e; vector<vector<int>> n;
 int i0=0,i1=11,j;double l; int ipt;
 //--
 {SPoint P; ifstream f("vertex.txt"); while(f>>P)p.push_back(P);}//ввод вектора
вершин из файла с к-тами вершин в формате x y
 {SEdge E;  ifstream f("edges.txt"); while(f>>E)e.push_back(E);} //ввод вектора
ребер из файла с номерами вершин ребра и его длиной в формате i0 i1 d
 n.resize(p.size());
 for(size_t i=0;i<e.size();i++)
 {n[e[i].i0].push_back(i);n[e[i].i1].push_back(i);}
 //--
 for(ipt=i0,p[i0].l=0,p[i0].s=1;ipt!=i1;)
 {
  for(vector<int>::iterator i=n[ipt].begin();i!=n[ipt].end();i++)
   if((l=p[ipt].l+e[*i].d)<p[j=e[*i](ipt)].l)//мы сознательно не проверяем, что
p[j].s==0
    {p[j].iback=ipt;p[j].l=l;}
  l=1.e100;ipt=-1;
  for(size_t i=0; i<p.size(); i++)if(p[i].s==0&&p[i].l<l)
  {ipt=i;l=p[i].l;}
  if(ipt<0)goto le;
  p[ipt].s=1;
 }
//--
 cout<<"l="<<p[i1].l<<endl;
 for(int k=i1;k>=0;k=p[k].iback)cout<<k<<" "; cout<<endl;
 getchar();return 0;
le:
 cout<<"can't find way"<<endl;
 getchar();
 return 0;
}
```

Отметим, что содержательный цикл в данной программе занимает места не больше, чем вышеприведенный алгоритм на псевдоязыке.

Проанализировав алгоритм для случая планарных графов без кратных ребер и петель, мы сможем заметить, что его можно улучшить. Действительно, при коррекции l_v для каждой вершины перебираются все смежные ребра и это происходит ровно один раз. Поэтому суммарное количество операций по коррекции l_v не превосходит ***O(q)***, где ***q*** – количество ребер, а для рассматриваемого случая ***O(q)= O(p)***, где ***p*** – количество вершин графа.
Допустим, что мы сможем создать структуру данных, содержащую вещественные числа, в которую можно добавлять элементы, удалять из нее минимальный элемент, искать минимум элементов, модифицировать положение элемента при изменении его значения, причем каждая из этих операций должна выполняться за время ***O(log N)***, где – ***N*** количество элементов, занесенных в данную структуру. Назовем множество, реализованное с помощью указанной структуры данных, ***P***.
На каждом шаге вышеописанного алгоритма мы можем хранить в множестве ***P*** ссылки на все элементы, содержащие вершины, смежные с вершинами очередного Q_n, которые сами в Q_n не содержатся. Заметим, что именно среди этих элементов происходит коррекция l_v и поиск минимально l_v. На каждом шаге алгоритма Дейкстры мы должны сначала добавить в ***P*** ссылки на все элементы, смежные ***c***, при изменении значений l_v модифицировать положение соответствующих элементов, а в конце извлечь из ***P*** ссылку на элемент с минимальным значением l_v. Т.о. алгоритм придет к следующему виду

Алгоритм Дейкстры модифицированный

———————

c=a; s_c=TRUE; l_c=0; back$_c$=NULL *//Q_0={(a,0, true,NULL)}*
P={Æ }
Вечный цикл
Для всех *вершин* ***v,*** *смежных* ***c***
Если s_c==FALSE и l_v==+INF то
Добавить ссылку на ***v*** *в* ***P***
Если P *пусто* ***то l_z =+INF; ВЫЙТИ//*** *в графе не осталось элементов*
Для всех *вершин* ***v,*** *смежных* ***c***
Если s_c==FALSE и (l_v==+INF или l_c+\|(c,v)\|< l_v) то
l_v = l_c+\|(c,v)\|; back$_v$=c;
Скорректировать *положение ссылки на* ***v*** *в* ***P***
Извлечь из P *ссылку на минимальный элемент* ***и***
поместить *минимальный элемент в* ***z***
l = l_z
Если z==b то ВЫЙТИ *// дошли до конца пути*
s_z=TRUE; c= z *// Q_{n+1}= Q_n È {(z,l,true)}*
Конец вечного цикла

Осталось заметить, что для корректировки положения ссылки на ***v*** в ***P*** нам еще необходимо уметь находить эту ссылку на ***v*** в ***P***, поэтому в каждом элементе исходного множества ***v*** еще придется хранить ссылку на ссылку на ***v*** в ***P***. Т.о. при всех перемещениях элементов в ***P*** ссылки на эти элементы в исходном множестве придется модифицировать.

Итак для случая планарных графов без кратных ребер и петель, в каждом переборе смежных вершин на протяжении всего алгоритма каждое ребро встречается только два раза, поэтому суммарное время работы первого цикла ***Для всех…*** равно ***O(p log p)***, суммарное время работы второго цикла ***Для всех…*** имеет такую же ассимптотику. Наконец, извлечение из ***P*** ссылки на минимальный элемент выполняется за время ***O(log p)***, а т.к. таких элементов ***O(p)***, то суммарное время этих извлечений = ***O(p log p)***. Таким образом, доказана следующая

Теорема 1. *Для случая планарных графов без кратных ребер и петель алгоритм Дейкстры работает за время* ***O(p²)****, а модифицированный алгоритм Дейкстры работает за время* ***O(p log p)****, где* ***p*** *– количество вершин в графе.*

До сих пор мы не предъявили реализации структуры данных, содержащей вещественные числа, такой, что она должна уметь выполнять следующие операции:

- добавлять элемент,
- удалять минимальный элемент,
- искать минимум элементов,
- модифицировать положение элемента при изменении его значения,

причем каждая из этих операций должна выполняться за время ***O(log N)***, где – ***N*** количество элементов, занесенных в данную структуру.
В качестве подобной структуры данных может выступать пирамида, определенная для сортировки ***Heapsort***. При ее определении мы требовали, чтобы в ее вершине лежал максимальный элемент (исходя из того, что $A_{[i/2]} \geq A_i$). Мы можем не менять определения пирамиды, потребовав, чтобы в нее заносились исходные элементы, умноженные на ***–1***, в таком случае в вершине будет храниться минимальный элемент исходного множества (умноженный на ***–1***).
Удалять максимальный элемент из пирамиды за время ***O(log N)*** мы научились в алгоритме ***Heapsort***. Находить максимальный элемент за время ***O(1)*** можно элементарно – он лежит в первом элементе массива пирамиды. Осталось научиться вставлять новый элемент в пирамиду и корректировать его положение при изменении значения.

Добавление элемента
Пусть дан пирамидально-упорядоченный массив A_i, (***i=1,…,N***). Требуется добавить к нему еще один элемент A_{i+1} и переупорядочить массив. Для этого

введем текущий номер элемента в массиве ***k***. В начальный момент положим ***k= N+1***.
Свойство пирамидально-упорядоченности выполняется для всех соответствующих пар элементов, кроме, быть может, пары ***(k,k/2)***. Если для этой пары свойство $A_{[k/2]} \geq A_k$ выполняется, то массив пирамидально упорядочен и ничего больше делать не надо. Иначе поменяем местами пару элементов с индексами ***k*** и ***k/2***. При этом элемент с индексом ***k/2*** не уменьшится, поэтому поддерево, с него начинающееся, но идущее по другой ветке, чем ***(k,k/2)*** останется пирамидально-упорядоченным (т.е., например для четного ***k***, останется верным $A_{[k/2]} \geq A_{k+1}$).
Перед обменом местами элементов с индексами ***(k,k/2)***, элемент $A_{[k/2]}$ был не меньше всех элементов в поддереве, начинающегося с $A_{[k/2]}$, кроме, быть может, A_k. Поэтому, после обмена местами элементов с индексами ***(k,k/2)***, элемент A_k будет не меньше всех элементов в поддереве, начинающегося с A_k. Осталось присвоить ***k=[k/2]***, и выполнить те же самые действия для нового ***k***.
Таким образом, мы произведем не более ***[log N]+1*** обменов местами соседних элементов, из чего следует, что процедура добавления элемента в пирамиду может быть произведена за время ***O(log N)***.

Корректировка положения элемента
Пусть дан пирамидально-упорядоченный массив A_i, (***i=1,…,N***). Для текущего элемента в массиве с номером ***k*** изменим значение элемента A_k. Требуется переупорядочить массив.
Если $A_k \geq A_{2k}$ и $A_k \geq A_{2k+1}$, то задача полностью сводится к предыдущей.
Пусть, для определенности, $A_k < A_{2k}$. Но тогда $A_{[k/2]} \geq A_k$ (т.к. $A_{[k/2]} \geq A_{2k} > A_k$), из чего получаем, что все элементы из поддерева, начинающегося с ***k***-го элемента массива не превосходят $A_{[k/2]}$. В этом случае возможно применить процедуру ***Heapify(A,k,N)***. Время ее работы ***O(log N)***.
■

Алгоритм Дейкстры на основе STL.
Весьма соблазнительно было бы реализовать алгоритм Дейкстры на основе STL, поскольку в STL присутствует реализация приоритетной очереди (скорее всего, на основе кучи!). Увы, для описанного выше алгоритма это невозможно, т.к. приоритетная очередь в STL не позволяет модифицировать элементы, содержащиеся внутри очереди (в алгоритме Дейкстры это требуется). Однако, мы можем модифицировать алгоритм Дейксты так, чтобы он стал совместимым с STL. Приведем два примера таких модификаций. Первая модификация идеальна с математической точки зрения, но при реализации на компьютере в ней требуется сравнение на равенство вещественных чисел, что может привести к непредсказуемым последствиям при работе с графом, в котором присутствуют предельно короткие ребра. Вторая реализация чуть сложнее, но не несет недостатков предыдущей реализации.

Основная идея обеих реализаций сводится к тому, что мы (также как и в предыдущей реализации) будем добавлять элементы в ***P***, но модифицировать элементы в ***P*** не будем. Т.е. вместо модификации элементов в ***P*** мы будем просто добавлять в ***P*** еще одну ссылку на вершину графа вместе с длиной пути до данной вершины (т.о. длина пути до данной вершин будет храниться дважды: в самой вершине в векторе вершин и в элементе, заносимым в ***P***). Таким образом, у нас в ***P*** образуется, вообще говоря, несколько ссылок на одну и ту же вершину, из которых только последняя ссылка должна быть действующей, а остальные должны быть фиктивными (неиспользуемыми). Процедура извлечения минимального элемента из ***P*** должна, при этом, превратиться в цикл, в котором из вершины ***P*** извлекаются элементы до тех пор, пока не появится рабочий (нефиктивный) элемент. Простейший вариант проверки *является ли элемент в **P** рабочим* сводится к проверке равенства длин пути до данной вершины, хранящимися в вершине в векторе вершин и в извлеченном элементе из ***P***.

Первая реализация алгоритма Дейкстры на основе STL.

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
using namespace std;
struct SPoint{SPoint(){l=-1;iback=-1;} double x,y,l;int iback;}; //l=длина кратч.пути, iback=номер пред.вершины в кратч.пути
struct SEdge{int i0,i1; double d; int operator()(int i){return i==i0?i1:i0;}}; //l=длина кратч.пути
struct SPointP{SPointP(int I=0,double L=0){i=I;l=L;} int i;double l;};
ifstream &operator>>(ifstream &f, SPoint &a){f>>a.x>>a.y; return f;}
ifstream &operator>>(ifstream &f, SEdge &a){f>>a.i0>>a.i1>>a.d; return f;}
bool operator<(const SPointP &a, const SPointP &b){return a.l>b.l;}
int main(void)
{
 vector<SPoint> p; vector<SEdge> e; vector<vector<int>> n;
 int i0=0,i1=11,j; double l; SPointP pt; priority_queue<SPointP> q;
 {SPoint P;ifstream f("vertex.txt"); while(f>>P)p.push_back(P);} //ввод вектора вершин из файла с к-тами вершин в формате x y
```

```
 {SEdge E; ifstream f("edges.txt");  while(f>>E)e.push_back(E);} //ввод вектора ребер из файла с номерами вершин ребра и его длиной в формате i0 i1 d
 n.resize(p.size());
 for(size_t i=0;i<e.size();i++)//создание вектора векторов номеров ребер, инцидентных вершине
 {n[e[i].i0].push_back(i);n[e[i].i1].push_back(i);}
 //----
 for(pt.i=i0,p[i0].l=0,pt.l=0;pt.i!=i1;) //пока не дошли до конца…
 {
  for(vector<int>::iterator i=n[pt.i].begin(); i!=n[pt.i].end(); i++)//перебираем ребра, инцидентные текущей вершине и добавляем их в очередь
  {
   j=e[*i](pt.i); l=p[pt.i].l+e[*i].d;
   if(p[j].l<0||p[j].l>l)
   {p[j].iback=pt.i;p[j].l=l;q.push(SPointP(j,l));}
  }
  do{if(q.empty())goto le;pt=q.top(); q.pop();}while(pt.l!=p[pt.i].l); //ищем первую рабочую вершину и записываем ее в текущую вершину
 }
 cout<<"l="<<pt.l<<endl;
 for(int k=i1;k>=0;k=p[k].iback)cout<<k<<" "; cout<<endl; //номера вершин кратчайшего пути в обратном порядке
 getchar();return 0;
le:
 cout<<"can't find path"<<endl;getchar();
 return 0;
}
```

Чисто формально, существенным недостатком данной реализации является наличие сравнения на равенство двух вещественных чисел в условии цикла *do…while*. Но на самом деле, реальная неприятность кроется не здесь (более того, если хорошо разобраться, здесь проблем быть не должно!). В программе присутствует присваивание *l=p[pt.i].l+e[*i].d;* , а после него – сравнение: *l<p[j].l* . Совершенно непонятно, что будет оптимизатор компилятора иметь в виду в последнем сравнении: значение переменной *l* в регистре (полученное после присваивания суммы), или обрезанное значение до типа *double*. В первом случае получится, что в последующем присваивании *p[j].l=l;* будет присвоено не то значение, которое принимало участие в сравнении, что может привести к непредсказуемым последствиям.

Вторая реализация алгоритма Дейкстры на основе STL.

Для разрешения вышеописанной проблемы предлагается в структурах *SPoint* и *SPointP* дополнительно хранить номер версии точки. При занесении в очередь версия заносимой точки в структуре *SPoint* будет увеличиваться на 1 и полученное значение будет записываться в структуру *SPointP*, которая и будет заноситься в очередь. Т.о. проверка на то, что извлекаемая из очереди вершина является рабочей сводится к проверке совпадения версии извлекаемой вершины в извлеченной структуре *SPointP* и версии данной вершины в векторе вершин в структуре *SPoint*.

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
using namespace std;
struct SPoint{SPoint(){l=-1;iback=-1;v=0;} double x,y,l;int iback,v;}; //l=длина кратч.пути, iback=номер пред.вершины в кратч.пути
struct SEdge{int i0,i1; double d; int operator()(int i){return i==i0?i1:i0;}}; //l=длина кратч.пути
struct SPointP{SPointP(int I=0,double L=0,int V=0){i=I;l=L;v=V;} int i;double l;int v;};
ifstream &operator>>(ifstream &f, SPoint &a){f>>a.x>>a.y; return f;}
ifstream &operator>>(ifstream &f, SEdge &a){f>>a.i0>>a.i1>>a.d; return f;}
bool operator<(const SPointP &a, const SPointP &b){return a.l>b.l;}
int main(void)
{
 vector<SPoint> p; vector<SEdge> e; vector<vector<int>> n;
 int i0=0,i1=11,j; double l; SPointP pt; priority_queue<SPointP> q;
 {SPoint P;ifstream f("vertex.txt"); while(f>>P)p.push_back(P);} //ввод вектора вершин из файла с к-тами вершин в формате x y
 {SEdge E; ifstream f("edges.txt");  while(f>>E)e.push_back(E);} //ввод вектора ребер из файла с номерами вершин ребра и его длиной в формате i0 i1 d
 n.resize(p.size());
 for(size_t i=0;i<e.size();i++)//создание вектора векторов номеров ребер, инцидентных вершине
 {n[e[i].i0].push_back(i);n[e[i].i1].push_back(i);}
```

```
//----
 for(pt.i=i0,p[i0].l=0,pt.l=0;pt.i!=i1;) //пока не дошли до конца…
 {
  for(vector<int>::iterator i=n[pt.i].begin(); i!=n[pt.i].end(); i++)//перебираем ребра,
инцидентные текущей вершине и добавляем их в очередь
  {
   j=e[*i](pt.i); l=p[pt.i].l+e[*i].d;
   if(p[j].l<0||p[j].l>l)
   {p[j].iback=pt.i;p[j].l=l;q.push(SPointP(j,l,++p[j].v));}
  }
  do{if(q.empty())goto le;pt=q.top(); q.pop();}while(pt.v!=p[pt.i].v); //ищем первую
рабочую вершину и записываем ее в текущую вершину
 }
 cout<<"l="<<pt.l<<endl;
 for(int k=i1;k>=0;k=p[k].iback)cout<<k<<" "; cout<<endl; //номера вершин
кратчайшего пути в обратном порядке
 getchar();return 0;
le:
 cout<<"can't find path"<<endl;getchar();
 return 0;
}
```

Легко доказать теорему о времени работы построенного алгоритма:

Теорема 2. *Для случая планарных графов без кратных ребер и петель алгоритм Дейкстры на основе STL работает за время* ***O(p log p)****, где* ***p*** *– количество вершин в графе.*

Действительно, тело цикла по ребрам, инцидентным текущей вершине суммарно (т.е. в двойном цикле) выполняется не более ***2*#E*** раз (где ***#E*** – количество ребер графа), т.к. каждое ребро в данном цикле встречается не более двух раз. По этой же причине размер приоритетной очереди не может превосходить ***2*#E***. А так как по доказанному выше утверждению ***#E£ 3p*** и учитывая, что извлечение элемента из приоритетной очереди осуществляется за время = ***O(log p)***, то получим, что суммарное время выполнения цикла перебора ребер, инцидентных текущей вершине равно ***O(p log p)***. Цикл извлечения элементов из очереди суммарно выполняется количества раз не более, чем количество занесений элементов в очередь, поэтому (учитывая, что время извлечения элемента из приоритетной очереди также логарифмическое) мы имеем время работы данного цикла также равным ***O(p log p)***. Т.о. получаем, что суммарное время работы алгоритма Дейкстры на основе STL равно ***O(p log p)***.
■

Лекция 10

Бинарные деревья поиска

Бинарными деревьями называют деревья, в которых, как правило, задан *корневой элемент* или *корень* и для каждой вершины существует не более двух *потомков*. Например, для задания одной вершины бинарного дерева целых чисел в языке С можно использовать следующую структуру

```
        typedef struct STree_
        {
 int value;
        struct STree_ *prev;
        struct STree_ *left, *right;
} STree;
```

здесь указатель ***prev*** указывает на родительский элемент данной вершины, а ***left*** и ***right*** – на двух потомков, которых традиционно называют ***левым*** и ***правым***.
Величина ***value*** называется ***ключом*** вершины.
Бинарное дерево называется ***деревом поиска***, если для любой вершины дерева ***a*** ключи всех вершин в правом поддереве больше или равны ключа ***a***, а в левом – меньше. Неравенства можно заменить на строгие, если известно, что в дереве нет равных элементов.
Отметим, что бинарные деревья поиска можно реализовывать либо с помощью вершин **без** указателя на родителя (в этом случае почти все операции с деревом реализуются через рекурсию, что дает довольно простой код, но вводит жесткие ограничения на размер дерева), либо (как показано выше) с помощью вершин дерева **с** указателем на родителя (сложный код, но нет искусственных ограничений на размер дерева).
Далее мы постараемся придерживать синтаксиса языка С, т.е., как правило, мы будем использовать в качестве вершины дерева ***a*** переменную типа

```
        STree *a;
```

Отсутствие потомка обозначается нулевым значением соответствующего указателя.

Часто, если это не будет приводить к двусмысленностям, под сравнением элементов мы будем подразумевать сравнение соответствующих ключей.
Ветвью дерева называется последовательность вершин дерева, каждый последующий элемент в которой является потомком предыдущего.
Длиной ветви дерева называется количество элементов в ветви.
Высотой дерева называется максимальная длина всех ветвей дерева.
Основные операции, которые можно совершать с бинарными деревьями:

- создание дерева
- очистка дерева
- удаление дерева
- пусто ли дерево?

- поиск элемента в дереве (т.е. элемента с ключом, равным заданному)
- добавление элемента в дерево
- удаление элемента из дерева
- поиск минимального и максимального элемента в дереве
- если рассмотреть дерево поиска, как упорядоченную по возрастанию последовательность элементов, то: для текущего элемента поиск следующего/ предыдущего
- для данной вершины дерева ***v*** разбиение дерева поиска ***T*** на два дерева поиска T_1 и T_2 таких, что все элементы в T_1 меньше или равны ***v***, и все элементы в T_2 больше или равны ***v***
- для двух деревьев поиска T_1 и T_2 таких, что все элементы в T_1 меньше или равны всех элементов в T_2 (будем далее для таких деревьев говорить, что ***дерево T_1 меньше или равно дерева T_2***: $T_1 \pounds T_2$): слияние деревьев в одно дерево поиска ***T***

Покажем, что все эти операции можно совершить за время ***O(h)***, где ***h*** – высота рассматриваемого дерева.

Поиск элемента в дереве

Требуется найти элемент, равный ***v***, в дереве. Введем понятие текущей вершины дерева ***c***. Сначала в качестве ***c*** выберем корень дерева. Рекурсивно вызывается следующая процедура:

_____

Если c==NULL то ИСКОМЫЙ ЭЛЕМЕНТ В ДЕРЕВЕ НЕ СОДЕРЖИТСЯ
Если v==c то return c
Если v^3c то выполнить эту же процедуру для c->right
иначе выполнить эту же процедуру для c->left

_____

На языке С это будет выглядеть следующим образом

```
STree *Find(STree *root, int v)
{
if(root==NULL)return NULL;
if(root->value==v)return root;
if(root->value>=v)return Find(root->right,v);
else return Find(root->left,v);
}
```

или более коротко:

```
STree *Find(STree *root, int v)
{
return root==NULL?NULL: root->value==v? root: v>=root->value?
  Find(root->right,v): Find(root->left,v);
}
```

Добавление элемента в дерево

Требуется добавить в дерево вершину ***v***.

Для этого ищем лист, после которого следует вставить ***v*** и вставляем ***v*** после него.

Алгоритмом, аналогичным поиску элемента, найдем лист ***c***, после которого следует вставить элемент ***v*** и вставим его. На языке С вставка вершины в дерево может быть выполнена следующим образом:

```
STree *Insert(STree *root, STree *v)
{
if(v->value>=root->value)
  return root->right==NULL ?
 (v->back=root,v->right=v->left=NULL,root->right=v) : Insert(root->right, v);
else
  return root->left==NULL ?
  (v->back=root,v->right=v->left=NULL,root->left=v) : Insert(root->left, v);
}
```

Отметим, что здесь активно используется разделитель выражений – запятая. Напомним, что, если несколько арифметических выражений в языке С разделены запятой, то значение всего выражения равно последнему из них.

Поиск минимального и максимального элемента в дереве

```
STree *SearchMin(STree *root)
{ return root->left==NULL?root: SearchMin(root->left);}

STree *SearchMax(STree *root)
{ return root->right==NULL?root: SearchMin(root-> right);}
```

Удаление элемента из дерева

Удаление вершины ***v*** из дерева поиска не представляет проблем, если данная вершина является листом или имеет всего одного потомка. Иначе, например, из правого поддерева ***v*** можно изъять минимальный элемент (самый левый) и поместить его на место удаленного. При этом, дерево останется деревом поиска.

Поиск следующего/предыдущего элемента в дереве

Если у текущего элемента ***v*** есть правый потомок, то следующим элементом будет минимальный элемент в поддереве, которое имеет корень ***v->right***. Иначе мы должны подниматься вверх по дереву, пока не встретиться вершина ***v***, являющаяся левым потомком своего родителя. В этом случае родитель этой вершины будет следующим элементом дерева.

```
STree *SearchNext(STree *cur)
```

```
        {
if(cur==NULL)return NULL;
if(cur->right!=NULL)return SearchMin(cur->right);
while(cur->back && cur!=cur->back->left)cur=cur->back;
        return cur->back;
}
```

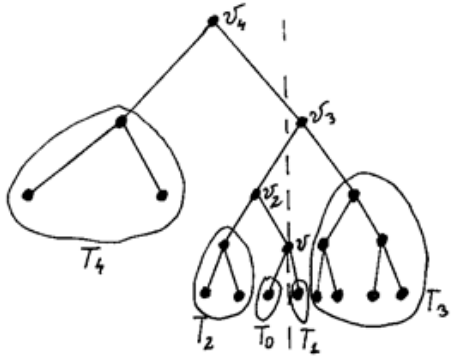
Аналогично ищется предыдущий элемент.

Слияние двух деревьев

Для двух деревьев поиска T_1 и T_2 таких, что все элементы в T_1 меньше или равны всех элементов в T_2 : слияние деревьев в одно дерево поиска T.
Выбираем из дерева T_2 наименьший элемент (самый левый), исключаем его из дерева T_2 и делаем его корнем нового дерева T. Его левым потомком будет корень дерева T_1, а правым – корень дерева T_2 . Дерево T будет деревом поиска.
Далее нам встретится немного другая задача – пусть задан некоторый элемент v и два дерева T_1 и T_2 такие, что все элементы в T_1 меньше или равны v, а все элементы из T_2 - больше или равны. Слить все указанные данные в одно дерево поиска T. Элемент v, в этой ситуации, называется ***стыковочным***. Задача в данной формулировке тривиальна.

Разбиение дерева по разбивающему элементу

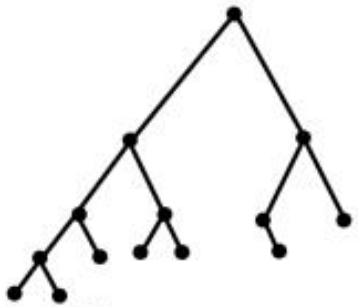
Для данной вершины дерева v разбиение дерева поиска T на два дерева поиска T_0 и T_1 таких, что все элементы в T_0 меньше v, и все элементы в T_1 больше или равны v.



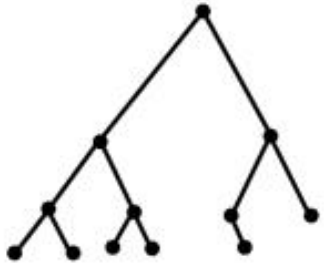
Для наглядности рассуждений расположим геометрически дерево поиска таким образом, чтобы вершины были упорядочены по оси ***OX***. Проведем на графике из вершины v вертикальную линию. Все элементы в дереве слева от этой линии – меньше или равны v, а справа – больше или равны.
Вершина v имеет два поддерева, из нее выходящих, в одном из которых все элементы меньше v , а в другом – больше или равны. Назовем эти деревья T_0 и T_1 , соответственно.
Вершины ветви, от v до корня дерева r назовем $V' = \{v_1=v, v_2, v_3, \ldots, v_n=r\}$.
Поддерево, выходящее из вершины, являющейся потомком $v_i \in V'$, назовем T_i (см. рисунок выше).
Легко доказать, что верны следующие факты для любого i:
либо для любого $j<i$: $T_i < v_i \le T_j$, $T_i < v_i \le v_j$ ($v_j \in$ правому поддереву v_i),
либо для любого $j<i$: $T_i \ge v_i > T_j$, $T_i \ge v_i > v_j$ ($v_j \in$ левому поддереву v_i).
Легко увидеть, также, что все дерево T состоит из вершин, принадлежащих либо некоторому T_i, либо V'.
Итак, конструирование двух требуемых поддеревьев будем производить следующим способом. Рассмотрим, сначала поддеревья T_0 и T_1. Будем последовательно добавлять в них данные, чтобы получить искомые деревья.
Будем далее последовательно перебирать вершины v_i для i от 2 до n.
На каждом шаге если $v_i \le v_{i-1}$, то $T_i < v_i \le T_0$ и мы сливаем деревья T_i и T_0 с с помощью стыковочного элемента v_i. Иначе $v_i > v_{i-1}$, тогда $T_i \ge v_i > T_1$ и мы сливаем деревья T_i и T_1 с с помощью стыковочного элемента v_i .
В силу вышесказанного, в конечном итоге, деревья T_0 и T_1 будут искомым разбиением исходного дерева поддеревья T с помощью вершины v.
■

Сбалансированные и идеально сбалансированные бинарные деревья поиска

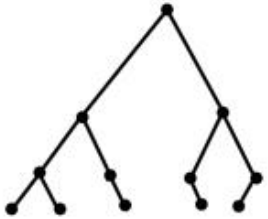
Бинарное дерево называется ***сбалансированным***, если для любой его вершины v высоты левого и правого поддерева, выходящих из v (т.е. поддеревьев с корнями ***v->left*** и ***v->right***), отличаются не более чем на 1.
Бинарное дерево называется ***идеально сбалансированным***, если длины всех ветвей, начинающихся в корне дерева и заканчивающихся в узле с хотя бы одним из нулевых указателей ***v->left*** и ***v->right***, отличаются не более чем на 1.
В литературе встречается другое (неравносильное) определение идеально сбалансированных деревьев. Будем называть ***идеально сбалансированными'*** деревьями такие деревья, для которых для каждой вершины количество элементов в левом и правом поддереве отличается не более, чем на единицу.



Сбалансированное дерево



Идеально сбалансированное дерево



Идеально сбалансированное дерево

Следующее условие равносильно условию идеально сбалансированности дерева: длины любых двух ветвей, начинающихся в одной вершине дерева, отличаются не более чем на 1. Доказательство тривиально. Из данного условия сразу же вытекает

Теорема. *Идеально сбалансированное дерево является сбалансированным.*

Иными словами, можно сказать, что для идеально сбалансированного дерева полностью заполнены элементами все слои дерева, кроме, быть может, последнего. Т.о. для идеально сбалансированного дерева высоты ***h*** количество элементов лежит в пределах $2^{h-1} \pounds N < 2^h$. из чего сразу же вытекает следующая элементарная

Теорема. *Для идеально сбалансированного дерева, состоящего из **N** вершин, высота дерева **h** лежит в пределах*

$$\log_2 N < h \pounds 1 + \log_2 N.$$

Оказывается, верна следующая

Теорема. *Идеально сбалансированное' дерево является идеально сбалансированным.*

Доказательство. Докажем данную теорему по индукции. Для деревьев высоты не более 1 теорема верна. Пусть для деревьев высоты *h* теорема верна, докажем ее для деревьев высоты *h +1*.

По определению идеально сбалансированных' деревьев, каждое поддерево такого дерева – идеально сбалансировано', а по условию индукции левое и правое поддеревья корня дерева высоты *h+1* – идеально сбалансированы. В идеально сбалансированных деревьях высоты *l* для количества элементов дерева ***N*** выполнено соотношение $2^{l-1} \pounds N < 2^l$, из чего сразу вытекает: если у двух идеально сбалансированных деревьев количество элементов в них отличается не более, чем на единицу, то либо их высоты равны, либо (если количество элементов в них, соответственно, равно 2^l-1 и 2^l) в меньшем дереве последний слой полностью заполнен. В обоих случаях длины всех ветвей обоих деревьев, начинающихся в корне, заканчивающихся в вершинах, не имеющих хотя бы одного потомка, отличаются не более, чем на единицу. Отсюда сразу вытекает, что и для дерева, полученного с помощью объединения двух таких деревьев с помощью общего корневого элемента, длины всех ветвей, начинающихся в корне, заканчивающихся в вершинах, не имеющих хотя бы одного потомка, отличаются не более, чем на единицу.

■

Для сбалансированного дерева верна оценка высоты дерева через количество вершин в нем, аналогичная по порядку оценке для идеально сбалансированных деревьев.

Теорема. *Для сбалансированного дерева, состоящего из **N** вершин, высота дерева **h** имеет оценку:*

$$h = Q(\log_2 N).$$

Доказательство. Пусть t_n – минимальное количество элементов в сбалансированном дереве высоты ***n***. Тогда верна рекурсивная формула

$$t_n = t_{n-1} + t_{n-2} + 1$$

т.е. для сбалансированного дерева высоты ***n*** с минимальным количеством вершин одно из поддеревьев, дочерних корневому элементу, должно быть сбалансированным деревом высоты ***n-1*** с минимальным количеством вершин, а другое - сбалансированным деревом высоты ***n-2*** с минимальным количеством вершин.

Уравнение $t_n - t_{n-1} - t_{n-2} = 1$ имеет общее решение вида

$$t_n = c_1 l_1^n + c_1 l_2^n - 1$$

где l_i являются решениями уравнения $l^2 - l - 1 = 0$. Из этого следует, что

$$t_n = ((1+\mathrm{sqrt}(5))/2)^n (1+o(1))$$

после логарифмирования последнего равенства мы сразу получаем требуемое соотношение:

$$\log_2 C_1 + \log_2 t_n \geq n \log_2 ((\mathrm{sqrt}(5)+1)/2)$$

$$n \leq (\log_2 t_n + \log_2 C_1) / \log_2 ((\mathrm{sqrt}(5)+1)/2)$$

для некоторого $C_1 > 0$. Или, в исходных обозначениях,:

$$h \leq \log_2 N / \log_2 ((\mathrm{sqrt}(5)+1)/2) + \log_2 C_2 \leq 1.45 \log_2 N + C$$

■

Операции с идеально сбалансированным деревом

Выше мы описали ряд алгоритмов выполнения базовых операций для деревьев поиска. К сожалению, не существует быстрых алгоритмов, для выполнения этих операций для идеально сбаланированных деревьев. Однако, определение идеально сбаланированного' дерева, фактически, дает нам алгоритм его построения, что сразу же дает нам возможность строить и идеально сбаланированное дерево по тому же набору элементов. Для построения идеально сбаланированного' дерева по набору из ***N*** элементов упорядочим этот набор. После этого алгоритм построения дерева сводится к разбиению полученной последовательности $\{a_i\}_{i=1,\ldots,N}$ на последовательности $\{a_i\}_{i=1,\ldots,[N/2]}$ и $\{a_i\}_{i=[N/2]+2,\ldots,N}$. Эти последовательности либо имеют равную длину (для нечетных ***N***), либо их длина отличается не более, чем на единицу (для четных ***N***). В корень создаваемого дерева помещаем элемент $a_{[N/2]+1}$, а левое и правое поддеревья строим таким же алгоритмом, соответственно, для последовательностей $\{a_i\}_{i=1,\ldots,[N/2]}$ и $\{a_i\}_{i=[N/2]+2,\ldots,N}$.

Итак, приведенный алгоритм доказывает следующую теорему:

Теорема. *Идеально сбалансированное' (а значит и идеально сбалансированное) дерево поиска, состоящее из* ***N*** *вершин, можно построить за время, равное* $O(N \log_2 N)$.

Таким образом, идеально сбалансированные деревья поиска удобны в тех случаях, когда перестройку дерева требуется производить, относительно редко, но требуется часто искать элементы в дереве. Приведенная оценка на высоту дерева гарантирует, что данная структура данных является наилучшей, с точки зрения времени поиска элементов в множестве (отметим, что здесь мы рассматриваем толко алгоритмы поиска, основанные на сравнениях). Все остальные операции (добавления элементов, удаления элементов и т.д.) в идеально сбалансированном дереве производятся с помощью полной перестройки дерева, т.е. они производятся медленно.

Отметим, что наша задача выродилась. Идеально сбалансированные деревья полезны только для поиска элементов в этих деревьях, но тогда дерево вообще не нужно строить! Достаточно ограничиться упорядоченным массивом исходных элементов, а поиск производить с помощью бинарного поиска. В этом случае, согласно построению идеально сбалансированного дерева', поиск по дереву будет в точности совпадать с поиском по массиву, что делает само дерево ненужным.

Операции со сбалансированным деревом

Оказывается, что для сбалансированных деревьев все описанные выше операции можно модифицировать так, что они будут сохранять сбалансированность дерева, при этом, время их выполнения не будет превышать $O(\log_2 N)$ операций. Везде далее, говоря о сбалансированных деревьях, будем подразумевать, что мы имеем дело с сбалансированными деревьями поиска.

Правым и ***левым поддеревьями*** некоторой вершины дерева ***v*** называются поддеревья с корнями ***v->left*** и ***v->right***, соответственно.

Балансом вершины дерева будем называть разность высот левого и правого поддеревьев этой вершины.

Поиск элемента в дереве

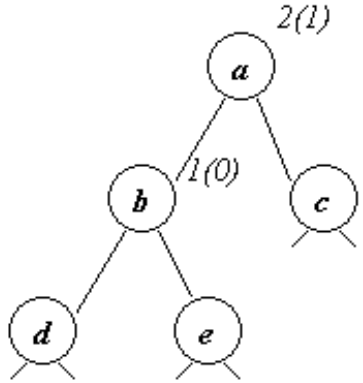
Не отличается от случая стандартных деревьев поиска.

Добавление элемента в дерево

Рассмотрим вершину дерева ***a***, в которой нарушается баланс после добавления элемента. Все нижесказанное будет верным, если это не оговорено особо, и для случая какого-либо изменения одного из поддеревьев вершины ***a***, приводящего к удлинению/укорачиванию соответствующего дерева на ***1***.

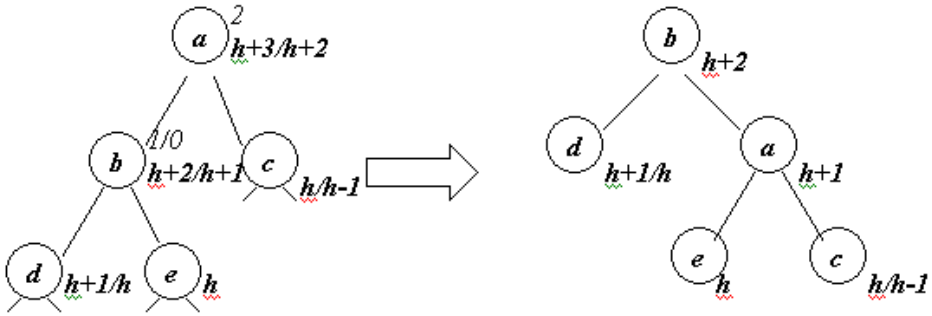
Будем предполагать, что для всех вершин, лежащих ниже ***a***, баланс по модулю не превосходит ***1***.

Пусть, для определенности, элемент добавляется в левое поддерево вершины ***a***. Справа и сверху от вершины будем писать баланс вершины после добавления новой вершины, а рядом, в круглых скобках, - баланс до добавления. Высоту соответствующего данной вершине поддерева будем писать справа снизу от вершины.



То, что дерево после добавления вершины разбалансировалось, означает, что до добавления вершины ***[a]=1***, а после добавления ***[a]=2*** (будем обозначать баланс вершины с помощью квадратных скобок: *баланс вершины* ***a = [a]***). Возможны три случая баланса вершины ***b*** после изменения дерева: ***1, 0, -1***. Рассмотрим их

1.После добавления вершины [b]=1 или 0 (или изменения поддерева с корнем b) ([b]=1 *соответствует удлинению левого поддерева* ***b)***



На рисунке варианты ***[b]=1 или 0*** печатаются через слеш (косую черту).
Приведенную здесь перестановку вершин (с соответствующими поддеревьями) будем называть ***правым поворотом*** (в соответствии с перемещением вершин ***d-b-a***).
Будем обозначать высоты дерева с корнем в вершине ***x*** h_x , а баланс этой вершины ***[x].***
Пусть $h_e=h$, тогда для случая ***[b]=1***

$h_d=h+1$ (т.к. ***[b]=1***)
$h_b=h+2$
$h_a=h+3$
$h_c=h$ (т.к. ***[a]=2***)

Из чего сразу следует, что после правого поворота
[a]=[b]=0
[c], [d], [e] не изменились

Т.о. данное дерево сбалансировалось. При этом, если изменение дерева произошло в результате добавления вершины, его высота не изменилась. Действительно, перед добавлением вершины $h_d=h$ из чего следует, что перед добавлением вершины $h_a=h+2$. После добавления высота не изменилась. Т.о. в случае ***[b]=1*** процесс балансировки дерева завершен.

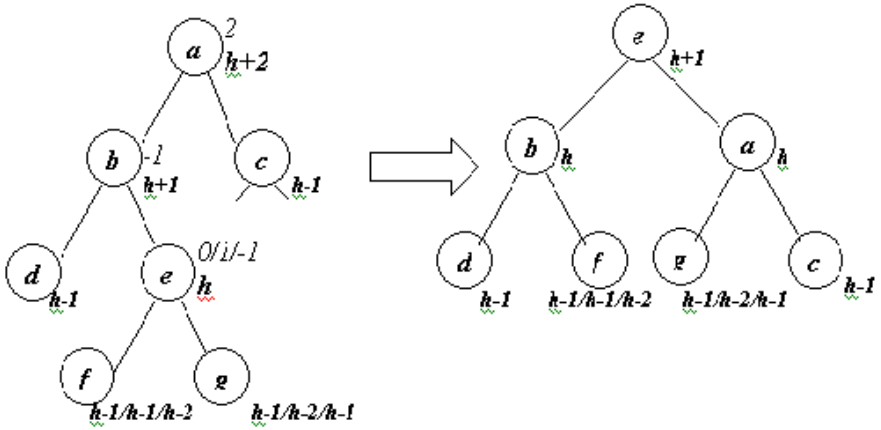
Для случая ***[b]=0*** нарисованное дерево тоже остается сбалансированным:

$h_d=h$ (т.к. ***[b]=0***)
$h_b=h+1$
$h_a=h+2$
$h_c=h-1$ (т.к. ***[a]=2***)

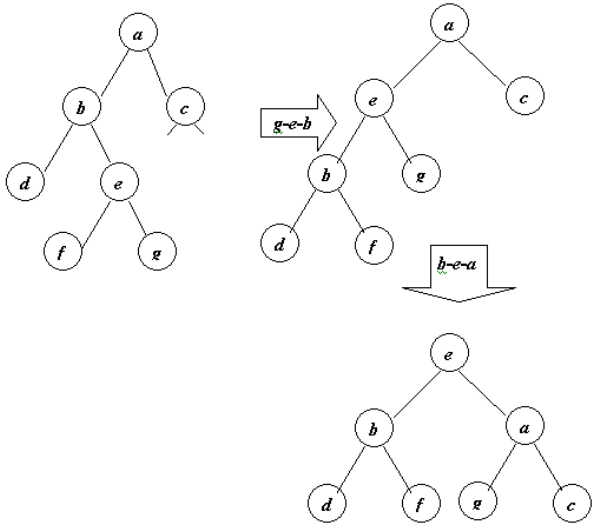
Из чего сразу следует, что после правого поворота
[a]=1, [b]=-1
[c], [d], [e] не изменились.
Однако высота всего нарисованного дерева изменяется (была до добавления $h_a=h+1$, стала $h_a=h+2$).

Итак, если перед изменением дерева $h_b=1$, то процесс балансировки завершен. Иначе, может разбалансироваться родитель вершины ***a***, и для нее нужно выполнить тот же алгоритм.

2.После добавления вершины [b]=-1 (или изменения поддерева с корнем b) *(удлинение правого поддерева **b**)*



Описанная перестановка может быть проведена за два вращения: левого ***g-e-b*** и правого ***b-e-a***:



Возможны три варианта баланса ***c: [c]=0/1/-1***. Пусть $h_e=h$, тогда, в соответствии с возможными вариантами ***[c]*** имеем:

$h_f=h-1/h-1/h-2$
$h_g=h-1/h-2/h-1$
$h_d=h-1$
$h_b=h+1$
$h_a=h+2$
$h_c=h-1$

Из чего сразу следует, что после правого поворота

[b]=0/0/1
[a]=0/-1/0
[e]=0

[d], [f] , [g] , [c] не изменились.

Т.о. данное нарисованное дерево сбалансировалось. При этом, если изменение дерева произошло в результате добавления вершины, его высота не изменилась. Действительно, перед добавлением вершины $h_e=h-1$ из чего следует, что перед добавлением вершины $h_a=h+1$. После добавления высота не изменилась. Т.о. в случае добавления вершины при ***[b]=-1*** процесс балансировки дерева завершен.

Т.о. процедура вставки вершины в сбалансированное дерево поиска сводится к нахождению вершины ***v***, после которой следует вставить новую вершину и проверке условия сбалансированности всех вершин на ветке, завершающейся в ***v***. Причем, проверку надо вести от вершины ***v*** к корню и если встретится разбалансированная вершина, то с помощью одного или двух вращений все дерево можно привести к сбалансированному.

Итак, мы доказали следующую теорему

Теорема. *В сбалансированное дерево поиска, состоящее из **N** вершин, можно добавить одну вершину за время $O(\log_2 N)$. При этом, для балансировки дерева потребуется не более двух поворотов.*

Отметим, что хотя балансировок требуется не более двух, весь процесс балансировки, все же, требует времени $O(\log_2 N)$, т.к. требуется еще найти – в какой вершине следует производить балансировку.

Удаление элемента из дерева

Удаление вершины из дерева поиска описано в параграфе ***Бинарные деревья поиска***. Нам остается только сбалансировать, возможно, разбалансированное дерево.

Т.о. процедура удаления вершины ***v*** из сбалансированного дерева поиска сводится к следующему

- нахождению вершины ***v***, которую следует удалить,
- ее удаления из дерева поиска (с помещением на ее место некторой вершины v'),
- для каждой вершины ветви дерева от v' до корня следует проверять условие балансировки – если оно нарушилось, то операциями вращения следует произвести балансировку соответствующего поддерева.

Итак, в силу построения алгоритма удаления вершины из сбалансированного дерева, верна

Теорема. *Из сбалансированного дерево поиска, состоящего из **N** вершин, можно удалить одну вершину за время $O(\log_2 N)$.*

Поиск минимального и максимального элемента в дереве

Не отличается от случая стандартных деревьев поиска.

Поиск следующего/предыдущего элемента в дереве

Не отличается от случая стандартных деревьев поиска.

Слияние двух деревьев

Для двух сбалансированных деревьев поиска T_1 и T_2 таких, что все элементы в T_1 меньше или равны всех элементов в T_2 : слияние деревьев в одно дерево поиска ***T***.

Выбираем из дерева T_2 наименьший элемент ***v*** (самый левый) и удаляем его из дерева T_2 . Элемент ***v*** называется ***стыковочным***. Для него верно: $T_1 \le v \le T_2$.

Возможна ситуация, когда стыковочный элемент присутствует уже в постановке задачи.

Будем, для определенности, предполагать, что высота дерева T_1 больше или равна высоте дерева T_2.

Рассмотрим правую ветвь дерева T_1 : $\{v_1,\ldots,v_K\}$. В силу сбалансированности дерева имеем: $h(v_i) - h(v_{i+1}) \le 2$, тогда на этой ветви найдется вершина v_l, такая что

$$h(v_l) = h(T_2) \text{ или } h(v_l) = h(T_2)+1$$

Сольем дерево с корнем в v_l и T_2 с помощью стыковочной вершины ***v*** и подставим новое дерево на место старой вершины v_l.

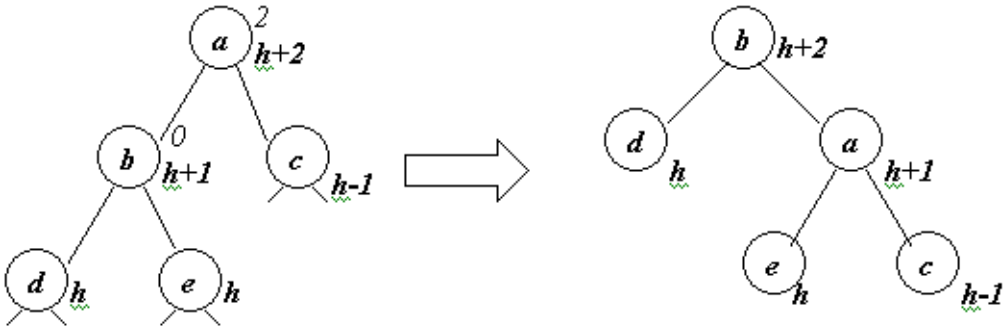
Вершина v_l и все дерево, с нее начинающееся, окажутся сбалансированными. Высота же дерева, начинающегося с v_l, увеличится на ***1***, т.к. $h(T_2) \le h(v_l)$.

Итак, в результате изменений дерева у одной вершины ***w*** длина соответствующего поддерева увеличилась на ***1***. Далее нам следует запустить стандартную процедуру балансировки дерева. Мы должны пройти ветвь, заканчивающуюся на ***w***, от ***w*** до корня и в каждой вершине проверить баланс. Если он будет по модулю больше ***1***, то баланс в данной вершине следует скорректировать одним или двумя вращениями.

Если, при этом, длина данного поддерева восстановится до значения, которому она была равна до слияния деревьев, то далее процесс проверки

сбалансированности производить не надо (т.к. дерево T_1 до слияния было сбалансированным). Иначе, процесс проверки следует продолжить.

Отметим, что, в отличии от добавления к дереву одной вершины, в данном случае после уравновешения одной вершины процесс может не завершиться, т.к. возможен следующий вариант



Этот вариант не мог реализоваться при добавлении одной вершины к дереву. В этом случае до слияния деревьев высота дерева с корнем ***a*** была равна ***h+1***, а после слияния она стала равной ***h+2***.

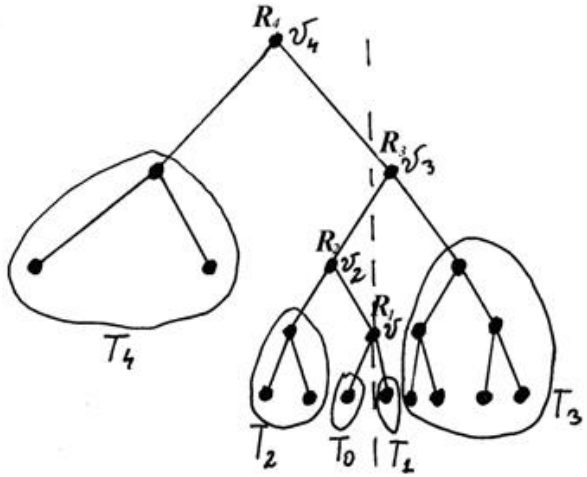
Итак, в силу построения алгоритма слияния двух сбалансированных деревьев, верна

Теорема. *Для двух сбалансированных деревьев поиска* T_1 *и* T_2 *, состоящих из* N_1 *и* N_2 *вершин, имеющих высоты* h_1 *и* h_2*, и элемента* v*, таких, что все элементы в* T_1 *меньше или равны* v *и* v *меньше или равно всех элементов в* T_2 *: слияние деревьев с помощью стыковочного элемента* v *в одно сбалансированное дерево поиска* T *можно произвести за время* $T=O(\log_2 (N_1+N_2))$ *или за время* $T=O(|h_1-h_2|)$*. Указанные деревья* T_1 *и* T_2 *можно слить за время* $T=O(\log_2 (N_1+N_2))$**.**

Отметим здесь, что для слияния двух сбалансированных деревьев требуется сначала извлечения стыковочный элемент из дерева с большей высотой, поэтому эта операция требует большего времени, чем слияние деревьев с готовым стыковочным элементом.

Разбиение дерева по разбивающему элементу

Для данной вершины дерева ***v*** разбиение сбалансированного дерева поиска ***T*** на два сбалансированных дерева поиска T_1 и T_2 таких, что все элементы в T_1 меньше или равны ***v***, и все элементы в T_2 больше или равны ***v***.



Алгоритм, практически полностью, совпадает с алгоритмом разбиения обычного дерева поиска. Только, теперь, нам следует пользоваться алгоритмом слияния деревьев для сбалансированных деревьев поиска.

Пусть высота дерева T_0 равна s_0. Пусть с деревом T_0 будут последовательно сливаться деревья T с индексами $i_1,\ldots,i_K$ ($i_K \pounds h$), в результате чего будут получаться деревья $S_1, S_2,\ldots, S_K$. Будем считать $S_0= T_0$. Т.е. $S_{j-1}+ T_{ij} \to S_j$.

Высота дерева S_j равна либо $MAX(h(S_{j-1}),h(T_{ij}))$, либо $MAX(h(S_{j-1}),h(T_{ij}))+1$.

Пусть R_i – деревья с корнями в вершинах v_i. Высота дерева R_i равна либо $h(T_i)+1$ либо $h(T_i)+2$. $h(R_i)$ строго возрастают.

Покажем по индукции , что высота дерева S_j равна либо $h(R_{ij})$, либо $h(R_{ij})-1$, либо $h(R_{ij})-2$. Пусть данное свойство выполнено для $l<j$, тогда

$h(S_j)=MAX(h(S_{j-1}),h(T_{ij})) \mid MAX(h(S_{j-1}),h(T_{ij}))+1$ следовательно

$h(S_j)= MAX(h(S_{j-1}),h(R_{ij}) -1) \mid MAX(h(S_{j-1}),h(R_{ij}) -2) \mid MAX(h(S_{j-1}),h(R_{ij}) -1)+1 \mid MAX(h(S_{j-1}),h(R_{ij}) -2)+1$ следовательно по индукции

$h(S_j)= (h(R_{ij}) -1) \mid ((h(R_{ij}) -1) \mid h(R_{ij}) -2) \mid h(R_{ij}) \mid (h(R_{ij}) \mid h(R_{ij}) -1)$ следовательно

$h(S_j)= h(R_{ij}) \mid (h(R_{ij}) -1) \mid (h(R_{ij}) -2)$

Здесь под вертикальной чертой подразумевается разделение возможных вариантов.

Время работы всего алгоритма

$T=O(|h(T_0)- h(T_{i1})|+1 +| h(S_1)- h(T_{i2})|+1 +\ldots+| h(S_{K-1}) - h(T_{iK})|+1)=$

$=O(|h(T_0)- h(T_{i1})| +| h(S_1)- h(T_{i2})| +\ldots+| h(S_{K-1}) - h(T_{iK})|)+O(h)=$

$=O(|h(T_0)- h(R_{i1})|+4 +| h(R_{i1})- h(R_{i2})|+4 +\ldots+| h(R_{i(K-1)}) - h(R_{iK})|+4)+O(h)=$
(в силу возрастания $h(R_i)$ *)*
$=O(|h(T_0)- h(R_{i1})|+| h(R_{i1})- h(R_{i2})|+\ldots+| h(R_{i(K-1)}) - h(R_{iK})|)+O(h)= O(h)$

Т.о. $T= O(h)=O(\log_2 N)$, где N – количество вершин в суммарном дереве.

Итак, верна следующая

Теорема. *Для данной вершины* ***v*** *сбалансированного дерева поиска* ***T*** *разбиение на два сбалансированных дерева поиска* T_1 *и* T_2 *таких, что все элементы в* T_1 *меньше или равны* ***v****, и все элементы в* T_2 *больше или равны* ***v****. может быть произведено указанным алгоритмом за время* $= O(\log_2 N)$*, где* N *– суммарное количество вершин в деревьях* T_1 *и* T_2*.*

Лекция 11

Красно-черные деревья

Красно-черными деревьями называют бинарные деревья поиска, у которых для каждой вершины добавляется дополнительное свойство: вершина является черной или красной. При этом требуется выполнение следующих свойств:

- корень дерева – черный
- у каждой красной вершины потомки – черные
- в любых двух ветвях от корня до листа количество содержащихся черных вершин равно (здесь *листом* называется вершина, у которой есть не более одного потомка)

Для простоты реализации в дерево добавляются фиктивные черные вершины: для каждой вершины дерева, при отсутствии у нее потомка, на место соответствующего потомка вставляется фиктивная черная вершина. Вершины, отличные от фиктивных, называются ***внутренними***. Будем далее называть ***листьями*** вершины, у которых хотя бы один потомок фиктивный. При определении высоты дерева фиктивные вершины учитывать не будем.

Например, для задания одной вершины красно-черного дерева целых чисел в языке С можно использовать следующую структуру

```
typedef struct SBTree_
{
int IsRed;
int value;
struct STree_ *par;
struct STree_ *left, *right;
} SBTree;
```

здесь указатель ***par*** указывает на родительский элемент данной вершины, а ***left*** и ***right*** – на двух потомков, которых традиционно называют ***левым*** и ***правым***. Целая переменная ***IsRed*** указывает – является ли данная вершина красной. Величина ***value*** называется ***ключом*** вершины.

Отступление на тему языка С. Поля структур.

В вышеприведенном примере кажется весьма накладным использовать целую переменную для хранения всего одного бита информации. Можно попробовать отвести под эту переменную меньше памяти:

```
typedef struct SBTreeX_
{
char IsRed;
int value;
struct STree_ *par;
struct STree_ *left, *right;
} SBTreeX;
```

Однако, в силу наличия выравнивания в структурах, для большинства современных машин размеры структур ***SBTreeX*** и ***SBTree*** окажутся равными.

Можно попробовать `отщипнуть' один бит для переменной ***IsRed*** из целой переменной с ключом данной структуры ***value***. Это можно сделать с помощью ***полей*** в структурах. Поля в структурах это – переменные целого типа, при описании которых после имени переменной пишется двоеточие и вслед за ним – количество бит, которые должны быть отведены под данную переменную. Например, в нашем случае, можно определить вершину дерева следующим образом:

```
typedef struct SBTree1_
{
unsigned int IsRed :1;
unsigned int value :31;
struct STree_ *par;
struct STree_ *left, *right;
} SBTree1;
```

При этом, следует понимать, что теперь каждая операция с членами структуры ***IsRed*** и ***value*** будет происходить довольно сложно (имеется реализация данной операции в кодах). Действительно, например, для изменения переменной ***value*** ее сначала требуется извлечь из структуры (используя битовые операции), изменить, а затем – поместить обратно.

Следует ожидать, что на IBM-совместимых ЭВМ работа со следующей структурой ***SBTree2*** будет происходить медленнее, чем со структурой ***SBTree1***:

```
typedef struct SBTree2_
{
unsigned int value :31;
```

```
        unsigned int IsRed :1;
struct STree_  *par;
struct STree_  *left, *right;
        } SBTree2;
```

Здесь используется следующий факт: на IBM-совместимых ЭВМ переменные типа ***int*** занимают 4 байта и байты располагаются в обратном порядке: от старшего к младшему. Поэтому для извлечения целой переменной из структуры ***SBTree1*** требуется скопировать первые 4 байта структуры в отдельную переменную и обнулить старший бит этой переменной. Для структуры ***SBTree2*** после извлечения первых четырех байт из структуры во внешнюю целую переменную надо еще дополнительно сдвинуть все биты целой переменной вправо на 1 бит.

Простейшие тесты подтверждают данное предположение. Естественно, что разные компиляторы по разному оптимизируют работу с битовыми полями. Так, например, компилятор *Microsoft Visual C++* почти нивелирует разницу в скорости работы со всеми описанными типами структур (разница в скорости элементарных операций с данными структурами оценивается примерно 10-20%). Для используемого же компилятора ***gnu C++*** разница в скорости оказалась – вдвое.

Отметим, что данный подход применим далеко не всегда. Поля в структурах обязаны иметь тип *unsigned int***.** В современных версиях языка ***C*** это требование немного ослабло и вместо этого типа часто можно использовать другие целые типы, но, например, тип *float* все равно использовать нельзя. Пример использованной программы прилагается.

Отступление на тему языка C. Бинарные операции.

В языке C есть возможность стандартной работы с битами, в рамках возможностей, предоставляемых обычными ассемблерами. Для работы с битами используются следующие арифметические операции:

- арифметическое ***и***: &
- арифметическое ***или***: |
- арифметическое ***не***: ~
- арифметическое ***исключающее или***: ^
- сдвиг влево на ***k*** разрядов: <<***k***
- сдвиг вправо на ***k*** разрядов: >>***k***

С помощью этих операций можно осуществить базовые операции с битами:

k-ый бит целого числа ***i == 0?*** *:* ***(i&(1<<k))==0***

Положить ***1*** *в* ***k****-ый бит целого числа* ***i*** *:* ***i|=(1<<k)***

Положить ***0*** *в* ***k****-ый бит целого числа* ***i*** *:* ***i&=~(1<<k)***

Присвоить ***l****-ый бит целого числа* ***j*** ***k****-тому биту* ***i*** *:*

```
i = ((j&(1<<l))==0) ? (i&(~(1<<k))) : (i|(1<<k))
```

Высота красно-черного дерева

Наводящим соображением на то, что в красно-черном дереве, состоящем из ***N*** вершин, высота ***h*=Q(*$log_2 N$*)** , является следующий факт: в каждой ветви дерева, начинающейся с корня дерева, не менее половины вершин – черные (т.к., по определению красно-черного дерева, вслед за красной вершиной всегда следует черная и ветвь, начинающаяся с корня дерева, начинается с черной вершины), с другой стороны: в каждой ветви находится равное количество черных вершин (следует заметить, что, тем не менее, в красно-черном дереве черных вершин может быть меньше половины количества всех вершин).

Назовем ***черной высотой*** дерева с корневой вершиной ***r*** максимальное количество черных вершин во всех ветвях, начинающихся в ***r*** и заканчивающихся в листьях, не считая саму вершину ***r***. Будем обозначать ее ***hb(r)***.

Заметим, что требование черноты корня красно-черного дерева, вообще говоря, не является обязательным. Действительно, если не использовать это свойство в определении красно-черного дерева, то в таком дереве цвет корня дерева можно заменить с красного на черный с сохранением всех остальных свойств красно-черных деревьев. Будем называть дерево *красно-черным'* если из определения красно-черного дерева убрать требование черноты корня. Легко показать, что для красно-черного дерева любое его поддерево является красно-черным'.

Верна следующая

Лемма. *В красно-черном дереве с черной высотой* ***hb*** *количество внутренних вершин не менее* $2^{hb+1}-1$.

Доказательство. Заметим, что смена цвета корня дерева не повлияет на черную высоту дерева. Поэтому данную лемму можно доказать для красно-черных' деревьев. Будем доказывать лемму по индукции по высоте красно-черного' дерева (обычной). Если рассмотреть дерево, состоящее из одного элемента, то для него лемма верна.

Рассмотрим внутреннюю вершину ***x***. Пусть ***hb(x)=h***. Тогда если ее потомок ***p -*** черный, то высота ***hb(p)=h-1***, а если – красный, то ***hb(p)=h***. Т.о., по предположению индукции, в поддеревьях (а они тоже являются красно-черными' деревьями) содержится не менее 2^h-1 вершин, а во всем дереве, соответственно, не менее $2^h-1 + 2^h-1 + 1=2^{h+1}-1$.

■

Если обычная высота красно-черного дерева равна ***h***, то черная высота дерева будет не меньше ***h/2-1*** и, по лемме, количество внутренних вершин в дереве

$$N \geq 2^{h/2}-1.$$

Прологарифмировав неравенство, имеем:

$$\log_2 (N+1) \geq h/2$$

$$2\log_2(N+1) \geq h$$
$$h \leq 2\log_2(N+1)$$

Итак, учитывая, что для любого бинарного дерева $h > \log_2 N$, получаем, что доказана следующая

Теорема. *Для красно-черного дерева, имеющего N внутренних вершин, верна следующая оценка для его высоты*

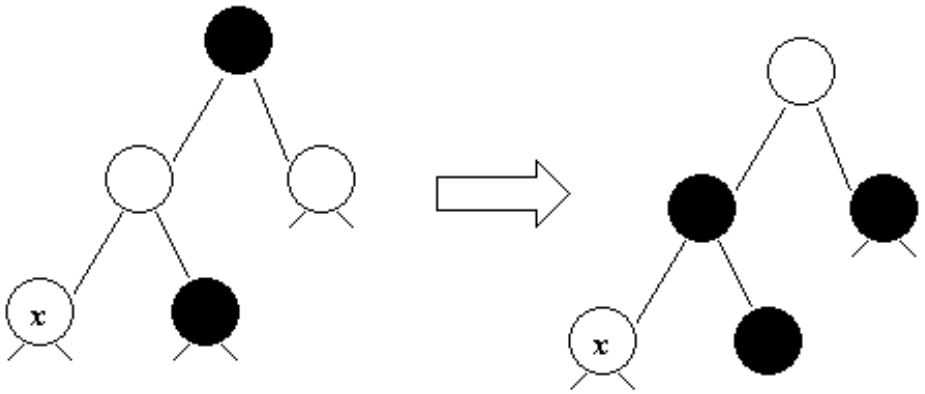
$$h=\Theta(\log_2 N),$$

или, более точно,

$$\log_2 N < h \leq 2\log_2(N+1).$$

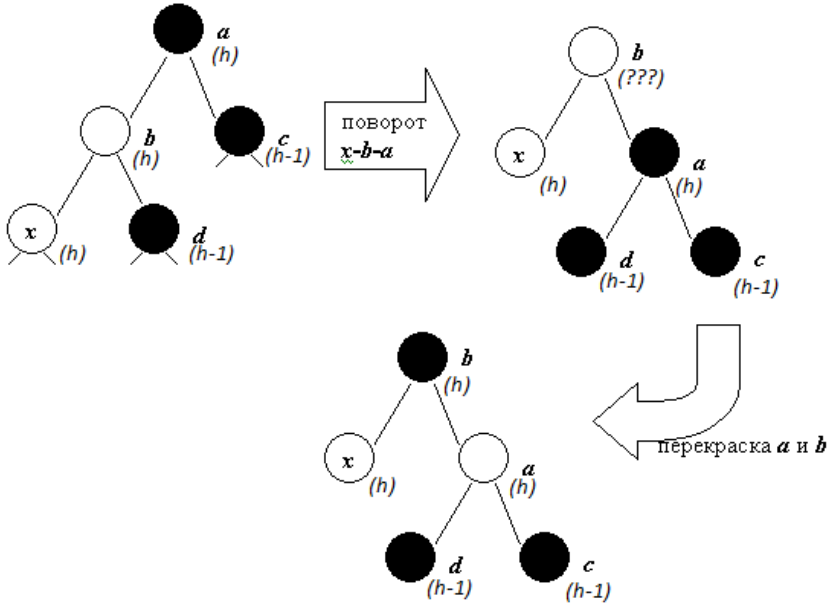
Добавление элемента в красно-черное дерево

Новая вершина вставляется в красно-черное дерева в два этапа.
На первом этапе вершина вставляется, как в обычное дерево поиска (без фиктивных вершин). Новая вершина красится в красный цвет. Следует отметить, что, в реальности фиктивных вершин может вообще не быть. Их наличие может обозначаться соответствующими нулевыми указателями у родительских вершин.
Добавление красной вершины ***x*** не меняет баланса дерева по черным вершинам.
Т.к. потомки новой вершины – фиктивные, то они – черные, по определению, что соответствует определению красно-черного дерева.
Единственная проблема, которая может возникнуть, это то, что у вставленной красной вершины ***x*** может оказаться красный родитель. Требуется изменить дерево, чтобы решить эту проблему.
При преобразованиях дерева мы будем сохранять указанное свойство: у нас будет сохраняться балансировка по черным вершинам и единственная возможная проблема это – некоторая красная вершина ***x*** будет иметь красного родителя.
Итак, ***x->par*** – красная, то ***x->par->par*** – черная (т.к. единственная проблема – нестыковка ***x->par*** и ***x***, с другой стороны, у красной вершины может быть только черный родитель).
Будем называть вершину ***x->par->par->next***, где ***next*** это – ***left*** или ***right дядей*** вершины ***x***, если ***x->par->par->next!= x->par***.
Рассмотрим все возможные случаи.
0. Если вершина вставляется в пустое дерево, то она просто перекрашивается в черный цвет.
1. У вершины ***x->par*** нет родителя, т.е. эта вершина – корневая. В таком случае мы просто перекрашиваем вершину ***x->par*** в черный цвет и процесс завершается.
2. Дядя вершины - ***x*** красный.

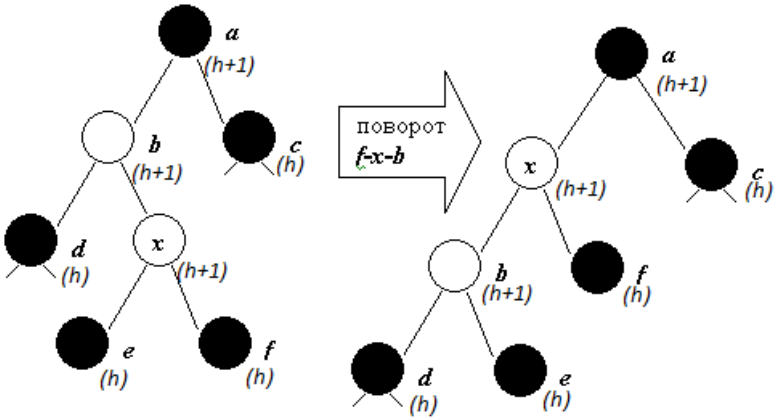


Перекрашиваем родителя, деда и дядю вершины ***x*** и рассматриваем в качестве вершины ***x*** ее деда: ***x=x->par->par.***
Т.о. мы перенесли проблему выше по ветви дерева.
Осталось рассмотреть случаи, когда дядя вершины ***x*** - черный.

3. Дядя вершины - ***x*** черный, ***x*** – левый потомок ***x->par.*** Будем в скобках справа от вершины писать черную высоту поддерева (поддерево должно быть красно-черным'), начинающегося с данной вершины. Символами *???* будем обозначать дисбаланс высот (=поддерево не является красно-черным').



В этом случае мы проводим правый поворот ***x-b-a*** и в получившемся дереве перекрашиваем две вершины: ***a*** и ***b*** .
Вершина получившегося дерева – черная, проблем с цветами нет, сохранилась черная высота дерева, начинающегося с корня, баланс черного сохранился. Т.о. дерево сбалансировано. Последующая балансировка не требуется.
4. Дядя вершины - ***x*** черный, ***x*** – правый потомок ***x->par.***



Делаем левый поворот ***f-x-b*** и ситуация сводится к предыдущему случаю. Заметим, что при этом сохранилась черная высота дерева, начинающегося с корня, баланс черного сохранился.

Все случаи рассмотрены.

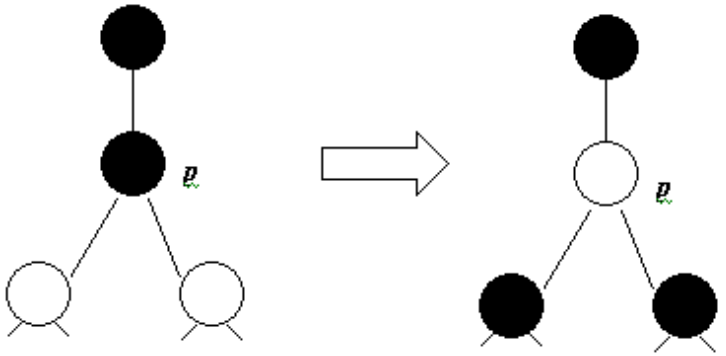
Итак, после добавления вершины процесс приведения дерева к виду красно-черного дерева сводится к некоторому количеству процедур перекраски ***1*** (не более ***h*** раз, где ***h*** – высота дерева) и не более чем к двум поворотам. Причем, после поворотов дерево не требует дальнейших изменений.

Итак, мы доказали следующую теорему

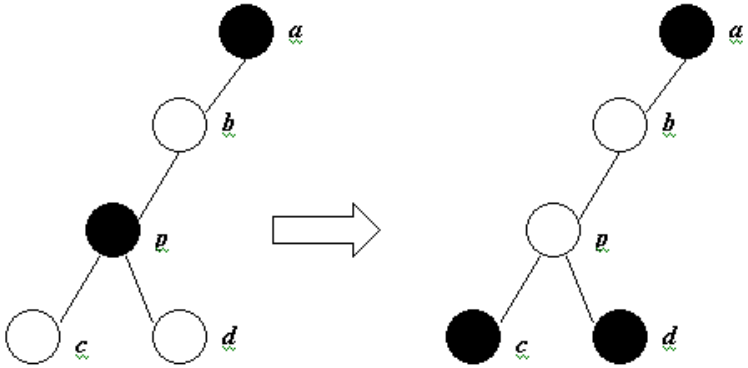
Теорема. *Указанный алгоритм позволяет добавлять вершину к красно-черному дереву за время* $T=O(\log_2 N)$ *операций, где* N *– количество вершин в дереве.*

Однопроходное добавление элемента в красно-черное дерево

Отметим, что красно-черные деревья имеют несколько худшую оценку высоты в зависимости от количества вершин в дереве, чем сбалансированные деревья. В реальной практике высоты деревьев различаются не существенно (имеется в виду – в среднем). Преимущество красно-черных деревьев является то, что добавление вершин может быть осуществлено за один проход по соответствующей ветви дерева. В сбалансированных деревьях требуется два прохода: один – для того, чтобы найти вершину, после которой следует вставить новую вершину, а второй – для балансировки дерева.
Итак, все, что нам нужно, это – не допустить в процессе поиска элемента, после которого будет вставлен новый элемент, реализации случая ***1***, т.к. случай ***3*** сводится к случаю ***2***, а последний завершает алгоритм. Это можно обеспечить, перекрашивая вершины при поиске листа, после которого следует вставить новую вершину. Иными словами, нам следует обеспечить, чтобы либо у вставляемой вершины был бы черный родитель (тогда ничего больше делать не надо), либо у вставляемой вершины был бы черный дядя (тогда дерево можно сделать красно-черным за один или два поворота).
При поиске листа, после которого следует вставить новую вершину, мы, сначала рассматриваем в качестве текущей вершины ***p*** корень дерева. Далее в качестве ***p*** рассматриваем один из потомков корня, и т.д. Пусть, для определенности, от вершины ***p*** мы переходим к вершине ***p->left***, тогда нам следует обеспечить, чтобы в случае если ***p->left*** была красной, то ***p->right*** должна стать черной вершиной.
Рассмотрим все возможные случаи. Следует рассматривать только случаи, когда оба потомка ***p*** красные (легко увидеть, что случаи, когда ***p->left*** или ***p->right*** – черные нас устраивают).
0. p – корень дерева, оба потомка ***p*** – красные. Тогда, все, что нужно сделать – перекрасить обоих потомков ***p*** в черный цвет и перейти к рассмотрению следующей вершины ***p->left***.
1. p->par – черный, оба потомка ***p*** – красные. Тогда, все, что нужно сделать – перекрасить ***p*** и его потомков и перейти к рассмотрению следующей вершины ***p->left***.

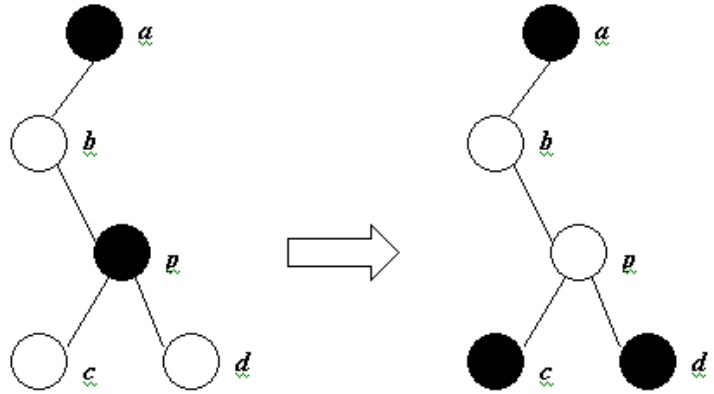


2. p->par – красный, причем ***p*** – левый потомок ***p->par***, ***p->par*** – левый потомок ***p->par->par***, оба потомка ***p*** – красные (случай, когда оба потомка – правые аналогичен) .



Сначала, мы перекрашиваем ***p*** и его потомков. Теперь мы попали в ситуацию, аналогичную случаю ***3***, рассмотренному выше (заметим, что правый потомок ***a*** должен быть черным, что обеспечено на предыдущем шаге прохода). Как было показано выше, за один поворот и одну перекраску проблему можно решить.

3. p->par – красный, причем ***p*** – правый потомок ***p->par***, ***p->par*** – левый потомок ***p->par->par***, оба потомка ***p*** – красные (случай, когда ***p*** – левый потомок ***p->par***, ***p->par*** – правый потомок ***p->par->par*** - аналогичен).



Сначала, мы перекрашиваем ***p*** и его потомков. Теперь мы попали в ситуацию, аналогичную случаю ***4***, рассмотренному выше. Как было показано выше, за два поворота и одну перекраску проблему можно решить.

На самом деле, стоит отметить некоторую тонкость: несмотря на то, что нашей целью при проходе сверху вниз была ликвидация случая наличия двух красных братьев (после которых вставляется новый элемент!), мы все-равно можем иметь после прохода ситуации двух красных братьев вдоль пройденной ветви (случай ***3*** при вставке элемента создает именно такую ситуацию!). Предлагается самостоятельно осознать, что это замечание не мешает нам решать поставленную задачу.

Итак, мы показали, что алгоритм добавления вершины к красно-черному дереву можно реализовать за один проход по соответствующей ветви дерева.

Удаление элемента из красно-черного дерева

Сначала мы удаляем вершину, как в обычном дереве поиска.

Если у удаляемой вершины ***y*** всего один внутренний потомок ***x***, то мы просто ставим ***x*** на место ***y***. Если вершина ***y*** была красной, то проблем не возникает (черная длина дерева не изменяется). Если вершина ***y*** – черная, а ***x*** – красная, то проблем тоже нет: мы перекрашиваем вершину ***x***, вставшую на место вершины ***y***, в черный цвет и RB-свойства будут выполняться. Наконец, если обе вершины ***x*** и ***y*** – черные, то нам придется присвоить вершине ***y*** двойную черноту. Как с ней бороться – будет ясно далее.

Если у удаляемой вершины ***y*** два внутренних потомка ***w=y->right, z=y->left***, то мы извлекаем следующий элемент за ***y*** (минимальный в дереве с корнем ***w***) и ставим его на место ***y***.

Теперь все проблемы сместились к вершине, у которой нет внутренних потомков. При ее удалении она становится фиктивной, что не будет противоречить дальнейшему алгоритму. Если данная вершины была красной, то она просто перекрашивается в черный цвет (уже – в качестве фиктивной вершины). Если же она была черной, то ей необходимо приписать двойную черноту.

Итак, задача сводится к следующей. *Есть вершина в красно-черном дереве **x**, обладающая двойной чернотой. Все свойства красно-черного дерева выполняются. Требуется привести дерево к такому виду, что в нем все вершины будут просто черными или красными.*

Мы будем производить некоторые манипуляции с окрестностью вершины ***x***, которые будут или перебрасывать двойную черноту вверх по дереву, или окончательно приведут дерево к требуемому виду. Если корень дерева приобретет двойную черноту, мы просто сделаем его черным, что завершит работу алгоритма.

Рассмотрим различные варианты:

1)брат ***x*** – красный;

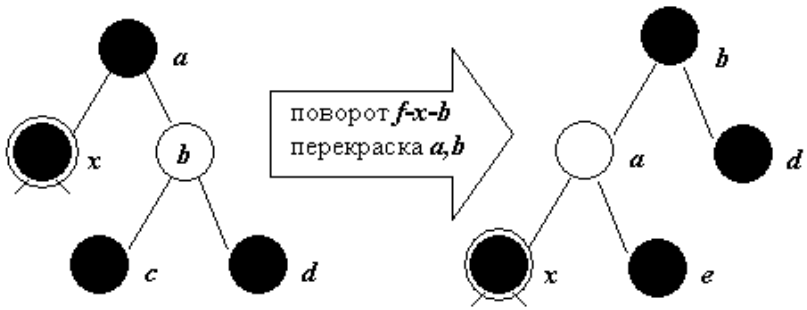
2-4: брат ***x*** – черный:

2)потомки брата ***x*** – черные;

3)правый потомок брата ***x*** – черный, левый – красный;

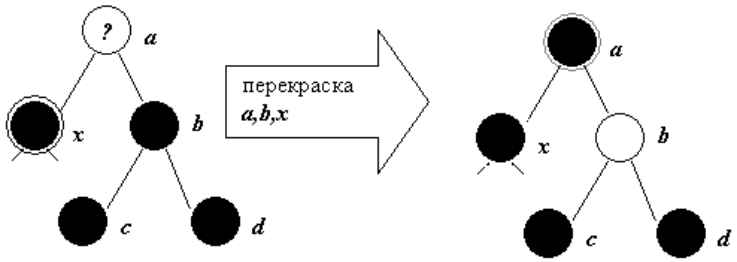
4)правый потомок брата ***x*** – красный.

1) брат ***x*** красный.



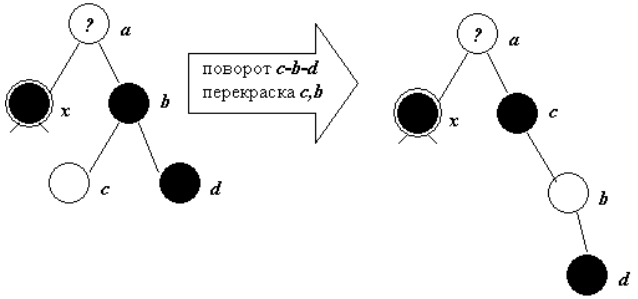
x остается с двойной чернотой, но получает черного брата. Ситуация сводится к вариантам 2-4.

2) брат ***x*** – черный, потомки брата ***x*** – черные.



Одна чернота ***x*** и чернота ***b*** переходят к их родителю. Если родитель был красным, то процесс на это завершается. Иначе рассматриваем далее в качестве вершины ***x*** вершину ***a***.

3) правый потомок брата ***x*** – черный, левый – красный.



Делаем правый поворот ***c-b-d*** и перекрашиваем вершины ***b*** и ***c***. В результате, получаем, что правый потомок брата ***x*** – красный, т.е. приходим к случаю 4.

4) правый потомок брата ***x*** – красный, левый – не важно.

Делаем левый поворот ***d-b-a*** и делаем указанную перекраску (***x*** становится просто черной). При этом цвет корня дерева и вершины ***c*** не должны меняться. Разберемся с балансировкой. Пусть ***hb(x)=h*** (не забывать, что в ***hb(x)*** не учитывает сама вершина ***x***). То ***hb(a)=h+2, hb(b)=h+1=hb(d)=*** количеству черных вершин в любой ветви, начинающейся на ***c***. Отсюда, в результате простой проверки, получаем, что новое дерево является красно-черным.

Итак, мы завершили разбор операции удаления вершины в красно-черном дереве.

Лекция 12

B-деревья

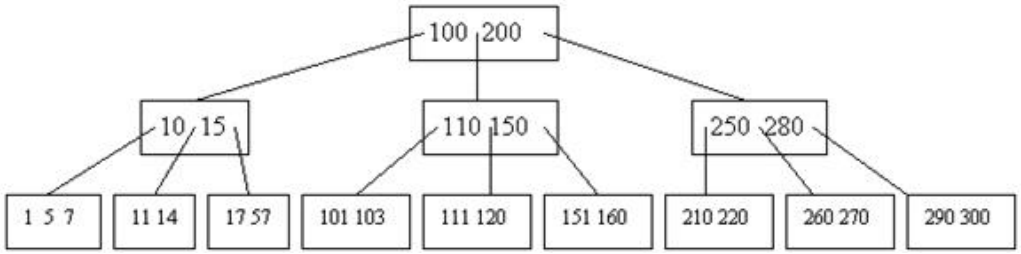
До сих пор мы рассматривали только бинарные деревья. Теперь рассмотрим деревья, имеющие бОльшую степень ветвления. При этом хочется, чтобы сохранялись свойства, аналогичные свойству сбалансированности в сбалансированных деревьях. Этим условиям удовлетворяют ***B-деревья***.
Отметим, что B-деревья являются основным инструментом построения многих современных файловых систем (NTFS, RaiserFS, JFS, XFS).

В B-деревьях в каждой вершине может содержаться несколько элементов (ключей). Высота дерева определяется как максимальное количество вершин в ветвях. Будем далее рассматривать случай, когда все элементы (ключи) в дереве различны.

B-дерево степени ***n*** определяется следующим образом

- каждая вершина дерева, кроме корня, содержит от ***n-1*** до ***2n-1*** элемента (ключей) и от ***n*** до ***2n*** ссылок на дочерние элементы; корень дерева содержит не более ***2n-1*** элементов (ключей) и не более ***2n*** ссылок на дочерние элементы
- B-дерево идеально сбалансировано, более того, длины всех ветвей от корня до листа совпадают;
- элементы в каждой вершине упорядочены по возрастанию
- если в вершине содержится ***k*** элементов, то в ней содержится ***k+1*** ссылок на дочерние вершины (кроме листьев, ссылок на дочерние вершины не содержащих);
- элементы в вершине и ссылки на дочерние вершины сопоставляются следующим образом: про первую ссылку говорят, что *она располагается* ***до*** *первого элемента*, про последнюю – что *она располагается* ***после*** *последнего элемента*, остальные ссылки располагаются каждая – ***между*** некоторой парой элементов в вершине;
- все элементы x_i в поддереве V, ссылка на которое расположена после некоторого элемента y, больше y; все элементы x_j в поддереве V, ссылка на которое расположена до некоторого элемента z меньше z.

Пример B-дерева степени 3:



Как правило, B-деревья имеют достаточно большие степени. Например, их задают исходя из того, что одна вершина должна занимать один блок на диске. На языке C тип данных для хранения одной вершины B-дерева степени 100 целых чисел можно определить следующим образом

```
#define NB 100
typedef struct BNode_
{
 struct BNode_ *par;
 int n;
 struct BNode_ *child[2*NB];
 int value[2*NB];
} BNode;
```

Здесь ***n*** – количество элементов, содержащихся в вершине, ***value[i]*** – значение ***i***-го элемента, ***child[i]*** – указатель на соответствующего потомка. Заметим, что мы отвели на один целый элемент больше, чем нам требуется для хранения данных. Этим мы воспользуемся позднее – при поиске элемента в B-дереве.
Если элемент дерева занимает много места, то имеет смысл в вершинах хранить не сами данные, а указатели на них. Так, например, допустим, мы хотим создать дерево для хранения строк, в понимании языка C, тогда тип вершины можно определить следующим образом

```
#define NB 100
typedef struct BNode_
{
 struct BNode_ *par;
 int n;
 struct BNode_ *child[2*NB];
 char  *str[2*NB-1];
} BNode;
```

Инициализировать такую структуру можно очень простой функцией:

```
void Init(BNode *node){memset(node,0,sizeof(BNode));}
```

После инициализации занесение строки в ***k***-ый элемент вершины можно осуществить следующей функцией

```
void Insert(BNode *node, int i_elem, char *str)
{
 if(node->str[i_elem]!=NULL)free(node->str[i_elem]);
 node->str[i_elem]=strdup(str);
}
```

Высота B-дерева

Получим оценку на высоту B-дерева через количество элементов в нем.

Корень дерева содержит не менее одного элемента. На втором уровне содержится не менее двух вершин, а в каждой вершине – не менее ***n-1*** элементов. На каждом следующем уровне количество вершин увеличивается не менее чем в ***n*** раз (т.к. каждая вершина имеет не менее ***n*** потомков). Т.о. на ***k-***ом уровне будет не менее $2n^{k-2}$ вершин для ***k>1***, и, соответственно, не менее $2(n-1)n^{k-2}$ элементов.

Т.о. получаем оценку на количество элементов ***N*** в дереве высоты ***h***

$$N \geq 1+\sum_{k=2}^{h} 2(n-1)n^{k-2}=1+2(n-1)\sum_{k=0}^{h-2} n^k=$$
$$=1+2(n-1)(n^{h-1}-1)/(n-1)= 2n^{h-1}-1$$

Т.о., учитывая то, что оценка сверху на число элементов в дереве получается аналогичным образом, мы получаем, что верна следующая

Теорема. *Для B-дерева степени* ***n****, содержащего* ***N*** *элементов, высоты* ***h*** *верна оценка для высоты*

$$h=\Theta(\log_n N).$$

Верна точная оценка

$$h \leq \log_n((N+1)/2)+1.$$

Поиск вершины в B-дереве

Поиск вершины, содержащей заданный элемент (или элемент с ключом, равным заданному), осуществляется аналогично поиску в двоичном дереве поиска. Единственное отличие – для каждой вершины процедура поиска данного элемента более сложная, чем для случая дерева поиска. На языке C поиск элемента, равного ***v***, в B-дереве с корнем ***root*** можно оформить в виде следующей функции

```
BNode *BSearch(BNode *root, int v)
{
 if(root==NULL)return NULL;
 for(i=0;i<root->n;i++)
  if(root->value[i]==v)return root;
  else if(root->value[i]>v)return BSearch(root->child[i],v);
 return BSearch(root->child[i],v);
}
```

Отступление на тему языка C. Быстрый поиск и сортировка в языке C

В конкретных реализациях степень B-дерева может быть весьма большой. Поэтому поиск элемента в одной вершине при больших степенях B-деревьев следует производить с помощью двоичного поиска. В языке C есть стандартная функция для поиска в упорядоченном массиве ***bsearch***. Например, в *Microsoft Visual C* эта функция имеет следующее описание

void *bsearch(const void *****key**, const void *****base*****, size_t*** *nmemb****, size_t*** *size****, int (__cdecl *****compare*** *) (const void **elem1, const void **elem2));*

В других компиляторах описание этой функции аналогично, быть может, с точностью до тонкостей. Например, в компиляторе *gcc* описание этой функции имеет следующий вид

void *bsearch(const void *****key**, const void *****base*****, size_t*** *nmemb****, size_t*** *size****, int (*****compare*** *)(const void * elem1, const void * elem2));*

Здесь *key* – указатель на искомый элемент, *base* – указатель на массив с данными, *nmemb* – количество элементов в массиве, *size* – размер в байтах одного элемента массива, *compare* – указатель на функцию, получающую указатель на два элемента массива и возвращающую результат сравнения элементов: +1 – если первый элемент больше второго, -1 – если второй элемент больше первого, 0 – если элементы равны.

Для нашего случая – поиска целого числа в массиве функция сравнения может быть определена следующим образом

```
int compare(const void *v0,const void *v1)
{ return *(int*)v0>*(int*)v1 ? 1 : *(int*)v0<*(int*)v1 ? -1 : 0 ; }
```

К сожалению, если данный элемент в массиве не найдет, то функция ***bsearch*** возвращает ***NULL***, при этом информация о том – между какими элементами находится искомый, теряется. Если нам все же хочется непременно воспользоваться функцией ***bsearch***, то мы можем применить некоторый трюк: мы можем воспользоваться информацией о том, что реально – первый параметр ***bsearch*** это – адрес искомого элемента, а второй – адрес некоторого элемента в массиве. Исходя из алгоритма двоичной сортировки, если искомый элемент в массиве отсутствует, то последний элемент ****v1*** при вызове функции *compare*, для которого оказалось, что

****(int*)v0<*(int*)v1***

будет ближайшим элементом массива, большим искомого (=****(int*)v0***) . Адрес этого элемента можно запомнить в соответствующей глобальной переменной. Чтобы указанное свойство было верным и для последнего элемента исходного массива, поместим вслед за последним элементом массива самое большое из всех чисел типа ***int***, и, соответственно, запретим его использование в обычной работе. Поиск же элемента будем производить в расширенном массиве. В этом случае функцию сравнения следует оформить следующим образом

```
int *v_gt_save=NULL;
int compare (const void *v0,const void *v1)
{
 if(*(int*)v0>*(int*)v1)return 1;
 if(*(int*)v0<*(int*)v1){v_gt_save=(int*)v1;return -1;}
 return 0;
}
```

Функция поиска вершины может тогда выглядеть следующим образом

```
BNode *BSearchQ(BNode *root, int v)
{
 if(root==NULL)return NULL;
 root->value[root->n]= INT_MAX;
 if(bsearch(&v, root->value, root->n+1, sizeof(int),compare))return root;
 return BSearchQ(root->child[v_gt_save-root->value], v);
}
```

Здесь константа ***INT_MAX*** обозначает максимальное число типа ***int***. Данная константа (определяемая через ***#define***) является, фактически, стандартной в разных версиях языка С. Так, например, в *Microsoft Visual C* и *GCC* эта константа определяются в стандартном файле ***include.h***.

Отметим, что данный подход, возможно, не является оптимальным как в плане скорости счета (присутствует лишняя операция присваивания ***v_gt_save=(int*)v1***), так и в плане выполнения правил хорошего тона (в алгоритме использовались глобальные переменные). Однако этот подход немного экономит время программиста (не надо программировать алгоритм двоичного поиска). В нашем же случае он, скорее, служит примером использования функции ***bsearch***.

Еще одним примером использования указателей на функцию является использование функции быстрой сортировки. Функция имеет следующее описание в *Microsoft Visual C*

void qsort(void * *base*, **size_t** *num*, **size_t** ***size***, **int (__cdecl *** *compare* **)(const void *** *elem1*, **const void *** *elem2* **));**

а в *GCC:*

void qsort(void * *base*, **size_t** *num*, **size_t** ***size***, **int** ***(*compar)*(const void*,const void*));**

здесь *base* – указатель на массив с данными, *num* – количество элементов в массиве, *size* – размер в байтах одного элемента массива, *compare* – указатель на функцию, получающую указатель на два элемента массива и возвращающую результат сравнения элементов: +1 – если первый элемент больше второго, -1 – если второй элемент больше первого, 0 – если элементы равны.

Например, в нашем случае отсортировать массив элементов одной вершины ***node*** B-дерева можно следующим образом

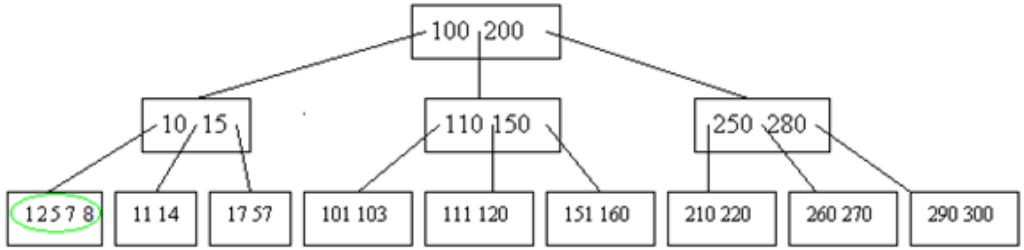
```
Bnode *node;
qsort(node->value, node->n, sizeof(int),compare);
```

Добавление вершины в B-дерево в два прохода

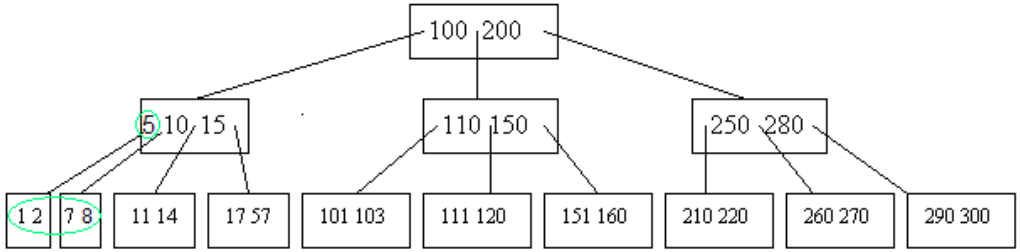
1. По аналогии с деревом поиска сначала ищется лист, в который можно вставить новый элемент (это – первый проход по дереву).
2. Заметим, что листом дерева называется элемент без потомков. Если найденный лист ***V*** не заполнен, то новый элемент вставляется в лист ***V*** B-дерева степени ***n*** и на этом процедура завершается.
3. Иначе в элементах данной вершины находится медиана ***x*** и вершина разбивается на две вершины по ***n-1*** элементу в каждой, причем элементы в первой вершине V_- должны быть меньше ***x***, а во второй V_+ – больше ***x***.
4. Элемент ***x*** вставляется в массив элементов в родительской вершине между элементами, между которыми находилась ссылка на вершину ***V.*** Ссылки на вершины V_- и V_+ должны расположиться непосредственно слева и справа от ***x***.
5. Теперь новый элемент можно вставить в одну из вершин V_- или V_+.
6. После этого, если в родительской вершине количество элементов становится меньше ***2n-1***, то на этом процедура завершается. Иначе процедура разбиения вершины рекурсивно применяется для родителя (т.е. переходим к шагам 3, 4, 6). В этом заключается второй проход по дереву.

В худшем случае процедура будет последний раз применена для корня дерева и дерево увеличит свою высоту на 1.

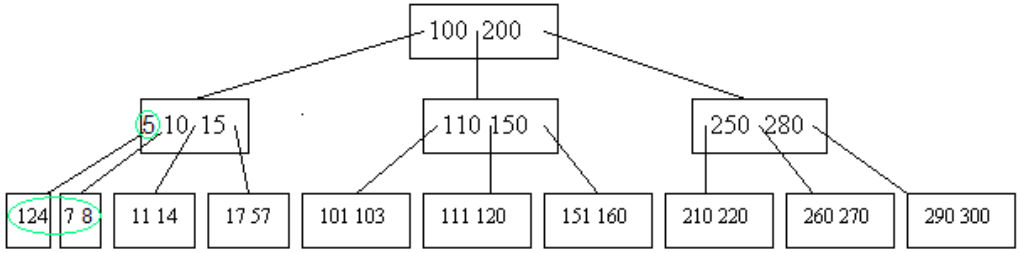
Приведем пример. В следующее B-дерево степени 3 требуется вставить элемент со значением 4.



Элемент 4 надо вставлять в вершину со значениями *{1, 2, 5, 7, 8}*. Вершина заполнена, поэтому мы разбиваем ее на две вершины *{1, 2}* и *{7, 8}* и медиану 5. Родительская вершина не заполнена, поэтому мы вставляем ключ 5 в родительскую вершину (с заменой ссылки на старый лист на ссылки на два новых листа):



Осталось вставить элемент 4 в лист *{1, 2}*:



Добавление вершины в B-дерево за один проход

Не составляет труда выполнить процедуру вставки нового элемента в B-дерево степени ***n*** за один проход. Для этого сначала проверяется – заполнен ли корень дерева. Если корень заполнен, то он разбивается на две новые вершины, состоящие из первых и последних ***n-1*** элемента, и создается новый корень дерева, в который добавляется медиана старого корня со ссылками на две новые вершины. Далее при поиске требуемого листа следует для каждой пройденной вершины (включая лист) проверять – заполнена ли она. Если вершина заполнена, то (по аналогии с корнем дерева) ее следует разбить на две вершины, состоящие из первых и последних ***n-1*** элемента, и вставить медиану элементов из данной вершины в вышестоящую вершину. При этом ссылка на данную вершину (от родителя) заменяется на две ссылки на новые вершины (слева и справа от вставленного значения). Данная вставка не приведет к переполнению родительской вершины, т.к. на предыдущем шаге было обеспечено, что родительская вершина не заполнена.

Удаление вершины из B-дерева за один проход

Для удаления элемента, равного заданному, требуется, сначала, его найти. При осуществлении поиска мы параллельно будем обеспечивать условие, гарантирующее, чтобы в вершине, из которой будет удаляться элемент, было бы не менее ***n*** элементов (по определению B-дерева достаточно, чтобы в вершине присутствовало не менее n-1 элемента).

Итак, в процессе поиска вершины, содержащей удаляемый элемент ***v***, мы вводим понятие текущей вершины ***x***. Текущая вершина перемещается от корня дерева по соответствующей ветви к вершине, содержащей удаляемый элемент.
Для каждого очередного значения текущей вершины возможен один из следующих вариантов

1) *Вершина является листом.*
Если, при этом, вершина - корень (все дерево состоит из одной вершины), то мы просто удаляем найденный элемент из этой вершины (если элемент найден).
Если дерево состоит более, чем из одной вершины, то в результате выполнения следующих пунктов, данная вершина содержит не менее ***n*** элементов и найденный элемент можно исключить из данной вершины (если он нашелся, иначе - элемент отсутствует в дереве).

2) *Вершина* ***x*** *- внутренняя. Элемент* ***v*** *в вершине* ***x*** *не найден.*
Ищем потомка ***x->child[i]*** вершины ***x***, с которого начинается поддерево, содержащее элемент ***v*** (если он вообще есть в дереве). По условию мы должны гарантировать, чтобы в вершине ***x->child[i]*** содержалось бы не менее ***n*** элементов. Если это выполняется, то переходим к рассмотрению этой вершины:

x=x->child[i].

Иначе мы либо `перетаскиваем' один элемент из брата вершины ***x->child[i]*** в

вершину ***x->child[i]***, либо, если это невозможно, объединяем данную вершину с братом. Более подробно, есть два варианта:

a) *У вершины* ***x->child[i]*** *есть брат, содержащий не менее* ***n*** *элементов.*

Пусть, для определенности, это - правый брат, т.е. ***x->child[i+1]->n*** ^{3} ***n***.

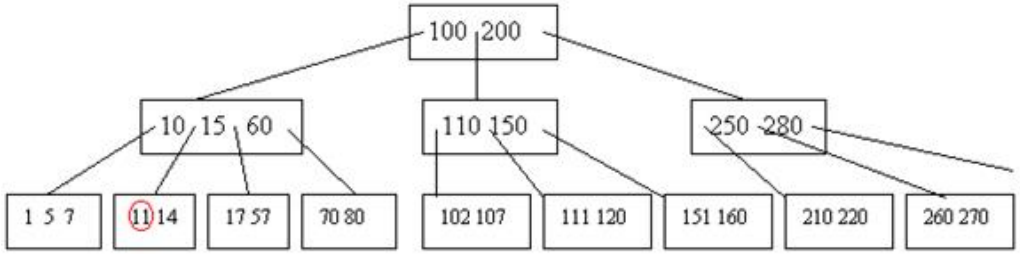
Тогда мы переносим элемент ***x->value[i]*** в конец массива элементов ***x->child[i]***

(соответственно, увеличив на 1 значение ***x->child[i]->n***) и элемент ***x->child[i+1]->value[0]*** переносим на место ***x->value[i]***

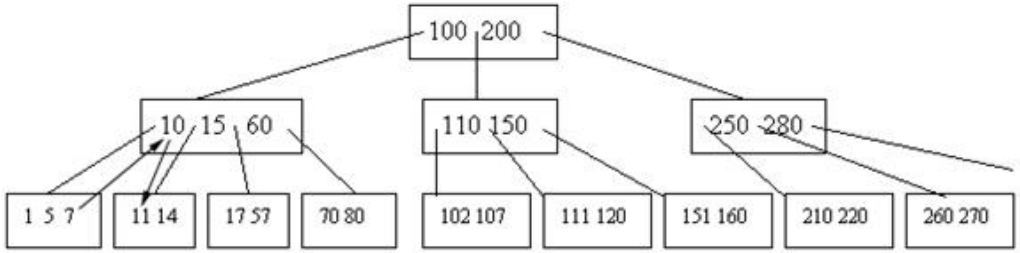
(соответственно, уменьшив на 1 значение ***c->child[i+1]->n***):

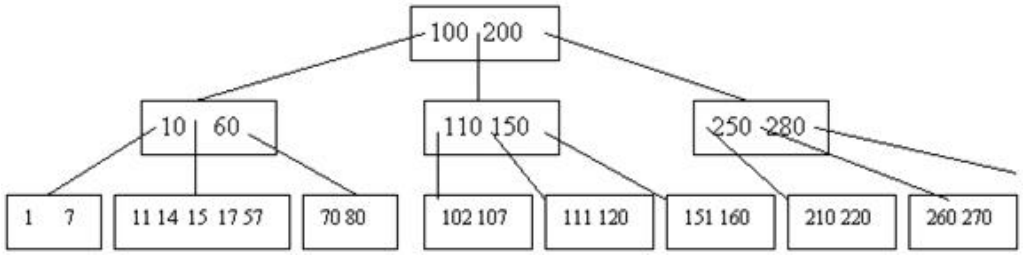
```
x->child[i]->value[++x->child[i]->n]=x->value[i];
x->value[i]=x->child[i+1]->value[0];
for(i=1;i<x->child[i+1]->n;i++)
  x->child[i+1]->valuie[i-1]=x->child[i+1]->valuie[i];
x->child[i+1]->n--;
```

Например, требуется в следующем дереве удалить вершину ***11*** в B-дереве степени ***3***:



У данной вершины есть сосед (левый), содержащий 3 элемента. Тогда мы максимальный элемент из этой вершины – ***7*** – помещаем на место ***10***, а ***10*** помещаем на место ***11***:





Теперь мы можем перейти к рассмотрению следующей вершины:
x=x->child[i];

3. Вершина x - внутренняя, в вершине найден элемент ***x->value[i]==v***.
Сначала удаление производится аналогично дереву поиска: в одном из поддеревьев, соседних данной вершине, например, для определенности, в правом соседнем поддереве данной вершины (т.е. в поддереве, начинающемся с вершины ***x->child[i+1]***) находим элемент v_0, ближайший к ***v***. В нашем случае это – минимальный элемент поддерева, начинающегося с вершины ***x->child[i+1]***. Заметим, что минимальный элемент в поддереве B-дерева является первым элементом некоторого листа.
Далее, помещаем элемент v_0 на место элемента ***v*** и запускаем процедуру удаления старого (т.е. удаленного) элемента v_0.
Здесь, чтобы не запутаться в `старом' $v_0$ и `новом' v_0 лучше сначала запомнить адрес элемента, на который мы должны скопировать v_0 и само значение v_0, далее можно осуществить процедуру удаления элемента v_0 и только потом скопировать запомненное v_0 по запомненному адресу.

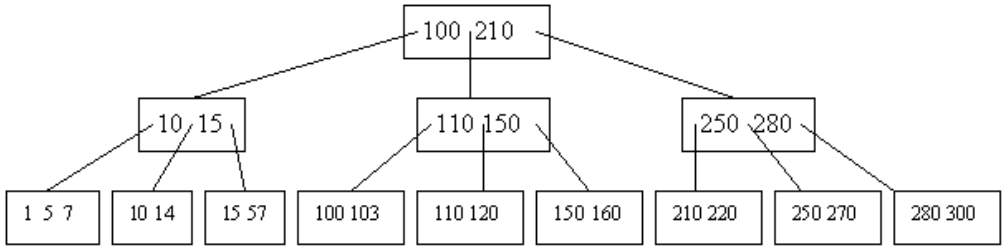
B⁺-деревья

Интересной и очень важной модификацией B-деревьев являются B⁺-деревья. Приведем далее формальное определение B⁺-дерева, выделив заглавными буквами формальные отличие между B-деревьями и B⁺-деревья.

B⁺-дерево степени ***n*** определяется следующим образом

- каждая вершина дерева, кроме корня, содержит от ***n-1*** до ***2n-1*** элемента (ключей) и от ***n*** до ***2n*** ссылок на дочерние элементы; корень дерева содержит не более ***2n-1*** элементов (ключей) и не более ***2n*** ссылок на дочерние элементы
- B-дерево идеально сбалансировано, более того, длины всех ветвей совпадают;
- элементы в каждой вершине упорядочены по возрастанию
- если в вершине содержится ***k*** элементов, то в ней содержится ***k+1*** ссылок на дочерние вершины (кроме листьев, ссылок на дочерние вершины не содержащих);
- элементы в вершине и ссылки на дочерние вершины сопоставляются следующим образом: про первую ссылку говорят, что *она располагается* ***до*** *первого элемента*, про последнюю – что *она располагается* ***после*** *последнего элемента*, остальные ссылки располагаются каждая – ***между*** некоторой парой элементов в вершине;
- все элементы x_i в поддереве ***V***, ссылка на которое расположена после некоторого элемента ***y***, ***БОЛЬШЕ ИЛИ РАВНЫ y***; все элементы x_j в поддереве ***V***, ссылка на которое расположена до некоторого элемента ***z*** меньше ***z***.
- ***ССЫЛКИ НА ДАННЫЕ ЛЕЖАТ ТОЛЬКО НА НИЖНЕМ УРОВНЕ ДЕРЕВА (В ЛИСТЬЯХ); ВО ВСЕХ ОСТАЛЬНЫХ ВЕРШИНАХ ЛЕЖАТ ТОЛЬКО КОПИИ КЛЮЧЕЙ ЭЛЕМЕНТОВ С НИЖНЕГО УРОВНЯ, ИСПОЛЬЗУЮЩИЕСЯ ДЛЯ ИНДЕКСИРОВАНИЯ***.

Пример B+-дерева степени 3:



Основным преимуществом данной структуры данных является то, что все реальные ключи данных, содержащихся в дереве, лежат на одном уровне (на нижнем). Поэтому не сложно завязать эти элементы в список, что существенно упрощает процедуру последовательно перебора данных их списка. Например, именно B+-деревья используются во всех прогрессивных файловых системах (NTFS, RaiserFS, XFS и т.д.).

Поиск вершины в B+-дереве

Поиск вершины, содержащей ссылку на заданный элемент, осуществляется аналогично поиску в B-дереве, с той лишь разницей, что искать надо в любом случае вплоть до листа. На языке C поиск элемента, равного ***v***, в B+-дереве с корнем ***root*** можно оформить в виде следующей функции

```
BNode *BSearch(BNode *root, int v)
{
 if(root==NULL)return NULL;
 if(root->child[0]==NULL)//если мы работает с листом
 {
  for(i=0;i<root->n;i++) if(root->value[i]==v)return root;
  return NULL;
 }
 for(i=0;i<root->n;i++) if(root->value[i]>v)return BSearch(root->child[i],v);
 return BSearch(root->child[root->n],v);
}
```

Добавление вершины в B+-дерево в два прохода

Вставка нового элемента в B+-дерево делается аналогично вставке элемента в B-дерево, но случай заполнения вершины до *2n-1* элементов обрабатывается немного по-другому:

- Вершина ***P*** (лист), в которой образуется *2n-1* элемент, разбивается на две вершины P_1 и P_2 с, соответственно, *n* и *n-1* элементами;
- Ссылка на ***P*** заменяется на ссылку на P_1;
- Пара (*Ключ минимального элемента из* P_2*, ссылка на* P_2) вставляется в родителя данной вершины после ссылки на P_1;
- Если после этого родительская вершина оказывается заполненной, то с ней рекурсивно осуществляется аналогичная процедура.

Удаление вершины из B+-дерева

Удаление элемента из B+-дерева делается аналогично удалению элемента из B-дерева, но оно требует аккуратной корректировки индексов в родительских вершинах.

Лекция 13

STL

Александр Степанов. Менг Ли. *Руководство по стандартной библиотеке шаблонов (STL).* Все предельно по делу. Уровень формализации очень высок (=читать тяжело, но для мехмата в самый раз).

Мэтью Уилсон. *Расширение библиотеки STL для C++. Наборы и итераторы.* Много идеологии и терминов (=читать тяжело, но для фанатов доставит удовольствие).

В упрощенном понимании *STL* (*Standard Template Library*) является стандартной библиотекой шаблонов, обеспечивающих основные потребности программистов в часто-используемых объектах/конструкциях. Возможна поддержка данной библиотеки на уровне компилятора. Вторая вышеприведенная книга содержит более 600 страниц и невозможно изложить (даже тезисно) все рассмотренные в ней вопросы. Рассмотрим лишь бегло несколько основных тем, включающихся в STL.

Последующие разделы идут в смысловом порядке. Т.е. сначала идет описание основных (в смысле использования) понятий, а потом понятий с помощью которых идет работа с уже описанными понятиями. Но STL является весьма целостной вещью, поэтому невозможно описывать базовые понятия без использования вторичных понятий. Поэтому часто при формальном описании STL нижеприведенные разделы следуют в обратном порядке. Т.е. сначала выписываются способы работы с объектами, а только потом описывают сами объекты.

Реализации структур данных. Контейнеры

Везде далее будем обозначать тип объектов, хранящихся в контейнере, *T*. Тогда контейнер *Container* объектов типа *T* можно определить как

```
Container<T> x;
```

Далее мы будем часто пользоваться следующими функциями (определяемыми пользователем)

```
void print(int v){cout<<v<<" ";}
```

```
void printp(pair<int,double> p){cout<<"(key="<<p.first<<",v="<<p.second<<")
";}
void printss(pair<string,int>p){cout<<"("<<p.first<<","<<p.second<<") ";}
int fun_if(int x){return x%2==0;}
int comp(int x,int y){if(x>y)return 1; return 0;}
int Eq5(int v){return v==5;}
```

Везде последующей работы потребуются include-файлы:

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <list>
#include <vector>
#include <deque>
#include <queue>
#include <stack>
#include <map>
#include <set>
#include <strstream>
#include <sstream>
#include <fstream>
#include <string>
```

Последовательные контейнеры

В стандарте STL присутствует три *последовательных контейнера*: *vector, deque, list*. Также есть три адаптации последовательных контейнеров: *stack, queue, priority_queue*. Здесь под *адаптацией контейнера* имеется в виду некоторый производный контейнер, полученный из исходного сокращением функциональности и незначительной модификацией.

Последовательные контейнеры служат для работы с множеством элементов заданного типа с возможностью быстрого последовательного доступа к элементам данного контейнера.

vector. Служит для работы со структурой данных *вектор*.

К элементам вектора можно осуществлять **доступ** либо с помощью методы *T &at(int index),* либо с помощью оператора *T &operator[](int index)*. В первом случае происходит обязательная проверка выхода за границы вектора (в случае выхода генерируется исключение). Во втором случае проверка происходит лишь в особых ситуациях. Под особыми ситуациями, например, в *MSVisualStudio* понимается отладочный режим компиляции программы, а в *gcc* для проверки требуется следующая инструкция (инструкция должна быть вставлено ДО вставки include-файла):

```
#define  _GLIBCXX_DEBUG
```

Очистить вектор и задать **указанное количество одинаковых элементов** вектора можно методом *assign(int index, const T&value)*.

Изменить количество памяти, отведенной под вектор можно методом *reserve(int size)*. Здесь следует отметить, что под вектор отводится памяти больше, чем требуется для его хранения, на случай последующего увеличения размера вектора.

Изменить размер вектора можно методом *resize(int size)*.

Вставить элемент на указанную позицию можно методом *insert(vector::const_iterator place, const T&value)*.

deque. Служит для работы со структурой данных *дек*. При этом деком можно пользоваться как вектором.

С STL-деком работают операции для вектора, например, *T &at(int index), T &operator[](int index)*, *assign(int index, const T&value), resize(int size)*, *insert(deque::const_iterator place, const T&value)*.

Есть родные для дека операции:

```
push_back(T&value)
push_front(T&value)
pop_back(T&value)
pop_front(T&value)
```

list. Служит для работы со структурой данных *L2-список*. При этом деком можно пользоваться как деком.

С STL-списком работают операции для дека *push_back(T&value), push_front(T&value), pop_back(T&value), pop_front(T&value).*

Передвижение по списку осуществляется с помощью итераторов.

Вставка элемента до текущего (до итератора):

```
insert(list::const_iterator place, const T&value)
```

Уничтожение элемента/элементов:

```
erase(list::const_iterator place)
erase(list::const_iterator from, list::const_iterator to)
```

queue. Служит для работы со структурой данных *очередь*.

С очередью можно работать только как с очередью. Следующий кусок кода последовательно загружает в очередь числа 2,1,3,4, распечатывает длину очереди и далее по одному достает (с удалением) числа с головы очереди и печатает их:

```
queue<int> v;  v.push(2);v.push(1);v.push(3);v.push(4);   cout<<"
size="<<v.size()<<endl;
```

```
while(v.size()>0){cout<<v.front()<<" ";v.pop();} cout<<endl;
```

priority_queue. Служит для работы со структурой данных *приоритетная очередь*. Отличается от очереди тем, что в голову очереди встает элемент с максимальным значением. В наших лекциях приоритетная очередь реализовывалась с помощью *кучи* (=*пирамиды*) в сортировке *HeapSort*.

С приоритетной очередью можно работать только как с очередью. Следующий кусок кода последовательно загружает в очередь числа 2,1,3,4, распечатывает длину очереди и далее по одному достает (с удалением) числа, начиная с максимального и далее по убыванию, и печатает их:

```
queue<int> v;  v.push(2);v.push(1);v.push(3);v.push(4);   cout<<"
size="<<v.size()<<endl;
while(v.size()>0){cout<<v.top()<<" ";v.pop();} cout<<endl;
```

Ассоциативные контейнеры

В стандарте STL присутствует четыре *ассоциативных контейнера*: *set, multiset, map, mutimap*.

Ассоциативные контейнеры служат для работы с множеством элементов, к которым требуется произвольный доступ по ключу. Т.е. это – аналог массива, индексированного ключом произвольного типа.

set. Служит для работы с множествами (элемент множества = ключ).

multiset. Служит для работы с множествами с дубликатами (т.е. допускаются равные элементы множества).

Следующий кусок кода будет работать для *set* и для *multiset*, но будет выдавать разные результаты:

```
multiset<int> v;  v.insert(1);v.insert(2);v.insert(3); v.insert(4); v.insert(4);
cout<<" size="<<v.size()<<endl;
for_each(v.begin(),v.end(),print);
```

map. Служит для работы с множествами элементов, идентифицирующихся ключами. Фактически в множестве хранятся пары *<Ключ,Элемент >* . Одному ключу соответствует не более одного элемента.

Пример подсчета количества различных слов в файле:

```
map<string,int> v; ifstream f("t.txt"); istream_iterator<string> it;
        for(it=f;it!=istream_iterator<string>();it++)
        {
         map<string,int>::iterator itt=v.find(*it);
         if(itt!=v.end())
          itt->second++;
         else
          v.insert(pair<string,int>(*it,1));
        }
        for_each(v.begin(),v.end(),printss);
```

multimap. Служит для работы с множествами элементов, идентифицирующихся ключами. Фактически в множестве хранятся пары *<Ключ,Элемент>* . Одному ключу может соответствовать более одного элемента.

Для работы с подобными типами в STL существует шаблон структуры типа *pair*, с помощью которой (структуры в виде шаблона) можно задавать пары элементов различных типов. Например, пару элементов типов *int* и *double* можно задать в одном из следующих видов:

```
pair<int,double> p(1,10.);
pair<int,double>(1,10.)
```

Следующий кусок кода (кроме последней строки) будет работать для *map* и для *multimap*, но будет выдавать разные результаты:

```
        map<int,double> v; map<int,double>::iterator it; double val;

v.insert(pair<int,int>(1,1));v.insert(pair<int,int>(2,2));v.insert(pair<int,int>(3,40));v.
insert(pair<int,int>(4,30));
        for_each(v.begin(),v.end(),printp); cout<<endl;
        it=v.find(3);  cout<<"[3]="<<it->second<<endl;
        val=v[3];  cout<<"[3]="<<val<<endl;//работает только для map
```

Итераторы

Далее под *итераторами* мы будем иметь в виду *STL-итераторы*.

Максимально упрощенный подход

В максимально упрощенном виде итераторы являются обобщением/заменой понятия *указателя,* используемому вне STL. Поэтому работа с итератором часто напоминает работу с обычным указателем. В реальности, итератор является классом, в которым (скорее всего) содержится только указатель на объект, хранящийся в контейнере, с переопределенными операциями, необходимыми для работы с данным объектом.

Итераторы не используются для работы с очередью и приоритетной очередью. Существует несколько типов итераторов (!!!), но пока не будем заострять на этом внимания. Для каждого контейнера существует свой тип итератора, который можно использовать для работы с данным контейнером:

```
vector::iterator
list::iterator
deque::iterator
```

и т.д.

но все они имеют одинаковый (на самом деле, не всегда) набор функций, что унифицирует работу с итераторами.

Следует помнить, что итераторы (как указатели на объекты в контейнере) можно использовать после получения их значения только пока контейнер не подвергается изменению. После изменения контейнера полученный ранее итератор становится неопределенным.

В каждом типе контейнера (в классе контейнера) содержатся следующие функции, возвращающие итераторы текущего контейнера (читай: псевдоуказатели на объекты, хранящиеся в текущем контейнере):

iterator begin() возвращает итератор, указывающий на **первый** элемент в контейнере

iterator end() возвращает итератор, указывающий на элемент, следующий **после последнего** элемента в контейнере

iterator rbegin() возвращает итератор, указывающий на **первый** элемент в контейнере для обратного перебора элементов

iterator rend() возвращает итератор, указывающий на элемент, следующий **после последнего** элемента в контейнере, используемого при обратном переборе элементов

К итераторам применима префиксная операция ++, возвращающая итератор, указывающий на следующий элемент в контейнере (постфиксная операция ++ тоже может существовать, но это нечто более сложное).

К итераторам применима префиксная операция *, возвращающая значение элемента данных, содержащееся в элементе контейнера, на который указывает итератор.

Приведем пример перебора элементов контейнер *set.*

```
set<int> v;  set<int>::iterator it;
        v.insert(2);v.insert(1);v.insert(3);v.insert(4);v.insert(4);
        for(it=v.begin();it!=v.end();++it)cout<<*it<<" ";
```

Итераторы ввода

Все, что было сказано в предыдущем разделе относится к итераторам ввода. Под вводом имеется в виду ввод данных из контейнера в окружающую программу (например, **ввод** данных из контейнера требуется при **выводе** содержимого контейнера на экран). Полный список операций для итераторов различных типов можно найти, например, в вышеприведенной книге ***Александра Степанова***. Итераторы ввода поддерживают только перемещение вперед по элементам контейнера и дают возможность получить значение элемента в контейнере без права его изменения. Итераторы ввода можно присваивать друг другу и их можно сравнивать на равенство/неравенство.

```
istream_iterator<int> itIn,itEnd=istream_iterator<int>(); int m[10],n;
        ostream_iterator<int> itOut(cout," ");
        itIn=cin;
        for(n=0;itIn!=itEnd;++itIn,n++)m[n]=*itIn;
        for_each(m,m+n,print);cout<<endl;
```

Итераторы вывода

Итераторы вывода не дают возможности для сравнения итераторов, но для них можно использовать оператор * слева от знака присваивания.

```
vector<int> v; v.resize(10); size_t n;
        istream_iterator<int> it; ostream_iterator<int> itOut(cout," ");
        for(it=cin,n=0;it!=istream_iterator<int>();++it,n++)v[n]=*it;
        for(vector<int>::iterator it=v.begin();it!=v.end();++it)*itOut=*it;
cout<<endl;
```

Последовательные итераторы

Последовательные итераторы объединяют возможности итераторов ввода и вывода.

Двунаправленные итераторы

Двунаправленные итераторы поддерживают все свойства последовательных итераторов плюс операцию -- . Например, итераторы ассоциативных контейнеров является двунаправленными:

```
        multimap<int,double> v; multimap<int,double>::iterator it;

v.insert(pair<int,int>(1,1));v.insert(pair<int,int>(2,2));v.insert(pair<int,int>(3,40));v.
insert(pair<int,int>(4,30));v.insert(pair<int,int>(4,40));
        it=v.find(4);  cout<<"[4]="<<it->second<<endl;
        --it; cout<<"[3]="<<it->second<<endl;
```

Итераторы произвольного доступа

К свойствам двунаправленного итератора добавляются возможности прибавления/вычитания целого числа и оператор []. Например, итератор вектора (vector) является итератором произвольного доступа.

```
        vector<int> v;  vector<int>::iterator it;  v.resize(10);
        for(it=v.begin();it!=v.end();++it)*it=(it-v.begin());
        for_each(v.begin(),v.end(),print);
```

Итераторы вставки

Все вышеперечисленные итераторы позволяют перебирать и модифицировать элементы контейнеров, но для списков требуется дополнительная операция вставки, поэтому, соответственно, пришлось создавать итератор вставки. Достаточно рассмотреть итераторы вставки в начало объекта, в хвост объекта и на определенную позицию объекта. Начнем с того, что полегче ☺

inserter - функция, возвращающая итератор вставки на определенную позицию

back_inserter - функция, возвращающая итератор вставки в хвост контейнера

front_inserter - функция, возвращающая итератор вставки в голову контейнера

```
int x[]={1,2,3,4,5}; list<int> l; list<int>::iterator it;
        copy(x,x+sizeof(x)/sizeof(x[0]),front_inserter<list<int>>(l));//
вставляем в начало
        copy(x,x+sizeof(x)/sizeof(x[0]), back_inserter<list<int>>(l));//
вставляем в конец
        it=l.begin(); ++it;
        copy(x,x+sizeof(x)/sizeof(x[0]), inserter<list<int>>(l,it));//вставляем со
второй позиции (после первого элемента)
        copy(l.begin(),l.end(), ostream_iterator<int>(cout," ")); cout<<endl;
```

Не плохо понимать, что происходит в предыдущем примере. В нем используются не итераторы, а функции, возвращающие итераторы. В следующем примере итераторы используются напрямую (это сложнее!!!)

inserter - функция, возвращающая итератор вставки на определенную позицию

back_inserter - функция, возвращающая итератор вставки в хвост контейнера

front_inserter - функция, возвращающая итератор вставки в голову контейнера

```
int x[]={1,2,3,4,5}; list<int> l; list<int>::iterator it;
        front_insert_iterator<list<int>> itF(l);
        copy(x,x+sizeof(x)/sizeof(x[0]), itF);
        back_insert_iterator<list<int>> itB(l);
        copy(x,x+sizeof(x)/sizeof(x[0]), itB);
        copy(l.begin(),l.end(), ostream_iterator<int>(cout," ")); cout<<endl;
        insert_iterator<list<int>> itI(l,++l.begin());
        copy(x,x+sizeof(x)/sizeof(x[0]), itI);
        copy(l.begin(),l.end(), ostream_iterator<int>(cout," ")); cout<<endl;
```

Надо помнить, что итераторы вставки используются только для вставки!!! Эта фраза подразумевает, что, например, в последнем примере операция *++itI* не приведет к изменению позиции итератора (хотя, синтаксически не приведет к ошибке!).

Не удается использовать итераторы вставки для вектора и ассоциативных контейнеров (хотя, функция *insert* для вектора существует), но их можно использовать для дека.

Функциональные объекты

Функциональными объектами называются объекты (шаблоны объектов), к которым можно применять *operator()* с одним или двумя параметрами. Функциональные объекты служат заменой унарных или бинарных операций. Функциональные объекты используются в алгоритмах, в которых требуется указать операцию, которую необходимо применить для набора элементов в исходных контейнерах.

Существует набор стандартных (реализованных в STL) функциональных объектов. Например:

```
template <class T>struct plus : binary_function<T, T, T> {    T operator()(const T& x,
const T& y) const { return x + y; } };
template <class T>struct minus : binary_function<T, T, T> {    T operator()(const T&
x, const T& y) const { return x - y; } };
template <class T> struct negate : unary_function<T, T> {    T operator()(const T& x)
const { return -x; } };
```

Ничто не мешает пользователю создавать свои функциональные объекты:

```
template <class T> struct sqr : unary_function<T, T> {    T operator()(const T& x)
const { return x*x; } };
```

Везде, где применяются функциональные объекты можно применять обычные указатели на функции (или указатели на шаблоны функций).

Алгоритмы

Алгоритмами называются шаблоны для работы (перебора/поиска/изменения и т.д.) с контейнерами.

Например, шаблон for_each(), используемый выше.

Алгоритм *find*() (*find_if*()):

```
vector<int> v;  vector<int>::iterator it;  v.resize(10);  for(it=v.begin();it!=v.end();+
+it)*it=(it-v.begin());
        it=find(v.begin(),v.end(),5);
        if(it!=v.end())cout<<*it<<endl;


         int v[10]; for(int i=0;i<10;i++)v[i]=i;
        it=find_if(v,v+10,Eq5);
        if(it!=v.end())cout<<*it<<endl;
```

Алгоритм поиска подпоследовательности в последовательности *search*():

```
vector<int> s; s.push_back(1);s.push_back(2);
 int x[]={1,1,2,3,4};
 int *it=search(x,x+sizeof(x), x.begin(),x.end());
 if(it==y+sizeof(y))cout<<"not found"<<endl;
 else cout<<"position="<<it-y<<endl;
```

Алгоритм *partition*() (*stable_partition*()) (используется поток вывода и итератор вывода):

```
vector<int> x; vector<int>::iterator it;  ostream_iterator<int> itOut(cout," ");
         for(int i=0;i<20;i++)  x.push_back(rand()%10-5);
         it=stable_partition<vector<int>::iterator>(x.begin(),x.end(),lt0);
```

```
stable_partition<vector<int>::iterator>(it,x.end(),eq0);
copy(x.begin(),x.end(), itOut); cout<<endl;
```

Алгоритм *transform*() (применяются функциональные объекты):

```
vector<int> x,y,z;
for(int i=0;i<20;i++)  { x.push_back(rand()%10-5); y.push_back(rand()
%10-5);}
transform(x.begin(),x.end(), y.begin(), z.begin(), multiplies<int>());
```

Здесь следует обратить внимание на синтаксис использования функционального объекта (сравните с синтаксисом задания указателя на обычные функцию в предыдущем примере).

Алгоритм *sort*() (*stable _sort* ()):

```
vector<int> v;  vector<int>::iterator it;  v.resize(10);  for(it=v.begin();it!
=v.end();++it)*it=(it-v.begin());
sort<vector<int>::iterator>(v.begin(),v.end(),comp);
for_each(v.begin(),v.end(),print);
```

Про эффективность алгоритма sort() можно судить по времени работы алгоритма на моем ноутбуке при работе под MSVisualStudio для массива из 10^8 случайных элементов (значение элемента задается как *m0[i]=rand()|(rand()<<14*)

) в сравнении с другими алгоритмами:

Алгоритм sort():	26сек
Стандартная функция qsort():	28сек
Алгоритм деления пополам с рекурсией из лекции:	34сек
Функция быстрой сортировки QSort2() из лекции:	22сек

Для `менее' случайных элементов (значение элемента задается как *m0[i]=rand()*) имеем:

Алгоритм sort():	14сек
Стандартная функция qsort():	20сек
Алгоритм деления пополам с рекурсией из лекции:	17сек
Функция быстрой сортировки QSort2() из лекции:	13сек

Алгоритм copy():

```
int m[5]={1,2,3,4,5}; vector<int> v(m+0,m+5),v2(m+0,m+5);
copy(v.rbegin(),v.rend(), v2.begin());
for_each(v2.begin(),v2.end(),print);cout<<endl;
```

Алгоритм copy_if():

```
int m[5]={1,2,3,4,5}; vector<int> v(m+0,m+5),v2(m+0,m+5);
copy_if(v.rbegin(),v.rend(), v2.begin(),funif);
for_each(v2.begin(),v2.end(),print);cout<<endl;
```

Потоки

Работа с итераторами хорошо сочетается с потоками ввода/вывода. Например, в качестве потоков можно рассматривать поток вывода на стандартный поток вывода (cout) (отметим, что здесь дважды употреблено слово *поток* в разных смыслах), поток ввода со стандартного потока ввода (cin), строковый поток ввода/вывода. Например:

```
stringstream str(""); int m[10]={1,2,3,4,5,6,7,8,9,10}; ostream_iterator<int> it(str,"
");
copy_if(m,m+sizeof(m)/sizeof(m[0]), it,fun_if);
cout<<str.str()<<endl;

stringstream str("1 2 3 4 5"); vector<int> m; m.resize(10); istream_iterator<int>
it=str; vector<int>::iterator rez;
 rez=copy_if(it,istream_iterator<int>(), m.begin(),funif);
 m.resize(rez-m.begin());
 for_each(m.begin(),m.end(),print);
```

Также удобно работать с файловыми потоками. В следующем примере из файла читается `кривой' массив и выводится на экран двумя способами:

```
ifstream fl("t.txt"); vector<vector<int>> v; string Str;
 while(getline(fl,Str))//the same: getline(fl,Str)!=NULL
 {
  stringstream str(Str); istream_iterator<int> it;
  v.push_back(vector<int>());
  for(it=str;it!=istream_iterator<int>();it++)(v.end()-1)->push_back(*it);
  cout<<"l="<<(v.end()-1)->size()<<endl;
 }
 for(vector<vector<int>>::iterator ity=v.begin(); ity!=v.end(); ity++)//output by
        means of iterators
 { for(vector<int>::iterator itx=ity->begin(); itx!=ity->end(); itx++)cout<<*itx<<"
        "; cout<<endl; }
 for(int i=0;i<v.size();i++)//output my means of indexes
 { for(int j=0;j<v[i].size();j++)cout<<v[i][j]<<" ";cout<<endl; }
```

В данном примере нетривиальна конструкция *while(getline(fl,Str))*. Тонкость здесь в том, что *getline()* в качестве результата возвращает ссылку на свой первый параметр. Здесь ссылку можно использовать в качестве логического выражения т.к. для типа *ifstream* существует оператор преобразования типа к типу *void** (в C++11 используется преобразование к типу *bool*), а этот тип уже можно использовать как логический.

Безусловно, для типа *ifstream* можно использовать операторы << и >>, но в нашем случае (ввода кривого массива) они бы нам не помогли.

Лекция 14

Хеширование

Фактически, алгоритмы работы со всеми структурами данных, связанными с деревьями, основаны на операции сравнения. Можно использовать другой подход. Попробуем на основе значения элемента ***x***, заносимого в структуру данных, вычислять некоторую функцию ***h(x)***, которая будет так или иначе отражать положение элемента ***x*** в структуре данных (например, индекс элемента в массиве). Такая функция называется ***хэш-функцией***. Сама структура данных, поиск элементов в которой использует хэш-функцию, называется ***хэшируемой***. Наиболее прямолинейным способом хранения хэшируемых данных является массив массивов элементов. Т.е. для каждого значения хэш-функции отводится свой массив, в котором хранятся элементы, рассматриваемого типа. Например, для работы с множеством целых чисел, при использовании хэш-функции ***h(x)*** со значениями $0 \le h(x) < M$, можно использовать массивы

int h_array[M][N], l_array[M];

Здесь константа ***N*** задает ограничение на количество чисел, содержащихся в структуре данных, для каждого значения хэш-функции. Данные, соответствующие значению хэш-функции ***h(x)=i***, хранятся в массиве ***h_array[i]***, количество элементов в этом массиве хранится в переменной ***l_array[i]***.

Преимущества и недостатки такого подхода очевидны: основным преимуществом является простота и удобство работы при равномерном распределении значений хэш-функции, а недостатком – неэффективность при неравномерной работе хэш-функции. Отметим также, что время работы для добавления элемента меньше времени работы для удаления элемента, т.к. в последнем случае приходится сдвигать часть массива.

Закрытая адресация. Метод многих списков (он же – метод цепочек)

Модификацией вышеописанного алгоритма является алгоритм, хранящий данные ***методом многих списков.*** В нем каждому значению хэш-функции сопоставляется свой список значений, содержащий хранимые данные. В этом случае на языке С при использовании стандартных списков (L1 или L2) для организации данных следует завести массив указателей на вершину списка:

CList *h_list[M];

здесь ***M*** – (как и выше) константа, ограничивающая максимальное значение хэш-функции; ***CList*** – тип переменной для хранения одной вершины списка.

Инициализация структуры данных тривиальна:

void Init(Clist *h_list[]){memset(h_list,0,M*sizeof(Clist*));}

Отметим, что вместо списков в данной ситуации можно брать другие структуры данных: например, динамические массивы (чтобы сэкономить память) или деревья (для более быстрого поиска элементов в них).

Можно оценить среднее время поиска элемента в такой структуре данных в ситуации, когда у нас используется `идеальная' хэш-функция, т.е. время ее работы равно ***O(1)*** и она с равной вероятностью выдает все свои значения для потока входных данных. В этом случае среднее время поиска элемента пропорционально среднему количеству элементов в произвольном списке из массива ***h_list***. Заметим, что последний факт верен как для удачного, так и для неудачного поиска.

Итак, пусть у нас хранится всего ***N*** элементов в ***M*** списках. Вероятность попадания элемента в один определенный список равна $p=1/M$. Тогда вероятность попадания ***k*** элементов в один конкретный список равна $p_k=C_N^k p^k (1-p)^{N-k}$. Средняя длина списка равна

$$l_N = \sum_{k=0}^{k \le N} k\, p_k = Np = N/M$$

Данная формула является весьма очевидной, но, все же, ее можно доказать в лоб:

$(x-q)^N = \sum_{k=0}^{k \le N} C_N^k x^k q^{N-k}$; продифференцируем по x:

$(x-q)^N{}' = N(x-q)^{N-1} = \sum_{k=0}^{k \le N} k\, C_N^k x^{k-1} q^{N-k}$

Теперь, если взять $x=p,\ q=1-p$, то получим

$l_N = \sum_{k=0}^{k \le N} k\, p_k = p\, N(p-(1-p))^{N-1} = Np$

Т.о., мы доказали следующую теорему

Теорема. *Если хэш-функция* ***h(x)*** *с равной вероятностью принимает все свои значения* $0 \le h(x) < M$*, то среднее время поиска, добавления, удаления элемента в хэшируемом множестве, реализованном с помощью метода многих списков,*

$$T_{N,M} = \Theta(N/M).$$

В худшем случае для поиска, добавления, удаления элемента требуется время, равное $\Theta(N)$.

Здесь следует понимать, что мы рассматриваем хешируемое множество именно как **множество**, поэтому и для удаления и для добавления элемента требуется сначала осуществить поиск элемента. После удачного или неудачного поиска, собственно, удаление или добавление элемента осуществляются за постоянное время.

Для случая хэширования с помощью массивов оценки аналогичны (отметим, что в этом случае усложняется удаление элемента из множества).

Несомненным преимуществом данного подхода является возможность прямого перебора всех элементов множества.

Недостатком данного подхода являются большие накладные расходы на реализацию списка.

В случае использования STL реализация данного подхода становится предельно простой, что не снижает накладных расходов на реализацию списка.

Открытая адресация. Метод линейных проб

Можно попробовать использовать для хранения данных таблицу без ссылок. Будем дополнительно в каждой ячейке таблицы хранить информацию о том – занята ли ячейка или нет. Если при занесении в таблицу нового элемента хэш-функция от заносимого значения укажет на пустую ячейку, то проблем никаких нет. Иначе, получается ситуация, называемая ***коллизией***. Разрешение коллизий является основной целью при создании алгоритмов работы с хешируемыми множествами.

Простейший способ разрешения коллизий в таблице следующий: если элемент, на который указывает хэш-функция, занят, то мы рассматриваем последовательно все элементы таблицы от текущего, пока не найдем свободной место. Новый элемент помещается в найденное свободное место.

Отметим, что можно рассматривать и другие алгоритмы поиска свободного места в хэш-таблице (а следовательно, и алгоритма поиска элементов с тем же значением хеш-функции). Единственное ограничение на метод перебора элементов таблицы: перебор должен гарантировать просмотр всех элементов таблицы.

Например, вместо того, чтобы брать следующий элемент, можно брать элемент с шагом h, где h взаимно-просто с длиной таблицы (=**линейное пробирование**).

Можно вычислять значение h отдельно для каждого значения хэш-функции (=**двойное хеширование**). В этом случае мы можем, например, использовать в качестве длины таблицы только простые числа, а шаг h будем вычислять с помощью второй хеш-функции, которая выдает значения от 1 до длины таблицы.

Можно увеличивать шаг с каждым шагом поиска следующего элемента (=**квадратичное пробирование**). В этом случае шаги могут вычисляться, например, по формулам: $h_1= 1,\ h_2= 2,\ \ldots,\ h_k= k$ для таблицы длины 2^P (соответствует индексам $i_k= i_0+k^2/2+k/2 \pmod{2^P}$). В общем случае, шаги могут увеличиваться каждый раз на нечетное целое: $h_1= 2h+1,\ h_2= 2(2h+1),\ \ldots,\ h_k= k(2h+1)$ (где $h\geq 0$) для таблицы длины 2^P. Докажем, что при таком выборе шагов полученные подряд 2^P целых чисел будут различны и, следовательно, покроют весь набор целых чисел от 0 от 2^P (не включительно).

Сначала выведем явную формулу соответствующих x_i таких, что $x_i - x_{i-1} =(2h+1)i$. Легко получить, что

$x_i= x_0+(h+1/2)i^2+(h+1/2)i \pmod{2^P}$.

Отметим, что, несмотря на дробные коэффициенты, x_i целые. Осталось решить уравнение

$(h+1/2)i^2+(h+1/2)i = (h+1/2)j^2+(h+1/2)j \pmod{2^P}$

для $0\leq i,j<2^P$

Имеем:

$(h+1/2)i^2+(h+1/2)i - (h+1/2)j^2-(h+1/2)j = t\,2^P$

$(2h+1)(i-j)(i+j+1) = t\,2^{P+1}$

$(i-j)(i+j+1) = 2^{P+1}\,t/(2h+1)$

Возможны два случая: либо $i-j$ четно и $i+j+1$ нечетно, либо $i-j$ нечетно и $i+j+1$ четно.

a) $i-j$ четно и $i+j+1$ нечетно

в этом случае $i-j$ должно делиться на 2^{P+1}и в силу ограничений на i,j получаем, что $i=j$.

b) $i-j$ нечетно и $i+j+1$ четно

в этом случае $i+j+1$ должно делиться на 2^{P+1} , но в силу ограничений на i,j получаем, что $i+j+1\leq 2^P-1+2^P-1+1=2^{P+1}-1$, *следовательно* $i=j$.

Таким образом мы доказали следующую теорему:

Теорема. *Квадратичное пробирование с шагами* $h_k= k(2h+1)$ *(где* $h\geq 0$*) для таблицы размером* 2^P *гарантирует, что первые* 2^P *перебираемых элементов будут различны и покроют все значения таблицы. Данный перебор соответствует прямой формуле для вычисления индекса i-го элемента:* $k_i= k_0+(h+1/2)i^2+(h+1/2)i \pmod{2^P}$.

Чтобы отследить ситуацию с переполнением таблицы, мы введем переменную ***M*** – счетчик количества занятых ячеек в таблице. Если значение ***M*** достигло величины ***N-1***, то мы считаем, что наступило переполнение. Это гарантирует нам, что в таблице всегда будет в наличии хотя бы одна пустая позиция.

При таком способе разрешения коллизий для поиска элемента ***x*** в таблице требуется вычислить хэш-функцию от значения элемента ***h(x)***. Если позиция с номером ***h(x)*** пуста, то элемент ***x*** в таблице отсутствует. Иначе перебираются элементы таблицы от позиции с номером ***h(x)*** до первой пустой позиции. Если среди этих элементов ***x*** найден не будет, то он отсутствует в таблице.

На языке С алгоритм поиска элемента ***value*** в таблице из ***Nmax*** целых чисел можно реализовать следующим образом.

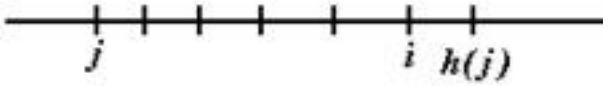
```
#define Nmax 1000
typedef struct CNode_{int value; int is_empty;} CNode;
CNode node[Nmax];
int hash(int value);
int search(CNode node[])
{
 int  i;
 for(i=hash(value); !node[i].is_empty; i=(i==0? Nmax-1 : i-1))
  if(node[i].value==value)return i;
 return –1;
}
```

здесь ***hash(int value)*** – хэш-функция; член структуры ***is_empty*** равен нулю, если элемент пуст, единице – иначе. В данной реализации поиск идет по направлению уменьшения индекса. Наличие в таблице хотя бы одной пустой позиции гарантирует нам, что цикл не будет вечным.
Чтобы сэкономить память признаки пустоты элементов таблицы можно реализовать отдельно от самой таблицы. Тогда под каждую ячейку можно будет отвести ровно 1 бит информации:

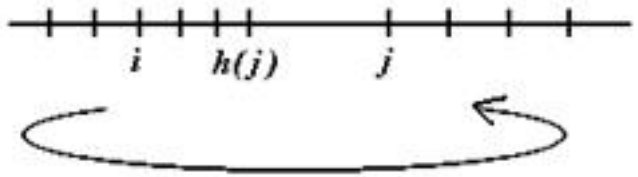
```
#define Nmax 1000
int value[Nmax], is_empty[(Nmax+31)/32];
int hash(int value);
int search(CNode node[])
{
 int  i;
  for(i=hash(value); !(is_empty[i/32]&(i%32)); i=(i==0? Nmax-1 : i-1))
  if(node[i].value==value)return i;
 return –1;
}
```

Удаление элемента из таблицы несколько более сложное, чем добавление и поиск. Нельзя просто объявить позицию ***i***, в которой требуется удалить элемент, пустой. Если это сделать, то элемент ***value[j]*** с индексом ***j<i***, для которого ***h(value[j])≥i*** и для которого все элементы с индексами между ***j*** и ***i*** заняты, окажется потерянными для последующего поиска (здесь мы рассматриваем случай отсутствия `перескока' в конец массива при поиске очередного элемента). Действительно, при поиске этого элемента мы обязательно натолкнемся на пустую позицию ***value[i]*** и поиск будет завершен.

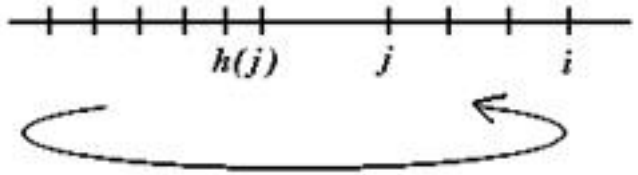
Чтобы исправить ситуацию мы должны найти первый такой элемент ***value[j]***, перенести его значение в позицию ***i*** и свести задачу к удалению элемента ***value[j]***. Естественно, мы должны учитывать возможность `перескока' в конец массива при поиске такого ***value[j]***. Т.е., более строго, условия переноса ***value[j]*** в позицию ***i*** следующее:
h(value[j])≥i>j (отсутствие перескока)



или
j>h(value[j])≥i (перескок)



или
i>j>h(value[j]) (перескок)



Условием остановки алгоритма является пустота позиции ***j***.
Подпрограмма удаления элемента с индексом ***i*** из списка может выглядеть следующим образом

```
void remove(CNode node[], int  i)
{int j; node[i].is_empty=1;
 for(j=(i==0? Nmax-1 : i-1); !node[j].is_empty; j=(j==0? Nmax-1 : j-1))
 if( (hash(value[j])>=i && i>j) || (hash(value[j])>=i && j>hash(value[j])) ||
  (i>j && j>hash(value[j])))
 {
  node[i].is_empty=0; node[i].value=node[j].value;
  remove (node,j);
  break;
 }
}
```

Оценим время, необходимое для поиска, вставки, удаления элементов из таблицы, хэшируемой методом линейных проб.
Мы будем использовать следующее ***предположение***: все расположения ***m*** элементов в хешированной таблице, состоящей из ***n*** записей равновероятны.

Лемма. $\sum_{i=0}^{M} C_{L+i}^{L} = C_{L+M+1}^{L+1}$
Доказательство.
M=0: $C_L^L = C_{L+1}^{L+1}$
M: $\sum_{i=0}^{M} C_{L+i}^{L} = \sum_{i=0}^{M-1} C_{L+i}^{L} + C_{L+M}^{L} = C_{L+M}^{L+1} + C_{L+M}^{L} =$
$=(L+M)!/((L+1)!M!) + ((L+M)!/(L!M!)) = (L+M+1)!/((L+1)!M!) = C_{L+M+1}^{L+1}$
■
Оценим время добавления нового элемента к хэшированной таблице методом проб в случае, когда в таблице из ***N*** записей занято ***M*** позиций. Это время, фактически, совпадает с неудачным временем поиска значения в таблице (т.е. поиска элемента, если его в таблице нет).
Если вслед за позицией, на которую указала хэш-функция следует ***k-1*** занятая запись (включая запись, на которую указала хэш-функция), то время вставки есть $Q(k)$ (т.к. для поиска свободной позиции надо будет осуществить ***k*** проб).
Пусть вероятность того, что начиная с данной позиции следует ***k-1*** занятых позиций, а следующая позиция свободна, равна p_k. Тогда, учитывая то, что количество подряд занятых позиций не может превзойти ***M***, среднее время вставки элемента пропорционально
$T_M' = \sum_{i=1}^{M+1} k\, p_k$
Согласно вышеприведенному предположению, вероятность p_k равна количеству перестановок оставшихся ***M-k+1*** занятых позиций среди оставшихся ***N-k*** записей, деленное на общее число перестановок C_N^M. Итак
$p_k = C_{N-k}^{M-k+1} / C_N^M$

$T_M' = \sum_{k=1}^{M+1} k\, p_k =$
(учитывая $(N+1)\sum_{i=1}^{M+1} p_i = (N+1)$)
$N+1 - \sum_{k=1}^{M+1} (N+1-k)\, p_k =$
$N+1 - \sum_{i=1}^{M} (N+1-k)\, C_{N-k}^{M-k+1} / C_N^M =$
(учитывая $C_n^k = C_n^{n-k}$)
$=N+1 - \sum_{k=1}^{M+1} (N+1-k)\, C_{N-k}^{N-M-1} / C_N^M =$
$=N+1 - \sum_{k=1}^{M+1} ((N-k+1)! / ((N-M-1)!(M-k+1)!)) / C_N^M =$
$=N+1 - \sum_{k=1}^{M+1} (N-M)\, C_{N-k+1}^{N-M} / C_N^M =$
(замена $i=M-k+1,\ k=M-i+1$)
$=N+1 - \sum_{i=0}^{M} (N-M)\, C_{N-M+k}^{N-M} / C_N^M =$
(согласно лемме)
$N+1 - (N-M)\, C_{N-M+M+1}^{N-M+1} / C_N^M = N+1 - (N-M)\, C_{N+1}^{N-M+1} / C_N^M$

$=N+1-(N-M)\ ((N+1)!/((N-M+1)!M!))\ \ M!(N-M)!/N! =$
$= N+1 - (N-M)(N+1)/(N-M+1) = (N+1)/(N-M+1)$

Оценим теперь время удачного поиска. Легко увидеть, что все операции, производимые для удачного поиска некоторого элемента списка, совпадают с операциями, которые производились для вставки этого элемента в список. Среднее время удачного поиска равно среднему из всех средних времен, затрачиваемых на поиск каждого элемента в списке. Поэтому, среднее время удачного поиска равно среднему из времен вставки элементов, находящихся на данный момент в списке, в список. Т.о., если рассматривать элементы в списке в порядке их поступления в список, то получим, что среднее время удачного поиска равно
$T_M = 1/(M+1) \sum_{i=0}^{M} T_i' = 1/(M+1) \sum_{i=0}^{M} (N+1)/(N-i+1) =$
$= 1/(M+1) \sum_{i=0}^{M} (N+1)/(N-i+1) = 1/(M+1) \sum_{j=N-M+1}^{N+1} (N+1)/j \approx$
$(N+1)/(M+1) \int_{t=N-M+1}^{N+1} 1/t\, dt = (N+1)/(M+1)\ \log((N+1)/(N-M+1))$

Осталось заметить, что, согласно алгоритму удаления элемента, в худшем случае удаление элемента работает за время пропорциональное неудачному поиску элемента.
Т.о. если ввести коэффициент заполненности таблицы $a=M/N$, то верна следующая теорема
Теорема. *Среднее время неудачного поиска элемента в таблице (=время добавления нового элемента), состоящей из* ***N*** *записей,* ***M*** *из которых заполнены* $T_M' = Q(1/(1-a))$, *где* $a=M/N$ *– коэффициент заполненности таблицы.*
Среднее время удачного поиска имеет следующую оценку
$T_M = Q((1/a)\ \log(1/(1-a)))$
Среднее время удаления элемента равно
$S_M' = O(1/(1-a))$.

Метод цепочек для открытой адресации
Для разрешения коллизий метод проб можно немного модифицировать. В каждый элемент таблицы, состоящей из ***N*** ячеек, можно добавить ссылку на элемент таблицы, с помощью которой можно ускорить поиск следующего элемента.

Для локализации свободного места введем переменную ***P***, которая изначально указывает на конец списка: ***P=N+1***. При возникновении коллизии свободную ячейку мы будем брать исходя из значения ***P:*** для этого ***P*** уменьшается на 1 до тех пор, пока ячейка с индексом ***P*** не станет свободной. Найденная ячейка будет использована для нового значения, заносимого в таблицу. Таким образом, все ячейки с индексами больше ***P*** всегда заняты.
Каждому значению хэш-функции будет сопоставлена цепочка ячеек, в которой будут находиться все значения, соответствующие данному значению хэш-функции. При добавлении нового значения ***x*** в ячейку таблицы с индексом ***P*** (если возникла коллизия) ссылка из последней ячейки соответствующей цепочки должна направиться на ячейку ***P***.
Отметим, что цепочки могут сливаться, поэтому в одной цепочке могут одновременно находиться значения с различными значениями хэш-функции. Этим данный алгоритм нисколько не отличается от метода линейных проб.
Более формально, для хранения хэшированной методом цепочек таблицы, состоящей из не более чем ***1000*** целых чисел, необходимо определить следующие данные

```
#define Nmax 1000
typedef struct CNode_
{int value; int is_empty; int next;} CNode;
CNode node[Nmax];
int P;
int hash(int value);
```

Здесь мы использовали те же самые обозначения, что и раньше, плюс к тому, появился член структуры ***next***, служащий номером следующей ячейки в данной цепочке и целая переменная ***P*** для указания на номер последнего включенного в список элемента, при включении которого произошла коллизия.
Функция инициализации данных может выглядеть следующим образом

```
        void Init(CNode node[])
        {int i;
for(i=0;i<Nmax;i++){node[i].is_empty=1; node[i].next=-1;} P=Nmax;
        }
```

здесь мы использовали информацию о том, что ячейки в таблице имеют индексы от ***0*** до ***Nmax-1*** и, поэтому, ***P*** указывает на первую (не существующую) ячейку после таблицы.

Процедура поиска в таблице будет иметь следующий вид

```
int search(CNode node[])
{
 int  i; i=hash(value);
 if(node[i].is_empty)return –1;
 for(; node[i].next>=0; i=node[i].next)
  if(node[i].value==value)return i;
 return –1;
}
```

Процедура занесения значения ***value*** в таблицу будет иметь следующий вид

```
int add(CNode node[], int value)
{
 int  i; i=hash(value);
 if(node[i].is_empty)//если нет коллизии
 {node[i].value=value; node[i].next=-1; return 0;}
//иначе – если коллизия есть: ищем конец цепочки:
  for(; node[i].next>=0; i=node[i].next);
//ищем свободное место:
 do{ P--;
      if(P<0)return –1;//переполненние
    }while(!node[P].is_empty) ;
//помещаем элемент на найденное место:
 node[P].value=value; node[P].is_empty=0; node[P].next=-1;
 node[i].next=P;
 return 0;
}
```

Для того, чтобы понять преимущество метода цепочек над методом линейных проб, рассмотрим следующую ситуацию. Пусть таблица сильно заполнена. Пусть в данный момент в таблицу еще не заносились элементы в позицию с индексом ***i***. При первом появлении элемента x_1 такого, что $h(x_1)=i$ все происходит почти также, как и в методе проб: элемент заносится в таблицу, ссылка на следующий элемент устанавливается в пустоту:

```
node[i].value= x1
node[i].next=-1
node[i].is_empty=0
```

Если же появится еще один элемент x_2 такой, что $h(x_2)=i$, то сразу станет ясно отличие от метода проб: для нового элемента ищется свободное место с помощью уменьшения ***P*** до тех пор, пока не станет выполняться ***is_empty[P]==1***; новый элемент помещается в позицию ***P*** и ссылка ***next[i]*** устанавливается на позицию ***P***.
Т.о., с одной стороны, мы не просматриваем в поисках свободного места заведомо заполненную часть таблицы, а с другой - поиск элемента x_2 теперь займет всего два сравнения. Вообще, переменная ***P*** может смещаться на одну позицию не более, чем ***N*** раз, а т.к. в таблицу заносится не более, чем ***N***

элементов, то получается, что в среднем для поиска свободного места для одного элемента переменная ***P*** изменяется всего на 1, т.е. в среднем проводится не более одной проверки на пустоту очередной позиции с индексом ***P***.

Хэш-функции

Существует два наиболее простых метода построения хэш-функций: на основе деления и на основе умножения.

Хэш-функции на основе деления

Пусть требуется для числа ***A*** получить значение хэш-функции. Предлагается в качестве хэш-функции использовать остаток от деления ***A*** на некоторое ***K***

$$h(A)=A \pmod K$$

Если ***A*** имеет довольно большую длину (например, ***A*** – строка текста), то данный алгоритм применим и в этом случае. Будем далее остаток от деления обозначать через оператор ***%***. Представим ***A*** как число в позиционной системе счисления

$$A=\sum_{k=0}^{k<N} r^k c_k$$

где ***r=256*** (в общем случае ***r*** – основание системы счисления, в которой представляется число ***A***), c_k–значение ***k***-ой цифры в представлении ***A*** (в нашем случае = код ***k***-ого символа строки), ***N*** – количество знаков (цифр) в представлении ***A***. Будем предполагать, что

$$K<r.$$

Легко увидеть, что

$$(r^{N-1} c_{N-1} + r^{N-2} c_{N-2})\%K = (r^{N-2} ((r c_{N-1} + c_{N-2})\%K))\%K$$

Из чего сразу получаем, что

$$h(A)=(\sum_{k=0}^{k<N} r^k c_k)\%K = (\sum_{k=0}^{k<N-1} r^k d_k)\%K,$$

где $d_{N-2}=(r c_{N-1} + c_{N-2})\%K$, $d_i = c_i$ $(i<N-2)$.

Т.о. следующая подпрограмма вычисляет хэш-функцию на основе деления от строки текста

```
int hash(unsigned char *str, int K)
{unsigned short int p=0;
 if(str[0]=='\0')return 0;
 if(str[1]=='\0')return str[0]%K;
 p=str[0];
 while(str[1]!='\0')
 {
  p=(p<<8)|str[1];
  p=p%K;
  str++;
 }
 return p;
}
```

Хэш-функции на основе умножения

Алгоритм вычисления хэш-функции, основывающийся на умножении, задается следующей формулой

$$h(A) = [M(\{AK/ R\})]$$

здесь фигурные скобки являются оператором взятия дробной части, квадратные скобки являются оператором взятия целой части, ***R*** – размер машинного слова, в котором размещается ***A*** (например, если ***A*** размещается в целой 32-битной переменной, то $R=2^{32}$), ***K*** – некоторое число, взаимно простое с ***R***. В качестве ***M*** часто выгодно брать $M=2^m$.

Алгоритм является обобщением алгоритма, основанного на делении. Действительно, пусть ***K*** есть некоторое приближение к ***R/S, M=S,*** то

$$h(A) = [M(\{AK/ R\})] = [S(\{A / S\})] \approx A\%S$$

Практически, алгоритм сводится к следующему: ставится десятичная точка перед числом ***A***, полученное число умножается на ***K***, из результата берутся первые ***m*** бит, расположенных после десятичной точки (здесь под *десятичной точкой* имеется в виду разделитель целой и дробной части в позиционной системе отсчета).

В случае, когда ***A*** представляет собой строку текста, состоящую из ***n*** байт, то, опять же, ***A*** рассматривается как число в позиционной системе исчисления. В этом случае $R=256^n$.

Умножение можно производить `столбиком', тогда для $m \pounds 16$ подпрограмма вычисления хэш-функции имеет следующий вид

```
int hashm(unsigned char *str, int K, int m)
{union ICHAR {unsigned int i; unsigned char c[4];}s,srez,sm;
 int rez,l,i; l=strlen((char*)str); srez.i=s.i=sm.i=0;
 for(i=l-1;i>=0;i--)
 {
  srez.c[0]=s.c[0];    //кладем младший байт из пред.знач.
  s.i=0; s.c[0]=str[i];
  s.i*=K;              //умножаем K на очередную цифру
  s.i+=sm.i;           //добавляем запомненное
  sm.c[0]=s.c[1];sm.c[1]=s.c[2];sm.c[2]=s.c[3];sm.c[3]=0;
 srez.c[1]=s.c[0];
 }
 rez=srez.i>>(16-m); //извлекаем m бит после точки
 return rez;
}
```

Здесь мы ввели ***union ICHAR*** для того, чтобы иметь возможность обращаться к отдельным байтам целой переменной. В цикле мы умножаем ***K*** на каждый байт

строки и добавляем к результату то, что осталось от переполнения в предыдущем умножении. При этом под переполнение отводится 3 старших байта переменной ***s***, а под основную цифру – один младший байт. Результат переполнения хранится в переменной ***sm***. Т.к. конечный результат может занимать до двух байт, приходится вводить дополнительную переменную ***srez***, для хранения последних байт произведения ***str*K***. Либо то же самое чуть проще и с конкретными константами:

```
unsigned short HashM(char *s_,unsigned short K=253*253,int m=8) //[M*{AK/R}];
M=1<<8, R=1<<(strlen(s_)*8), K=253*253
{unsigned int h=0,h0; unsigned char *s=(unsigned char*)s_; //будем уверены в
беззнаковости char
 int i,l=strlen(s_);
 if(s[0]=='\0')return 0;
 for(i=l-1;i>=0;i--)
 {
  h0=h;
  h=s[i]*K+(h>>8);
 }
 h=(((h&255)<<8)|(h0&255))>>(16-m);
 return h;
}
```

CRC-алгоритмы обнаружения ошибок

Вообще говоря, использование в структурах данных – не единственное применение хэш-функций. Следует упомянуть еще хотя бы о двух областях, в которых появляются аналоги хэш-функций: составление контрольных сумм и криптография. Часто эти области имеют большое пересечение.
Контрольные суммы применяются в случаях, когда требуется уверенность в неизмененности передаваемых данных. Они активно используются при передаче данных по сети, при хранении данных на различных носителях и т.д.
Криптографические хэш-функции представляют собой такие функции, которые, вообще говоря, легко вычислить, но вычисление обратной функции к которым практически невозможно. Например, при хранении паролей в ЭВМ принято хранить не сами пароли, а значение некоторой криптографической функции от них. Процедура проверки правильности вводимого пароля сводится к вычислению от него той же самой функции и сравнению полученного значения с сохраненным.
Здесь мы немного расскажем об одном из наиболее часто используемых способов создания контрольных сумм – CRC (данный метод не является криптографически стойким!). Данный класс алгоритмов достаточно хорошо выявляет ошибки в потоке входных данных. Однако, алгоритм не лишен недостатков. Так, например, существуют алгоритмы, позволяющие добавлять к данным дополнительные байты таким образом, чтобы значение CRC не изменялось бы. Это ограничивает, например, возможности использования данных алгоритмов при подсчете контрольных сумм выполняемых файлов (действительно, из сказанного следует, что злоумышленник может изменить содержимое выполняемого файла, а затем, добавив в него необходимые байты, подогнать значение контрольной суммы к исходной).
Алгоритмы CRC основаны на понятии *полиномиальная арифметика*. В ней коэффициенты в позиционном представлении числа $a=\{a_0,a_1,\ldots,a_N\}$ рассматриваются, как коэффициенты многочлена $P_a = a_0 + a_1 x + \ldots + a_N x^N$.
Арифметические действия, при этом, переопределяются как действия над многочленами. Нам интересен случай, когда рассматривается двоичное представление числа, а сами коэффициента многочлена рассматриваются как элементы кольца вычетов по модулю 2. Для данного модуля кольцо является еще и полем, т.е. в нем корректно определены операции сложения, вычитания, умножения и деления.
Можно забыть о многочленах и тогда говорить о соответствующих операциях с точки зрения правил выполнения операций над самими числами. Тогда мы получим арифметику, аналогичную обычной, но без операций выполнения переносов:
1+1=0
1-1=0
101+011=110

Деление целых чисел в этих терминах производится как обычно – столбиком. Далее под ***делением*** мы будем иметь в виду деление именно в указанном смысле. Отметим, что операция сложения и вычитания здесь не отличаются.
Исходные данные представляются как одно большое двоичное число ***X***. Вычисление контрольных сумм сводится к вычислению остатка от деления числа ***X*** на некоторое заранее заданное число ***m***. Приведем примеры стандартных значений ***m*** для различных алгоритмов (в скобках номера единичных битов):

```
16 бит:  (16,12,5,0)
           [стандарт X25]

         (16,15,2,0)
["CRC-16"]

32 бит:  (32,26,23,22,16,12,11,10,8,7,5,4,2,1,0)
[Ethernet]
```

Отметим, что для получения ***n***-битного остатка от деления требуется ***n+1*** -битный делитель.
Деление столбиком сводится к следующему. Заводится переменная ***x*** длиной ***n+1*** бит, которую мы будем называть ***аккумулятором***. В нее записываются первые ***n+1*** бит данных. Далее циклически выполняется следующий шаг
Если старший бит ***x*** *равен 1, то* ***x=x^m****. Далее* ***x*** *сдвигается влево на 1 бит и в младший бит записывается следующий бит данных.*
В конце в переменной ***x*** будет лежать остаток от деления на ***m***.
При использовании ***n+1*** -битного делителя ***m***, реальные данные дополняются с конца ***n*** нулями, от полученных данных считается остаток от деления на ***m***. Полученное значение записывается на место ***n*** последних нулей данных. Последнее эквивалентно вычитанию (=сложению) из данных ***m***, поэтому остаток от деления полученного большого числа на ***m*** станет равным 0. Именно это свойство и можно использовать при проверке сохранности данных. Например, данный подход используется при сетевой передаче данных по протоколу Ethernet.
Обычно, алгоритм немного модифицируется. Проблема заключается в том, что остаток от деления не зависит от добавления некоторого количества нулей в начало данных. Для избежания этой проблемы в начало данных можно записывать некоторые стандартные биты. Данную операцию можно осуществить иначе – в аккумулятор, можно изначально не просто помещать первые ***n+1*** бит данных, а выполнять еще $x=x\hat{}x_0$, где x_0 – некоторое начальное значение. Используется, также, конечное значение x_1, для которого выполняется аналогичная операция с конечным результатом.
Стандартный алгоритм *CRC16* не использует начальные и конечные значения. Модификация *CRC16/CITT* использует стартовое значение *FFFFh*. Алгоритм *CRC32* использует *FFFFFFFFh* в качестве начального и конечного значений.
В следующем примере рассчитывается 16-битная CRC от заданной строки с двумя добавленными нулевыми байтами. Далее CRC записывается в два последних нулевых байта и проверяется, что CRC от полученной строки =0.

```
short int X=(1<<15)|(1<<2)|(1<<0);
unsigned short CRC16(char *s,int l)
{int i; unsigned short R; //R: s[0], s[1]
 for(R=(s[0]<<8)|s[1],i=0;i<l-16;i++)
  if(R&(1<<15))
   R=((R<<1)|GetBit(s,i+16))^X;
  else
   R=((R<<1)|GetBit(s,i+16));
 return R;
}
int main(void)
{char str[256]="int Find(int v){int i=Hash(v); for(list<int>::iterator j=m[i].begin();j!
=m[i].end();j++)if(*j==v)return 1; return 0;}";
 int l=strlen(str)*8+16,l0=strlen(str); short int crc=0;
 str[l0]=0;str[l0+1]=0;//добавляем два нулевых байта
 crc=CRC16(str,l); printf("crc=%hd\n",crc,crc);//считаем CRC
 str[l0+1]=(crc>>0);
 str[l0]=crc>>8;//записываем CRC в последние нулевые байты
 crc=CRC16(str,l); printf("check crc=%hd\n",crc);//убеждаемся, что CRC==0
return 0;
}
```

Лекция 15

Поиск строк

Пусть имеется последовательность символов $S=\{ s_i \}$ из алфавита $S: s_i \in S$, $i=1,\ldots,N$ и последовательность $W=\{ w_i \}$ из алфавита $S: w_i \in S$, $i=1,\ldots,M$, $M \le N$.
Ставится задача поиска всех таких целых $0 \le k \le N-M$, что для всех $i=1,\ldots,M$: $s_{k+i}=w_i$.
Стандартной интерпретацией данной задачи является задача поиска заданного слова в строке или задача поиска слова в файле.
У данной задачи существует прямое решение, при котором происходит последовательная проверка совпадения подстроки ***W*** со всеми подряд идущими подстроками строки ***S*** длины ***M***. Легко увидеть, что данный алгоритм требует времени порядка $Q(MN)$ (реализация данного алгоритма приведена в следующем параграфе). На самом деле задачу можно решить существенно быстрее, о чем и пойдет речь далее.

Отступление на тему языка C. Ввод-вывод строк из файла

Стандартной интерпретацией поставленной задачи является задача поиска заданного слова в текстовом файле. В ОС UNIX имеется стандартная программа ***grep*** поиска слов по шаблону в текстовых файлах. Ее простейший формат вызова следующий:
grep шаблон список_файлов_поиска
здесь вместо слова ***шаблон*** можно подставить просто слово, которое требуется найти в тексте файлов из списка ***список_файлов_поиска***. Имена файлов в списке разделяются пробелом. Если список имен файлов пуст, то слово ищется в стандартном потоке ввода.

Следующая программа демонстрирует, как можно простейшим способом реализовать функцию ***grep*** для случая, когда строки в файлах имеют длину не более 512 символов и когда вместо шаблона поиска вводится простое слово.

```
#include <stdio.h>
#include <string.h>
int main(int npar,char **par)
{FILE *f; int i,istr; char str[512]; if(npar<=1)return -1;
 for(i=(npar==2?1:2);i<npar;i++)
 {
  f=(npar==2?stdin:fopen(par[i],"r"));
  if(f)
  {
   for(istr=1;fgets(str,512,f);istr++)
    if(strstr(str,par[1]))
    {printf("%s: %d: %s",par[i],istr,str);}
   fclose(f);
  }
 }
 return 0;
}
```

Программа демонстрирует следующие возможности:

- Передачу параметров из командной строки
- Открытий/закрытие файлов
- Ввод текста из файла
- Использование стандартного потока ввода
- Стандартную процедуру поиска слова в тексте

Детальное описание всех указанных возможностей следует искать в документации к языку С.

Алгоритм поиска подстроки с использованием хеш-функции (Алгоритм Рабина-Карпа)

Идея алгоритма проста: для каждой подстроки S_i строки S, используемой при сравнении с W (т.е. подстроки длины, равной длине W), вычисляется значение некоторой хэш-функции $h(S_i)$. Если $h(S_i)= h(S)$, то данная подстрока является хорошим претендентом на равенство и для нее производится полное сравнение, иначе переходим к следующей подстроке S_{i+1}. При вычислении $h(S_i)$ мы можем использовать тот факт, что строка S_i отличается всего на два символа от строки S_{i-1}, поэтому есть шанс использовать уже вычисленное значение $h(S_{i-1})$ для вычисления $h(S_i)$. Действительно, это можно сделать, если в качестве хэш-функции использовать остаток от деления на некоторое число K. При этом строка должна интерпретироваться как одно большое целое число.

Действительно

$$S_i=(s_{i+0},\dots, s_{i+M-1}) =(s_{i-1}, s_{i+0},\dots, s_{i+M-1})-256^M s_{i-1}=$$
$$=256(s_{i-1}, s_{i+0},\dots , s_{i+M-2}) + s_{i+M-1} - 256^M s_{i-1}$$

из чего вытекает

$$S_i\%K = (256 ((s_{i-1}, s_{i+0},\dots , s_{i+M-2}) \%K) + s_{i+M-1} - (256^M\%K)\ s_{i-1}) \%K$$
$$h(S_i) = (256\ h(S_{i-1}) + s_{i+M-1} - (256^M\%K)\ s_{i-1}) \%K$$

Единственное большое число, возникающее в последней формуле, это 256^M, поэтому $(256^M\%K)$ следует вычислить заранее. Наконец, если K выбрать таким образом, чтобы $256K<2^{31}-257$, то $(256^M\%K)\ s_{i-1}<2^{31}-257$, $256\ h(S_{i-1}) <2^{31}-257$ и тогда все вычисления могут производиться в рамках обычных целых чисел.

Действительно

$$|256\ h(S_{i-1}) + s_{i+M-1} - (256^M\%K)\ s_{i-1}|$$
$$\le MAX(|256\ h(S_{i-1})|,|(256^M\%K)\ s_{i-1}|)+ s_{i+M-1}$$
$$\le 2^{31}-1$$

что помещается в переменную ***int.***

Осталось заметить, что K должно быть простым числом. В качестве K можно взять $K =8388593$. Действительно

$$256K = 2147479808$$
$$2^{31}-257 = 2147483391$$

При идеальном распределении значений хэш-функции каждое ее значение будет появляться с вероятностью $1/K$, поэтому время работы алгоритма для неудачного поиска будет складываться из времени предварительных вычислений $Q(M)$, времени поиска при отсутствии коллизий $Q(N)$ и времени поиска при наличии коллизий $Q(MN/K)$. Полное время поиска при наличии в строке S n вхождений строки W будет следующим

$$T=Q(M+N+MN/K)+ Q(Mn)$$

Итак, мы доказали следующую теорему

Теорема. *При идеальном распределении значений хэш-функции в среднем алгоритм Рабина-Карпа требует времени*

$$T=Q(M+N+MN/K)+ Q(Mn)$$

где M – длина искомой подстроки, N – длина строки входных данных, n – количество вхождений искомой строки в строку входных данных, K – модуль, используемый при вычислении остатка от деления в хэш-функции.

В худшем случае алгоритм работает за время

T*=Q*(MN).

Конечные автоматы

Начнем с примера тривиального конечного автомата.
Пусть у нас имеется некоторая кучка камней. В каждый момент ***состояние*** кучки ***q*** отражается числом, равным количеству камней в кучке. В ***начальный момент*** в кучке находилось q_0 камней. Последовательно подаются запросы на добавление или удаление камня из кучки. Нас интересует момент, когда в кучке камней не останется или, что, то же самое, когда кучка придет к состоянию ***q=0***.
Запросы на добавление/удаление камней можно проинтерпретировать как последовательность элементов a_i из ***алфавита*** S, состоящего всего из двух чисел: ***1*** и ***–1***.
При появлении элемента a_i состояние кучки изменяется, причем новое состояние можно вычислить как ***функцию*** от исходного состояния q_{i-1} и пришедшего элемента a_i : $q_{i-1} = d(q_i, a_i) = q_i + a_i$. Функцию d нам будет удобнее задавать таблицей

состояние \ входные символы	***-1***	***1***
0	0	1
1	0	2
2	1	3
3	2	3

Заметим, что данная табличная функция описывает кучку, состоящую не более чем из трех камней. Попытки занесения лишних камней, также как и попытка изъятия камня из пустующей кучки, игнорируются.

Осталось добавить, что выделенное состояние ***q=0***, с точки зрения конечных автоматов, называется ***принимающим***. Пример можно считать завершенным. Сведя вместе все выделенные понятия из данного примера, можно дать строгое определение конечного автомата.
Конечным автоматом называется объект, состоящий из пяти множеств:
Q – конечное ***множество состояний;***
A Ì ***Q*** – подмножество ***принимающих состояний;***
q_0 Î ***Q*** – ***начальное состояние;***
S – конечный ***входной алфавит;***
d: ***Q*** × ***A*** → ***Q*** – ***функция перехода.***

Функция перехода, обычно, задается таблицей, поэтому считается, что вычисление одного значения функции требует времени ***O(1)***.
Отметим, что алгоритмы, основанные на конечных автоматах, принципиально не вкладывается в схему алгоритмов, основанных на сравнениях, т.к. значение табличной функции не может быть вычислено в рамках алгоритмов, основанных на сравнениях, за время ***O(1)***.

Отступление на тему языка С. Работа со строками

В языке С есть очень удобная библиотека для работы со строками. Большинство функций библиотеки, безусловно, следует выучить и активно ими пользоваться. Описания функций содержатся в файле ***string.h***. Подробное описание функций следует прочитать в документации по языку С. Здесь мы кратко приведем описание нескольких функций, которые будем использовать в дальнейшем при объяснении алгоритмов, чтобы не вводить новых понятий.
int strlen(const char *);*//длина строки*
char * strcpy(char *, const char *);*//копирование второй строки в первую*
char * strcat(char *, const char *);*//подклеивание второй строки к первой*
char * strstr(char *, const char *);*//поиск второй строки в первой*
int strcmp(const char *, const char *);*//лексикографич.сравнение строк*
int strncmp(const char *, const char *,int n);*//лекс.сравн. первых **n** байт строк*

Алгоритм поиска подстроки, основанный на конечных автоматах

С этого момента мы будем говорить о строках в понимании языка С.
Итак, в строке ***S***, ***strlen(S)==N***, следует найти все вхождения подстроки ***W***, ***strlen(W)==M,*** т.е. следует найти все такие ***0*** £ ***i*** £ ***N-M***, что ***strncmp(S+i,W,M)==0***.

Будем говорить, что строка ***b*** является ***префиксом*** строки ***a***, если
strlen(b)<=strlen(a) && strncmp(b,a,strlen(b))==0.

Будем говорить, что строка ***b*** является ***суффиксом*** строки ***a***, если
strlen(b)<=strlen(a) && strcmp(b,a+strlen(a)-strlen(b))==0.

Основная идея алгоритма следующая: будем последовательно добавлять к входной строке ***S*** по одному символу из входного потока данных. При этом, каждый раз будем вычислять значение функции ***h(S,W)***, равной максимальной длине ***l*** суффикса строки ***S***, совпадающего с префиксом строки ***W*** длины ***l***:

strncmp(S+strlen(S)-l,W,l)==0

Например, для ***S=(ababa), W=(abac): h(S,W)***=3.
Если, при этом, выполнится условие

h(S,W)==strlen(W)

то это будет обозначать, что найдено вхождение ***W*** в строку ***S***.

Допустим, что в некоторый момент мы знаем значение функции ***h(S,W)***. Пусть строка ***S2*** получена с помощью добавления очередного символа ***a*** из входного потока данных в конец строки ***S***.

Легко увидеть, что ***h(S2,W)***<= ***h(S,W)+1*** (иначе, мы сразу получим, что строка ***S*** имеет суффикс длины большей ***h(S,W),*** совпадающий с префиксом ***W***), но зная значение ***h(S,W)*** мы сразу получаем значения ***h(S,W)*** последних символов ***S*** (это – первые ***h(S,W)*** символов строки ***W***). Т.о. значение функции ***h(S2,W)*** может быть вычислено исходя из знания значения ***h(S,W)*** и ***a***.

Итак, мы строим конечный автомат, в котором ***состояние*** системы задается величиной ***H= h(S,W)***. В качестве входного ***алфавита*** будут выступать символы, текста. ***Принимающим*** будет такое состояние ***H***, когда ***H==strlen(W).*** ***Начальное*** состояние $H_0=0$. О вычислении функции перехода поговорим позднее.

Итак, легко увидеть, что, если не задумываться о вычислении функции перехода, то основная часть алгоритма поиска выполняется за время ***T***=Q***(N)***, где ***N*** – длина входной последовательности текста.

Функцию перехода предлагается вычислять в лоб. Т.е. для случая, когда ищется строка W и когда алфавит состоит из 256 символов, строится таблица ***tab*** из 256 столбцов и ***strlen(W)*** строк. ***j-***ый столбец будет соответствовать появлению символа с кодом ***j***, а ***i-***ая строка будет соответствовать состоянию автомата ***i***.

Для получения значения ***tab[i][j]*** следует рассмотреть строку, состоящую из ***i*** первых символов строка ***W*** с добавленным в конец символом с кодом ***j***. Длина максимального суффикса полученной строки, совпадающего с префиксом ***W***, будет искомым значением ***tab[i][j]***.

Для получения значения ***tab[i][j]*** нужно не более ***i*** раз сравнить подстроку ***W*** с подстрокой полученной строки. Итого, ***tab[i][j]*** вычисляется за время $O(M^2)$. Все значения ***tab[i][j]*** вычисляются за время $O(256M^3)$, где 256 – количество символов входного алфавита, ***M*** – длина искомого слова. Легко увидеть, что для данного алгоритма данная оценка точна. Т.о. мы доказали следующую теорему

Теорема. *Поиск подстроки длины **M**, состоящей из символов алфавита из **K** символов, в тексте длины **N** с помощью предложенного алгоритма, использующего конечные автоматы, требует основного времени* T_1=Q***(N)***. *На подготовку, зависящую только от искомой подстроки и размера входного алфавита, требуется время* T_0=Q***(K M³)***.

Лекция 16

Алгоритм поиска подстроки Кнута-Морриса-Пратта (на основе префикс-функции)

Основная проблема алгоритма поиска подстроки, основанного на конечных автоматах – необходимость вычисления функции перехода. Алгоритм Кнута-Морриса-Пратта обходит эту проблему за счет некоторого удорожания, собственно, процесса поиска и существенного сокращения предварительных вычислений.

Основная идея алгоритма следующая. Пусть S_k – подстрока строки ***S*** длины ***k***. Пусть нам известно значение функции перехода $h(S_k,W)$ (см. предыдущий параграф). Требуется вычислить значение функции $h(S_{k+1},W)$, т.е. найти максимальный префикс ***W***, являющийся суффиксом S_{k+1}.

Если $S[k]==W[h(S_k,W)]$, то $h(S_{k+1},W)= h(S_k,W)+1$ (как уже отмечалось ранее – больше быть не может, а то, что в этой ситуации $h(S_{k+1},W)^3\ h(S_k,W)+1$ – получается по определению). Пример:

char S[]="ababab",W[]="abaa"; int k=4;

h(S,4,W)==2
S : abab ab
W : __ab

h(S,5,W)==3
S : ababa b
W : __aba

Пусть $S[k]!=W[h(S_k,W)]$, то $h(S_{k+1},W)< h(S_k,W)+1$. В приведенном примере:

char S[]="abababab",W[]="abaa"; int k=5;

h(S,5,W)==3
S : abababab
W : __aba

h(S,6,W)==2
S : abababab
W : ____ab

Для вычисления $h(S_{k+1},W)$ при отсутствии функции перехода можно не перебирать все префиксы ***W***. Действительно, $h(S_{k+1},W)$ == длине ***l*** максимального префикса ***W***, для которого ***S[k]==W[l]***, плюс 1. Тогда, для вычисления $h(S_{k+1},W)$ следует перебрать все префиксы ***W***, являющиеся

суффиксами S_k, в порядке убывания их длины и найти первый из них, для которого $S[k]==W[l]$, где l – длина префикса. Тогда $h(S_{k+1},W) = l+1$.
Итак, если бы мы могли быстро вычислять длины всех префиксов W, являющиеся суффиксами S_k, в порядке их убывания, то задача поиска подстроки выполнялась бы за время T_1=Q(N). Действительно, исходя из рассуждений, приведенных в предыдущих абзацах, T_1 пропорционально количеству изменений переменной l в процессе работы алгоритма. Но переменная l может увеличиваться на 1 не более N раз, поэтому и уменьшаться она может не более N раз. Что и требовалось доказать.

Осталось понять, *как вычислять длины префиксов W, являющихся суффиксами S_k.*

Легко заметить, что если мы знаем, что имеется префикс W, являющийся суффиксом S_k, длины l, то для вычисления максимального префикса W меньшей длины, являющегося суффиксом S_k, не надо ничего знать о S. Достаточно информации только о строке W. Действительно, т.к. W_l - суффикс S_k, то следует найти максимальный префикс W, длины меньше l, являющийся суффиксом W_l.
Введем функцию p: {1,…,N}®{1,…,N-1}, такую что ***p(l)=длина максимального префикса W_l, являющегося суффиксом W_l, длиной меньше l***.
Теперь заметим, что $W_{p(l)}$ является, одновременно, суффиксом W_l , поэтому следующий по длине (в порядке убывания) суффикс W_l , являющийся префиксом W_l , является суффиксом $W_{p(l)}$. Осталось найти длину максимального суффикса $W_{p(l)}$, с длиной меньше $p(l)$, являющегося префиксом $W_{p(l)}$. Данная величина, по определению, равна $p(p(l))=_{по\ определению}=p^2(l)$.
Т.о., по индукции, получаем, что последовательность длин суффиксов W_l, совпадающих с префиксами W_l и расположенных по убыванию длин, совпадает с последовательностью $\{l,p(l),p(p(l))\ldots\}=\{p^0(l), p^1(l), p^2(l), \ldots\}$. Т.о., если бы мы имели таблицу значений функции $p(*)$, то задача вычисления длин префиксов W, являющихся суффиксами S_k, оказалась бы решенной, что, в свою очередь, решило бы задачу поиска подстроки в строке.

Займемся вычислением табличной функции $p(*)$.

Префикс-функция $p(*)$ вычисляется в точности по уже приведенному алгоритму.
Пусть требуется вычислить $p[k+1]$, если $p[i]$ для $i£k$ уже известны.
Если $W[k]==W[p[k]]$, то $p[k+1]= p[k]+1$.
Если $W[k]!=W[p[k]]$, то, как и ранее, перебираем в порядке уменьшения длин l все префиксы W , совпадающие с суффиксами W_k , пока не выполнится

$W[k]==W[l]$

Каждое последующее l получается из предыдущего по формуле

$l=p(l);$

Положим в начале цикла $l= p[k]$, то случай $W[k]==W[p[k]]$ подпадет под вычисления внутри последнего цикла и его отдельное рассмотрение будет излишним.
Внутренний цикл следует продолжать пока k^30. Если окажется, что $k<0$, то $p[k+1]=0$. Иначе, в конце внутреннего цикла имеем: $p[k+1]= l+1$.
Отметим, что мы можем положить

$p[0]=-1;$

после чего случай $k<0$ перестанет быть выделенным (в этом случае $l=-1;p[k+1]=l+1;$ из чего сразу получаем $p[k+1]=0$).
Итак, на языке С подготовка функции (массива) p может выглядеть следующим образом

```
void MakeP(int *p, char *W, int M)
{int k,l; p[0]=-1; p[1]=0; l=0;
 for(k=1;k<M;k++)
 {
  l=p[k];
  while(l>=0 && W[k]!=W[l])l=p[l];
  p[k+1]=l+1;
 }
}
```

Основная функция, ищущая первое вхождение строки W в строку S, может выглядеть следующим образом

```
char *Search(char *S,int N, char *W, int M, int *p)
{int l=0,k;
 for(k=0;k<N;k++)
 {
  while(l>=0 && S[k]!=W[l])l=p[l];
  l++;
  if(l==M)return S+k-l+1;
 }
 return NULL;
}
```

Пример программы на языке С, использующей данные функции прилагается. Программа написана по аналогии с функцией ***grep***, которые мы обсудили в начале лекции.

Алгоритм поиска подстроки Бойера-Мура (на основе стоп-символов/безопасных суффиксов)

Алгоритм напоминает элементарный алгоритм поиска подстроки в строке. Основное отличие – сравнение искомой строки ***W*** с соответствующей частью строки ***S*** осуществляется не слева направо, а справа налево. По результатам сравнения делается вывод – на сколько можно сместиться вправо для сравнения ***W*** со следующей подстрокой ***S*** (в элементарном алгоритме поиска сдвиг всегда происходит на одну позицию). При этом, есть два независимых алгоритма, позволяющих вычислять, на сколько можно смещаться вправо для сравнения со следующей подстрокой ***S***. Выбирается максимальный из сдвигов, получаемых по этим алгоритмам. Рассмотрим эти два алгоритма.

Эвристика стоп-символа

Рассмотрим несколько примеров.

Пример 1. S="abababababa", W="cccc".

Сначала сравниваем суффикс S_4 и ***W*** (S_4 - подстрока ***S***,состоящая из ее первых четырех символов). Как уже отмечалось, сначала сравниваем ***S[3]*** и ***W[3]***. Они не равны, следовательно, S_4 и ***W*** не равны. Более того, т.к. ***S[3]*** вообще не встречается в ***W***, то далее можно сравнивать с ***W*** уже $S_{4+strlen(W)}=S_8$, т.к. суффиксы $S_{4+1},\ldots, S_{4+strlen(W)-1}$ заведомо не совпадают с ***W***.

Пример 2. S="ababacaba", W="abac".

Сначала сравниваем суффикс S_4 и ***W***. Как уже отмечалось, сначала сравниваем ***S[3]*** и ***W[3]***. Они не равны, следовательно, S_4 и ***W*** не равны. ***S[3]*** встречается первый раз в ***W*** (при просмотре с конца) в позиции ***1***, то далее можно сравнивать с ***W*** уже $S_{4+strlen(W)-1-1}=S_6$, т.к. при таком сдвиге впервые ***S[3]*** совпадет с соответствующим символом ***W***.

Вообще говоря, пусть сравнивается суффикс S_k и ***W***. Пусть ***W[l]*** – первый справа символ ***W***, не совпавший с соответствующим символом строки ***S***, т.е.

S[k-strlen(W)+l]!=W[l], S[k-strlen(W)+i]==W[i] (strlen(W)>i>l).

Если ***l==0***, то мы нашли вхождение ***W*** в ***S***. Переходим к анализу S_{k+1}.

Рассмотрим случай ***l>0***. Обозначим ***s= S[k-strlen(W)+l]*** .

Пусть ***m(s)*** – функция, выдающая самое правое вхождение символа ***s*** в строку ***W***. В случае, если символ ***s*** в строке ***W*** не найден, пусть ***m(s)=-1.*** Тогда, следующим претендентом на сравнение будет $S_{k+MAX(1,strlen(W)-1-m(s))}$.

Здесь и далее ***MAX*** и ***MIN*** в языке С можно определить следующим образом

```
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))
```

Функция ***m*** является табличной, поэтому ее можно заменить соответствующим массивом. Например, если мы производим поиск строк в языке С, то ***m*** можно определить следующим образом

```
unsigned char m[256];
```

Все значения массива изначально инициализируются значением ***-1***. Далее для всех символов ***W[i]*** строки ***W*** от первого до последнего следует положить

```
m[W[i]]=MAX(m[W[i]],i);
```

На самом деле, с точки зрения языка С, последнее утверждение **не верно!!!** Это связано с тем, что переменная ***m[k]*** , вообще говоря, знаковая. Следующая попытка исправить ситуацию тоже не верна

```
m[(unsigned)W[i]]=MAX(m[(unsigned)W[i]],i);
```

Связано это с тем, что преобразование

signed char -> unsigned int

происходит, на самом деле, более сложно:

signed char -> signed int -> unsigned int

в результате получаем, что отрицательное 8-битное число преобразуется, сначала, в отрицательное 32-битное число, а только потом произойдет преобразование к беззнаковому числу. Итого

'a'=-32 -> 4294967264

Правильное преобразование показано в следующей функции, вычисляющей массив ***m***

```
void MakeM(char W[], int l, int m[256])
{int i;
 for(i=0;i<256;i++)m[i]=-1;
 for(i=0;i<l;i++)m[(unsigned char)W[i]]=MAX(m[(unsigned char)W[i]],i);
}
```

Осталось написать подпрограмму, осуществляющую поиск первого вхождения строки ***W*** длины ***lW*** в строку ***S*** длины ***lS***

```
char *Search(char S[], int lS, char W[], int lW, int m[256])
{int l,k; if(lS<lW)return NULL;
 for(k=lW;k<=lS;k=k+MAX(1,strlen(W)-1-m[(unsigned char)W[l]]))
 {
  for(l=lW-1;l>=0;l--)if(W[l]!=S[k-lW+l])break;
  if(l<0)return S+k-lW;
 }
 return NULL;
}
```

Эвристика безопасного суффикса

Рассмотрим несколько примеров.

Пример 1. S="abababbbaab", W="abbaab".

Сначала сравниваем суффикс S_6 и ***W***. Как уже отмечалось, сравнение производим справа налево. Выясняется, что максимальный совпадающий

суффикс S_6 и W ***"ab"*** состоит из двух символов. Ближайшее справа вхождение подстроки ***"ab"*** в строку W начинается с позиции ***0***, поэтому далее можно сравнивать с W уже $S_{6+strlen(W)-strlen("ab")+0}=S_{10}$, т.к. при таком сдвиге впервые та же самая подстрока ***"ab"*** строки S совпадет с соответствующей подстрокой W.
Иными словами, в этом примере мы искали максимальное $i<6$, такое что ***"ab"*** являлась суффиксом W_i. Следующий претендент на сравнение вычислялся по формуле $S_{6+strlen(W)-i}$.

Пример 2. S="abababbaab", W="bbbaab".
Сначала, как и в предыдущем примере, сравниваем суффикс S_6 и W. Как уже отмечалось, сравнение производим справа налево. Выясняется, что максимальный совпадающий суффикс S_6 и W ***"ab"*** состоит из двух символов. Подстрока ***"ab"*** больше в строку W не входит. Однако максимальное начало строки W, совпадающее с соответствующим суффиксом ***"ab"***, имеет длину ***1***, поэтому далее можно сравнивать с W уже $S_{6+strlen(W)-strlen("ab")+1}=S_{11}$.
Действительно, при таком сдвиге впервые часть той же самой подстрока ***"ab"*** строки S (имеется в виду подстрока ***"b"***) совпадет с соответствующей подстрокой W.
Иными словами, в этом примере мы искали максимальное $i£2$, такое что W_i являлась бы суффиксом ***"ab"***. Следующий претендент на сравнение вычислялся по формуле $S_{6+strlen(W)-i}$ (сравнить с предыдущим примером).

Введем обозначение. Будем говорить, что строки A и B ***сравнимы:*** $A \sim B$, если A является суффиксом B или B является суффиксом A.
Обобщая приведенные примеры, мы можем сказать, что мы искали максимальное $i<strlen(W)$, такое что $W_i \sim$ ***"ab"***.
Введем функцию g, такую что $g(l)$ равна максимальному $i<strlen(W)$, такому, что W_i сравнима с суффиксом W длины l. Если такого не нашлось, то $g(l)=0$.
Пусть сравнивается суффикс S_k и W. Пусть C - максимальный по длине общий суффикс S_k и W. Следующим претендентом на сравнение будет

$$Sk+ strlen(W) - g(strlen(C)) .$$

Осталось выяснить – каким образом задать функцию $g(l)$.
По определению $g(l)=Max\{i<strlen(W): W_i \sim Suff(W,l)\}$, где $Suff(W,l)$ – суффикс W длины l. То же самое можно переписать иначе:

$g(l) = Max\{ Max\{i<strlen(W): W_i$ – суффикс $Suff(W,l)\}, Max\{i<strlen(W): Suff(W,l)$ – суффикс $W_i \} \}$

Выше мы ввели функцию $p(i)$, равную максимальной длине суффикса строки W_i, являющегося префиксом W. По определению имеем, что $W_{p(strlen(W))}$ является суффиксом W, поэтому $W_{p(strlen(W))} \sim Suff(W,l)$. Из последнего вытекает, что
$g(l)^3 p(strlen(W))$
Т.о. получаем

$g(l) = Max\{ Max\{i<strlen(W): W_i$ – суффикс $Suff(W,l)\}, Max\{i<strlen(W): Suff(W,l)$ – суффикс $W_i \} \}$

Более того, $Max\{i<strlen(W): W_i$ – суффикс $Suff(W,l)\}$ не может превзойти $W_{p(strlen(W))}$, т.к. если бы это произошло, то мы получили бы суффикс $Suff(W,l)$ (а следовательно и суффикс W), являющийся префиксом W, длиной больше максимально возможной длины суффикса W, являющегося префиксом W. Т.о. получаем

$g(l) = Max\{ p(strlen(W)), Max\{ p(strlen(W)) £ i<strlen(W): Suff(W,l)$ – суффикс $W_i \} \}$

Легко увидеть, что поиск

$w(l)=Max\{ p(strlen(W)) £ i<strlen(W): Suff(W,l)$ – суффикс $W_i \}$

сводится к поиску самого правого участка строки W, совпадающего с $Suff(W,l)$ (естественно, рассматриваются участки левее самого $Suff(W,l)$).

Пример:

$l=2; W="abacabacab";$ *//выделен Suff(W,l) и его правое вхождение в W*

Отметим, что такого участка может не существовать. Если рассмотреть строку W', представляющую собой перевернутую строку W, то задачу можно свести к поиску самого левого вхождения строки W'_l в строку W' (правее начальной позиции):

$w(l)=Max\{ p(strlen(W)) £ i<strlen(W): Suff(W,l)$ – суффикс $W_i \}=$
$=strlen(W) - Min\{ i>l: W'_l$ – суффикс $W_i \}+l$

Пример:

$l=2; W'="bacabacaba";$ *//выделен* W'_l *и его левое вхождение в W*

Рассмотрим начало строки W', завершающееся найденным левым вхождением W'_l в строку W (в примере это – ***"bacaba"***). Более формально: рассмотрим W'_l, где $I=argMin\{ i>l: W'_l$ – суффикс $W_i \}$.
Легко увидеть: $l=p'(I)$, где p' – префикс-функция W'. Действительно, если бы нашелся больший суффикс W'_I, являющийся одновременно префиксом W', то, соответственно, нашлось бы и более левое вхождение подстроки W'_l в строку W (т.к. начало более длинного суффикса должно совпадать с W'_l).
С другой стороны равенство $l==p'(I)$ влечет за собой тот факт, что W'_l является суффиксом W'_I.

Т.о. имеем

Min{ i>l: W'$_l$ – суффикс W_i }= Min{ i>l: l==p'(i)}

Тогда получаем

w***(l)= strlen(W) +l - Min{ i>l: l==p'(i)}***

Последнее равенство дает алгоритм вычисления w***(t)***: следует перебрать все значения ***i*** в порядке убывания и для каждого из них выполнить присвоение

w***[p'(i)]= strlen(W) + p'(i) – i*** *если* ***i>p'(i)***

В конце концов, получаем

g***[l] = Max{ p(strlen(W)),*** w***[l]}***

В двух последних формулах мы реализовалиw и g как массивы.

В прилагаемой программе реализованы функции создания массивов ***m, p*** *и* g. *Реализованы функции поиска, использующие только эвристику стоп-символа, только эвристику безопасного суффикса и, наконец, функция поиска по обоим эвристикам.*

Форматы BMP и RLE

Формат ***BMP*** исторически является основным форматом представления изображений в ОС ***Microsoft Windows****. Постараемся дать, по возможности, полное описание формата.

Обычно формат не использует алгоритмов сжатия. Поэтому не представляет особого труда написать собственную программу, позволяющую считывать изображения с диска (т.е. конвертировать их из ***BMP*** формата в формат, удобный для непосредственного использования) и записывать их на диск.

Данный формат дает возможность представлять изображения с палитрой или без нее. При наличии палитры значение цвета в каждом пикселе задается номером цвета. Сам цвет определяется по его номеру в палитре, представляющей собой массив

unsigned char pal[256][4];

Каждая строка в массиве палитры задает один цвет с помощью указания его ***RGB*** составляющих, соответственно, в ячейках с номерами ***0, 1, 2***. Количество строк зависит от количества используемых цветов и, обычно, не превосходит 256 (что соответствует изображению, в котором на один пиксел отводится 8 бит).

Данные можно представлять в формате ***true color***, когда каждый пиксел задается 4-мя байтами. Из них используется 3 байта для размещения ***RGB*** компонент цвета (по одному байту на каждую компоненту).

Возможно большое количество нестандартных вариаций данного формата. Например, отсутствие палитры в изображениях с толщиной 8 бит на пиксел может обозначать, что кодируется серое изображение, в каждом байте которого хранится яркость пиксела. Некоторые программы понимают форматы ***BMP*** в которых отводится 3 байта на пиксел (формат ***true color***) или даже 2 байта на пиксел (в пикселе хранится только яркость, т.е. кодируется серое изображение).

Файл состоит из следующих разделов:

- Заголовка
- Возможно – палитры
- Собственно данных

Заголовок файла представляется следующей структурой

```
struct BMPHEAD
 {
  unsigned short int Signature ;          // Must be 0x4d42 == "BM"          //0
       unsigned  long  FileLength  ;                                  // в байтах
                                        //2
      unsigned  long  Zero  ;                                         // Must be 0
                                            //6
  unsigned long Ptr ;                     // смещение к области данных       //10
  unsigned long Version ;// длина оставшейся части заголовка=0x28       //14
  unsigned long Width ;       // ширина изображения в пикселах            //18
  unsigned long Height ;      // высота изображения в пикселах            //22
  unsigned short int   Planes ;           // к-во битовых
плоскостей                  //26
  unsigned short int   BitsPerPixel ;     // к-во бит на папиксел     //28
  unsigned long Compression ;             // сжатие: 0 или 1 или 2           //30
  unsigned long SizeImage ;               // размер блока данных в байтах    //34
  unsigned long XPelsPerMeter ;           // в ширину: пикселов на метр      //38
  unsigned long YPelsPerMeter ;           // в высчоту: пикселов на метр     //42
  unsigned long ClrUsed ;     // к-во цветов в палитре               //46
  unsigned long ClrImportant ; // к-во используемых цветов в палитре //50
 } ;
```

Предполагается, что ***sizeof(unsigned long)==4, sizeof(unsigned short int)==2***.

Соответствующие поля в файле с данным форматом непрерывно располагаются в указанном порядке.

Палитра (если она есть) следует сразу за заголовком. Данные начинаются с байта номер *Ptr*, начиная от начала файла.

BMP без сжатия.

Поле *Compression* определяет способ сжатия данных. Обычно значение этого поля=0, что соответствует отсутствию сжатия. При этом данные записываются по битам подряд. ***BMP*** формат со сжатием часто называется ***RLE*** форматом. Длина каждой строки округляется в большую сторону до кратности 32 битам (4 байта). Т.о., например, при отсутствии сжатия если *Width=3*, то каждая строка на диске будет занимать

(Width BitsPerPixel + 31)/8=4* байт.

Предполагается, что байты располагаются в порядке их нумерации, старший бит слева. Т.о., если *BitsPerPixel* =1, то самый первый пиксел ляжет в старший бит самого первого байта данных.

Для хранения всего изображения структуру ***struct BMPHEAD*** следует дополнить массивом палитры и массивом самих данных. Если предположить, что мы будем иметь дело с изображениями не более 8 бит на пиксел, то для данных можно завести, например, массив ***unsigned char **v*** . Пиксел с координатами ***(i,j)*** можно хранить в переменной ***v[i][j]***.

Итак, все изображение можно хранить в структуре

```
struct CBMP
 {
  unsigned short int Signature ;          // Must be 0x4d42 == "BM"          //0
       unsigned long FileLength ;                                          // в байтах
                                  //2
      unsigned long Zero ;                                                 // Must be 0
                                        //6
  unsigned long Ptr ;                     // смещение к области данных      //10
  unsigned long Version ;// длина оставшейся части заголовка=0x28          //14
  unsigned long Width ;       // ширина изображения в пикселах             //18
  unsigned long Height ;      // высота изображения в пикселах             //22
  unsigned short int  Planes ;            // к-во битовых
плоскостей                    //26
  unsigned short int  BitsPerPixel ;      // к-во бит на папиксел      //28
  unsigned long Compression ;             // сжатие: 0 или 1 или 2          //30
  unsigned long SizeImage ;               // размер блока данных в байтах   //34
  unsigned long XPelsPerMeter ;           // в ширину: пикселов на метр     //38
  unsigned long YPelsPerMeter ;           // в высчоту: пикселов на метр    //42
  unsigned long ClrUsed ;     // к-во цветов в палитре                 //46
  unsigned long ClrImportant ; // к-во используемых цветов в палитре //50
  unsigned char pal[256][4];
  unsigned char **v;
 } ;
```

Отвести память можно, например, следующим образом:

```
struct CBMP pic;  int i;
…
pic.v=(unsigned char**)malloc(pic.Height*sizeof(char*));
for(i=0;i<pic.Height;i++)pic.v[i]= (unsigned char*)malloc(pic.Width);
```

Однако следующий способ гораздо более эффективен:

```
struct CBMP pic;  int i;
…
pic.v=(unsigned char**)malloc(pic.Height*sizeof(char*)+pic.Height*pic.Width);
pic.v[0]= (unsigned char**)(pic.v+pic.Height);
for(i=1;i<pic.Height;i++)pic.v[i]=pic.v[i-1]+pic.Width;
```

Преимуществом такого способа отведения памяти является уменьшение накладных расходов и упрощение процедуры освобождения памяти. Для освобождения отведенной памяти надо вызвать всего один оператор:

```
free(pic.v);
```

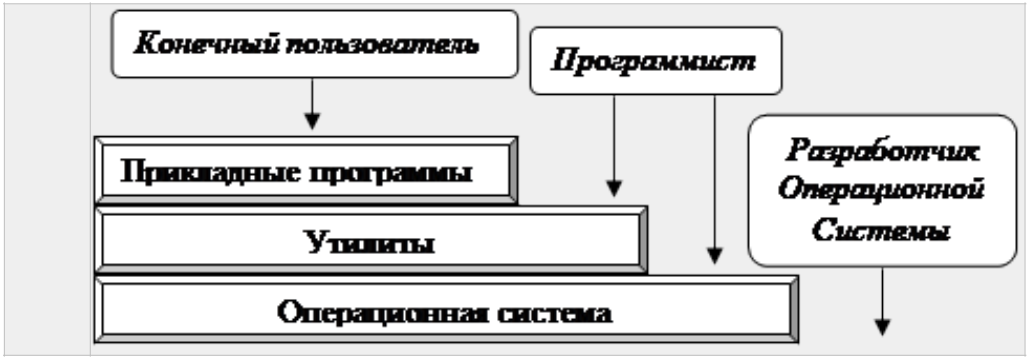
Лекция 17

Операционные системы

Операционная Система - это программа, контролирующая работу [*и позволяющая запускать?*] прикладных и системных приложений и исполняющая роль интерфейса между пользователем, другими приложениями и аппаратным обеспечением компьютера.

Данное определение весьма условно. На данный момент сложилось некоторое общее представление о том, что должна делать ОС. Это представление зафиксировано в стандарте POSIX (*Portable OS Interface based on uniX*) и при разработке новых ОС всегда происходит оглядка на данный стандарт. Исходя из самого понятия POSIX следует, что не все ОС жестко ориентируются на это определение. Например, ОС Windows* явно претендуют на выполнение функций запуска приложений, в то время, как ОС UNIX последовательно отделяет от себя эту функцию.

Определить место ОС во всей вычислительной системе (по идеологии POSIX!!!) можно из следующей картинки, определяющей уровни вычислительной системы и различные точки зрения на нее:



По современным представлениям, ОС должна уметь делать следующее:

- Обеспечивать загрузку пользовательских программ в оперативную память и их исполнение.
- Обеспечивать работу с устройствами долговременной памяти, такими как магнитные диски, ленты, оптические диски и т.д. Как правило, ОС управляет свободным пространством на этих носителях и структурирует пользовательские данные.
- Предоставлять более или менее стандартный доступ к различным устройствам ввода/вывода, таким как терминалы, модемы, печатающие устройства.
- Предоставлять некоторый пользовательский интерфейс. Слово *некоторый* здесь сказано не случайно - часть систем ограничивается командной строкой, в то время как другие на 90% состоят из средств интерфейса пользователя.

Существуют ОС, функции которых этим и исчерпываются. Одна из хорошо известных систем такого типа - дисковая операционная система *MS DOS*.
Более развитые ОС предоставляют также следующие возможности:

- Параллельное (точнее, псевдопараллельное, если машина имеет только один процессор) исполнение нескольких задач.
- Распределение ресурсов компьютера между задачами.
- Организация взаимодействия задач друг с другом.
- Взаимодействие пользовательских программ с нестандартными внешними устройствами.
- Организация межмашинного взаимодействия и разделения ресурсов.
- Защита системных ресурсов, данных и программ пользователя, исполняющихся процессов и самой себя от ошибочных и зловредных действий пользователей и их программ.

В силу вышесказанного можно провести грубую классификацию операционных систем:
Дисковые операционные системы. *ДОС*. Это системы, берущие на себя только первые четыре функции. Классический пример – MS DOS. Система по определению может работать только с одной задачей (если не рассматривать недокументированные возможности). После загрузки задачи полный контроль над системой передается задаче и ОС, практически, никак не может помешать задаче сделать с системой что угодно – например, порушить саму ОС. Если пользовательская программа была абсолютно корректной, то после ее завершения управление системой передается опять ОС и она продолжает интерфейс с пользователем.
Безусловно, не стоит недооценивать ОС MS DOS. В ней можно найти много красивых идей, украденных у UNIX, например, в ней существуют подобия файлов устройств (например, *PRN, NUL, CON*), понятие стандартных потоков ввода вывода, перенаправления в стандартные потоки ввода/вывода/вывода сообщений об ошибках (в точности так же, как и в UNIX). Существует понятие конвейера. Хотя задачи в конвейере (в силу однозадачности системы) запускаются в порядке их написания (т.е. вторая задача запускается после окончания работы первой задачи), а данные передаются через промежуточный буфер.
Обычные операционные системы. ОС. К этому классу относятся такие широко распространенные системы, как *VAX/VMS*, системы семейства*Unix, OS/2*, Windows*. Здесь под ОС подразумеваются системы ``общего назначения", т.е. рассчитанные на интерактивную работу одного или нескольких пользователей в режиме разделения времени. Как правило, в таких системах уделяется большое внимание защите самой системы, программного обеспечения и пользовательских данных от ошибочных и злонамеренных программ и пользователей. Обычно такие системы используют встроенные в архитектуру процессора средства защиты и виртуализации памяти.

Эволюция операционных систем

Последовательная обработка данных

Идеи создания вычислительных машин давно витали в воздухе.
В дневниках *Леонардо да Винчи* можно найти эскизы вычислительной машины, на основе зубчатых элементов.
В 1642г. *Блез Паскаль* создал реально работающее счетное устройство на основе зубчатых колес, которое было в состоянии складывать и вычитать десятичные числа.

1830-1846 гг. *Чарльз Беббидж* разрабатывает проект Аналитической машины - механической универсальной цифровой вычислительной машины с программным управлением (!). Машина состоит из пяти устройств - арифметического устройства (АУ), запоминающего устройства (ЗУ), устройства управления (УУ), ввода и вывода (все как в первых ЭВМ, появившихся 100 лет спустя). АУ строилось на основе зубчатых колес, на них же предлагалось реализовать ЗУ (на тысячу 50-разрядных чисел - итого 50 тыс. зубчатых колес). Для ввода программы и данных использовались перфокарты. Предполагаемая скорость вычислений: сложение и вычитание за 1 сек, умножение и деление - за 1 мин. Помимо арифметических операций, имелась команда условного перехода. Лекции Беббиджа, опубликованные на итальянском языке, были переведены на английский *Адой Августой Лавлейс* (дочерью Джорджа Байрона). Она же начала писать программы для этой машины, поэтому ее называют первым программистом (в ее честь назван язык Ada).

Первые работающие компьютеры появились в конце 40-х – начале 50-х гг. Фактически, ОС на них не было. Управление ЭВМ осуществлялось с тумблеров на пульте управления. Существовали некоторые устройства ввода данных (перфоленты, перфокарты), с которых можно было загрузить программу в память машины. В крайнем случае, задать значения ячеек памяти можно было непосредственно с тумблеров. О наличии каких-либо ошибок сообщали соответствующие сигнальные лампы. С их помощью можно было проанализировать состояние памяти и регистров ЭВМ. Результаты работы программы можно было распечатать на принтере.

В 1941 на основе разработок Чарльза Беббиджа была со субподрядному договору с IBM была создана электронно-механическая машина MARK-I (4.5 тонны, 765 тысяч деталей, оперировала с 75 числами из 23 десятичных знаков), которая и считается первым реальным созданным компьютером. Машина не имела операций условного перехода, циклы реализовывались с помощью склеивания перфоленты в петлю.

Простые пакетные системы

Высокая стоимость процессорного времени привела к необходимости более эффективного его использования. В результате появилась концепция *пакетной операционной системы*. Первые пакетные ОС были разработаны в середине 50-х гг. в компании General Motors для машин IBM 701. В основе пакетных ОС лежит программа, называемая *монитор*. Основная часть монитора находится постоянно в оперативной памяти. Пользователь не имеет непосредственного доступа к машине. Вместо этого он общается с оператором. Оператор загружает последовательность перфокарт или перфоленту сразу нескольких заданий, после чего монитор загружает очередное задание, при необходимости производит с ним некоторые действия и отсылает на счет. После завершения счета управление возвращается монитору. Он сразу же закачивает следующее задание, и система продолжает работу.

Многозадачные пакетные системы

Многозадачные пакетные системы являются логичным продолжением развития простых пакетных систем. При обработке одного задания могут возникать паузы, связанные, например, с операциями ввода/вывода. В простых пакетных системах процессор вынужден в таких ситуациях простаивать. Если же мы позволим нескольким задачам выполнять ``параллельно'', последовательно выдавая каждой задаче свой квант времени, то мы получим возможность при приостановке одного процесса заставлять процессор обслуживать другой процесс. Это подразумевает наличие прерываний для осуществления операций ввода/вывода и наличие механизма управления памятью, т.к. все выполняемые программы обязаны одновременно находиться в физической памяти.

Системы, работающие в режиме разделения времени

Эти системы дают возможность нескольким пользователям работать в системе параллельно в режиме on-line. Каждому процессу последовательно выделяется свой квант времени для обслуживания процессором (величина кванта обычно измеряется в миллисекундах). Отклик системы на действия пользователя обычно наступает в течение нескольких секунд.

Считается, что первой операционной системой реального времени является ОС CTSS (Compatible Time-Sharing System). Она работала с оперативной памятью, состоящей из 32К 36-битных слов. Первые 5000 слов занимал монитор, а строго с адреса 5000 грузилась программа пользователя. С интервалом 0.2 секунды на место этой программы грузилась программа другого пользователя, а предыдущая спасалась на жестки диск. В качестве оптимизации этого процесса было предусмотрена выгрузка на диск не всей программы предыдущего пользователя, а только той части, которая затиралась новой программой, поэтому при восстановлении исходной программы с диска существовала в некоторых случаях возможность считывания не всей программы, а только ее затертой части. Распределение машинного времени происходило с помощью административных ресурсов: каждый пользователь записывался на определенное время. Если этого времени не хватало, то пользователь был вынужден прерывать работу.

В отведенное время пользователь должен был ввести в машину свою программу, откомпилировать ее (как правило, обычные программы были написаны на языке высокого уровня), скомпоновать программу с библиотечными функциями и запустить на счет. Возникновение каких-либо ошибок приводило к необходимости начинать все заново.

Системы реального времени. Многопоточность. Потоки и процессы

Эти системы дают возможность нескольким пользователям работать в системе параллельно в режиме on-line, причем особые требования задаются для времени отклика системы на действия пользователей. Они должны происходить в режиме реального времени, т.е., практически, без задержек. В реальности, требование

быстрого отклика системы заменяется требованием гарантированного отклика системы на любое событие в течение заданного интервала времени (действительно, если этот интервал зафиксирован, то, скорее всего, при развитии аппаратной части ЭВМ данный интервал будет уменьшаться, и в некоторой момент он, в любом случае, станет приемлемым).
Такому требованию заведомо не удовлетворяла ОС Windows 3.11. Windows 3.11 является *кооперативной система*, т.е. каждое приложение обязано само периодически проверять очередь сообщений. Только при каждой проверке система дает возможность другим приложениям передать управление процессором. Т.о., если приложение где-то зациклилось и не производит проверку очереди сообщений, то это подвешивает всю систему. Отметим, что это же свойство сохранилось для 16-битных приложений в Windows95, хотя для обычных 32-битных приложений здесь были нормальные механизмы управления процессами. Последующие ОС Windows* уже полноценно можно считать системами реального времени.
К системам реального времени применимо требование *многопоточности*. С общим понятием `задачи' обычно ассоциируются два понятия: *поток* (= нить = thread) и *процесс*. Обычно, под потоком понимают диспетчирезуемую единицу работы, с которой связывается контекст процессора (например, регистр флагов), собственный стек (который обычно задается регистром стека). Обычно потоки внутри одного процесса имеют одно адресное пространство. Процесс является объединением некоторого количества потоков вместе с набором ресурсов, связанных с потоками. В отличие от потоков, различные процессы, вообще говоря, должны иметь различные адресные пространства. Одна задача, вообще говоря, может состоять из нескольких процессов, в каждом из которых может быть несколько нитей.
Вообще говоря, существуют и другие схемы соотношений потоков и процессов. Например, в ОС Clouds один поток может переходить от одного процесса к другому. Это очень удобно для реализации распределенных вычислений: поток ассоциируется с некоторой выполняемой последовательностью команд. При этом, можно переходить от одного адресного пространства к другому. Более того, эти адресные пространства могут относиться даже к разным ЭВМ.
Обычно к операционным системам реального времени предъявляют следующие требования:
1. ОС должна быть *многонитевой и прерываемой*.
2. Должно существовать понятие *приоритета нити*. Нить с высшим приоритетом должна иметь возможность прервать выполнение нити с низшим приоритетом.
3. ОС должна обеспечивать предсказуемые *механизмы синхронизации задач* (блокировка и посылка сигналов).
4. Должна существовать система *наследования приоритетов*. Это понятие связано с понятием *инверсии приоритетов*. Пусть запущено три процесса. Процесс с низшим из трех приоритетов может захватить некоторый ресурс А. Пусть сразу после этого запускается второй процесс с бОльшим приоритетом, который вытесняет первый процесс. Далее запускается третий процесс с самым большИм приоритетом и он тоже требует ресурс А. В результате получается блокировка: третий процесс не может продолжать работу, пока первый процесс не освободит ресурс А, а первый процесс вынужден ждать завершения работы второго процесса. Т.о. на работу процесса с высшим приоритетом оказывает влияние процесс с низшим приоритетом. Эта ситуация называется *инверсией приоритетов*. Отметим, что эта ситуация весьма часто встречается в различных версиях Windows (в ранних версиях чаще, в более новых - реже).
Для выхода из этой ситуации ОС должна уметь повышать приоритет процесса с низшим приоритетом до приоритета процесса с высшим приоритетом, если второй процесс как-то обращается к первому процессу. Отсюда вытекают понятия *статического приоритета, динамического приоритета, покупаемого приоритета.*
5. Поведение ОС должно быть известно. Например, должны быть известны времена выполнения всех системных вызовов, времена задержки от поступления прерывания до выполнения соответствующей процедуры и т.д.

Режимы адресации оперативной памяти в Intel-совместимых компьютерах

Реальный режим. Само название появилось с появлением процессоров 80286 (1982г). До этого (процессоры 8086) не было даже такого названия. Использует *сегментную адресацию памяти*. Адрес задается двумя 16-битными регистрами:
CS = адрес, смещенного на 4 бита вправо, 64К-сегмента памяти,
IP = смещение адреса внутри сегмента.
Т.о. физическое значение адреса вычислялось по формуле CS*16+IP. Реальный размер адресуемой памяти определялся особенностями процессоров. В процессорах 8086 размер адресуемого пространства был 1М (20 бит адресации), в процессорах 80286 размер адресуемого пространства вырос до 16М (24 бита адресации). При данном способе адресации каждая программа имеет доступ ко всей оперативной памяти компьютера.
Защищенный режим. Появился вместе с процессорами 80386 (1985г), в которых была реализована *страничная организация* памяти. Для процессоров 80386 данный режим позволял получать доступ ко всем 4Г памяти (32бита). Основная идея режима: память разбивается на страницы (от 4К до 1Г в архитектуре x86-64). Формируется таблица страниц, в которой по номеру страницы можно получить смещение к данной странице в физической памяти. Если физической памяти не хватает, то некоторая страница может быть вытеснена на жесткий диск и данная страница будет помечена как несуществующая. Далее при обращении к данной странице происходит прерывание, при обработке которого на ее место может быть загружен образ

данного куска памяти с жесткого диска. Данный процесс называется *подкачкой данных с жесткого диска.*
Адрес также задается двумя 16-битными (на процессорах 80286) регистрами:
CS = селектор страницы памяти,
IP = смещение адреса внутри страницы.
Линейная адресация. Данный режим используется во всех современных ОС (появился в процессорах 80386). В данном режиме чисто внешне происходит работа с обычным линейным адресом. На самом деле, данный режим является просто существенным усовершенствованием защищенного режима. Усовершенствование заключается в создании каталога страниц (=массива таблиц страниц; страницы имеют фиксированный размер), в котором лежат таблицы страниц (в защищенном режиме таблица страниц одна). Создание каталога страниц позволяет для каждого процесса использовать свою таблицу страниц, что делает адресные пространства процессов непересекающимися. В архитектуре x86-32 под номер таблицы страниц отводятся старшие 10 бит, под номер страницы в таблице страниц отводятся следующие 10 бит, оставшиеся 12 бит отводятся под смещение внутри страницы (т.о. одна страницы занимает 4К). Размеры страниц в других архитектурах могут отличаться. Например, в архитектуре x86-64 размеры страниц могут быть 4К, 2М, 1Г. Более того, в разных архитектурах количество уровней адресации может быть больше двух. Например, в 64-битной архитектуре их 4.
В архитектуре x86-32 каждая запись в каталоге страниц и в таблице страниц занимает 4 байта. Старшие 20 бит в записи задают номер таблицы страниц/ страницы, а далее следуют различные флаги: 0)наличие страницы в памяти, 1)разрешение на запись, 2)разрешение обычному пользователю обращаться к данной странице и т.д. Заметим, что старшие 20 бит номера страницы задают физический адрес страницы (если заполнить младшие биты нулями, учитывая, что размер страницы = 4К=2^{12}б)
Здесь следует упомянуть об механизме *copy-on-write*, существенно основанном на данном подходе. Данная техника используется, например, в UNIX при реализации функции *fork()*, которая вызывает системный вызов, создающий копию текущего процесса (единственное отличие: в родительском процессе функция возвращает ID дочернего процесса, а в порожденном процессе функция возвращает 0). Все процессы в сессии UNIX создаются как копии других процессов, в основе генеалогии которых лежит один процесс, с которого начинается работа системы. Механизм *copy-on-write* заключается в том, что перед созданием копии данного процесса запрещается запись в страницы памяти родительского процесса. В порожденном процессе создаются копии таблиц страниц родительского процесса, т.е. оба процесса ссылаются на одну и ту же память, что дает возможность спокойно считывать эту память в обоих процессах. Но при попытке записи генерируется исключение, прерывающее программу, в котором, собственно, и разделяется затронутая странница памяти в двух рассматриваемых процессах. После обработки исключения нормальная работа программы продолжается. Механизм *copy-on-write* имеет широкое использование. Например, он используется в реализации строк в библиотеке MFC, в различных файловых системах и т.д.

Процесс выполнения нити

Рассмотрим данный вопрос на уровне ассемблера. Мы будем оперировать понятиями *оперативная память* (память, обращение к которой идет по *адресу*) и *регистры* (именованные ячейки памяти, физически расположенные на самом процессоре). Каждая архитектура обладает своей спецификой при рассмотрении вопросов на этом уровне. Для определенности будем рассматривать архитектуру *x86* (*x86-32* или *x86-64*). Отметим, что ныне используемая архитектура на ПК была изначально предложена фирмой AMD и носит название AMD64, после чего Intel была вынуждена создать свою, совместимую с AMD64, архитектуру EM64T. На данный момент архитектура, поддерживаемая Intel-процессорами *Core-2,...,Core-9,* официально называется *Intel 64*. До появления 64-битных систем на уровне ПК господствовала архитектура IA32, обеспечивающая доступ до 4Г оперативной памяти (при этом приложения не могли занимать более 2Г памяти).
Итак, для объяснений нам потребуются знание о существовании в архитектуре *x86-64* регистра номера следующей выполняемой команды *RIP* и регистра стека *RSP*. Выполнение нити сводится к загрузке команды, адрес которой лежит в регистре RIP в процессор, увеличения значения регистра RIP на величину команды и выполнения загруженной в процессор команды.

Лекция 18

Вызов функций. Механизмы передачи параметров в функции. Функции с переменным количеством параметров

На уровне ассемблера (т.е. с точки зрения, собственно, архитектуры) вызов функции осуществляется с помощью инструкции *CALL* с адресом перехода в виде параметра, а возвращение из функции осуществляется инструкцией *RET*. Вызов *CALL* записывает адрес перехода в регистр следующей выполняемой команды, а адрес, следующий за данной инструкцией, записывает в вершину стека (с помощью регистра, указывающего на вершину стека). Инструкция *RET* извлекает из стека записанный там ранее адрес и помещает его в регистр с адресом следующей команды. Т.о. происходит возврат из функции. Этот механизм вообще не предполагает наличия понятия *параметров функции*.

В языке C (и во многих других языках) фактически, существует два стандарта передачи параметров в функции (в виде надстройки над вызовом функции на уровне ассемблера). Какой из них используется – личное дело компилятора.
В 32-битных системах стандартно используется *механизм передачи параметров через стек*. Т.е. все параметры функций просто записываются в стек. При этом, в языке C/C++ обычно действуют некоторые соглашения, которые, например в 32-битных приложениях Microsoft, требуют, чтобы переменные типа char и short int передавались в функцию в виде переменных типа int, bли, в gcc переменные типа float передаются преобразованными к типу double. При передаче переменных по ссылке реально передается адрес переменной. Для лучшего понимания ситуации полезно позапускать у себя следующую простую программу в разных режимах (отладка/рабочий режим) на разных платформах и в разных компиляторах:

```
#include <stdio.h>
#include <stdlib.h>
int f(int a, char b, long c, short int d, float e, double g,int h,int &i,int z)
{
 printf("%d %d %d %d %d %d %d %d\n",(int)(((char*)&b)-((char*)&a)),
                        (int)(((char*)&c)-((char*)&b)),
                        (int)(((char*)&d)-((char*)&c)),
                        (int)(((char*)&e)-((char*)&d)),
                        (int)(((char*)&g)-((char*)&e)),
                        (int)(((char*)&h)-((char*)&g)),
                        (int)(((char*)&i)-((char*)&h)),
                        (int)(((char*)&z)-((char*)&i))
                        );
 return 0;
}
int f1(int a, char b, long c, short int d, float e, double g,int h,int &i,int z)
{
 printf("%d %d %d %d %g %lg\n",
  a,
  *(((char*)&a)+4),
  *(((char*)&a)+4+4),
  *(((char*)&a)+4+4+4),
  *(double*)(((char*)&a)+4+4+4+4),
  *(double*)(((char*)&a)+4+4+4+4+8)
  );
 return 0;
}
```



```
int f2(int a,...)
{
 printf("%d %d %d %d %g %lg\n",
  a,
  *(((char*)&a)+4),
  *(((char*)&a)+4+4),
  *(((char*)&a)+4+4+4),
  *(double*)(((char*)&a)+4+4+4+4),
  *(double*)(((char*)&a)+4+4+4+4+8)
  );
 return 0;
}
int main(void)
{int a=1; char b=2; long c=3; short int d=4; float e=5; double g=6;int z=7;
printf("%d\n",sizeof(int*));
 f(a,b,c,d,e,g,z,z,z);
 f1(a,b,c,d,e,g,z,z,z);
 f2(a,b,c,d,e,g,z,z,z);
return 0;
}
```

Здесь функция f2() имеет переменное количество параметров, и мы пытаемся достать параметры функции, исходя из предположения о том, что все параметры лежат в стеке с использованием соответствующих преобразований переменных при передаче их в стек. Ту же самую попытку мы делаем для функции f1() (с теми же предположениями, что и для функции f2()). Из вывода данной программы можно сделать так много выводов, что описывать их здесь просто нерационально.
В 64-битных системах (архитектура AMD64) стандартно первые параметры передаются через регистры (четыре первых целых и четыре первых вещественных), если же их много, то опять же используется стек. Исходя из вывода предыдущей программы, можно сделать вывод, что данный принцип может использоваться и для 32-битных систем.
Для работы с функциями с переменным количеством параметров используются директивы *va_list, va_start, va_arg, va_end*. Например, в нашем случае для функции f2() в случае компилятора Microsoft аргументы корректно извлекать с помощью следующей программы:

```
int f3(int a,...)
{char b; long c; short int d; float e; double g;int h;int i;int z;
 va_list args;
```

```
 va_start(args,a);
 b=va_arg(args,char);
 c=va_arg(args,long);
 d=va_arg(args,short);
 e=va_arg(args,double); //!!!!!!!!
 g=va_arg(args,double);
 h=va_arg(args,int);
 i=va_arg(args,int);
 z=va_arg(args,int);
 printf("%d %d %d %d %g %lg %d %d %d\n",a,b,c,d,e,g,h,i,z);
 return 0;
}
int main(void)
{int a=1; char b=2; long c=3; short int d=4; float e=5; double g=6;int z=7;
printf("%d\n",sizeof(int*));
f3(a,b,c,d,e,g,z,z,z);
return 0;
}
```

А для случая компилятора gcc аргументы корректно извлекать с помощью следующей программы:

```
int f3(int a,...)
{char b; long c; short int d; float e; double g;int h;int i;int z;
 va_list args;
 va_start(args,a);
 b=va_arg(args,int); //!!!!!!!!
 c=va_arg(args,long);
 d=va_arg(args,int); //!!!!!!!!
 e=va_arg(args,double); //!!!!!!!!
 g=va_arg(args,double);
 h=va_arg(args,int);
 i=va_arg(args,int);
 z=va_arg(args,int);
 printf("%d %d %d %d %g %lg %d %d %d\n",a,b,c,d,e,g,h,i,z);
 return 0;
}
int main(void)
{int a=1; char b=2; long c=3; short int d=4; float e=5; double g=6;int z=7;
printf("%d\n",sizeof(int*));
f3(a,b,c,d,e,g,z,z,z);
return 0;
}
```

Алгоритмы динамического выделения памяти

Существуют различные алгоритмы выделения/освобождения памяти.

Использование стека задачи

Существуют ситуации, когда есть гарантия того, что память освобождаться в строго обратном порядке ее отведению. Например, эта ситуация реализуется при отведении памяти под локальные автоматические переменные в языке С. Действительно, для этих переменных память отводится при входе в некоторый блок ***A***, а освобождается при выходе из блока ***A***. При этом, память, отведенная под все переменные, находящиеся во внутренних блоках блока ***A,*** отводится *после* отведения памяти переменных блока ***A,*** а освобождается *до* освобождения переменных блока ***A.***

В этом случае переменные могут размещаться в стеке данной задачи. Компилятор видит определения всех локальных переменных в данном ядре, поэтому при появлении в блоке каждой локальной переменной компилятор ассоциирует ее с текущей позицией стека и перемещает указатель стека на размер этой переменной. При обработке куска кода внутри блока, компилятор преобразует все ссылки на локальные переменные в ссылки на соответствующие адреса внутри стека (т.е. у него есть таблица адресов локальных переменных для каждого блока). При выходе из блока, в соответствии с таблицей локальных переменных блока, компилятор отодвигает указатель на стек назад на размер локальных переменных в блоке.

Все остальные алгоритмы базируются на использовании *общей кучи* (*heap*) – линейного куска памяти с произвольным обращением по адресу.

Списки блоков фиксированного размера

Рассмотрим простейшую ситуацию. Пусть нам требуется выделять блоки памяти фиксированного размера ***L***. Тогда напрашивается элементарное решение: объединить все свободные блоки в список свободного места. Элементом списка будет один блок. В каждый блок нам придется поместить указатели на предыдущий и следующий элементы списка.

Поиск свободного места (т.е. поиск свободного блока длиной ***L***) становится элементарным: мы просто берем первый попавшийся блок в списке и исключаем

его из списка свободного места. Освобождение памяти не сложнее: мы добавляем блок к списку свободного места.
Подобный механизм отведения памяти часто используется, например, при работе с жесткими дисками. Неиспользуемая дисковая память во многих файловых системах объединяется в список блоков. Также подобный механизм отведения памяти используется в реальном отведении памяти в языке С для выделения памяти кусками небольшого размера. Для кусков небольшого размера создаются списки под отдельные размеры кусков выделяемой памяти.

Алгоритм близнецов (для блоков размером 2^k)
Если существует необходимость иметь блоки нескольких размеров, то эту задачу можно свести к предыдущей: мы просто создадим отдельный список свободных блоков для каждого размера. На этом основан *алгоритм близнецов*.
Алгоритм предполагает, что будут выделяться блоки размером только 2^k. В каждом таком блоке следует выделить служебную область. По сути, требуется всего один бит, указывающий: занят ли данный блок, или нет. В реальных ЭВМ обычно адресация возможна только к большим кускам памяти (например, к одному байту), поэтому и служебная область будет больше одного бита (как минимум, один байт). На самом деле (см. ниже), в блоке еще должно быть достаточно места для хранения трех указателей (эти данные будут использоваться только в случае, когда данный блок относится к свободной памяти).
Для каждого размера блока 2^k создается двунаправленный список $List_k$ свободных блоков размером 2^k. Т.е. для каждого **свободного** блока в блоке вводятся три дополнительных служебных поля: указатели на предыдущий и следующий блоки в $List_k$ и переменная со значением k, задающим размер блока. Предполагается, что размер всей отведенной памяти равен $M=2^N$ и рассматриваются блоки длиной не менее $m=2^n$. Из сказанного выше для 64-битной системы получаем $m=5$.
В начальный момент все списки $List_k$ пусты, кроме $List_N$, в котором располагается всего один элемент.
Алгоритм отведения блока памяти размером 2^k следующий: если список $List_k$ не пуст, то мы просто берем в нем первый элемент, исключаем из списка, помечаем его как занятый и используем для работы (например, возвращаем его адрес пользователю; если пользователь не знает о внутренней структуре алгоритма, то возвращаем адрес рабочей части блока).
Если список пуст, то мы рекурсивно отводим блок памяти размером 2^{k+1}. Если это невозможно, сообщаем, что память отвести нельзя (например, функция отведения памяти возвращает ***NULL***; в глубине рекурсии это может произойти только, если потребуется отвести блок памяти размером 2^{N+1}). Блок размером 2^{k+1} разбиваем на две половины. Эти половины называются *близнецами*. Первую из этих половин добавляем в список свободных блоков размером 2^k, а вторую – помечаем как занятый блок и используем для работы.
Легко увидеть, что алгоритм отведения памяти выполняется за время $T=O(N-n)=O(\log M - \log m)$.

Для освобождения памяти, занятой под блок, нам необходимо знать адрес начала блока (если пользователь получил адрес рабочей части блока, то, зная ее смещение, относительно начала блока, адрес начала блока легко получить). Пусть исходный блок размером 2^N начинается с ячейки памяти с адресом 0.
Освобождение памяти базируется на следующем факте: для каждого блока мы можем легко найти адрес его блока-близнеца. Действительно, блоки близнецы размером 2^k объединены в блок размером 2^{k+1}. Адреса блоков размером 2^{k+1} кратны 2^{k+1} (легко доказать по индукции: для самого большого блока это верно; пусть это верно для блоков размером 2^{k+1}, но блоки размером 2^k располагаются либо в начале блоков размером 2^{k+1} с нулевым смещением, либо со смещением 2^k, что и доказывает утверждение). Получаем, что в двоичном представлении адреса блока размером 2^{k+1} имеется $k+1$ нулей в младших битах. Соответствующие блоки-близнецы размером 2^k из данного блока B размером 2^{k+1} имеют либо тот же самый адрес что и блок B, либо тот же адрес, в котором в k позиции прописана 1 (нумерация с нулевой позиции). Т.о. адрес блока близнеца *Addr2* для блока с адресом *Addr1* вычисляется по формуле:
Addr2=Addr1^(1<<k);
Если требует освободить блок с адресом *Addr1*, то мы ищем его блок-близнец. Если близнец оказывается занят или его не существует (это возможно только для блока размером 2^N), то все, что нам остается, это – добавить блок с адресом *Addr1* к списку свободных блоков соответствующего размера k (размер блока указан в служебной области блока). Если близнец свободен, то мы извлекаем его из списка $List_k$ (для этого нам и нужен двунаправленный список!), объединяем данный блок и его близнец в блок размером 2^{k+1} (все, что для этого надо сделать – в первом из двух блоков увеличить на 1 значение поля, задающего длину блока) и рекурсивно применяем эту же процедуру для освобождения нового блока с размером 2^{k+1}.
Легко увидеть, что асимптотика максимального времени работы алгоритма освобождения памяти совпадает с асимптотикой времени работы алгоритма отведения памяти. Т.о. алгоритм показывает очень хорошую скорость работы, но не позволяет отводить блоки памяти произвольного размера. В определенных ситуациях на последнее можно закрыть глаза. Т.е. если требуется выделить кусок памяти длиной l, то можно отводить кусок памяти длиной $2^{\lceil \log l \rceil}$. Легко увидеть, что при этом размер лишней памяти ($=l - 2^{\lceil \log l \rceil}$) не превосходит размера требуемой памяти.
В ОС ***Linux*** данный алгоритм используется в ядре системы как один из алгоритмов отведения памяти (функция ***kalloc***).

Списки блоков свободной памяти в общем случае

В общем случае мы можем организовать свободную память в виде двухсвязного списка блоков. Кроме указателей на предыдущий и следующий элементы списка, в блоке будем хранить его длину. Если требуется выделить новый кусок памяти длиной ***l***, то нам придется пролистать список до первого устраивающего нас блока. Длину блока можно хранить в начале блока. При этом, она будет храниться и в случае, если блок относится к свободной памяти, и в случае если он относится к выделенной памяти. В последнем случае пользователю можно выдавать указатель на кусок памяти, следующий сразу после длины блока. Указатели на следующий и предыдущий блоки надо хранить только для блока в списке свободного места, поэтому их можно расположить сразу после длины блока.

Далее, если ***l*** меньше длины найденного блока l_b, то мы устанавливаем новую длину блока равной $l_b - l$ и оставшийся кусок длины ***l*** предоставляем для использования. Если ***l*** совпадает с l_b, то мы просто исключаем данный блок из списка свободного места.

Существуют разные стратегии поиска подходящего блока. Две основные это: поиск первого подходящего (*first fit*) и наилучшего (*best fit*). При реализации стратегии *first fit* ищется первый блок, для которого $l_b \geq l$. При реализации стратегии *best fit* ищется блок, для которого $l_b - l$ минимально среди всех блоков, для которых $l_b \geq l$.

Алгоритмы отведения памяти, основанные на стратегии *best fit* , как правило, более экономичны (т.е. позволяют сохранять в течение большего времени большие блоки свободной памяти), но время работы таких алгоритмов прямо пропорционально длине списка. Время работы алгоритмов отведения памяти, основанных на стратегии *first fit*, зависит от распределения блоков длины не менее ***l*** среди всех блоков списка.

В указанной ситуации алгоритм очистки отведенной памяти (= алгоритм добавления блока в список + слияние его с соседними свободными блоками) оказывается также весьма дорогостоящим. Действительно, добавление блока в список выполняется за константу операций. Но, кроме этого, мы должны еще проверить – не свободны ли блоки памяти, стоящие непосредственно слева и справа от данного блока. Если они свободны, то требуется объединить эти блоки с данным блоком.

Т.о. суммарное время работы каждого из указанных алгоритмов ***T=O(L)***, где ***L*** – длина списка свободного места (в худшем случае).

Модифицированные списки блоков свободной памяти в общем случае (алгоритм парных меток)

Если мы готовы пойти на некоторые дополнительные накладные расходы, то скорость работы со списками свободной памяти можно существенно увеличить.

Введем две дополнительные служебные ячейки памяти, расположенные в начале и конце блока (эти ячейки будут использоваться как в свободных, так и в занятых блоках). Договоримся, что для свободных блоков мы будем хранить в этих ячейках размер блока, а для занятых ячеек будем хранить размер блока со знаком `минус'. Эти ячейки называются *парными метками*.

Пространство между парными метками будем называть *рабочей областью блока*. Адрес именно этой части блока будет возвращаться пользователю. Соответственно, когда пользователь захочет освободить память, то именно этот адрес он передаст в программу освобождения памяти.

Как и ранее, в свободных блоках будут также храниться указатели на предыдущий и следующий блоки в списке свободного места.

Пусть ***h*** – размер машинного слова, т.е. размер переменной, в которой может размещаться адрес или длина любого объекта в памяти. Например, сейчас для обычных персональных компьютеров ***h=8*** (в байтах; ***h=sizeof(size_t)***).

Итак, если мы имеем адрес рабочей области отведенной памяти ***Addr***, то по адресу ***Addr-h*** располагается длина данного блока памяти. В терминах языка С длина данного блока памяти это: ****((size_t*)(((char*)Addr)-h))*** или ***((size_t*)Addr)[-1]***. Здесь мы предположили, что длина любого объекта в памяти (а следовательно, и адрес памяти) может размещаться в переменной типа ***size_t***. Можно сразу договориться, что длину блока мы будем измерять в машинных словах, т.е. в пересчете на размер переменной типа ***size_t***. Тогда, размер блока будет также лежать в переменной ***((size_t*)Addr)[((size_t*)Addr)[-1]-2]***. Размер этой переменной должен быть на 2 больше размера рабочей части блока.

Для отведения памяти мы должны использовать ту же процедуру отведения памяти, что и для списков блоков свободной памяти в общем случае.

Естественно, не надо забывать, что каждый блок должен быть погружен в парные метки, т.е. реально для каждого блока требуется на две ячейки памяти больше (на две переменные типа ***size_t***), чем требуется пользователю.

Для очистки памяти мы можем использовать существенно более быструю процедуру. Для этого сначала, мы должны определить – нет ли свободных блоков непосредственно слева и справа от данного блока. Если соседние блоки заняты, то все, что надо сделать, это – добавить текущий блок к списку свободного места. Иначе, текущий блок следует объединить со свободными соседями и добавить получившийся блок к списку свободного места.

Более точно, существует четыре возможных ситуации, задающиеся знаками переменных ***((size_t*)Addr)[-2]*** и ***((size_t*)Addr)[((size_t*)Addr)[-1]-1]***.

1) *Оба соседа заняты* ***(((size_t*)Addr)[-2]<0*** и ***((size_t*)Addr)[((size_t*)Addr)[-1]-1]<0)***. Добавляем текущий блок к списку свободного места.

2) *Блок слева свободен, блок справа занят* ***(((size_t*)Addr)[-2]>0*** и ***((size_t*)Addr)[((size_t*)Addr)[-1]-1]<0)***. Если мы имеем адрес рабочей части удаляемого блока ***Addr***, то размер левого блока содержится в переменной ***((int*)Addr)[-2]***. Левый

блок принадлежит списку свободного места. Мы можем присоединить текущий блок к левому, не изменяя списка. Для этого надо только модифицировать парные метки блока, полученного объединением левого и текущего блоков. Т.е. в переменные ***((size_t*)Addr)[((size_t*)Addr)[-1]-2], ((size_t*)Addr)[-2-((size_t*)Addr)[-2]+1]*** следует внести длину объединения двух блоков ***l2=((size_t*)Addr)[-1]+ ((size_t*)Addr)[-2]***.

3) *Блок справа свободен, блок слева занят(**((size_t*)Addr)[-2]<0*** и ***((size_t*)Addr)[((size_t*)Addr)[-1]-1]>0**)*. Будем предполагать, что в свободном блоке указатели на предыдущий и следующий блоки лежат в указанном порядке в начале рабочей области блока (т.е. сразу после его длины). По сути, требуется исключить из списка правый блок, объединить текущий блок с правым и добавить их объединение к списку свободных блоков. Более коротко: можно скорректировать ссылки предыдущего и следующего блоков для правого блока, на текущий блок; установить ссылки на предыдущий и следующий блоки для текущего блока; модифицировать длину текущего блока:

size_t *right=((size_t*)Addr)+((size_t*)Addr)[-1]; // Указатель на рабочую часть правого блока
size_t *prev=((size_t**)right)[0]; // Указатель на пред. блок правого блока
size_t *next=((size_t**)right)[1]; // Указатель на след. блок правого блока
((size_t**)prev)[2]= ((size_t*)Addr)-1; *//предыдущий->следующий := текущий*
((size_t**)next)[1]= ((size_t*)Addr)-1; *//следующий->предыдущий := текущий*

((size_t**)Addr)[0]=prev; *// текущий-> предыдущий := предыдущий*
((size_t**)Addr)[1]=next; *// текущий->следующий := следующий*

((size_t*)Addr)[-1]+= ((size_t*)Addr)[((size_t*)Addr)[-1]-1]; *// длина=сумме длин*
((size_t*)Addr)[((size_t*)Addr)[-1]-2]= ((size_t*)Addr)[-1];

4) *Блоки справа и слева свободны (**((size_t*)Addr)[-2]>0*** и ***((size_t*)Addr)[((size_t*)Addr)[-1]-1]>0**)*. Исключим правый блок из списка, а затем объединим левый, текущий и правый блоки:
size_t *right=((size_t*)Addr)+((size_t*)Addr)[-1]; // Указатель на рабочую часть правого блока
size_t *prev=((size_t**)right)[0]; // Указатель на пред. блок правого блока
size_t *next=((size_t**)right)[1]; // Указатель на след. блок правого блока
((size_t**)prev)[2]=next; *//предыдущий->следующий := следующий*
((size_t**)next)[1]=prev; *//следующий->предыдущий := предыдущий*

int l2=((size_t*)Addr)[-2]+ ((size_t*)Addr)[-1]+(((size_t*)Addr)+((size_t*)Addr)[-1])[-1];
((size_t*)Addr)[-((size_t*)Addr)[-2]-1]=l2;
((size_t*)Addr)[-((size_t*)Addr)[-2]-1+l2-1]=l2;

Отметим, что мы использовали в формулах тот факт, что ***sizeof(size_t)==sizeof(size_t*)***. Также мы не рассматривали ситуации, когда рассматриваемый блок лежит на краю используемого куска памяти.

Гарантируется наличие ошибок в вышеприведенных формулах ☺.

Сборка мусора
Возможна принципиально другая стратегия выделения/освобождения памяти. Согласно этой стратегии, мы можем создавать объекты, но не имеем возможности их удалять. В этом случае в среде, в которой происходит выполнение программы, должен присутствовать механизм, называемый *сборкой мусора*. Память выделяется до тех пор, пока хватает системных ресурсов. Когда ресурсы заканчиваются запускается процесс сборки мусора, который освобождает память из под всех объектов, на которые не ссылаются никакие переменные из программы. Например, в рамках этой идеологии реализован язык программирования Java. Сборка мусора также присутствует в языке Python.
Простейшей реализацией системы сборки мусора является введение для каждого объекта *счетчика ссылок* на него. Если какая-то переменная становится ссылкой на объект, то счетчик ссылок увеличивается на 1. Если какая-то переменная уничтожается или становится ссылкой на другой объект, то счетчик на первоначальный объект уменьшается на 1. Считается, что объект, на который никто не ссылается уже никому и не нужен, поэтому он уничтожается в процессе сборки мусора. Подобный подход был реализован в языке *Perl*. Уничтожение объекта происходит сразу же в момент, когда счетчик обнуляется.
Данная реализация имеет очевидные проблемы: если отвести память под циклический список, а потом уничтожить ссылку на этот список, то на каждый объект списка все равно будет ссылаться предыдущий в списке элемент. Т.е. при подобном подходе список уничтожен не будет. Более того, даже если система обнаружит, что на данный список нет больше ссылок, то удалить она его все равно не сможет, т.к. не знает, в каком порядке это надо делать.
Другой разновидностью алгоритмов сборки мусора являются алгоритмы *трассировки*. При реализации этого алгоритма система с некоторой частотой запускает процесс сборки мусора. При этом, основной процесс приостанавливается, чтобы не вносить путаницы. В каждом объекте вводится флаг, обозначающий – используется ли данный объект, или нет. В процессе сбора мусора сначала все флаги обнуляются. Далее рассматривается множество

базовых салок, доступных в данный момент программе. В них флаг устанавливается равным 1. Далее процесс рекурсивно переходит к ссылкам, содержащимся в данных ссылках, и устанавливает флаги в них =1. И т.д. Естественно, что при этом, ссылки на объекты со значением флага, равным 1, далее не рассматриваются. Фактически, здесь реализуется рассмотренный ранее алгоритм волны. Т.о., происходит проход по дереву зависимостей объектов. Когда алгоритм волны остановится, то запускается второй этап алгоритма. Просматриваются ссылки на все существующие объекты и объекты с нулевым значением флага добавляются в список свободной памяти.
Существенным недостатком данного подхода является то, что основной процесс приходится приостанавливать для реализации сборки мусора. Если память сильно загружена, то, с одной стороны, сборка мусора выполняется долга, а с другой – довольно часто. Данный подход лишает нас возможности автоматического вызова деструктора – функции, которая должна вызываться системой в момент разрушения объекта. Эти недостатки привели к остановке развития одного из самых сильных объектно-ориентированных языков – Java, в котором реализован данный подход.

Лекция 19

Прерывания

Процесс выполнения программы сводится, в конечном счете, к следующему. Программа представляет собой набор последовательно записанных команд. Как правило, существует регистр, в котором хранится номер текущий команды. Элементарный шаг выполнения программы называется *циклом*. В процессе исполнения цикла происходит *выборка команды* и *исполнение команды*. Сразу после выборки команды счетчик команд увеличивается так, что он указывает на следующую команду.
Базовым понятием при описании принципов действия ЭВМ является *прерывание*. Бывают прерывания следующих типов:
*Внутренние прерывания (*они же *исключения=exceptions)*
*Внешние (*они же *аппаратные) прерывания*
Программные прерывания

При исполнении прерывания происходит приостановка выполнения основной программы, данные о адресе текущей команды (регистр с адресом текущей команды) и состояние системы (например, регистр флагов) сохраняются (как правило, в стек) и происходит передача управления на программу обработки прерывания. После ее завершения состояние системы восстанавливается, и управление передается на прерванную команду. Отметим, что часто перед обработкой прерывания приходится сохранять гораздо больше данных. Например, обычно сохраняются значения всех регистров. Но это уже - забота функции обработки прерывания.
Очевидно, что внешнее прерывание не может наступать в произвольный момент. Поэтому в цикл выполнения программы добавляется еще один этап: проверка наличия внешнего прерывания. Именно на этом этапе будет возможен вызов внешнего прерывания. Если его поместить после этапа исполнения команды, то в стеке надо будет сохранить текущий регистр с адресом исполняемой команды (он увеличился сразу после выборки команды), а после завершения прерывания надо вернуться по этому адресу.
В процессе выполнения некоторых прерываний следует запретить вызов других прерываний. Этот процесс называется *блокировкой вызовов прерываний*. Вызов прерываний всегда блокируется при, собственно, вызове прерывания, когда необходимые данные загружаются в стек и происходит переход к процедуре исполнения прерывания.
Обычно, прерываниям приписывается *приоритет*. При этом речь может идти об *относительных* и об *абсолютных* приоритетах. В обоих случаях при одновременном поступлении нескольких запросов на прерывание обслуживается запрос на прерывание с наибольшим приоритетом. Но для относительных приоритетов в случае если при обработке прерывания поступает запрос на прерывание с бОльшим приоритетом, то обработка первого прерывания прерывается на обработку второго. Для абсолютных приоритетов обработка второго прерывания может произойти только после завершения первого.
Прерывания могут *маскИроваться*. Под этим подразумевается возможность в любой момент выставлять маску на некоторое подмножество в множестве всех прерываний (например, на все прерывания), вызов которых далее будет запрещен. Данный механизм полезен, например, в случае, когда прерывания из одной группы обращаются к одному неразделяемому ресурсу.
Условно, механизм вызова аппаратных прерываний можно подразделить на *векторные* и *опрашиваемые*. *Векторный* механизм прерываний подразумевает, что с приходом аппаратного прерывания на шину, связывающую процессор и соответствующее оборудование, подается (тем или иным образом) *вектор прерывания*, характеризующий адрес процедуры обработчика прерывания. При использовании *опрашиваемых* прерываний при запросе на прерывание на шину приходит информация только об уровне прерывания. Одному уровню может соответствовать несколько прерываний (=процедур обработчиков прерывания). Процессор вызывает **все** функции обработчики прерывания данного уровня, а из этих процедур уже должно прийти подтверждение о том, что именно данное прерывание было вызвано. Возможна комбинация этих систем. Например, шины могут поддерживать опрашиваемые прерывания, но после прихода прерывания контроллер прерывания выясняет, от кого пришло прерывание (по механизму опрашиваемых прерываний) и выставляет процессору вектор прерывания. Т.е. процессор при этом работает с механизмом векторных прерываний.

В архитектуре x86 прерывания реализовывались с помощью *таблицы прерываний*. Таблица прерываний представляет собой массив, в каждой ячейке которой хранится адрес процедуры обработки прерывания (вектор прерывания) и, возможно, некоторая дополнительная информация. В этом случае каждое прерывание ассоциируется с некоторым номером ***n*** и вызов прерывания ***n*** ассоциируется с вызовом процедуры обработки прерывания, адрес которой находится в ячейке таблицы прерываний ***n***. В современных архитектурах на современных ОС реализация прерывания становится весьма сложным процессом.

Прерывания очень полезны, например, для обработки операций ввода/вывода. Эти операции весьма медлительны, поэтому было бы разумным (на самом деле так происходит далеко не всегда) запустить эту процедуру и вернуть управление текущей программе. Далее, когда от процессора будет требоваться вмешательство в процесс ввода/вывода (например, данные попали, наконец, в буфер ввода/вывода), то вызовется прерывание, которое обработает эти данные и вернет управление текущей программе.

Кэш-память

Cache (запас) обозначает быстродействующую буферную память между процессором и основной памятью. Кэш служит для частичной компенсации разницы в скорости процессора и основной памяти - туда попадают наиболее часто используемые данные. Когда процессор первый раз обращается к ячейке памяти, ее содержимое параллельно копируется в кэш, и в случае повторного обращения в скором времени может быть с гораздо большей скоростью выбрано из кэша. При записи в память значение попадает в кэш, и либо одновременно копируется в память (схема *Write Through* - прямая или сквозная запись), либо копируется через некоторое время (схема *Write Back* - отложенная или обратная запись). При обычной обратной записи, называемой также *буферизованной сквозной записью*, значение копируется в память в первом же свободном такте, а при *отложенной* (*Delayed Write*) - когда для помещения в кэш нового значения не оказывается свободной области; при этом в память вытесняются наименее используемая область кэша. Вторая схема более эффективна, но и более сложна за счет необходимости поддержания соответствия содержимого кэша и основной памяти. По понятным причинам особенно сложна реализация данной схемы в многопроцессорных системах.
Сейчас под термином Write Back в основном понимается отложенная запись, однако это может означать и буферизованную сквозную.
Сквозная запись имеет существенный недостаток: при выполнении подряд идущих операций записи в память кэш, практически, отключается и скорость записи определяется только скоростью работы основной памяти. С другой стороны, в многопроцессорных системах данный подход обеспечивает согласованность оперативной памяти: данные, записанные различными процессорами, оказываются сразу же в памяти. Однако, при этом кэш может оказаться несогласованным с памятью: один процессор может не знать о том, что другой записал что-то в память. В этом случае необходим цикл просмотра памяти для согласования памяти и кэша. Отметим, что в реальных системах операций чтения памяти гораздо больше операций записи (например, при обращении к командам используются вообще только операции чтения). Реально, примерно 10% операций являются операциями записи, а остальные – операциями чтения.
При использовании отложенной записи запись в память осуществляется или при вытеснении данных из Кэша или при особых событиях. Например, в Pentium-компьютерах это может осуществиться даже программным путем с помощью посылки специальной команды или с помощью аппаратного сигнала. Ясно, что смена строк Кэш-памяти для случая отложенной записи требует больше времени, чем в случае сквозной Кэш-памяти, т.к. перед записью строки в Кэш необходимо еще вытеснить текущую строку Кэша в память.
При записи в память при возникновении *промахов* (т.е. ситуаций, когда данные по записываемому адресу не размещены в Кэше) возможно использование двух стратегий: стратегия *с размещением* и стратегия *без размещения*. При использовании стратегии *с размещением* данные параллельно записываются и в Кэш и в память. На самом деле, обычно, сначала процессор записывает данные в основную память и продолжает свою работу, а Кэш-контроллер параллельно считывает соответствующую Кэш-строку из памяти в Кэш. Данная стратегия имеет очень сложную реализацию, поэтому часто при записи данных Кэш-память просто игнорируется (=стратегия *без размещения*).
Память для кэша состоит из собственно области данных, разбитой на блоки (строки), которые являются элементарными единицами информации при работе кэша, и области признаков (*tag*), описывающей состояние строк (свободна, занята, помечена для дозаписи и т.п.). В основном используются две схемы организации кэша: с прямым отображением (*direct mapped*), когда каждый адрес памяти может кэшироваться только одной строкой (в этом случае номер строки определяется младшими разрядами адреса), и n-связный ассоциативный (*n-way associative*), когда каждый адрес может кэшироваться несколькими строками. Ассоциативный кэш более сложен, однако позволяет более гибко кэшировать данные; наиболее распространены 4-связные системы кэширования.
Для современных процессоров можно говорить о трех видах кэш-памяти: *кэш данных* (для ускорения работы с данными), *кэш инструкций* (для ускорения загрузки машинного кода) и *буфер ассоциативной трансляции = TLB* (*Translation lookaside buffer*) (для ускорения трансляции виртуальных адресов в физические). При любом раскладе работа с кэш-памятью недоступна на программном уровне (кроме, быть может, отдельных общих инструкций). Кэш

данных может иметь несколько уровней. Сейчас считается стандартном использование трех уровней кэширования: L1, L2, L3.

Кэширование может быть либо *прямым* (одному адресу оперативной памяти соответствует строго одна строка в кэше), либо *ассоциативным*. Под *ассоциативностью* имеется в виду, что любая ячейка оперативной памяти, вообще говоря, может заноситься в любую строку кэша. Такие реализации встречаются в реальных процессорах, но полностью ассоциативный кэш обычно бывает очень маленьким (256 байт в Cyrix). В силу сложности реализации такого подхода обычно используется *частичная ассоциативность.* При этом подходе каждая строка оперативной памяти может записываться в одну из нескольких строк кэша. Понятие *n-канальной ассоциативности* подразумевает, что одна строка оперативной памяти может быть записана в одной из *n* строк кэша. Реализация прямого кэширования наиболее проста, т.к. для получения адреса в кэше достаточно просто взять требуемое количество бит из адреса оперативной памяти. Грубо говоря, данный вид кэширования полезен, если мы работаем с одним (не очень большим?) массивом данных из оперативной памяти. Но если параллельно используется два массива, то данный подход уже бесполезен.

Существует две стратегии вытеснения строк из кэша при записи в случае, когда все строки кэша, соответствующие записываемой строке памяти, заняты. Это – LRU (least recently used) и LFU (least frequently used).

Приведем пример конкретной реализации ассоциативной кэш-памяти для весьма древнего процессора. Отметим, что современная реализация ассоциативной кэш-памяти, по существу, не отличается ничем от приведенной ниже.

Организация Кэш-памяти и ассоциативная память в IBM PC-совместимых ЭВМ

Рассмотрим пример процессора i386, имеющего 16K ассоциативной Кэш-памяти. Для ее реализации используется быстрая (и дорогая) SRAM-память.

Кэш-строка содержит 16 байт.

Внутри Кэш-контроллера 32-битный адрес делится на три части:

- Биты B31-B12 (20 бит) - адрес *дескриптора*
- Биты B11-B4 (8 бит) - адрес *набора*
- Биты B3-B2 (2 бит) - адрес *слова* (имеются в виду 4-байтные слова).

T.o. для адреса ***x*** имеем:

D(x)=((x>>12)&((1<<20)-1)) *номер дескриптора*
S(x)=((x>>4)&((1<<8)-1)) *номер набора*
W(x)=((x>>2)&3) *номер слова*
B(x)=(x&3) *номер байта*

Ячейка Кэш-блока состоит из *Кэш-директории* и, собственно, *Кэш-записи*.

Кэш-директория содержит 20 бит со значением дескриптора данной Кэш-записи и 5 бит признаков: бит защиты, бит занятости, 3 бита LRU (об этом – позже).

Бит *защиты* предохраняет Кэш-запись от изменения и считывания в процессе цикла записи Кэш-строки. Во время цикла записи он устанавливается равным 1, а в остальное время он равен 0.

Бит *занятости* указывает, на то, что данная Кэш-строка реально отражает содержимое общей памяти. Например, при изменении памяти в режиме DMA данный бит должен быть обнулен. Если данные в Кэше соответствуют данным в основной памяти, то данный бит должен быть равным 1.

Отметим, что когда мы говорили о 16K памяти, то имели в виду лишь память, в которой будут храниться непосредственно данные. Реально еще необходима память для хранения Кэш-директорий. Более того, последняя память будет более часто использоваться и должна иметь большее быстродействие.

Вся Кэш память разбивается на *магистрали*. В нашем случае используется 4 магистрали. Каждая магистраль состоит из массива *строк* по 16 байт. Для данного адреса ***x*** внутри магистрали однозначно задается номер соответствующего набора = ***S(x)***. T.o., размер магистрали равен длине строки умножить на количество наборов = $2^8*2^4=4K$, а т.к. у нас всего 4 магистрали, то размер всей Кэш-памяти равен ***16K***.

Если требуется внести в Кэш ячейку памяти с адресом ***x***, то это можно сделать только вместе со всей строкой в памяти, содержащей ***x***. Итак, пусть требуется внести в память строку, соответствующую адресу ***x***. Мы сразу можем посчитать номер набора для этой строки = ***S(x)***. Этот набор может быть расположен в одной из четырех магистралей в наборе с номером ***S(x)***, поэтому мы должны сравнить ***D(x)*** со значениями дескрипторов в Кэш-директориях с номерами изо всех магистралей, где в данных Кэш-директориях бит защиты = 0 и бит занятости = 1.

Если нашелся дескриптор, равный дескриптору вносимой строки, то считается, что произошло Кэш-попадание и данные могут быть внесены в соответствующую Кэш-строку. Иначе, произошел Кэш-промах. Если среди Кэш директорий с номером ***S(x)*** нашлась свободная директория (т.е. значение ее бита занятости = 0), то мы можем разместить свои данные в соответствующей Кэш-записи. Иначе нам придется выбрать одну из занятых Кэш-записей с номером ***S(x)***, вытеснить ее в обычную память и занести нашу Кэш-строку на ее место, модифицировав значения дескриптора в Кэш-директории на значение ***D(x)***.

Простейшая стратегия выбора подходящей магистрали – выбор ее случайным образом. Возможен более аккуратный способ, когда выбирается давно

использующаяся Кэш-запись. Это делается с помощью ***LRU***-бит (*Least Recently Used*) в Кэш-директории. Например, это можно сделать следующим способом. Обозначим биты ***LRU: B0, B1, B2***.

Если произошел доступ к магистрали 0 или 1, то установим ***B0=1***, при этом если произошел доступ к магистрали 0, то установим ***B1=0***, иначе установим ***B1=1***.

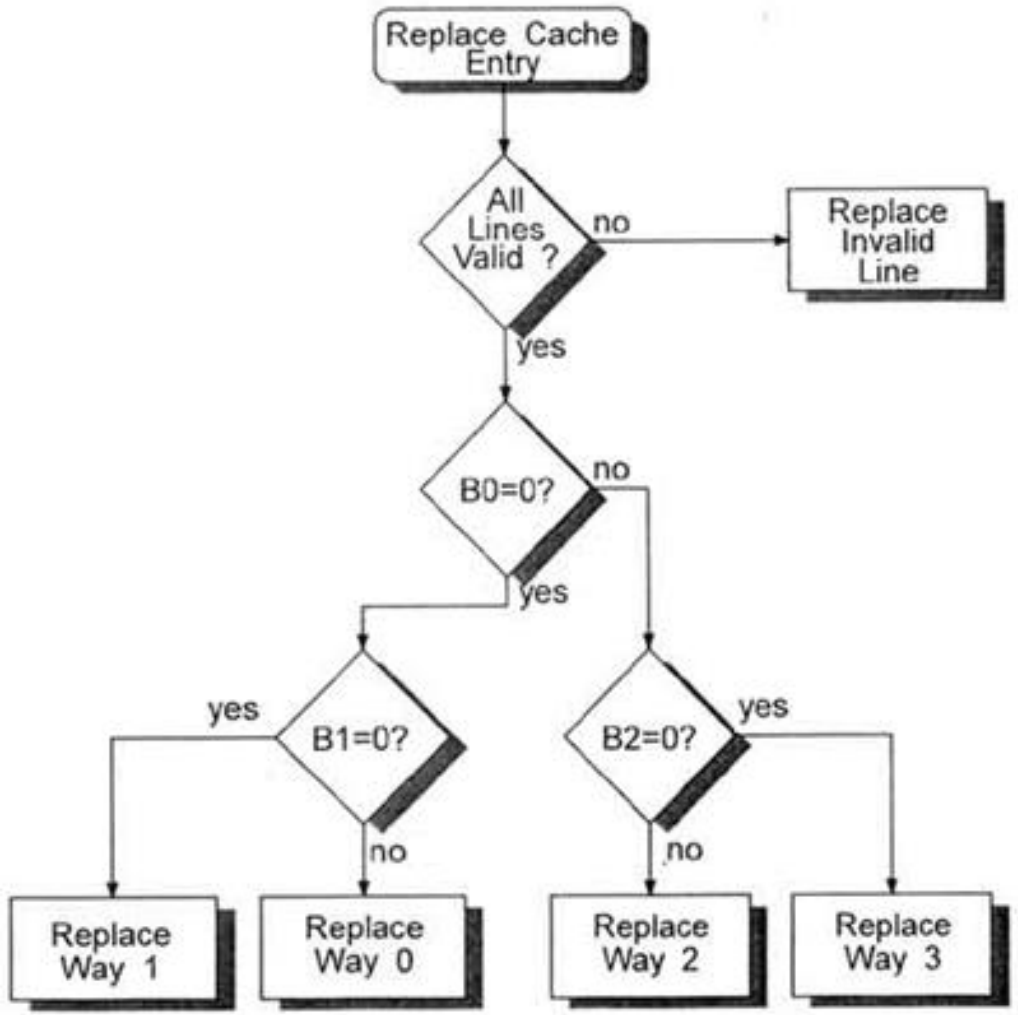
Если произошел доступ к магистрали 2 или 3, то установим ***B0=0***, при этом если произошел доступ к магистрали 2, то установим ***B2=0***, иначе установим ***B2=1***.

Т.о. образом, бит ***B0*** указывает, к какой паре магистралей произошел последний доступ. Т.е. если ***B0==0,*** то использовать для замещения нужно магистрали 0 или 1, а если если ***B0==1,*** то использовать для замещения нужно магистрали 2 или 3. Оставшиеся биты указывают, какая из двух магистралей использовалась последней. Т.е. если ***B0==0&& B1==0***, то последней из двух магистралей 0 и 1 использовалась магистраль 0, поэтому для замещения надо использовать магистраль 1.

Если ***B0==0&& B1==1***, то последней из двух магистралей 0 и 1 использовалась магистраль 1, поэтому для замещения надо использовать магистраль 0.

Если ***B0==1&& B2==0***, то последней из двух магистралей 2 и 3 использовалась магистраль 2, поэтому для замещения надо использовать магистраль 3.

Если ***B0==1&& B2==1***, то последней из двух магистралей 2 и 3 использовалась магистраль 3, поэтому для замещения надо использовать магистраль 2.



На практике оказывается, что стратегия случайного замещения Кэш-записей оказывается не намного хуже стратегии LRU.

Основные принципы параллельных вычислений

Параллельные вычисления имеет смысл рассматривать в двух принципиально (по крайней мере, на первый взгляд) различных ситуациях: *в случае однопроцессорной системы*, в которой выполняется псевдо-параллельное выполнение нескольких процессов и в случае много-процессорной системы, когда присутствует реальное параллельное выполнение нескольких процессов.

Можно говорить о следующих основных проблемах, возникающих при параллельных вычислениях:

- Проблемы, возникающие при некорректном разделении доступа к глобальным объектам
- Проблемы, связанные со сложностью для ОС оптимального разделения ресурсов между процессами; при неоптимальном распределении ресурсов возможны явления, имеющие названия *взаимоблокировки* и *голодание*
- Сложность устойчивого повтора создавшихся ситуаций при отладке программ

Первую проблему можно проиллюстрировать следующим примером:

*char *num2str(int n){static char s[100]; sprintf(s,"%d",n); return s;}*

Данная функция возвращает строку, в которой содержится текстовое представление параметра, передаваемого в функцию. Использование статической переменной удобно, т.к. не требует отведения/очистки памяти под строку. Однако, в многопоточной задаче использование подобной функции недопустимо, т.к. внутри одного процесса потоки имеют единое адресное пространство, и если два параллельных потока одновременно (или почти одновременно) вызовут эту функцию, то результат будет непредсказуемым.

Вторую проблему можно проиллюстрировать следующим простым примером: первый процесс сначала захватывает ресурс А, а потом требует захвата ресурса Б (не освобождая А). Параллельно второй процесс захватывает ресурс Б и далее требует захвата ресурса А. В результате, ни один из двух процессов не может продолжить свою работу. Данная ситуация называется *взаимоблокировкой*.

Третья проблема очевидна. При отладке многопроцессных приложений практически нереально воспроизвести создавшуюся ситуацию во второй раз, т.к. скорость работы каждого процесса зависит от очень большого количества разнообразных факторов. Более того, если мы пользуемся отладчиком, то он сам является процессом и, поэтому он также оказывает влияние на запущенные процессы. По сути, существует единственный способ отладки многопроцессных приложений: их надо сразу писать правильно ☺.

Вообще можно говорить о разных уровнях взаимодействия процессов, в зависимости от того, что процессы знают друг о друге. Если у процессов нет никакой информации друг о друге, то можно говорить о *конкуренции* процессов при попытке разделить общие ресурсы.

Если же какая-то информация друг о друге имеется, то говорят о *сотрудничестве* процессов при разделе ресурсов. При этом возможна ситуация, когда, на самом деле, процессы практически ничего друг о друге не знают, кроме того, что они используют один ресурс и изменения в этом ресурсе одного процесса могут использоваться другим процессом (*сотрудничество при разделении ресурса*). Или же процессы знают идентификатор друг друга и могут получать почти любую информацию друг о друге (*сотрудничество при работе*).

С каждым из описанных видов взаимодействия связаны соответствующие проблемы.

Конкуренция процессов в борьбе за ресурсы

Пусть есть два или более процессов, которые ничего не знают друг о друге, но используют один неделимый ресурс (см. пример функции, приведенной выше). В этом случае должен существовать механизм *взаимных исключений*. Этот механизм должен обеспечивать в определенных кусках кода каждого процесса возможность использования данного ресурса не более чем одним процессом. Ресурс, о котором идет речь, обычно называется *критическим ресурсом*, а кусок кода процесса, в котором должно осуществляться единовластие процесса над критическим ресурсом, принято называть *критической секцией* или *критическим разделом* (critical section).

Т.о., в коде программы должны быть обеспечены операторы начала критической секции (с указанием критического ресурса) и конца критической секции. При этом, мы сразу же получаем две новые проблемы: *взаимные блокировки* и *голодание*.

Сущность взаимной блокировки уже описана выше. *Голодание* наступает в результате более сложной ситуации. Пусть есть три процесса *P1, P2, P3*. Каждый из которых нуждается в каком-либо неразделяемом ресурсе. Пусть процесс *P1* получил доступ к ресурсу и успешно его использует. Параллельно ресурс потребовался процессам *P2* и *P3*. Пусть процессы *P1* и *P2* имеют больший приоритет, чем процесс *P3*. Тогда после завершения использования ресурса процессом *P1* начнет выполняться процесс *P2*. Пусть в процессе использования ресурса процессом *P2* данный ресурс вновь потребовался процессу *P1*, а потом он снова потребовался процессу *P2*, и т.д. В результате ресурс будет переходить от процесса *P1* к процессу *P2* и наоборот, а процесс *P3* так и не получит возможность продолжить работу, хотя, формально, взаимоблокировки не наступало. Данная ситуация носит название *голодания*.

Сотрудничество с использованием разделения

Пусть есть два или более процессов, которые используют один ресурс по принципу сотрудничества по использованию ресурса. При этом, результат воздействия на ресурс одного процесса интересует другой процесс.

Простейший пример. У нас есть счетчик обращений к ресурсу: глобальная переменная ***n***. Мы можем вызвать функцию увеличения счетчика:

void NInc(){n=n+1;}

а можем просто посмотреть значение переменной ***n***. Отметим, что, вообще говоря, свободно использовать функцию в параллельных процессах нельзя, т.к. можно себе представить следующую последовательность действий по ее фактической реализации:
*положить переменную **n** в регистр*
*увеличить переменную **n** на 1*
*положить значение регистра в переменную **n***

Теперь, если один процесс положит переменную в регистр и увеличит ее значение на 1, а другой процесс после этого прервет первый и сделает то же самое, то, в результате, в регистре будет лежать значение исходной переменной, увеличенное всего на 1, а не на 2, как хотелось бы. В результате при двух вызовах функции переменная ***n*** увеличится всего лишь на 1.
Т.о. в данной ситуации при вызове функции *NInc* следует помечать начало и конец критической секции. В отличие от предыдущей ситуации критическая секция предохраняет критический ресурс только от записи, но не от чтения. При выполнении функции *NInc* мы можем в любой момент использовать значение переменной ***n***.
Однако, здесь появляется требование *согласованности данных*. Например, если у нас есть две переменных ***n1, n2***, играющих в точности такую же роль, как и переменная ***n***:
void NInc2(){n1=n1+1;n2=n2+1;}

то разумно потребовать, чтобы в каждый момент времени эти переменные совпадали бы. Однако, т.к. мы разрешили в **любой** момент времени использовать эти переменные, то при выполнении данной функции это условие может нарушаться. Проблема решается опять же с помощью критических секций.
Следующий пример еще более неприятен:
В одном процессе выполняется функция
void Add1(){n1++; n2++;}
а в другом:
void Mult2(){n1=2; n2*=2;}*

Применение каждой функции не нарушает условие равенства переменных. Однако их одновременное применение может привести к следующей последовательности действий:
n1++;
n1=2;*
n2=2;*
n2++;

а в этом случае переменные перестанут быть равными.
В качестве решения проблемы можно предложить погружение каждого тела функции в критическую секцию для одного ресурса.

Сотрудничество с использованием связи

Пусть есть два или более процессов, которые знают идентификаторы друг друга и могут активно друг с другом общаться. Даже в этом случае возможны конфликты. Например, процесс *P1* постоянно обменивается данными с процессами *P2, P3*. Голодание может возникнуть от того, что процесс *P1* будет постоянно обмениваться данными с процессом *P2*, а процесс *P3* будет сколь угодно долго ждать своей очереди.

Семафоры

Одним из базовых понятий, используемых при реализации взаимных исключений, являются *семафоры*. Семафоры реализуются на уровне операционной системы, и их следует использовать просто как объекты, удовлетворяющие некоторым свойствам.
Семафор следует рассматривать как объект, содержащий целую переменную *Count* и очередь процессов *ProcQueue*. При этом, переменную *Count* следует рассматривать как переменную, которая никогда не может стать отрицательной. Т.е. при попытке ее уменьшения до отрицательного значения текущий процесс (нить) блокируется и уменьшение переменной (вместе с разблокировкой процесса) наступает только после того, как переменная станет положительной.
Над семафором можно совершать **только** три **атомарных** операции:

1. Инициализация переменной семафора *Count* неотрицательным числом.
2. Если значение *Count* меньше или равно 0, то блокировка текущего процесса и помещение его в очередь процессов *ProcQueue* Уменьшение переменной семафора *Count* на *1*;.
3. Увеличение переменной семафора *Count* на *1*; если значение *Count* становится больше или равно *0*, то из очереди процессов *ProcQueue* извлекается и разблокируется очередной процесс.

Вообще говоря, предполагается, что никаких других действий с семафорами осуществлять нельзя. На самом деле, в различных реализациях семафоров добавляются некоторые дополнительные действия, которые можно осуществить с семафорами. Например, в Linux реализованы следующие функции для работы с семафорами:

```
int sem_init(sem_t *sem, int pshared, unsigned int Count);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
```

```
int sem_post(sem_t * sem);
int sem_getvalue(sem_t * sem, int * sval);
int sem_destroy(sem_t * sem);
```

Функции описаны в файле *semaphore.h*.
Идентификатором семафора является объект типа *sem_t*.

Инициализируется семафор функцией *sem_init*. В ней задаются:
- семафор, который следует инициализировать (указатель на *sem_t*),
- целая переменная *pshared* указывает на то, что семафор используется в различных процессах (*pshared=1*; в этом случае переменная семафора должна размещаться в разделяемой памяти) или внутри одного процесса в различных нитях (*pshared=0*),
- целая переменная *Count* задает начальное значение семафора

Функция *sem_wait* проверяет значение семафора. Если значение семафора меньше или равно 0, то процесс (нить) блокируется. Далее (после разблоктровки) функция уменьшает на 1 значение семафора.
Функция *sem_post* увеличивает на 1 значение семафора и, если необходимо, извлекает из очереди процессов ждущий заблокированный процесс и разблокирует его.
Дополнительные функции это:

*int sem_trywait(sem_t * sem);*
Если *Count==0*, то возвращает *!0*, иначе уменьшает *Count* на *1* и возвращает *0.*

*int sem_getvalue(sem_t * sem, int * sval);*
Возвращает значение семафора.

*int sem_destroy(sem_t * sem);*
Очищает память из-под структуры данных, созданной *sem_init*.

Легко увидеть, что механизм взаимных исключений элементарно реализовать с помощью семафоров:

```
sem_t sem;
main()
{//инициализация семафора:
sem_init(&sem,0,1);
//...
}
```

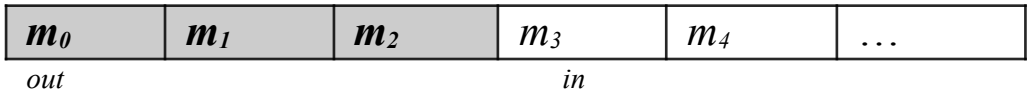


```
//функция начала критической секции:
void CriticalBegin()
{
 sem_wait(&sem);
}

//функция конца критической секции:
void CriticalEnd()
{
 sem_post(&sem);
}
```

Задача о производителях и потребителе
Пусть есть несколько производителей, которые производят штучный товар и кладут его на склад, и есть один потребитель, который берет товар со склада и потребляет. Следует разграничить права доступа производителей и потребителя к складу, так чтобы только один производитель или потребитель в один и тот же момент времени имел бы возможность пользоваться складом.
Задача является весьма распространенной. Например, несколько клиентов могут посылать текстовые сообщения серверу, а сервер должен выводить эти сообщения на экран по одному сообщению в строке (естественно, что в одной строке не должны смешиваться два сообщения от клиентов).

Пусть производитель производит целые числа, а склад реализован в виде очереди на основе бесконечного массива целых чисел *m*. Переменная *in* указывает на номер ячейки в массиве *m*, куда производитель должен класть очередное число. После помещения числа в массив *in* должна увеличиваться на *1*. Переменная *out* указывает на номер ячейки в массиве *m*, откуда потребитель должен забирать очередное число. После извлечения числа из массива *out* должна увеличиваться на *1*.



Выделены элементы массива, лежащие на складе.
Попробуем написать программу, реализующую описанную ситуацию. Чтобы не связываться с массивом=складом и переменными *in/out* введем целую переменную *n=in-out*, которая будет указывать на количество товара на складе. Будем использовать следующие рабочие функции:
Produce(); *//произвести единицу товара*

```
Append();          //занести единицу произведенного товара на склад
Take();            //взять единицу произведенного товара со склада
Consume();         //использовать товар по назначению
```

Неудачная попытка.

Будем использовать два семафора: один - отвечающий за блокировку при обращении к складу (*prod*), другой - позволяющий потребителю спать, пока на складе ничего нет (*cons*).
Инициализация семафоров:

```
sem_t prod,cons; int n=0;
main()
{
sem_init(&prod,0,1);//изначально склад готов к обслуживанию товара
sem_init(&cons,0,0);//изначально потребителю нечего потреблять
 //...
}
```

Итак, на первый взгляд, функция, отвечающая за производство в каждом процессе-производителе может иметь следующий вид:

```
void P0()
{
 while(1)//вечно производить
 {
  Produce();        //произвести единицу товара
 sem_wait(&prod);//начало критической секции
   Append();        //занести единицу произведенного товара на склад
   n++;
   if(n==1)//при поступлении первого товара разблокировать потребителя
    sem_post(&cons);
 sem_post(&prod); //конец критической секции
 }
}
```

функция, отвечающая за потребление в процессе-потребителе может иметь следующий вид:

```
void P1()
{
 sem_wait(&cons);//ждать первого товара на складе
 while(1)//вечно потреблять
 {
 sem_wait(&prod);//начало критической секции
  Take(); //изъять единицу произведенного товара со склада
  n--;
 sem_post(&prod); //конец критической секции
 Consume();          //употребить единицу товара
 if(n==0)//при отсутствии товара на складе заблокировать потребление
  sem_wait(&cons);
 }
}
```

Отметим, что, совершенно логично, функции производства/потребления вынесены за рамки критических секций, т.к. критическому ресурсу они не обращаются, а время их выполнения, наверняка, весьма существенно.
Оказывается, что приведенный алгоритм неверен. Вкратце, это можно проиллюстрировать следующим образом. Пусть выполнилась следующая последовательность действий:

- произвели товар (Þ *n==1* ; значение *cons == 1*)
- изъяли товар со склада и потребляем его в функции *Consume();* (Þ *n==0*)
- снова произвели товар (Þ *n==1* ; значение *cons == 2*)
- проверили в *P1* условие *if(n==0)*; оно оказалось ложью, поэтому значение *cons* не изменилось
- потребили товар (Þ *n==0* ; значение *cons == 2*)
- проверили в *P1* условие *if(n==0)*; оно оказалось истиной, поэтому значение *cons* уменьшилось (Þ *n==0* ; значение *cons == 1*), но блокировка для *cons* не наступила, т.к. *cons > 0*
- пытаемся снова потребить товар, но склад пуст. Неприятности.

Правильное решение.

Проблема в предыдущем примере заключалась в том, что у нас нарушился баланс между инкрементациями и декрементациями семафора. Мы инкрементировали семафор каждый раз, когда выполнялось *n==1* и должны были декрементировать его каждый раз, когда выполнялось *n==0*, но последнее требование мы не выполнили.
Чтобы не пропустить случай *n==0* , мы можем ввести временную переменную *n1* и присвоить ей значение *n* внутри критической секции. Теперь нулевое значение *n* не пропадет, т.к. сравнение *if(n==0)* мы заменим на сравнение *if(n1==0)*.
Итого, правильный вариант функции, отвечающей за потребление:

```
void P1()
{int n1;
 sem_wait(&cons);//ждать первого товара на складе
 while(1)//вечно потреблять
 {
  sem_wait(&prod);//начало критической секции
   Take(); //изъять единицу произведенного товара со склада
   n--; n1=n;
  sem_post(&prod); //конец критической секции
  Consume();              //употребить единицу товара
  if(n1==0)//при отсутствии товара на складе заблокировать потребление
   sem_wait(&cons);
 }
}
```

Более короткое решение.

На самом деле, переменная *n* является лишней, т.к. семафор *cons* сам является счетчиком. Поэтому функции могут иметь следующий вид:

```
void P0()
{
 while(1)//вечно производить
 {
  Produce();          //произвести единицу товара
  sem_wait(&prod);//начало критической секции
   Append();          //занести единицу произведенного товара на склад
  sem_post(&prod); //конец критической секции
   sem_post(&cons);//добавить 1 к семафору потребителя
 }
}

void P1()
{
while(1)//вечно потреблять
 {
  sem_wait(&cons);//ждать товара на складе
  sem_wait(&prod);//начало критической секции
   Take(); //изъять единицу произведенного товара со склада
  sem_post(&prod); //конец критической секции
  Consume();              //употребить единицу товара
}
}
```

Отметим, перестановка *sem_post(&prod);* и *sem_post(&cons);* в *P0* ничего не изменит, т.к.для изъятия товара надо разблокировать оба семафора. А вот перестановка *sem_wait(&cons);* и *sem_wait(&prod);* в *P1* приведет к неприятным последствиям. В этом случае, если потребитель войдет в критическую секцию при пустом складе, то производитель уже никогда не сможет воспользоваться складом. Возникнет взаимоблокировка.