ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

АЛГОРИТМЫ И АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ

БЕЛЕВАНЦЕВ
АНДРЕЙ АНДРЕЕВИЧ

ВМК МГУ

БЛАГОДАРИМ ЗА ПОДГОТОВКУ КОНСПЕКТА
СТУДЕНТКУ ФАКУЛЬТЕТА ВМК МГУ
ЧЕРНИКОВУ ПОЛИНУ ГЕОРГИЕВНУ

Оглавление

Лекция №1. Машина Тьюринга

Элементы теории алгоритмов

Под *алгоритмом* (или эффективной процедурой) в математике понимают *точное предписание*, задающее *вычислительный процесс*, ведущий от *начальных данных*, которые могут варьироваться, к *искомому результату**.
*Неформальное определение алгоритма, так как слова, выделенные курсивом еще точно не определены.
Алгоритм должен обладать следующими свойствами:

- **Конечность (результативность)**

Алгоритм должен заканчиваться за конечное (хотя и не ограниченное сверху число шагов).

- **Определенность (детерминированность)**

Каждый шаг алгоритма и переход от шага к шагу должны быть точно определены и каждое применение алгоритма к одним и тем же исходным данным должно приводит к одинаковому результату.

- **Простота и понятность**

Каждый шаг алгоритма должен быть четко и ясно определен, чтобы выполнение алгоритма можно было «поручить» любому исполнителю (человеку или механическому устройству)

- **Массовость**

Алгоритм задает процесс вычисления для множества исходных данных (чисел, строк, букв и т.п.), он представляет собой общий метод решения задач
В компиляторах gcc (основной открытый компилятор Linux) детерминированность настолько важна, что сам процесс сборки компилятора построен таким образом, чтобы как можно в большей степени протестировать этой свойство. То есть если на вход компилятора подается один и тот же файл, то компилятор должен выжать один и тот же объектный код. Рассмотрим процесс сборки. Часто так бывает, что для того, чтобы скомпилировать компилятор Си нужен компилятор Си. Этот процесс называется bootstrap, то есть «раскрутка». Первый этап раскрутки заключается в следующем: есть системный компилятор и компилятор, который компилируется. Системным компилятором компилируется исходный код нового компилятора. В итоге получается объектный код нового компилятора, который может выполняться на вашей машине. На втором этапе новым компилятором еще раз собираются исходники своего компилятора. Таким образом вы можете быть уверены, что эта программа оптимизирована вами же. Кроме того, вы можете сразу использовать все особенности вашего компилятора, а не поддерживать несколько системных компиляторов. Для написания стандартных библиотек это важно. Для того, чтобы проверить свойство детерминированности есть

третий этап, который заключается в следующем: вторая версия объектного кода gcc, которая собрана сама собой, собирается еще раз. Объектный код, полученный на втором и третьем этапе, сравнивается. Если по каким-то причинам он оказался разным, то свойство детерминированности отсутствует, что означает очень серьезную ошибку.

Пример. *Алгоритм Евклида* нахождения наибольшего общего делителя двух целых положительных чисел a и b НОД(a,b).
Даны два целых числа a и b, найти НОД(a,b):

1. Если a < b, то поменять их местами
2. Разделить нацело a на b, получить остаток r
3. Если r = 0, то НОД(a,b) = b
4. Если r ≠ 0, заменить a на b, b на r и вернуться к шагу 2

Алгоритм Евклида обладает всеми свойствами алгоритма.

Машина Тьюринга

Когда появились результаты, в том числе теорема Геделя о неполноте, означавшие, что можно выдвинуть набор каких-то аксиом и в их терминах сформулировать утверждение, ответа на которое не будет даже без всякого алгоритма, появились сомнения, что любую задачу можно алгоритмически разрешить. Появился вопрос: существуют ли задачи, для которых нельзя построить алгоритм вообще?
Не имея такого определения, невозможно доказать, что задача алгоритмически неразрешима, то есть алгоритм ее решения никогда не удастся построить.
Тезис Тьюринга-Черча. Для любой интуитивно вычислимой функции существует вычисляющая ее значения машина Тьюринга*.
*Для Тьюринга найдется машина Тьюринга, для Черча – лямбда-исчисление.
Тезис Тьюринга-Черча невозможно строго доказать или опровергнуть, так как он устанавливает эквивалентность между строго формализованным понятием частично вычислимой функции и неформальным понятием вычислимости.
Получается, что математиком, придумавшим формализацию, утверждается, что его формализация достаточно точна, чтобы покрыть все мыслимые алгоритмы. Но всех мыслимых алгоритмов без этой формализации не существует, поэтому это называется тезисом.

Формализация

Чтобы формализовать информацию, можно считать, что все данные записываются в виде некоторых символов, эти символы составляются в последовательности.
Последовательность этих символов и есть данные.
Алфавит – это конечное множество $A_p = \{a_1, a_2, \dots, a_p\}$.

Элементы алфавита A_p называются *символами.*
Последовательность из m символов алфавита A_p называется словом длины m над алфавитом A_p: $a_{i_1} a_2 \dots a_{i_m}$.
Слово длины 0 называется *пустым* словом и обозначается ε .
Множество всех слов над алфавитом A_p:
$A_p{}^* = \{ \varepsilon \} \cup A_p \cup A_p{}^2 \cup \dots \cup A_p{}^m \cup = \bigcup_{m=0}^{\infty} A_p{}^m$
Длину слова w $\in A_p{}^*$ будем обозначать |w|, в частности для пустого слова $| \varepsilon |$ = 0.

Утверждение. Для любой пары алфавитов A и B можно выполнить кодирование алфавита A с помощью алфавита в и обратно, возможно, с применением дополнительно служебного символа («конец кода символа»).

Следствие. Кодирование позволяет ограничиться одним алфавитом. Обычно рассматривается A_1 или A_2.

Пример. |A| > |B|.
|A| = p, |B| = m => p > m
Первые m символов из A кодируются однобуквенными словами из алфавита B, следующие m^2 символов кодируются двухбуквенными словами из алфавита B и так далее пока процесс не закончится. Далее, на любое слова из алфавита A записываются соответствующие кодировки из алфавита B, и возникнет знак, позволяющий отделить одно слово от другого.

Множество всех слов любого алфавита *счётно*, то есть можно пронумеровать все слова из одного алфавита и из другого и поставить между ними соответствие по номерам.
В компьютерах самый популярный алфавит $A_2 = \{0,1\}$. Но можно обойтись и алфавитом из одного символа A = {|}. Так считали дикари или дети, то есть одна палочка, две палочки…
Также часто про задачи обработки информации, то есть про алгоритмы, рассуждают со точки зрения вычислимой функции. Рассматривают функции, которые переводят некоторые натуральные числа в некоторые другие и строят формализм над ними.
Задача обработки информации – это задача построения частичного отображения (функции) F: $A^* \to A^*$.

Утверждение. Существует взаимно-однозначное отображение # : $A^* \leftrightarrow \mathrm{N}_0$, где N_0 – множество целых неотрицательных чисел, которое любому слову w $\in A^*$ ставит в соответствие его номер n $\in \mathrm{N}_0$. Это отображение # и называется нумерацией.
Можно заметить, что частично вычислимая функция и алгоритм похожи друг на друга, то есть одно задает другое. «Частично вычислимая» обозначает, что алгоритм не обязан быть определенным для любого слова из заданного множества.

Если есть частично вычислимая функция, которая одно число переводит в другое: f: $\mathrm{N}_0 \rightarrow \mathrm{N}_0$, то f одно и то же, что и F: $A^* \rightarrow A^*$.
Пусть задано слово A_p из m+1 символа $a_{i_0} a_1 \dots a_{i_m}$. Нулю ставим в соответствие эпсилон $0 \rightarrow \varepsilon$. Далее вычисляем номер слова через возведение в степень, соответствующую основанию системы счисления – p. То есть $p \times i_0 + p^2 \times i_1 + \dots + p^{m=1} i_{m+1}$. Таким образом по слову получаем его номер и, что менее очевидно, однозначным способом по номеру получим слово. Это следует из известной теореме о p-ых системах позиционного счисления, которая доказывает, что запись любого числа в p-ой системе счисления единственна. Поэтому каждому слову в соответствие ставится единственный номер, и обратно: зная основание системы счисления, число можно разложить на натуральные степени и понять, сколько символов нужно, чтобы закодировать это число.
Получается, что, если есть алгоритм F: $A^* \rightarrow A^*$, этому F можно построить f: $\mathrm{N}_0 \rightarrow \mathrm{N}_0$ с помощью нумерации и наоборот.

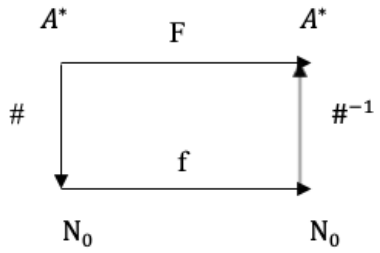


рис. 1.1 Переход отображений

Таким образом (см. рис. 1.1):

1. Каждый алгоритм F: $A^* \rightarrow A^*$ определяет частично вычислимую функцию f: $\mathrm{N}_0 \rightarrow \mathrm{N}_0$
2. Каждая частично вычислимая функция f: $\mathrm{N}_0 \rightarrow \mathrm{N}_0$ определяет алгоритм F: $A^* \rightarrow A^*$

Например, есть слово w $\in A^*$. Предположим, что с этим словом алгоритм выполним и применим его: F(w) = v $\in A^*$. Посмотрим, как выполняется f: $\mathrm{N}_0 \rightarrow \mathrm{N}_0$. # - нумерация, $\#^{-1}$- восстановление слова по номеру. #(w) – получаем номер, f(#(w)) – получаем другой номер. $\#^{-1}(f(\#(w)) = v$ – восстанавливаем слово по номеру. На самом деле $\#^{-1}(f(\#(w)) = v \equiv F(w)$.
И наоборот #(F($\#^{-1}(n)$) = m $\equiv f(n)$

Обработка информации

Как формализовать процесс перевода одного слова в другое? Идея Тьюринга заключалась в том, чтобы формализовать каждый шаг (чем меньше шаг, тем лучше)

алгоритма. То есть нужно придумать автомат, который будет выполнять этот вычислительный процесс.

Машина-автомат: предъявляется любое исходное слово w ∈ A^*, а в результате обработки получается v = F(w). Каждая частичная функция F, для которой можно построить МТ, называется *вычислимой по Тьюрингу.*

Машина Тьюринга устроена следующим образом:

- Алфавит состояний Q = $\{q_0, q_1, \dots a_n\}$
- Рабочий алфавит S = $A \cup A'$
- A – алфавит *входных* символов
- A' – алфавит *выходных* символов
- Лента, размеченная на ячейки (пустая ячейка - Λ)
- Управляющая головка (УГ)
- Рабочая ячейка (РЯ)
- Начальное состояние q_0, состояние останова q_s
- Начальные данные – слова из A*

Машина Тьюринга представляет из себя ленту бесконечной длины, разделенную на ячейки (без номеров). Управляющая головка для каждой ячейки выполняет какое-либо действие в зависимости и от состояний. В каждый момент времени УГ направлена на одну ячейку. Действие происходит только с рабочей ячейкой (ячейкой, на которую в данный момент направлена УГ). Мы можем записать в ячейку один символ, оставить ячейку пустой, сдвинуть головку на одну ячейку влево или вправо. Выбор действия зависит от обозреваемого символа в РЯ и от текущего состояния. Таким образом действие всегда определяется парой (символ, состояние).

- Конфигурация МТ: (n, F, q), где n - номер текущей рабочей ячейки, F: $\mathbb{Z} \to S$ – текущая запись на ленте, q – текущее состояние.
- Позиция МТ: пара (n, q)
- Такт работы МТ: $\langle$состояние, символ$\rangle \to \langle$состояние, символ, направление$\rangle$

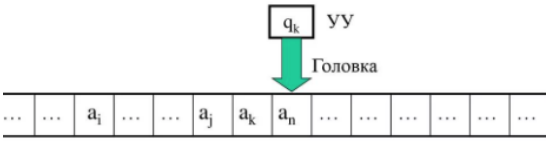


рис. 1.2 Машина Тьюринга

Машина останавливается в момент, когда переходит в конечное состояние q_s, символ, на который в данный момент находится в РЯ и будет выходным словом. Машина

считается определенной только для тех слов, для которых она приходит в состояние остановы.

Пример. Проверка правильности скобочных выражений.
МТ записать на ленту для правильного скобочного выражения результат 1 (для неправильного 0) и остановиться.

Правильное скобочное выражение*:

1. Число открывающихся скобок равно числу закрывающихся скобок
2. Каждая открывающаяся скобка предшествует закрывающейся скобке

(( ))( ) -правильное скобочное выражение
) (или (( ) – неправильные скобочные выражения

*для одного типа скобок

Решение:
Рабочий алфавит: S = {(,), 0, 1} ∪ { Λ, X}
Алфавит состояний Q = {q_0, q_1, q_2, q_3, q_s }
q_0 - начальное состояние МТЖ поиск ближайшей справа закрывающейся скобки
q_1 - поиск парной открывающейся скобки
q_2 - стирание маркеров, запись результата 1 и переход в состояние q_s
q_3 - стирание маркеров, запись результата 0 и переход в состояние q_s
q_s – состояние останова
R – направо
L - налево
H - ничего

В начальном состоянии УГ обозревает самый левый символ входного слова.
Чаще всего решение записывается в виде таблицы.

$q_i \downarrow \backslash s_j \rightarrow$	(	)	X	Λ
q_0	$q_0, (, R$	q_1, X, L	q_0, X, R	q_2, X, L
q_1	q_0, X, R	$q_1,), L$	q_1, X, L	q_3, Λ, R
q_2	q_3, Λ, H	—	q_3, Λ, H	$q_s, 1, H$
q_3	q_3, Λ, L	—	q_3, Λ, L	$q_2, 0, H$

Лекция №2. Машина Тьюринга часть II

Мы обсудили, способ формализовать численный процесс и его ответвление – связь алгоритмов и частично вычислимых функций. Алгоритм можно рассматривать как некоторую функцию, которая одно слова из алфавита переводит в другое, при этом важна частичная определенность, потому что не для всех слов из множества слов над алфавитом алгоритм обязан работать. Так как все слова можно занумеровать, функции, обрабатывающей слова (алгоритму) поставить в соответствие функцию, которая работает над числами. Далее можно рассуждать над частично определенными функциями, которые определены над множеством натуральных чисел и 0, который включается из-за удобства кодирования им пустого слова, а можно рассуждать над алгоритмом, по сути, это одни и те же объекты. Мы будем рассматривать 2 попытки ученых формализовать вычислительный процесс: Машину Тьюринга и алгоритмы Маркова.

Вернемся к задаче проверки скобочных выражений. Рассмотрим пример (рис. 2.1).

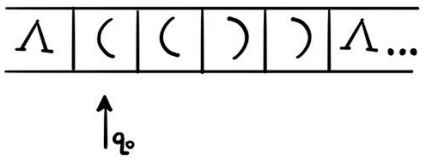


рис. 2.1 Начальное состояние

В начальном состоянии управляющая головка направлена на крайний левый непустой символ. В состоянии q_0 машина находит первую закрывающуюся скобку и помечает ее маркером (рис. 2.2) и переходит в состояние q_1.

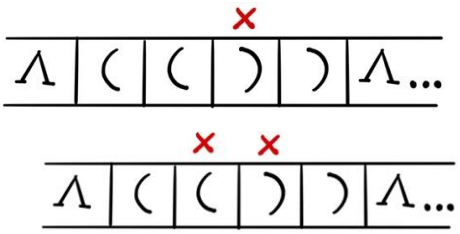


рис. 2.2 Маркеры

Состояние q_1 отвечает за то, чтобы найти ближайшую к этой скобке открывающуюся. После того, как пара состояний q_0 и q_1 отработает, будет найдена корректная пара скобок, которую мы вычеркнем, заменив на маркеры. Дальше так же рассматриваем

другие скобки. Если все хорошо, то при очередном поиске в состоянии q_0 мы не найдем открывающуюся скобку, будет достигнут конец слова, мы упремся в Λ. Это означает, что нам нужно проверить, действительно ли мы обработали все пары скобок. Поэтому мы переходи в состояние q_2 и движемся налево, проверяя, не осталась ли какая-нибудь скобка. Если скобка осталось, то выражение было неправильным, если не осталась и мы дойдем до Λ, то выражение было корректным, и мы запишем 1.
Если где-нибудь в середине мы ищем парную скобку и не находим ее, переходим в состояние q_3, в котором мы уже точно знаем, что последовательность неправильная и нам нужно записать 0.
Эта программа работает только в предположении о том, что мы начинаем только с конкретного места слова (с начала слова). Если головку поставить, например, на конец слова, то эта схема работать не будет.

Можно сделать следующие выводы:

- Машина Тьюринга очень чувствительна к начальному положению головки
- Техника маркеров. Так как у ленты Машины Тьюринга нет номеров ячеек, чтобы отличить одну ячейку от другой, одну скобку от другой можно использовать маркеры (вспомогательные символы). Например, на одну скобку помечать символом X, а другую – Y. Они будут означать разные смыслы. Альтернативный метод – увеличить количество состояний (будет рассмотрен позднее)

Важной особенностью формализации, которой, возможно, не ожидал даже сам Тьюринг, является возможность универсального вычислителя. Можно построить Машину Тьюринга, которая будет выполнять роль процессора в телефоне, то есть сможет брать любую другую машину и делать то же самое, что и она или решать любую задачу.
Тьюринг понял, что сама программа (таблица) также является данными. Таким образом, можно построить универсальную машину, которая в качестве данных будет использовать не только какой-то вход другой машины, но и программу этой машины, записанной определенным образом. По сути, в памяти компьютера хранится программа, записанная определенным образом, а задача процессора -выполнить эту программу. Процессор и есть универсальный вычислитель, и мы будем строить его. Но перед эти нам предстоит пройти еще несколько шагов: нормальные машины и машины с укороченными инструкциями. Объединив все эти машины, получим некоторый ограниченный класс машин Тьюринга, по мощности не уступающий всем машинам Тьюринга.

Любую МТ можно перестроить таким образом, что она будет ту же функцию и удовлетворять следующим условиям:

1. В начальном состоянии q_0, УГ установлена напротив пустой ячейки, которая следует за всеми исходными символами (рис. 2.3)

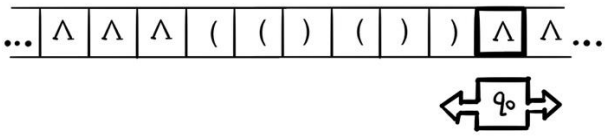


рис. 2.3 УГ напротив пустой ячейки

2. В состоянии q_s УГ установлена напротив пустой ячейки, которая следует за всеми символами результата (рис. 2.4)

МТ, удовлетворяющая условиями (1) и (2), называется *нормальной МТ*

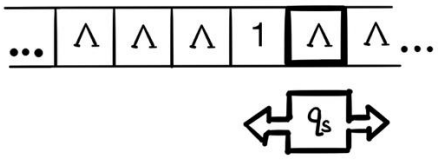


рис. 2.4 УГ напротив пустой ячейки в q_s

Как мы уже отметили, Машина Тьюринга очень чувствительна к первоначальному положению головки. Если у нас есть много Машин Тьюринга, скомпонованных в одну большую Машину Тьюринга (своего рода задача композиции) и для одной программы нужно, чтобы УГ стояла слева от слова, а для другой – справа от слова, а для третьей – нужно, чтобы еще как-то была записана программа, то очевидно, компоновать будет неудобно. Если одна программа закончилась, одна Машина Тьюринга закончилась, где-то на ленте болтается УГ и ее нужно поставить еще в нужное место для второй программы, необходимы какие-то связки между этими программами, подготавливающие УГ к новой программе. Этого можно избежать путем таких договоренностей как Нормальная МТ.
Нормальная МТ всегда начинает на первом пустом символе справа от входного слова и заканчивать на первом пустом символе справа от выходного слова. Это позволяет применить к одному слову подряд сразу несколько машин (композицию), так как позиции УГ на входном и на выходном слове буду совпадать.

Другим введением для упрощения построения композиций стала МТ с лентой, ограниченной с левого конца.
Чтобы построить МТ Т с ограниченной лентой слева,

1. Перегнем ленту по ячейке с номером 0 (рис.2.5)

2. Раздвинем ячейки правой части ленты: символ из ячейки номером n > 0 перепишем в ячейку с номером $2 \times n$
3. В освободившиеся ячейки с нечетными номерами перенесем содержимое ячеек правой части ленты: символ ячейки с номером n < 0 перепишем в ячейку с номером $2 \times |n| - 1$ (рис. 2.6)

МТ ограниченная слева нужна не столько для композиции, сколько для построения Универсальной Машины Тьюринга, потому что от машины, ограниченной с одного конца, и бесконечной с другого конца можно отрезать кусок ленты и выделить его под служебные данные, а на ленте бесконечной с двух сторон это невозможно. Такое преобразование сохраняет мощность машины, потому что натуральные и целые числа равны по мощности.

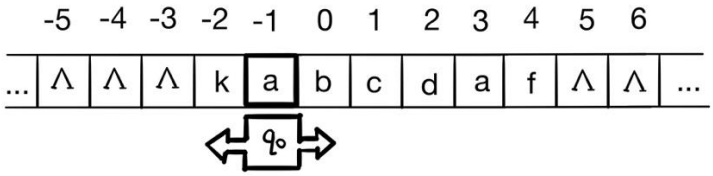


рис. 2.5 бесконечная лента, конфигурация T

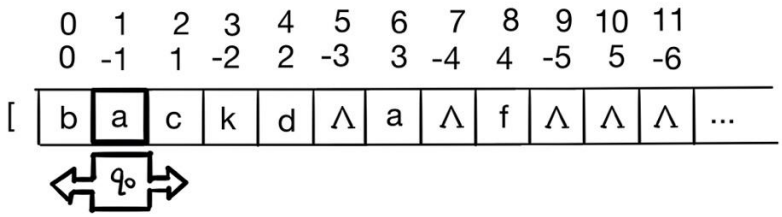


рис. 2.6 лента, ограниченная с левого конца, конфигурация T'

В конфигурации T' передвижение машины будет происходить по следующим правилам:

T	T' (четные)	T' (нечетные)	T' (ячейка 0)	T' (ячейка 1)
вправо	на 2 вправо	на 2 влево	на 2 вправо	на 1 влево
влево	на 2 влево	на 2 вправо	на 1 вправо	на 2 вправо

Но ведь мы не знаем номера ячейки. Как узнать четность ячейки? Для этого можно манипулировать либо множеством состояний, либо множеством действий (как в примере с последовательностью из открывающихся и закрывающихся скобок, где мы зарывающиеся скобки помечали символом X). Прием, где мы добавляем состояние, называется *размножением состояний*.

Допустим, изначально у нас было состояние q. В котором мы что-то делали. Для того, чтобы производить разные действия в зависимости от четности ячейки размножим это состояние на 2 других: q^+ и q^-, которые определяют соответственно состояния в четной и нечетной ячейках. Таким образом количество состояний в программе увеличивается в 2 раза.

Как узнать, что мы находимся в нулевой ячейке? В старой МТ такой проблемы вообще не было, так как лента была бесконечна. Это вопрос договоренности. Мы можем сказать, что не рассматриваем такие машины, то есть в описанных нами частично вычислимых функциях не бывает машин Тьюринга с такими программами, где возможно «упасть» с левого (конечного) края. Либо можно добавить специальное состояние q_H (halt) и сказать, что этот класс МТ мы не рассматриваем, либо говорим про него что-то специальное. Мы будем считать, что таких МТ просто нет.
Мы можем пометить нулевой (крайний) элемент каким-то маркером.

Для того, чтобы было удобнее впоследствии компоновать программы, превратим нашу МТ в *МТ с укороченными инструкциями.* Если за такт будет происходить 2 действия, (как, например, в программе с открывающимися-закрывающимися скобками, УГ писала символ и затем сдвигалась вправо в одном и том же состоянии), будет неудобно рисовать диаграмму Тьюринга.
Рассмотрим произвольную инструкцию МТ T: q, S, R
Разобьем ее на 2 инструкции:
q, a $\rightarrow$ q”, b, s только записывает символ
q’’, b $\rightarrow$ q’, b, R только сдвигает головку
Можно доказать, что для любой МТ T можно построить МТ T’, каждая инструкция которой либо только сдвигает, либо только записывает символ в РЯ.

Далее будем рассматривать класс МТ, который содержит только МТ с укороченными инструкциями и лентой, ограниченной слева. Кроме того, будем считать, что МТ, принадлежащие рассматриваемому классу, выполняют нормальные вычисления по Тьюрингу. Все эти предположения не являются ограничением общности, так как по произвольной МТ нетрудно построить МТ рассматриваемого класса. Основным преимуществом рассматриваемого класса МТ является возможность ввести понятие *действия.*

$$v = \{L, R, H, s_i \in S\}$$

Действие становится атомарным, то есть машина либо сдвигается, либо записывает символ в конкретном состоянии. Теперь можно начать построение сложных машин как композицию простых машин. Каждая элементарная машина выполняет одно из действий v. Наконец назовем МТ именем, чтобы далее использовать это имя в качестве «имени функции».

Запись символа в РЯ или сдвиг УГ вправо или влево называются *элементарными действиями*.
Чтобы записывать элементарные машины, одновременно с принципом композиции будем вводить принцип записи композиции.
У нас есть машина, которая только сдвигается на один символ влево, вправо или записывает символ a_i из алфавита A.

Элементная МТ	Программа	Диаграмма
l	$q_0\Lambda \to Lq_1, q_0a_1 \to Lq_1, \dots, q_0a_p \to Lq_1$	$\bullet\, l \,\bullet$
r	$q_0\Lambda \to Rq_1, q_0a_1 \to Rq_1, \dots, q_0a_p \to Rq_1$	$\bullet\, r \,\bullet$
a_i	$q_0\Lambda \to a_iq_1, q_0a_1 \to a_iq_1, \dots, q_0a_p \to a_iq_1$	$\bullet\, a_i \,\bullet$

Их программа очень простая: всегда в машине 2 состояния, последнее состояние – состояние останова. В машинах l, r программа такая: что бы мы не видели в состоянии q_0, сдвигаемся налево или направо и останавливаемся. В машине a_i, несмотря на то, что находится в РЯ, программа записывает символ a_i и останавливается. У каждой элементарной машины есть им. Имена этих машин l, r или просто символ. Начальное и конечное состояние будут отображаться двумя точками ••.

Для того, чтобы перейти от элементарных диаграмм к менее элементарным, нам нужна вариативность действий в зависимости от содержания РЯ. Для этого нужно к • конечного состояния прикрепить следующую машину. Например, мы хотим построить машину, которая будет 2 раза сдвигаться влево. Для этого рисуем следующую диаграмму: $\bullet\, l \bullet\bullet\, l \,\bullet$. Допустим у нас есть алфавит $A = \{0,1\}$, и мы хотим, если смотрим на 0, сдвинуться влево, если смотрим на 1, сдвинуться вправо. Вариативность реализуется следующим образом (рис. 2.7):

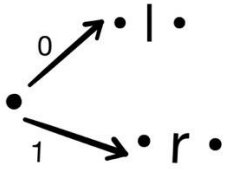


рис. 2.7 диаграмма состояний с разветвлением

Также можно воспроизвести цикл, например, сдвигаться влево, пока в РЯ не пусто. Это реализуется с помощью машин L, R, которые пропускают все слово (рис. 2.8).

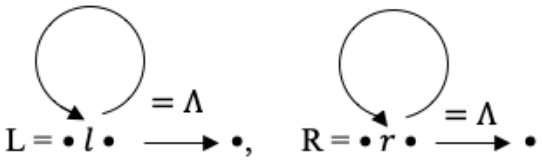


рис. 2.8 диаграмма состояний с циклом L, R

МТ L переводит конфигурацию на рис. 2.9 в конфигурацию 2.10

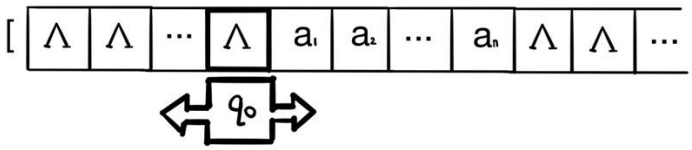


рис. 2.9 начальная конфигурация L

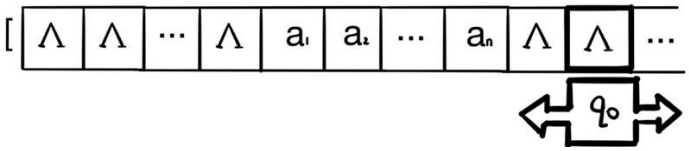


рис. 2.10 конечная конфигурация L

Чтобы не записывать целиком эти циклы, можно просто использовать буквы L и R. Построим машину К, которую так же далее будем использовать как подпрограмму. МТ К переводит конфигурацию на рис. 2.11 в конфигурацию на рис. 2.12:

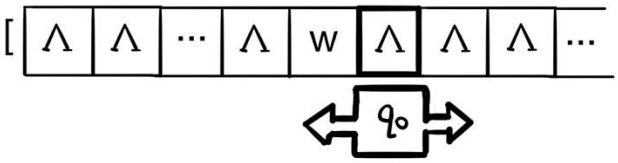


рис. 2.11 начальная конфигурация K

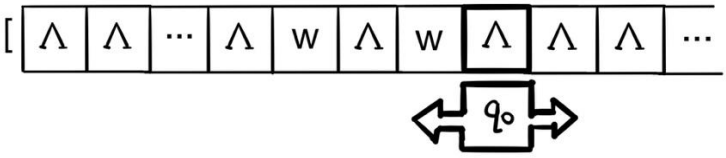


рис. 2.12 конечная конфигурация L

Машина К задает инструкцию для копирования слова (рис. 2.13). По соглашению НМТ машина начинает работу с первой правой пустой ячейки входного слова. Для того, чтобы скопировать слово, сначала необходимо попасть на первый символ слова, то есть крайний левый. Для этого запускаем цикл L, тем самым переходим в первый пустой символ перед началом слова. Далее идет цикл копирования, который организован следующим образом: переходим вправо и попадаем на символ слова, который мы будем копировать. Допустим наш алфавит A = {0,1, Λ} состоит из 2 символов и лямбды, означающей конец слова. Копируем символ и движемся вправо, пропуская все ячейки, не являющиеся Λ. Переносим скопированный символ в следующую после м пустую ячейку и таким образом строим копию слова.

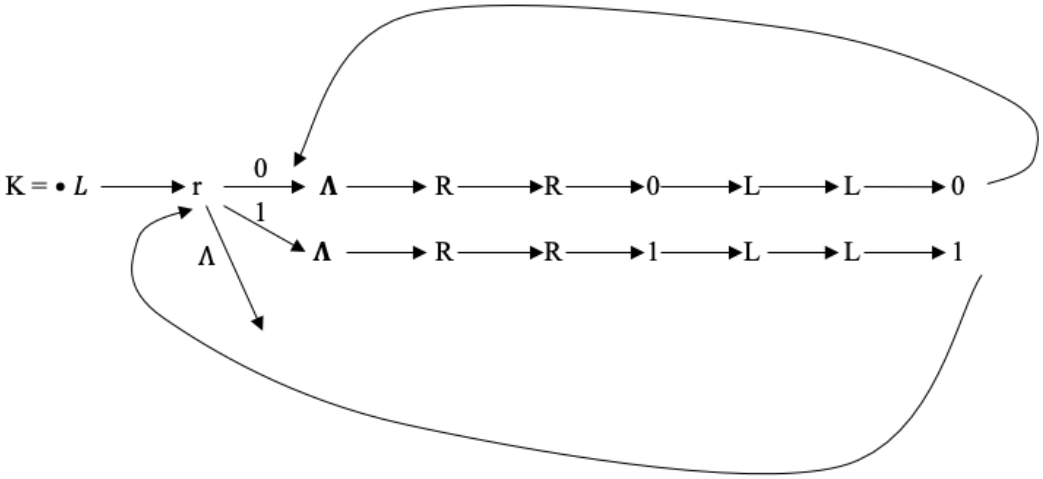


рис. 2.13 диаграмма K

Первая проблема, которая у нас возникает – формально мы не умеем копировать и присваивать символы, то есть не можем записать в пустую ячейку символ входного слова, но мы можем воспользоваться размножением состояний, которое обсуждали ранее. Поэтому в диаграмме мы используем разветвление в зависимости от символа (рис. 2.13). Дальше нам нужно пропустить все входное слово до первого пустого символа. Воспользуемся ранее описанной программой R. После этого потребуется пропустить символы уже построенной части копии (если только это не первая итерация), то есть снова воспользоваться R. Наконец мы нашли ячейку для записи скопированного символа. Чтобы записать символ воспользуемся элементарным действием записи символа, например, 0.

Теперь нам нужно вернуться обратно. Но как запомнить ячейку, откуда мы только что скопировали символ? Очевидно, нужно пометить ее маркером, например, X и только потом уже искать место для копии этого символа. После построения символа копии мы должны вернуться к символу-маркеру с помощью L, заменить маркер обратно на символ, который мы копировали, чтобы не испортить слово. Чтобы упростить

программу, можем в качестве маркера использовать не X, а Λ, так как программы L и R по сути ищут этот маркер на ленте.
Большую диаграмму можно сократить до следующего вида (рис. 2.14).
Если алфавит будет состоять из 3 символов, то в диаграмме программы K появится еще одна ветвь для 3 символа и так далее - с увеличением числа символов во входном алфавите увеличивается и количество ветвей в диаграмме.

Программа, записанная в виде таблицы, эквивалентна программе, записанной в виде диаграммы. Чтобы перейти от диаграммы к таблице:

1. Заменим упрощенную диаграмму полной
2. С помощью индексации добиваемся того, чтобы каждый символ МТ входил в диаграмму один раз
3. Сопоставим каждому символу МТ ее таблицу (таблицу запишем в виде набора соответствующих инструкций).
4. Перепишем все таблицы одну за другой (в любой последовательности)
5. Добавим в таблицу следующие строки:
   5.1 для каждого символа a входного алфавита, которому соответствует стрелка, ведущая из точки снова к ней же, добавим строку $q_0 a \to a q_0$
   5.2 для каждого символа a входного алфавита, которому не соответствует никакая стрелка, ведущая их точки к символу МТ M, добавим строку $q_0 a \to a q_{M_0}$
   5.3 для каждого символа a входного алфавита, которому не соответствует никакая стрелка, ведущая их точки, добавим строку $q_0 a \to H q_s$
   5.4 если два символа МТ M и M’ соединены стрелкой, над которой надписан символ a, то для состояния останова q_{Ms} из части таблицы, соответствующей M, добавляем строку $q_{Ms} a \to a q_{M'0}$ (аналогично для стрелки в состоянии останова)

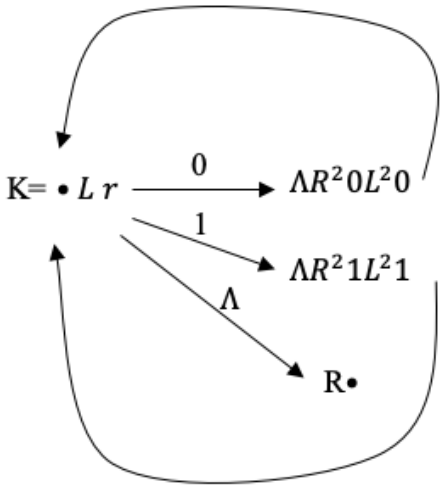


рис. 2.14 сокращенная диаграмма K

Лекция №3. Универсальная машина Тьюринга и нормальные алгоритмы Маркова

Моделирование машины Тьюринга

МТ М моделирует МТ М', если:

1. Данная машинная конфигурация вызывает машинный останов / переход за край ленты МТ **М** после конечного числа шагов тогда и только тогда, когда начальная конфигурация вызывает машинный останов / переход за край ленты МТ **М'** после конечного числа шагов.
2. Для последовательности (c_n') текущих конфигураций МТ М' для данной начальной конфигурации можно указать моделирующую подпоследовательность (c_n) последовательности текущих конфигураций МТ **М** для той же начальной конфигурации**:** для каждой конфигурации c_i' машины **М'** ее лента будет «частью» ленты конфигурации c_i машины **М**, УГ машины М будет находиться на ячейке, соответствующей положению рабочей ячейки машины **М'**, и по конфигурации c_i можно указать состояние машины **М'** в конфигурации c_i'.

Универсальная машина Тьюринга

Универсальная машина Тьюринга – машина, которая может промоделировать любую другую машину.

Универсальной Машиной Тьюринга (УМТ) для алфавита A[1] называется такая машина U, на которой может быть промоделирована любая МТ над алфавитом А.
[1]На самом деле можно эффективно построить УМТ, моделирующую любую МТ над любым алфавитом. Для этого фиксируется некоторый алфавит (например, $A_2 = \{0, 1\}$) и добавляется кодирование и декодирование.

Идея УМТ заключается в следующем:
На ленту УМТ записывается программа моделируемой МТ (таблица) и исходные данные моделируемой МТ. УМТ по состоянию и текущему символу МТ находит на своей ленте команду моделируемой МТ, выясняет, какое действие нужно выполнить, выполняет его.

Теперь должно быть понятно, чем нам удобна МТ с ограниченной слева лентой. Слева мы можем хранить программу, а дальше данные моделируемой МТ. Рассмотрим представление программы моделируемой МТ Т:

- Рабочий алфавит A_p

- Состояния $q_0, q_1, \dots, q_s$
- Правила $q_i a_j \rightarrow v_{ij} q_k$, где i, k = 0, …, n; j = 1, …, p; $v_{ij} \in \{a_1, a_2, \dots, a_p, l, r, h\}$

Универсальная машина:

- Алфавит $B_p = \{b_1, b_2, \dots, b_p\}$
- Дополнительные символы {l, r, h, +, -, O}

Для правила $q_i a_j \rightarrow v_{ij} q_k$ запись выглядит так

$$\begin{cases} b_j v_{ij} + {}^{k-i}, \text{если } k > i \\ b_j v_{ij} O, \text{если } k = i \\ b_j v_{ij} - {}^{i-k}, \text{если } k < i \end{cases}$$

$+ {}^{k-i}$ означает символ +, повторенный k-i раз
Правило $q_i a_j \rightarrow v_{ij} q_k$ укороченное и означает, что любой паре состояние-символ соответствует инструкция, описывающая действие (сдвиг или запись символа) и переход в новое состояние. B_p – это зеркальный алфавит для исходного алфавита, то есть каждый символ из алфавита является отличительной копией символа из исходного алфавита. Зеркальный алфавит используется для того, чтобы определить, что будет записываться на ленте слева, а что справа.
Рассмотрим укороченный пример. Допустим у МТ есть два состояния: q_0, q_1 и ее алфавит содержит два символа: 0,1. Соответственно есть группа из 4 правил:
$q_0, 0$
$q_0, 1$
$q_1, 0$
$q_1, 1$

Эта группа легко делится по состояниям на $q_0, 0$; $q_0, 1$ и $q_1, 0$; $q_1, 1$. Для каждого состояния нужно понять, как записать слово на ленте, соответствующее описанию этих правил. Как для пары состояние-символ записать инструкцию с новым состоянием? Можно просто записать номер этого состояния, для этого нужно завести какую-то систему счисления, закодировать все эти номера в этой системе счисления и просто записать номер. Тогда, чтобы понять, где на ленте будет запись для следующего состояния нужно будет как-то этот номер раскодировать, что не очень просто. Поэтому пользуются, по сути, относительной адресацией. То есть, если мы записываем для состояния q_2 правило: $q_2, 0 \rightarrow 0, q_4$. То есть мы в слове, где есть группа правил для q_2, записываем в зеркальном виде символ 1 как $\tilde{1}$, потом записываем, что нужно сделать: либо символ из исходного алфавита, либо действие. Дальше нужно указать, что мы попадем в четвертое состояние q_4. Для этого просто записываем, насколько 4 больше 2. Если разница положительна, ставим нужное количество «+», если отрицательна –

нужное количество «-», если разницы нет, ставится 0. В нашем случае достаточно поставить два «+».
В итоге получаем запись одного правила: $\tilde{1}0{+}{+}$

Пример.
$q_0, 0 \quad 0, q_0$
$q_0, 1 \quad 0, q_1$
$q_1, 0 \quad \text{r}, \ q_1$
$q_1, 1 \quad 1, q_0$

Записываем все правила в порядке номеров состояний, то есть сначала записываем все правила для состояния q_0: $\underbrace{\tilde{0}\emptyset O}\tilde{1}\emptyset+$

> Для 0 записываем 0 ($\emptyset$) и остаемся в том же состоянии (разница между состояниями равна 0, то есть О)

Аналогично для q_1: $\tilde{0}rO\tilde{1}1-$
Это слово описывает все правила для состояния q_1. Чтобы эти слова удобно отделить на ленте, используем маркер, например, С.
В общем виде:
Слово-программа: $cw_0cw_1 \ldots cw_n$§, где w_i – слово с записью подряд всех правил состояния q_i

- Слова правил разных состояний отделяются друг от друга вспомогательным маркером с.
- Вся программа заканчивается маркером §.

Лента в начальном состоянии (рис. 3.1):

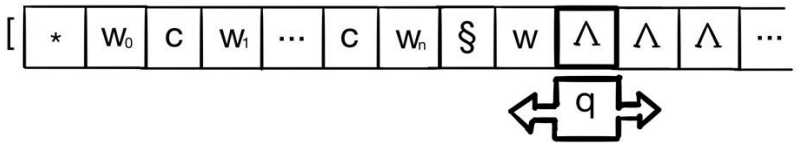


рис. 3.1 Лента в начальном состоянии

w- исходные данные моделируемой МТ
* - маркер начального состояния

Для интерпретации моделируемой МТ необходимо сделать следующее:

1. Найти правила для выполнения
   1.1 «Запоминаем» обозреваемый символ a_i размножением состояний
   1.2 Заменяем символ a_j на его зеркальную пару b_j

1.3 Ищем слово w_i, содержащее запись правила
1.4 Ищем запись правила для символа a_j

$$cw_0cw_1 \dots * \overbrace{b_0v_{io} + + \cdots \underbrace{b_j}_{\uparrow} v_{ij} - - - \cdots}^{w_i} cw_s\S a_{t1}a_{t2} \dots \overbrace{b_j}^{a_j} \dots a_{tw}\Lambda\Lambda \dots$$

2. Изменить текущее состояние моделируемой МТ
   2.1 Сдвигаемся на один символ вправо, пропуская v_{ij}
   2.2 По описанию сдвига пропускаем соответствующее количество символов-маркеров **c** и ставим маркер текущего состояния *****
   2.3 Возвращаемся на символ описания v_{ij} действия (см. рис 3.2)

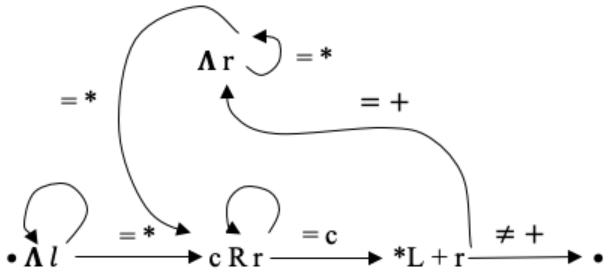


рис. 3.2 диаграмма изменения текущего состояния

3. Выполнить действия моделируемой МТ
   3.1 Ищем ячейку ленты, на которой находится УГ моделируемой МТ
   3.2 Выполняем считанное действие (запись символа или сдвиг [2])
   [2]Если при сдвиге УГ попала на символ §, отделяющий программу моделируемо МТ от данных, это означает, что моделируема МТ зашла за левый край ленты
4. Перейти на выполнение нового такта
   По сути, ничего не делаем, так как машина уже готова вернуться в пункт 1.

Это и есть процесс выполнения основного цикла интерпретации моделируемой машины. Находясь в состоянии, когда справа на данных моделируемой машины необходимо найти правило для выполнения, запоминаем размножением состояний обозреваемый символ, заменяем его на зеркальный, чтобы иметь возможность в него вернуться, передвигаемся налево до *, передвигаемся направо до запомненного символа. Таким образом находим правило. Потом нужно выполнить сдвиг звездочки на нужное количество **c** направо, либо налево в зависимости от + или -, либо остаться на месте, если 0. После этого в зависимости от действия вернуться на зеркальный символ, который был до этого справа от §, записать этот символ, либо сдвинуться и заменить

этот символ на незеркальный, обычный символ из алфавита A_p. На этом цикл закончился.
Мы выполнили основную работу по построению универсальной машины. Теперь нужно понять, как выйти из цикла, то есть сделать останову.
Если при сдвиге маркера текущего состояния (шаг 2.2) происходит переход на символ §, то следующим состоянием будет являться состояние останова. В таком случае УМТ нужно выполнить действие моделируемой машины, а потом остановиться. Лента в состоянии останова будет выглядеть следующим образом (рис.3.3). С остановом все просто: нужно либо записать H (вместо L или R), либо присваиваем состоянию останова самый большой номер. Например, на рис. 3.3 у нас два состояния q_0 и q_1, состоянию останова присваиваем q_2 и кодируем его так же, как и другие состояния, разницей из + и -. Далее, если мы попытаемся перейти в состояние остановы и будем сдвигать маркер * через **c**, в какой-то момент мы будем искать следующее **c**, которой не будет, потому что мы упремся в §. То есть мы захотим взять следующее слово, а его не будет. Это и будет признак остановы. В таком случае нужно выполнить последнее действие, иначе мы его потеряем и остановиться. В результате работа машины закончится на первой пустой ячейке благодаря нормальным вычислениям, а * исчезнет. Если бы мы кодировали состояние остановы символом H, мы могли бы оставить по желанию *.

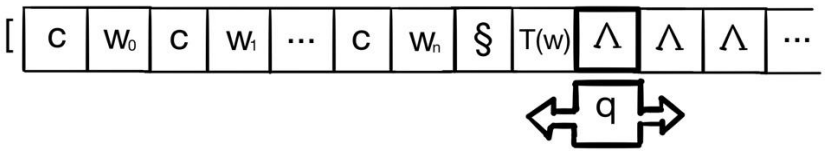


рис. 3.3 лента в состоянии останова

Таким образом мы доказали существование Универсальной Машины Тьюринга, построив ее.

Проблема останова и самоприменимости

Существует ли алгоритм, определяющий, произойдет ли когда-либо останов машины T, запущенный на входных данных w? Или иначе, остановится ли универсальная машина Тьюринга, моделирующая МТ T на входных данных w?

Утверждение. Проблема останова алгоритмически неразрешима.
Алгоритмическая неразрешимость означает, что мы не можем построить МТ, которая отвечает на этот вопрос для *всех возможных МТ* для *всех возможных входов*.
Неразрешимость доказывается от противного.

Пусть существует машина D, решающая проблему останова для всех МТ T и входных данных w. Построим машину E, которая по данным МТ T запускает машину D для МТ T и записи (описания) T на ленте.
Останавливается ли Машина E* (рис. 3.4), будучи примененной к описанию самой себя (то есть описанию машины E*)?
Считаем, что программа и вход записаны одним алфавитом (если нет, вопрос решается кодированием). Если проблема останова алгоритмически разрешима, машина D работает конечное время и выдает 1, либо 0.

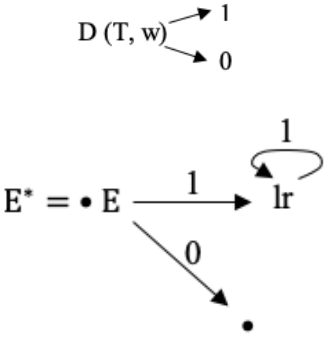


*рис. 3.4 машина E**

Но машина D достаточно сложная, так как имеет 2 входа. Давайте возьмем машину с 1 входом. Если у нас алфавиты совпадают, то мы можем взять машину E(T), которая задаются сразу МТ и запускает машину над МТ, используя функцию p(T), которая строит описание этой МТ. То есть вход для этой МТ — это описание самой МТ.
E(T) = D(T, p(T))
Теперь построим машину E* и попробуем применить ее к описанию самой себя: E*(p(E*)). Пусть машина E сказала, что машина E*, примененная сама к себе, останавливается, то есть машина E выдала 1. Глядя на диаграмму машины E* (рис. 3.4), мы понимаем, что в этом случае машина E* зацикливается. Это означает, что машина E не могла сказать, что машина E* останавливается, то есть она должна была выдать 0. Но если был выдан 0, то машина E* останавливается по построению. Противоречие. Значит, машина E не может существовать, следовательно и машина D не может существовать.
Доказывая неразрешимость проблемы останова, мы коснулись парной задачи проблемы останова – проблемы *самоприменимости*.

Машина Тьюринга T называется *самоприменимой*, если она останавливается, когда в качестве входного слова для нее используется описание самой машины T.
Как и ранее, будем считать, что с помощью кодирования описание задано во входном алфавите нашей машины.
Проблемой самоприменимости является вопрос о существовании алгоритма, определяющего самоприменимость любой заданной машины T.

Алгоритмическая неразрешимость проблемы самоприменимости может быть доказана тем же способом, что и неразрешимость проблемы останова: такой машиной является машина E из доказательства неразрешимости проблемы останова.

Нормальные алгоритмы Маркова

Другим примером формализма наряду с МТ являются Нормальные Алгоритмы Маркова (НАМ).
НАМ задаются следующим образом:
V – алфавит основных символов
V' – алфавит символов-маркеров
$\sigma, \sigma' \in V \cup V'$
Подстановка $\sigma \to \sigma'$ переводит слово $\tau = \alpha\sigma\beta$ в слово $\tau' = \alpha\sigma'\beta$, где $\tau, \tau', \alpha, \beta \in V \cup V'$.
Как слова α и β, так и слова σ и σ' могут быть пустыми.
Метасимвол → отделяет левую часть подстановки от правой.
Исполнитель в качестве одного такта работы делает *подстановку* (рис.3.5).

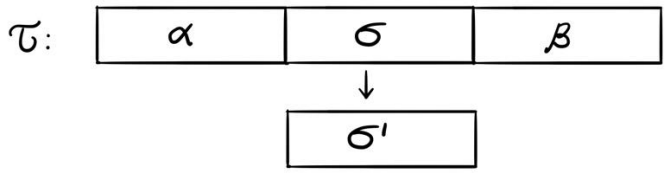


рис. 3.5 подстановка

Пример. Входное слово: ВМК

Подстановка: М → Ми	Подстановка: К→	Подстановка: → МГУ
Выходное слово: ВМиК	Выходное слово: ВМ	Выходное слово: МГУВМК

Нормальный алгоритм Маркова (НАМ) задается конечной последовательностью подстановок $p_1, p_2, \dots, p_n$
«Такт» работы алгоритма состоит в поиске подстановки применимой к текущему обрабатываемому слову:

- Поиск применимой подстановки ведется, начиная с левой подстановки в последовательности;
- Если ни одна подстановка не оказалась применимой, алгоритм завершается;
- Первая найденная применимая подстановка применяется: заменяется самое левое вхождение слова из левой части подстановки;
- Подстановка может быть применена как *терминальная,* тогда после ее применения алгоритм завершается

Терминальная подстановка обозначается как → • или⟼
Подстановки записываются в столбик и применяются по очереди. После выполнения какой-то подстановки процесс начинается с начала. Конец наступает, если ни одна из подстановок не применима, либо если сработала терминальная подстановка.

Формально: пусть задано входное слово $\sigma_0 \in (V \cup V')^*$ и набор подстановок $p_1, p_2, \dots, p_n$.

1. Положить i = 0
2. Положить j = 1
3. Если подстановка p_j применима к слову σ_i, перейти к шагу 5
4. Положить j = j + 1. Если $j \leq n$, то перейти к шагу 3, иначе остановиться
5. Применить подстановку p_j к слову σ_i и построить слово σ_{i+1}. Если p_j – терминальная подстановка, то остановиться, иначе положить i = i + 1 и перейти к шагу 2.

Говорят, что НАМ *применим* к слову σ_0, если в результате выполнения описанной процедуры интерпретации произойдет остановка.

Лекция №4. НАМ и введение в язык С

Рассмотрим пример НАМ: *Шифр Юлия Цезаря.*
i-я буква латинского алфавита шифруется (i+c) mod 26-буквой, где i – номер буквы (начиная с нуля), c – некоторая константна.

НАМ: $* a \to d *, * b \to e *, * c \to f *, \dots, * y \to b *, * z \to c *, * \shortmid \to, \to *$.

- Маркер устанавливается в начало слова с помощью подстановки с пустой левой частью (→*)
- Шифрование выполняется одной из 26 подстановок вида $* a_i \to a_{(i+3)} mod\ 26 *$, где $0 \leq i < 26, a + i \in A_{26}$={a,b,..,z}
- Последняя подстановка удаляет маркер из зашифрованного слова (* ǀ→)

Для построения цикла, обрабатывающего символы, в НАМ пользуются маркерами. Так как у нас нет никакого счетчика, чтобы помечать символы, используют механизм маркера, который изначально в слове не присутствует. Либо слева, либо справа от символа ставят маркер и записывают подстановки, содержащие этот маркер.
Вся программа делится на 3 большие части:

1. Обработка маркеров (*)

Обрабатывается символ и применяется определенное правило, * передвигается вправо, помечая новый символ.
Пример: было ade, после первой подстановки стало *ade, после второй подстановки стало $d * de$

2. Останова

Используем терминальную подстановку для остановы, переводя маркер в пустое слово * ǀ→

3. Добавление маркера * в слово

Перед словом с помощью правила →∗ добавляется маркер *. Если подставить это правило первым, получится бесконечный цикл, поэтому важно соблюдать порядок правил.

Другой пример: сложение чисел в единичной системе счисления: V = {+, |}, V' = {}.
Лобовая программа: $|+ \to +|, +| \to |, | \shortmid \to |$.
Более разумный способ - убрать +, схлопнув слово, с помощью правила + →.

Тезис Маркова. Любой алгоритм в алфавите V может быть представлен нормальным Алгоритмом Маркова над алфавитом V.
Примерно так же, как и для МТ, можно доказать алгоритмическую неразрешимость проблемы останова и самоприменимости.

Существуют различные НАМ решения одной и той же задачи. Проблема построения алгоритма, который может определить эквивалентность любых двух НАМ, *алгоритмически неразрешима.*
Можно построить *универсальный НАМ* U, который мог бы интерпретировать любой нормальный алгоритм, включая самого себя.

Подводя итог, можно сказать, что неформальное определение алгоритма для всех формализаций одинакова, а все формализации (МТ, НАМ, машина Поста) одинаковы по мощности.

Отметим также некоторые трудности программирования на МТ:

- Поиск данных осуществляется за линейное время: нет «адресов», которые позволили бы попасть в нужное место ленты за константное время.
- Нельзя «скопировать» символ программой константного размера: размножение состояний приводит к росту программы в зависимости от размера входного алфавита.

Следующим шагом улучшения вычислителя стала Машина фон Неймана (рис. 4.1).

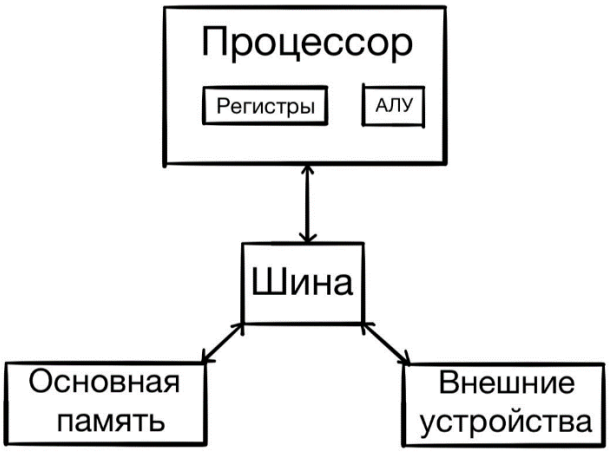


рис. 4.1 Машина фон Неймана

Аналогом автомата в МТ в машине фон Неймана является процессор, который организует все вычислительные процессы. То есть процессор является универсальным вычислителем. В процессоре есть блок арифметико-логических устройств (АЛУ), который выполняет различные операции. В МТ память была представлена в виде бесконечной ленты с непронумерованными ячейками, чтобы получить доступ к которым, мы пользовались маркерами. В машине фон Неймана уже появляется адресуемая память, то есть у ячейки появляется номер, по которому можно посмотреть содержимое или записать содержимое в ячейку за константное время. Шина – способ

обмена информации между процессором и памятью, а также способ взаимосвязи процессора с внешними устройствами. Самое главное в машине фон Неймана – это то, что память содержит как обрабатываемые данные, так и программу. Регистры представляют из себя именованные ячейки памяти, расположенные в процессоре. АЛУ чаще всего работают с именно с регистрами, потому что это быстрее. Существуют 2 основные команды: загрузка –достать данные из основной памяти и положить в регистр и запись –достать данные из регистра и положить в память.
Современные компьютеры похожи по устройству на фон Неймовскую машин, однако обладают несколькими центральными процессорами и большим количеством регистров.

Язык Си

Си разрабатывался как язык для реализации первой в мире универсальной операционной системы UNIX.

История:
1973 – первая версия Си
1978 – выход книги Б. Кернигана и Д. Ритчи «Язык программирования Си». Русский перевод вышел в 1985 году.
1989 – первый стандарт ANSI C (C89)
1999 - стандарт C99
2011 – стандарт C11 (ранее назывался C1X)
_Thread_local, _Generic, _Align*_, _Noreturn…
2018 - стандарт C18 (только исправление ошибок в C11)

Керниган и Ритчи писали на Си Unix для машины PDP-11. Поэтому часть команд в Си появилось прямиком из ассемблера PDP-11.

рис. 3.4 машина PDP-11

После этого началась стандартизация языка Си. Первый стандарт появился в 1999 году – C99, который мы и будем разбирать.
Приведем несколько причин популярности Си:

- Императивный язык
- Удобный синтаксис
- Позволяет естественно оперировать машинными понятиями
- Переносимость на уровне исходного кода, конфигурируемость

То есть на разных архитектурах свои компиляторы, и программа имеет разные типы, которые на разных архитектурах превращаются в разные понятия. И это нужно учитывать.

- Хорошие системные библиотеки
- Хорошие оптимизирующие компиляторы

Рассмотрим по традиции, заложенной Кернигеном и Ритчи, нашу первую программу “Hello, world”:

```
#include <stdio.h>

int main(void) {
        printf("Hello, world\n")
        return 0;
}
```

Поскольку у нас фон Неймовская архитектура, память адресуется. Но напрямую по адресу в Си мы обычно не обращаемся, а создаем имя. В дальнейшем имя может обозначать указатель, структуру, массив и т п. В нашем случае имя обозначает переменную. Кроме того, код подразделен на функции, к функции обращаемся по имени, чтобы в дальнейшем не повторять тело функции. Здесь появляется раздельная компиляция: есть функция main, которая вызывает функцию print, и мы должны суметь скомпилировать эту программу, не зная, как реализована функция print. Зная имя функции и ее аргументы, мы можем организовать вызов этой функции, так как у нас есть некоторая договорённость ABI о том, как называют и вызывают функцию. Благодаря этому и становится возможна раздельная компиляция.
Есть функции, тела которых нам не нужны для компиляции программы – это служебные функции, которые лежат в библиотеках. Чтобы использовать такие функции необходимо указать в заголовочном файле компилятору объявление функции.

Таким образом, программа:

- Объявления переменных или функций
- Определения функций
- Директивы препроцессора
- Системные библиотеки
- Строковые константы
- Управляющие последовательности

Лекция №5. Типы данных в С

Стандартные библиотеки

Вычислительный процесс в языке С представлен в виде функции. Чтобы одна функция смогла вызывать другую необходимо соглашение, чтобы реализовать которое, потребуется как минимум функция и ее аргументы, а также договоренность о количестве функций (больше одной) и их разделении. Тогда у нас есть тела функций, которые содержат код и вызывают другие функции, для которых нет исходного кода. Для организации такого вызова необходимо знать объявление или прототип функции: имя функции, набор параметров и возвращаемое значение. Зная это, компилятор может организовать вызов функции, не зная кода. Но объявление стандартных функций должны где-то лежать. Для этого в Си есть заголовочные файлы, в которых хранятся эти объявления. Соответственно, если это объявление функции, которая находится в системной библиотеке Си, идущей вместе с компилятором, то оно лежит в каком-то файле с фиксированным названием. В нашем примере:

```
#include <stdio.h>

int main(void) {
        printf("Hello, world\n")
        return 0;
}
```

Файл называется stdio.h, в нем лежат функции, которые имеют отношение к выводу и вводу (в данном случае к стандартному свойству вывода некоторой строки).

У нас есть файл, содержащий код этой программы и заголовочный файл stdio.h. Возникает вопрос, где лежит этот файл и что с ним делать? Для того, чтобы компилятор знал, где сложено объявление функции, было принято решение, что компилятор вообще не должен об этом знать, ему должна быть предъявлена программа, где эта функция уже есть, но, как несложно заметить, в нашем примере этих объявлений нет. Поэтому ввели предварительный, нулевой этап компиляции - препроцессирование. Раньше это была отдельная программа, потом ее интегрировали в компилятор. На данном этапе важно понимать следующее:

- Препроцессор работает перед компилятором
- Препроцессор понимает исключительно текст

Команды, которые начинаются с «#» — это директивы. Директива #include <stdio.h> обозначает, что нужно найти файл stdio.h, и в это место вставить вместо директивы содержимое файла, а ее выбросить. Далее есть флаг -E <имя файла>.

В командной строке можно написать gcc -E file.c и увидеть содержимое файла, которое получит компилятор на вход.
Еще раз напомним, что препроцессор чувствителен только к текстовым компонентам. И его не волнуют потенциальные ошибки в программе.

Часть стандарта С - описание функций, реализуемых стандартной библиотекой, приходящей с компилятором. Возможна ситуация, когда нет никакой стандартной библиотеки, и компилятор используется для компиляции базовых вещей. Стандартная библиотека С, как правило, тоже написана на С.

Выполнение программы должно быть начато с функции main, поэтому в стандартной библиотеки С есть компонента, разная для различных архитектур, которая, инициализирует структуры в стандартной библиотеке до запуска функции main бинарного файла программы, взводит все константы и вызывает функцию main. Компонент, который называется компоновщик, из разных частей стандартной библиотеки и текущей программы должен составить одну программу и проверить, все ли заявленные функции определены. Далее стандартная библиотека требует, чтобы функция main была определена. Если она не будет определена, то возникнет ошибка при компоновке.

Строковые константы все оформляются через двойные кавычки. Чтобы напечатать обычный символ, достаточно записать его в кавычках. То есть символы, хранящиеся в файле и задающие константу, отображаются, по сути, в себя же, если это печатаемые символы. Но в С также можно задать непечатаемые символы, которые выполняют какую-то функцию, например, перевод строки. Для этого нужно завести какие-то символы со спец значением.
Например, обратный слэш « \ » - типичный символ со спец значением, который обозначает, что его печатать не нужно, нужно посмотреть на символ, идущий за ним и превратить эти 2 символа в какой-то служебный символ. \n -перевод строки. Чтобы вывести «\» как символ, нужно написать «\\». Иногда это называется «управляющей» последовательностью, а не спецсимволом.

Поскольку функция main выполняется первой, то по соглашению ее возвращаемое значение принимается за возвращаемое значение всей запущенной программы. По соглашению у запущенной программы есть код возврата для того, чтобы по каким-то внешним признакам понять, хорошо ли отработала программа. Нулевой код возврата означает, что все хорошо, ненулевой возвращает разные классы ошибок. По сути, то, что возвращается из функции main становится кодом возврата программы, поэтому всегда в возвращаемом значении функции main всегда пишется 0.

Си-машина и классы памяти

Рассмотрим архитектуру машины Си (рис. 4.1). Главным отличаем архитектуры машины Си от архитектуры машины фон Неймана является расслоение основной памяти из-за различных типов данных. Есть данные, про которые мы точно знаем, когда они нужны, — это называется «время жизни» или scope. Есть данные, про которые мы точно не знаем, сколько времени они будут жить.
Данные, про которые мы точно все знаем, так же можно разделить на 2 категории: данные, которые нужны во время всей работы, и данные, которые нужны только в определенный момент. В Си, первый тип данных, живущих на протяжении всей программы, называется – статические данные. Статические данные также длятся на 2 типа: сама программа и константы, например, константная строка "Hello, world".
Данные, которые нужны только в определенный момент, хранятся в локальной (автоматической) памяти, которая управляется компилятором. Обычно этот тип данных используется при вычислениях в функциях с локальными переменными. Именно поэтому, чтобы избежать многократного повторения выделения и освобождения памяти для локальных переменных, используют автоматическую память, которой управляет сам компилятор. Эта память хранится в стеке.

Данные, чье время жизни и/или размер нам точно неизвестны, хранятся в динамической памяти, которой управляет программист. Она часто называется кучей (heap).

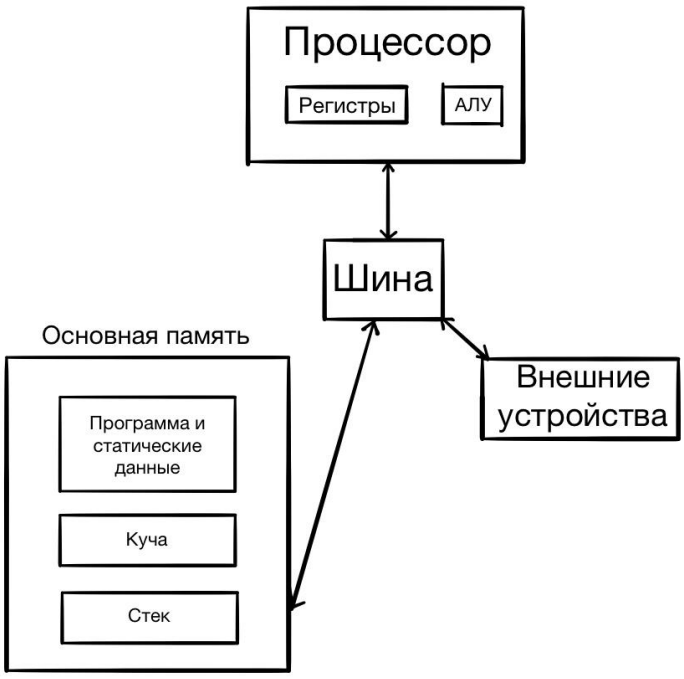


рис. 4.1 архитектура машины Си

Для того, чтобы управлять памятью, нужно завести переменную в эту сегменте памяти, то есть назвать ячейку памяти данного сегмента. Например, локальная переменная функции автоматически попадает в автоматическую память, управляемую компилятором.
Кроме того, есть регистровая память. Для управления регистрами есть специальное расширение языка, которое позволяет положить какое-то значение прямо в регистр, как в ассемблере. Если не пользоваться расширениям, то можно переменной указать регистровый класс памяти с помощью ключевого слова. Это означает, что мы предполагаем, что для этой переменной необходим быстрый доступ (как вы помните, регистры обеспечивают быстрый доступ к данным), и просим компилятор положить эту переменную на регистр. «Просим», потому что регистров может быть меньше, чем переменных или переменная слишком большая.
Этим активно пользовались раньше, потому что

1. регистров было мало
2. были программы-интерпретаторы, в которых была переменная, соответствующая текущей позиции в программе, к которой нужен был постоянный доступ. Поэтому ее просили положить в регистр
3. иногда для работы было необходимо распределение регистров, то есть некоторым переменным присваивался регистр, чтобы команды процессора могли работать.

Базовые типы

Ячейки с адресами в обычной памяти ничем не отличаются точно так же, как и ячейки в машине Тьюринга. С этим не очень удобно работать, потому что иногда важно понять, какие именно символы хранятся в ячейке: буквы ли это, образующие строки, или цифры, образующие числа, или пиксели, образующие изображения, и обрабатывать их по-разному. Поэтому в языке программирования есть *типы данных*.
Типы данных – это, по сути, классификация данных по диапазону значений и действий, которые возможно выполнять над ними. Чем богаче система типов в языке программирования, тем удобнее с ним работать. Система типов обычно устроена следующим образом: есть базовые типы, похожие на данные, которые удобно обрабатывать на процессоре и их модификации.

Базовые типы данных в языке Си:

- символьный – char
- целый – int
- с плавающей точкой – float
- двойной точности – double

- комплексный, C99 - _Complex

Тип без значения – void

Модификаторы базовых типов*:

- signed
- unsigned
- long – 32 бита
- short – 8 бит
- long long (C99) – 64 бита

*При этом:
- К типу int применимы все модификаторы
- К типу char только signed и unsigned
- К типу double только long (C99)

Размер целого типа выбирается максимально приближенным к целому числу в процессоре, то есть это размер регистров, наиболее удобный для процессора. Если регистр 16-битный, то и тип int будет 16-битный. Чтобы получить более короткий (short) или более длинный тип (long или long long), нужно воспользоваться модификацией типа. Кроме того, можно изменить целый тип на знаковый или беззнаковый (по умолчанию он знаковый).
Модификаторы можно указывать в любом порядке. При этом если какой-то модификатор однозначно определяет происходящее, то остальные можно убрать. Например, short int signed можно преобразовать в short, потому что short может быть только int, a int всегда signed.
Но short unsigned нужно указывать, при этом просто unsigned также будет int.

Рассмотрим, что происходит с числами с плавающей точкой. Аппаратно операции с числами с плавающей точкой реализовывать сильно сложнее, чем операции с целыми числами. Поэтому возможности представления этого типа в разных размерах гораздо медленнее развивались. Сначала был только тип float, позднее с развитием мощности процессоров появился тип double, который позволил увеличить точность. Так тип float – тип с одинарной точностью, double – с двойной точностью. С появлением более длинных регистров появился long double.

Тип char – это целый тип, то есть к нему применимы те же действия, что и к целому типу, но у него нет размера (соответственно к нему неприменим модификатор размера). В разных компиляторах char может быть по умолчанию как знаковым, так и беззнаковым. Это зависит от кодировки (implementation defined). У char размер 1 байт, если достаточна половина этого размера, то char можно сделать знаковым.

Тип void обозначает отсутствие значения и обычно используется в функция, чтобы показать, что они ничего не возвращают, или для специального типа указателей.

Представление целых чисел

В Си используется позиционная двоичная система счисления. При этом

- байты в представлении числа идут подряд
- порядок бат не гарантируется, то есть зависит от аппаратуры (big/little endian)
- Порядок бит в байте также не гарантируется (и его может быть невозможно узнать)
- Отрицательные числа часто представляются в дополнительном коде (n бит):
  - самый значащий бит (n-1) является знаковым
  - биты от 0 до n-2 – значение
  - положительные значения представлены как обычно
  - отрицательные значения записываются как $2^n - |n|$

Байт – минимальная адресуемая единица. В зависимости от аппаратуры число может записывать как слева направо – big endian, так и справа налево – little endian. Так, например, число $\underbrace{1111111}_{8}\underbrace{0000\ldots00}_{24}$ в big endian будет представлена как на рис. 4.2, а в Little endian как на рис. 4.3.

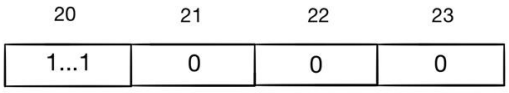


рис. 4.2 big endian

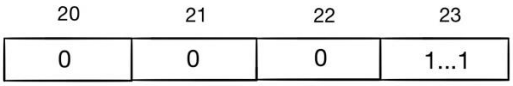


рис. 4.3 little endian

У каждого способа есть как свои преимущества, так и недостатки. Сейчас более распространен little endian. Если по адресу 20 хранится число размером 1 байт, то оно хранится по адресу 20. Если по адресу 20 хранится число размером 2 байта, то оно все равно хранится по адресу 20. То есть число растет вправо. Таким образом, мы можем прочитать 2 байта по адресу 20 и получить младшую часть числа, а можем по этому же адресу прочитать 4 байта, и начало сдвигать не придется.

Плюсом big endian является то, что можно считать первые 2 байта числа и понять, насколько оно большое.
Порядок бит в байте нам неважен, так как минимальной адресуемой единице является байт.

Чтобы хранить знаковые числа, нужно выделить место для кодирования знака. Для хранения знака необходим 1 бит (рис. 4.4).

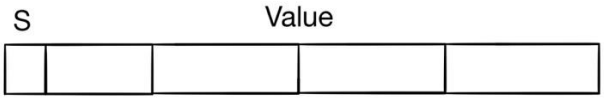


рис. 4.4 представление числа со знаком

0 в знаковом бите означает, что число без знака, 1 – число со знаком. Далее идет запись числа. Но нельзя в обоих случаях использовать прямую запись, так как, например, при записи 0 получатся два разных представления +0 и -0, которые по идее должны означать одно и то же. Поэтому используют дополнительный код для записи отрицательного числа. Дополнительный код, по сути, дополняет число до 2^n. Чтобы построить дополнительный код, необходимо инвертировать исходный код и добавить 1. Такую запись удобно использовать, потому что при сложении прямого и дополнительного кода получается 0.

Размеры типов

В языке Си есть оператор sizeof, который возвращает размер типа любого объекта.
$int\ x\ \rightarrow sizeof(x) == sizeof(int)$
Файл limits.h задает *минимальные* и *максимальные* значения типов.

Важно запомнить следующее:

$$sizeof(char) == 1$$
$$\wedge$$
$$sizeof(short) \geq 2$$
$$\wedge$$
$$sizeof(int) \geq 2$$
$$\wedge$$
$$sizeof(long) \geq 4$$
$$\wedge$$
$$sizeof(long\ long) \geq 8$$

Файл inttypes.h задает знаковые и беззнаковые целые типы фиксированных размеров (8, 16, 32, 64 бита). Фиксированные типы задаются следующим образом: int64 t x; суффикс t обозначает, что это не имя, а имя типа.

Таким образом, sizeof применим как к переменным, так и к типам.
Различные размеры переменных необходимы для того, чтобы язык Си оставался универсальным для различных архитектур.

Как вы заметили, в базовых типах мы не рассмотрели булевый и комплексный типы. Раньше вместо них пользовались типом char, либо описывали свой тип данных. В версии C99 появился тип _Bool и _Complex.
Для того, чтобы использовать тип _Bool (C99, переменная этого целого беззнакового типа принимает значения 0 или 1), необходимо включить stdbool.h для объявления bool, true, false.
Для того, чтобы использовать тип _Complex (C99, float/double/long double), необходимо использовать complex.h для объявления complex I и т.п.
Тип _Imagery (C99) является необязательным.
В C11поддержка комплексный чисел стала необязательной (_ _STDC_NO_COMPLEX_ _).
Булевский тип специально назвали _Bool, чтобы «не портить» программы, в которых программисты сами описывали этот тип и называли bool в течение 25 лет, что существовал язык Си. К тому же, если имя начинается с «_ _» или «_[A-Z]» (с двух подчеркиваний или подчеркивания и заглавной латинской буквы), то оно находится в пространстве имен, зарезервированных за реализацией языка. Поэтому все нововведения в язык делаются таким образом. Чтобы пользоваться булевским типом и не вызывать его так _Bool, можно включить файл stdbool.h для объявления bool, true, false. Таким образом сохраняется обратная совместимость.

Лекция 6. Переменные Ввод - вывод в С

Переменные

Чтобы завести переменную, мы должны определить ее имя, тип и значение, то есть переменная = тип + имя + значение. Каждая такая переменная является частью программы. Стоит отметить, что *ключевые слова* не могут быть именами переменных. Объявление переменной происходит следующим образом: type name [, name, name]; Можно задать класс памяти и начальное значение переменной int a, b; unsigned c = 2019. Ключевые слова – это служебные слова: базовые типы, встроенные в язык, ряд квалификаторов, слова, указывающие на операторы.

Перед тем как написать инициализацию переменной, определим, где мы можем пользоваться этой переменной, то есть определим область действий (scope).

Переменная может быть объявлена

1. Внутри функции или блока (локальная)
2. В объявлении функции (параметр функции)
3. Вне всех функций (глобальная)

Часто класс памяти переменной (storage) путают с областью действий переменных, потому что некоторые области действий автоматически влекут за собой определенный класс памяти, но это не всегда правда. Область действий

- *Локальной переменной* – блок, в котором она объявлена (C99 – начиная со строки объявления)
- *Глобальной переменной* – программный файл, начиная со строки объявления

В одной области действий нельзя объявлять более одной переменной с одним и тем же именем. Чтобы в языке программирования не возникал конфликт имен, можно всегда либо запретить это, либо сделать действие по умолчанию. Например, есть функция с двумя переменными x: глобальной и локальной

```
int x
    {
    int x
        x + y

    }
```

Чтобы здесь разрешить противоречие, нужно взять ближайшую область действий, то есть используется именно локальная переменная.

В таких случаях компилятор может выдавать предупреждение (warning Wshadow), на которые стоит обращать внимание.
Рассмотрим пример, который все это демонстрирует. У нас есть некоторая глобальная переменная count, ее областью видимости является весь файл. В функции func также появляется локальная переменная count. Присваивание, которое отнимает 2 из переменной count глупое, поскольку это действие никогда за пределы функции не выйдет.

```
#include <stdio.h>
int count;          /* global */
void func (void) {
        int count;      /* auto */
        count = count - 2;
}
static int mult = 0; /* static */
int sum ( int x, int y) {
        count ++;
        return (x + y) * (++mult);
}
int main(void) {
        register int s = 0; /* register */
        count = 0;
        s += sum(5, 7);
        func ();
        printf ("Sum is %d func is called %d times\n", s, count);
        return 0;
}
```

Отметим, что к глобальной переменной могут обратиться разные функции, раз разные функции могут обратиться к этой переменной, то компилятор не может управлять ее временем жизни, то есть она не может быть автоматической. Эта переменная должна жить какое-то время, в течение которого работают функции. Но какие функции работают точно неизвестно, поскольку в С раздельная компиляция. То есть можно объявить переменную таким образом, что с ней смогут работать функции из другого файла. Поэтому эту переменную мы вынуждены заводить на время работы всей программы, то есть она попадает в статическую память.

С локальными переменными ситуация обратная. По умолчанию локальная переменная – способ завести ячейку в автоматической памяти. Так в функции func компилятор выделит память для переменной count. Он знает ее размер и тип. Но можно нарушить это умолчание и завести переменную в статической памяти. Для этого есть квалификатор static.

Например,

```
int foo() {
        static int x = 1;
        .
        .
        .
        x= x + 1
}
```

Переменная x локальная, за пределами функции foo ей нельзя воспользоваться. Квалификатор static показывает, что переменная x хранится в статической памяти. По сути, x + x+1 означает, что переменная x будет счётчиком вызовов этой функции. Поскольку это статическая переменная, то после работы функции она никуда не исчезнет и значение тоже останется. При первом вызове функции x = 1, при втором x =2 и так далее. Инициализация же выполнится только один раз, перед началом программы.

Инициализация переменных

В зависимости от класса памяти и нахождения переменной инициализация выполняется по-разному. Так, при объявлении переменной int x = 42;

- Автоматические переменные инициализируются каждый раз при входе в соответствующий блок;
- Если нет инициализации, значение соответствующей переменной нее определено!
- Глобальные и статические переменные инициализируются только 1 раз в начале работы программы;
- Если нет инициализации, они обнуляются компилятором*;
- Внешние переменные инициализируются только в том файле, в котором они определяются;
- При инициализации переменной типа с квалификатором const она является константой и не может менять свое значение.

*Если нет инициализации переменные зануляются. Если регистровый класс памяти, выдаем инструкцию для записи 0 в регистр. Если у нас статический класс памяти, то в ко, который выполняется до начала работы запишем, чтобы статическая память была занулена. А у автоматической памяти свои нюансы. При вызове функции после выделения автоматической памяти нужно записать 0 на всю автоматическую память. Кроме того, это действие должно выполнятся каждый раз при вызове функции. А

функция может вызываться много раз. Решение следующее: если инициализации нет, то автоматическая память не инициализируется, то есть в ней хранится мусор. Поэтому автоматическую память всегда нужно инициализировать самому. Если же все-таки мы прочитаем, что находится в неинициализированной переменной, то столкнемся с неопределенным поведением (undefined behavior).

Компилятор может предупредить, когда глобальная и локальная переменные имеют одинаковое имя. Однако понять всегда ли переменная инициализируется в функции – алгоритмически неразрешимая задача.

Как записывать константы? Инициализация – по сути, присваивание переменной некоторого заранее заданного неизменяющегося значения, то есть константы. Литералы задают константу (фиксированное значение). При этом:

- Символьные константы задаются символом или его кодом в кавычках: ‘c’, ‘L’, ‘\0x4f’, ‘\040’. Тип символьной константы - int!

- Целые константы 100, -34l, 1000U, 999llu, u- unsigned, l (L) – long, ll- long long, порядок суффиксов роли не играет.
- Константы с плавающей точкой 11.123F, 4.56e-4f, 1.0, -11.123, 3.1415926l, -6.62068e-34L. Тип вещественной константы без суффикса f (F) – double! Можно опускать либо целую, либо дробную часть опускать, также запись с мантиссой.
- Шестнадцатеричные константы 0x80 (128). Префикс такой константы 0x.

  Вещественные шестнадцатеричные: 0x3.ABp3 ($3\frac{171}{256} \times 8 = 29.34375$
- Восьмеричные константы 012 (10). Префикс – 0.
- Строковые константы “a”, “Hello, World!”, L“Unicode string”
- Специальные символьные константы (управляющие последовательности) \n, \t, \b

Операции над целочисленными данными

Операции над целочисленными данными обычно группируются по местности, то есть по количеству операндов. Есть операции с 1, 2 и 3 операндами.

Арифметические операции

Одноместные: изменение знака (-), одноместный плюс (+)
Двухместные: сложение (+), вычитание (-), умножение (*), деление нацело (/), остаток от деления нацело (%)

(a / b) × b + (a % b) = = a – формула для определения остатка, остаток может быть меньше 0.

Отношения (результат 0/1 типа int)

Больше (>), больше или равно (> =), меньше (<), меньше или равно (<=)

Сравнения (результат 0/1 типа int)

Равно (= =), не равно (! =)

Логические операции

Отрицание (!), конъюнкция (&&), дизъюнкция (| |), ложное значение – 0, истинное значение – любое ненулевое
«Ленивое» вычисление && и | | означает, что если нужно вычислить выражение с конъюнкцией или дизъюнкцией, то операнды этого выражения вычисляются слева направо, вычислив первое выражение, уже можно предсказать результат операции. Например, если первое выражения равно 0, то второе вычисляться не будет. Таким образом у нас есть гарантия, что первое выражение вычислено. Это можно использовать так:

```
if a != 0  && b/a > 3 {
}
```

Если a = 0, то результат конъюнкции уже известен и b/a вычислено не будет, а если бы оно вычислялось, то мы бы делили на 0, что выдало бы ошибку.

Операции присваивания

В С есть понятие побочного эффекта (side effect). Побочный эффект заключается в изменении памяти, изменении объекта. Присваивание вводит в язык термины «левый» и правый» операнд присваивания, lvalue и rvalue соответственно, описывающие выражения, которые могут стоять слева и справа. lvalue = rvalue.

- lvalue – выражение, указывающее на объект памяти
- rvalue – выражение, генерирующее значение

Присваивание – это не оператор, а выражение, которое генерирует значение, правую часть, rvalue. Это не то же самое, что просто написать rvalue, у присваивания есть побочный эффект, который заключается в том, что в память, на которую указывает lvalue, заносится значение rvalue. Например, в выражении a = 0 значением является 0, а

побочным эффектом является то, что в a будет записан 0. Из-за того, что у выражения есть значение, мы можем писать цепочки. Пример: a = b = c = d = 0;
У них ассоциативность справа налево.

Укороченное присваивание: lvalue op = rvalue, где op -двухместная операция.
Пример: a += 15;
Укороченное присваивание построено на следующей идее: часто адрес результата совпадает с одним из операндов и можно результат записывать в этот операнд и таким образом сэкономить 1 адрес. Если нужно не испортить один из операндов, тогда можно сохранить значение в один из регистров. Таким образом, в выражении a = a + 2 a вычисляется 2 раза, а в выражении a += 2 – только один раз.

Инкремент и декремент тоже реализованы и действуют по логике двухместных операций. Когда люди писали ассемблер, они заметили, что частая операция сложения или вычитания — это сложение (вычитание) с 1. Поэтому для этого случая была придумана своя мнемоника. В ассемблере очень часто бывает инструкция inc и dec. В C это реализовано как ++ и -- соответственно. Инкремент (декремент) означает, что значение операнда должно быть увеличено (уменьшено) на 1 и записано обратно в операнд.
Если операция требует записи операнда, то этот операнд должен быть lvalue.
Назревает вопрос, что наступает раньше побочный эффект или вычисление выражения в записи a++? В выражении a = a + 1 такой проблемы нет, так как сначала происходит вычисление выражения (rvalue), а затем побочный эффект. Заметим, что побочный эффект в a++ и a = a+1 одинаковый. Можно записать выражение с ++ как в префиксной, так и в постфиксной форме. Выражение a++ генерирует значение до инкремента (верни a и увеличь на 1), а выражение ++a генерирует значение после инкремента (увеличь на 1 и верни a). Это легко запомнить, если считать, что операции выполняются слева направо. Например, если a = 3, то a++ вернет 3, в то время как ++a вернет 4.

Также есть двухместный операнд “,” для последовательного вычисления.
Пример: a = (b = 5, b+2);
Операция « , » вычислила операнд b = 5,у которого есть значение 5 и побочный эффект, который заключается в том, что в b записали 5. Далее вычисляем правый операнд, а побочный эффект уже произошел. Результат операции « , » - результат правого операнда, то есть в результате значение станет равно 7.

В языке Си нет никаких ограничений, в каком порядке будут вычисляться операнды. Например, в выражении a*2 + b/3 никто не гарантирует, что сначала будет выполняться умножение, может быть наоборот. В функции с большим количеством аргументов, аргументы могут быть вычислены в удобном нам порядке. То есть у компилятора есть полная свобода менять порядок вычислений операндов так, как ему покажется быстрее.

Однако есть некоторые ограничения. Эти ограничения записываются через *точку следования* (sequence point).
Точки следования – это, по сути, моменты, когда мы знаем, что все случилось и transaction completed. Пример из жизни: пришла смс, что деньги списались, пока не пришла смс непонятно, что случилось. Аналогично точки следования описывают момент, в который должны приходить смс, глядя на которые мы понимаем, сколько осталось денег.

Формально: *точка следования* – момент, во время выполнения программы, в котором все побочные эффекты предыдущих вычислений закончены, а новых – не начаты. В этот момент в rvalue мы можем быть уверены, что lvalue уже записал все в память. Ленивые вычисления обязаны влечь за собой точку следования.
Самый популярный способ получить в коротком выражении точку следования конъюнкцию или дизъюнкцию (&& или | |). Из-за ленивого вычисления, из-за того, что всегда сначала вычисляется левая часть, а потом правая, здесь наступит точка следования. При этом между двумя точками следования изменение значения переменной возможно не более одного раза.
Второй способ – это то, что называется full expression, самое длинное выражение, которое написали до ;. То есть c = d, это не full expression, a a = b = c = d = 0; - full expression. Это место, когда expression - операция в терминах стандарта превращается в expression statement, то есть в оператор, для этого нужно поставить ;. Поставили точку-произошел sequence point.
Напомним, что при вызове сложных функций с множеством аргументов, как, например, f(g(5), h(3)) вовсе необязательно, чтобы первой вычислялась g(5), может быть и наоборот.
В примере i++ + ++I нет точек следования, поэтому нельзя менять один и тот же объект 2 раза, это приведет к undefined behavior. Результат этого выражения будет зависеть от компилятора и его поведения.
В последних стандартах терминология несколько иная (sequenced before, unsequenced, indeterminately sequenced): точка следования влечет частичный порядок, его отсутствие делает возможным любые варианты.

Форматный ввод-вывод

Пока у нас нет указателей, мы фактически не можем ничего ввести. Есть функция scanf, в которую нужно передать указатели на числа], то есть их адреса, если мы хотим их считать два числа. Адреса передаются с помощью оператора &.

```
#include <stdio.h>
int main(void) {
        int s = 0;
```

```
        int a, b;
        sacnf("%d%d", &a, &b);
        s += a + b;
        printf ("Sum is %d\n", s);
        return 0;
}
```

У нас есть соглашения: стандартный поток ввода (stdin) и стандартный поток вывода (stdout). Эти функции читают стандартный поток ввода, который для нашего удобства является клавиатурой, и стандартный поток вывода, который является монитором. Также есть еще стандартный поток ошибок.
Чтобы ввести число типа int, например, мы могли бы написать функцию, но тогда для числа типа double пришлось бы писать свою функцию, что очень неудобно, особенно если учесть, что в С нет перегрузки имен, как в С++, и мы не можем одним и тем же именем назвать несколько функций.

В Си ввод-вывод реализован с помощью форматный строк, то есть мы подсказываем компилятору, какие типы данных мы считываем и записываем, зная это, компилятор может реализовать свой ввод-вывод для каждого типа. Форматный ввод-вывод так называются, потому что то, что мы считываем или записываем, определяется форматом, а формат определяется форматной строкой.

В нашем примере функция scanf – это первая функция с переменным количеством параметров, с который мы встретились. Мы можем считать как одно число, так и несколько чисел. Первым параметром обязана быть форматная строка, которая описывает, что мы считываем. % — это форматный символ (спецификатор), то, что идет после него — это тип того, что мы считываем или записываем.

Спецификатор	Печатает/считывает
%d, %ld, &lld	Число int, long, long long
%u, %lu, %llu	Число unsigned, unsigned long, unsigned long long
%f, %Lf	Печатает double, long double
%f, $lf, %Lf	Считывает float, double, long double
%c	Символ (char)

%4d: вывести число типа int минимум в 4 символа

%.5f: вывести число типа double с 5 знаками
%%: напечатать знак процента
Функция scanf возвращает количество удачно считанных элементов.

Лекция 7. Операции и операторы Си

Рассмотрим программу, которая решает квадратное уравнение.

```
#include <stdio.h>
#include <math.h>
int main (void) {
	int a, b, c, d;
	/*Input coefficients*/
	if (scanf("%d%d%d", &a, &b, &c) != 3) {
		printf("Need to input three coefficients!\n");
		return 1;
	}
	if (!a) {
		printf("That's not quadratic!\n");
		return 1;
	}
	d = b*b - 4*a*c;
	if (d < 0)
	  printf("No solutions\n");
	else if (d == 0){
		double db = -b;
		printf ("Solution^ %.4f\n", db/(2*a));
	} else {
		double db = -b;
		double dd = sqrt(d);
		printf("Solution 1: %.4f, solution 2: %.4f\n",
		     (db+dd)/(2*a), (db-dd)/(2*a));
	}
	return 0;

}
```

Очевидно, в этой программе нам необходим файл, содержащий функции ввода-вывода и файл, содержащий функции, отвечающие за математику, чтобы извлечь квадратный корень. Исторически основная библиотека C по умолчанию компилятором компоновалась с программой и все реализации std лежали в одном файле, а математические функции, будучи специфичными, лежали в отдельном файле. Поэтому, если требовалось использовать функцию из основной части библиотеки, компоновщику не надо было ничего говорить, но, если требовалось использовать функцию из

математической части, нужно было использовать флажок -lm (-l – запрос библиотеки, m – обращение к математической части).

Функция main, как вы помните, возвращает целое значение, поскольку мы хотим операционной системе рассказать, хорошо или плохо мы закончили свою работу. Функции scanf и printf обычно просто используют, однако они имеют возвращаемое значение, которое в случае printf скорее всего не понадобится, а в случае scanf оно очень полезно, так как говорит, сколько форматных спецификаторов было успешно обработано. Например, если вы хотите считать 3 числа, а на стандартном потоке ввода только 2 числа.

В нашем примере первый if проверяет, что мы ввели неправильное количество коэффициентов и возвращают 1. Как мы помним, ненулевой код возврата – признак того, что что-то пошло не так.

Второй if проверяет, что a не равно 0. Поскольку a – целое число, если $a = 0$, то это выражение истинно, если нет, то ложь. То есть это сокращенная запись сравнения a с 0. И в этом случае мы тоже возвращаем 1 (хотя можем и вернуть другой код возврата, чтобы в дальнейшем понимать, какая именно ошибка произошла).
У нас появляется double в строке double db = -b; потому что поскольку у нас все в целых числах, то в дальнейшем и результат деления db/(2*a) тогда был бы целым числом, а нам нужно вещественное. Чтобы результат деления был вещественным, нужно, чтобы один из аргументов был вещественным (изначально или преобразован к нужному типу). %.4f выведет вещественное число с 4 знаками, это удобно, потому что по умолчанию вещественное число выводится с экспонентой.

Присваивание

При присваивании a = b:

- «Широкий» целочисленный тип в «узкий» отсекаются старшие биты
- Знаковый тип в беззнаковый: знаковый бит становится «значащим»
  signed char c = -1; /* sizeof(c) = = 1 */
  ((unsigned char) c) → 255
- Плавающий тип в целочисленный: отбрасывается дробная часть
- «Широкий» плавающий тип в «узкий»: округление или усечение числа

Явное поведение типов: (type) expression
d = ((double) a + b) / 2;

В стандарте Си есть длинное описание того, что произойдет, если в двухместной операции будет операнды разного типа. Когда операнды двухместной операции имеют разные типы, происходит *неявное присваивание* типов по следующим правилам:

- Если один из операндов – long double, то и второй преобразуется к long double (так же для double и float)

long double + double → long double + long double
int + double → double + double
float + short → float + int → float + float

- Если все значения операнда могут быть представлены в int, то операнд преобразуется к int, так же и для unsigned int (англоязычный термин для этого integer promotion)

unsigned short(2 байта) + char(1байт) → int(4 байта) + int(4 байта)
unsigned short(4 байта) + char(1 байт) → unsigned int(4 байта) + int(4 байта)

- Если оба операнда – соответственно знаковых или беззнаковых целых типов, то операнд более «узкого» типа преобразуется к операнду более «широкого» типа

int + long → long + long
unsigned long long + unsigned → unsigned long long + unsigned long long

Легче всего это запомнить, если представить «прямую» типов (рис. 7.1). Меньший тип всегда будет приводиться к большему.

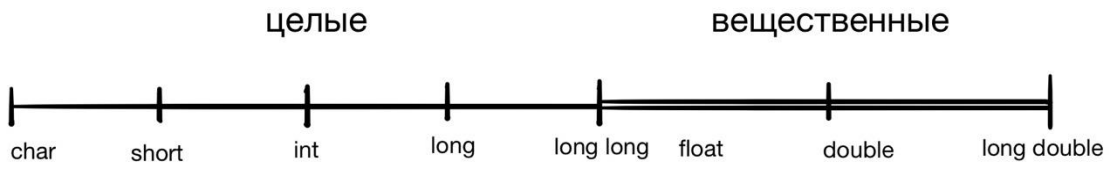


рис. 7.1 представление типов

Подробнее об integer promotion: типы, которые находятся слева от int - char, short (знаковые или беззнаковые), превращаются в int или unsigned int, потому что производить вычисления с короткими типами чревато переполнением. Таким образом, вычисления не происходят в типах меньше int.
При этом если больший тип беззнаковый, а меньший знаковый, то «побеждает» беззнаковый тип. Здесь могут возникнуть проблемы. Например, при делении int на unsigned int, int преобразовывается в unsigned int и делится на unsigned int. В итоге получается число unsigned int. Так -200 / 40 → ≅ $4*10^9$ / 40 = 10^9 , а не -50, которые мы хотели получить. Поэтому старайтесь не смешивать знаковые и беззнаковые типы в арифметике, для этого проще всего не использовать беззнаковый тип.

Таким образом,

- Если операнд беззнакового типа более или так же «широк», чем операнд знакового «узкого» типа, то операнд узкого типа преобразуется к операнду «широкого» типа

int + unsigned long → unsigned long + unsigned long
int(4) / unsigned int(4) → unsigned int(4) / unsigned int, что приводит к неверным значениям, как мы рассмотрели выше.

- Если типа операнда знакового типа может представить все значения типа операнда беззнакового типа, то операнд беззнакового типа преобразуется к операнду знакового типа

unsigned int(4) + long(8) → long(8) + long(8)
unsigned short + long long → long long + long long

- Оба операнда преобразуются к беззнаковому типу, соответствующему типу операнда знакового типа

unsigned int(4) + long(4) → unsigned long(4) + unsigned long(4)

- Числа типа float не преобразуются автоматически к double

Рассмотрим пример:
unsigned short a, b, c;

a + b

unsigned short меньше, чем int, поэтому число *a* превращается в int, как и *b*, соответственно результат так же будет int. unsigned short – 2-байтовый тип (максимальное числовое значение в районе 65000), и если a = b = 40000, то a+b = 80000, что хорошо помещается в 4-байтовом int. Но если мы запишем так: c = a + b, то произойдет следующее: unsigned short(2) = int(4), то есть int не поместится в unsigned short, придется отсечь 2 старших байта, и тогда получится примерно 23000.

Приоритеты операций

В любом языке программирования есть приоритеты выполнения операций. В C приоритеты выстраиваются таким образом, чтобы минимизировать употребления «()».

! ++ -- + - sizeof(type)	Справа налево
* / %	Справа налево
+ -	Справа налево

== !=	Слева направо
&&	Слева направо
||	Слева направо
= += -= /= %=	Справа налево
,	Слева направо

Операторы

Так как С -императивный язык, необходимо управление потоком, то есть задание циклов и условий. У нас есть statement expression, то есть мы умеем писать выражения, но пока не знаем, как превратить их в операторы. Чтобы превратить выражение в оператор, нужно поставить «;», которая, во-первых, отделит один оператор от другого, а во-вторых, превратится в точку следования.

Составной оператор: {} – способ записи сразу нескольких операторов, где по грамматике возможно записать только один. Например, когда тело функции состоит из составного оператора.

Условный оператор: if (expr) stmt; else stmt;

Обычно условный оператор дает выбрать 1 из 2 условий. Если выражение истинно, то условие выполняется, если нет – не выполняется. Блока с else можно опустить.
У условных операторов есть классическая *проблема разрешения неоднозначности в грамматике*. Это одна из первых проблем, которая возникает в языках. Она называется dangling else, то есть «висящий» else.
Рассмотрим эту проблему на примере:

```
if (x > 2)
   if ( y > z)
                y = z;
    else
                z=y;
```

К какому именно условию if относится else? Else всегда относится к ближайшему if. Чтобы отнести else у другому if, нужно поставить {}.

```
if (x > 2) {
        if (y > z)
        y = z;
   }
```

В С есть оператор выбора switch.

```
switch (expr) {
        case const-expr: stmt;
        case const-expr: stmt;
        default: stmt;
   }
```

Оператор break – немедленный выход из switch. В С есть некоторые проблемы с этим оператором: выражение должно быть исключительно целого типа. Ряды с case, по сути являются метками. То есть у нас происходит своего рода сложный go to переход по значению на ту метку, которая соответствует этому значению. Дальше идет просто блок, то есть выполняются операторы с той точки, на которую мы перешли. Метка default нужна для того, чтобы переходить куда-то в случае, если ни одна из меток switch не содержала значение вычисленного выражения. После каждого оператора, который мы хотим завершить, дописываем break. Если не записывать break, то весь блок выполнится поочередно, так как автоматический выход не предусмотрен.
Пример:

```
switch (a) {
        case 1: <...>; break;
        case 3: <...>; break;
   }
```

Если в нашем примере не было break, то выполнился бы сначала оператор из случая 1, а потом оператор из случая 2.

Чтобы понизить эту конструкцию до ассемблерного кода в случае, нужно:

- если в операторе switch несколько случаев, то конструкцию можно свести к цепочке if-else, так как в процессоре есть условный переход (и более сложный случай – переход по вычисленному адресу)

- если в операторе switch много случае, то можно построить структуру типа дерева, которая свела бы 10 сравнений к 2. Для этого нужно создать массив, который будет примерно, как дерево, все эти выражения, которые сравниваются в case, записать в этот массив и организовать какой-то поиск, что не очень просто и не всегда удобно.

Таким образом, часть компилятора, которая понижает код оператора switch, должна быть достаточно интеллектуальной.

Например,

```
switch (c) {
        case 2: <...>;
        case 1: <...>; break;
        case 3: <...>;
        default:  <...>;
   }
```

Когда произойдет переход по метке 2, выполнится 2 оператора: первый, который относится к case 2 и второй, который относится к case 1. Если c =1, то выполнится оператор оператор, который соответствует case 1. Если c=3, то выполнится оператор относящийся к case 3 и к default. Для того, чтобы отметить, что break намеренно отсутствует, пишут комментарий /* fall through */.

Теперь разберем циклы.
Цикл while: **while** (expression) stmt;

Цикл do-while: **do** {stmt;} **while** (expression);
Проверка условия выхода из цикла после выполнения тела!

Цикл for: **for** (decl1; expr2; expr3)
stmt;

```
decl1;
while (expr2) {
        stmt;
        expr3;
 }
```

for (; ;) stmt; - описание бесконечного цикла
while (1) – описание бесконечного цикла
Цикл for придумали как удобный способ сгруппировать в заголовке цикла условие инициализации и условие продолжения цикла. Сначала выполняется инициализация индуктивный переменных. В C89 сначала должны были быть все определения, а потом код. В C99 разрешили в середину кода вставлять объявление переменной и объявлять переменную в заголовке цикла: for (int i = 0; i < 100; i++ (или ++i));
Здесь есть свои нюансы:

1) Любая из частей может быть опущена (инициализация, условие или действие)
2) В отличие от Pascal i — это просто переменная, к ней можно будет обратиться после завершения цикла и она будет содержать значение, достигнутое на последней итерации

3) Есть операторы break и continue: выход из внутреннего цикла и переход на следующую итерацию.

Оператор **goto** – переход по метке
goto label;
…
label:

Областью видимости метки является *вся функция!*
goto нарушает структурированный поток управления, поэтому его используют только в случае крайней необходимости.
Например, функция, которая возвращает код статуса обработки.

```
it dofoo (int a, int b ...){
…
 }
```

Для того, чтобы функция выполняла какие-то действия, нужно, чтобы параметры удовлетворяли каким-то ограничениям. После проверки ограничений выдается тот или иной код. Но помимо получения кода, было бы хорошо вернуться и сделать подчистку. В более высокоуровневых языках для этого есть exception, в С можно сделать следующее:

```
if (bad1) {
        cleanup();
        return fail;
    }
    if (bad2) {
        cleanup();
        return fail;
    }
```

Если cleanup() – это не вызов функции, а какое-то количество выражений, которое прям в рамках этой функции делается несколько раз, это не очень удобно. Вместо этого можно использовать goto:

```
int ret = fail;
        if (bad1)
        goto err;
        if (bad2)
        goto err;
        ret =ok;
```

```
do ...;
err: cleanuo();
return ret;
```

Лекция 8. Строки и массивы в Си

Рассмотрим программу, которая вычисляет количество дней между двумя датами. Цикл while(1) – бесконечный цикл, чтобы выйти из которого есть несколько break. Например, первый break происходит, когда ввели не 6 чисел (вспомним, что scanf возвращает количество успешно прочитанных модификаторов. Функция check_date возвращает 0, когда все хорошо,
1, когда что-то из дня, числа или года равно 0, то есть нужно выйти из цикла,
и - 2, когда данные некорректные, но из цикла выходить не надо, надо запросить еще раз данные. Это пример типичной ленивой логики: если при проверки первой даты вернулась 1, то вторую дату проверять вообще не нужно.
Далее вызываются функция, которая вычисляет разницу между началом года и текущим днем, она вызывается 2 раза. Есть функция, которая возвращает разницу между годами в днях.

```
#include <stdio.h>

int main(void) {
	while (1) {
		int m1, d1, y1, m2, d2, y2;
		int t1, t2;
		int days1, days2, total;

		if (scanf ("%d%d%d%d%d%d", &d1, &m1, &y1, &d2, &m2, &y2) != 6)
		break;
		r1 = check_date(d1, m1, y1);
		if (t1==1 || (t2 =check_date(d2, m2,y2))==1)
		break;
		else if (t1 == t2 || t2 == 2)
		continue;
		days1 = days_from_jan1(d1, m1, y1);
		days2 = days_from_jan1 (d2, m2, y2);
		total = days_between_years(d2, m2, y2) + (days2 - days1);
		printf("Days between dates: %d",
			"weeks between days^ %d\n",
			rotal, total/7);
	}
	return 0;
}
```

Рассмотрим, как устроены эти функции.

Функция check_date возвращает целое число, мы это обсудили выше.

```
#include <stdio.h>
static int check_date (int d, int m, int y) {
        if (!d || !m || !y)
          return 1;
        if (d < 0 || m < 0 || y < 0)
        {
                printf("%d %d %d: wrong date\n", d, m, y);
                return 2;
        }
        return 0;
}
```

Чтобы посчитать количество дней в году, нужно понять, является ли он високосным. Для этого воспользуемся функцией leap_year().

```
static int leap_year (int y) {
        return (y % 400 ==0) || (y % 4 == 0 & y % 100 != 0);
}

static int days_in_year(int y) {
        return leap_year(y) ? 366 : 365;
}

static int days_between_years(int y1, int y2) {
        int i;
        int days = 0;
        for (i= y1; i<y2; i++)
          days += days_in_year(i);
        return days;
}
```

Год високосный, когда делится на 400 или когда выполняется второе условие. Здесь тоже используются ленивые вычисления.
Во второй функции используется трёхместный оператор **? :**
expr1 ? expr2 : expr3
Этот оператор представляет из себя сокращенную запись условного выражения. Вычисляется expr1, если она истинна, то результатом будет вычисление expr2, а если ложно, то вычисление expr3. Причем слева и справа от ? есть точка следования.

Функция days_between_years() считает количество дней между годами. Ее можно оптимизировать, например, пропуская заведомо невисокосные года.

В функции days_from_jan1() используется нетипичный switch. Здесь мы используем по умолчанию проваливание вниз, то есть fall through. Функция вычисляет сколько дней прошло сначала года, путем перехода на этот месяц и суммирования количества дней в предыдущих месяцах (путем проваливания вниз).

```
static int days_from_jan1(int d, int m, int y) {
	int days =0;
	switch(m) {
		case 12: days += 30;
		case 11: days += 31;
		case 10: days += 30;
		case 9: days += 31;
		case 8: days += 31;
		case 7: days += 30;
		case 6: days += 31;
		case 5: days += 30;
		case 4: days += 31;
		case 3: days += leap_year(y) ? 29 : 28;
		case 2: days += 31;
		case 1: break;
	}
	return days + d;
}
```

Символьный тип данных

В Си выделяют разные кодировки в зависимости от цели. Например, есть кодировка символов, которая находится в файле. Как правило, это ASCII. Так как символы представляют из себя чаще всего элементы какого-то алфавита, работают не с элементами алфавита, а с номерами этих элементов, что и является кодировкой. С переменными типа char работают как с переменными целого типа, а смысл char появляется в момент выполнения операций, например, сравнения с литералами, которые задают уже конкретные символьные константы. Если задана кодировка, то есть, по сути, таблица соответствий между символами и номерами, то, когда есть символьная константа в ‘ ‘, то, зная кодировку, мы можем поставить в соответствие этому символу номер и операцию сравнения переменной символьного типа с литералом свести к операции сравнения двух чисел.

Для ввода простого символьного типа (одного символа или строки), по сути, нужно организовать функцию printf для некоторого ограниченного числа спецификаторов, уметь разобрать форматную строку, что очень непросто. Но если мы хотим ввести один символ, то нет смысла использовать этот механизм. Для этого есть функция getchar(), которая считывает строку и не интерпретирует входной поток, а принимает только один символ.

```
#include <stdio.h>
int main(void) {
        int c, nl =0;
        while ((c = getchar()) != EOF)
          if (c == '\n')
            ++nl;
        printf("%d\n", nl);
        return 0;
}
```

Чтобы понять, успешно ли считала символ функция getchar(), (а символ может быть неуспешно считан, только когда он закончился), нужно сравнить возвращаемое значение с константой из файла stdio.h EOF (end of file). Если возвращаемое значение равно EOF, то поток ввода закончился. Обычно эта константа равна -1, но нам необязательно это знать, так как константа определена в файле stdio.h.

Каков должен быть тип, возвращаемый функцией getchar()? С одной стороны функция должна возвращать какой-то символ, а с другой – может вернуть константу EOF. Как это все уместить в тип char? Никак. По принципу Дирихле, если у нас есть 100 клеток и 101 кролик, то хотя бы в одно клетке сидит больше одного кролика. Соответственно возвращаемое значение не влезет в тип char и будет типа int.
В программе, представленной выше, считываются символы по одному, пока не закончился поток ввода. Если встречается /n, то увеличиваем некоторую переменную и ее выводят. Если вы имели дело с Linux, то вы, наверное, заметили, то мы написали часть кода WC (word count), который считает во входном потоке количество строк, слов и символов.

Вернемся к кодировке. Символьные данные представляются в некотором коде. Популярным кодом является ASCII (American Standard Code for Information Interchange). Каждому символу сопоставляется его код – число типа char. Требуется, чтобы в кодировке присутствовали маленькие и большие английские буквы, цифры, некоторые другие символы. Требуется, чтобы коды цифр 0, 1, …, 9 были последовательны.

Изначально char был 7-битным, но с появлением европейских алфавитов с диакритическими (надстрочными) знаками, появился еще 1 значащий бит.

- К символьным данным применимы операции целочисленных типов (но обычно – операции *отношения* и *сравнения*)
- Каждый символ-литерал заключается в одинарные кавычки ' и '
- Последовательность символов (строка) заключается в двойные кавычки " и "
- Специальные (управляющие) символы представляются последовательностями из двух символов.

Примеры:
\n – переход на начало новой строки
\t – знак табуляции
\b – возврат на один символ с затиранием

Если мы запишем '*a*', то в памяти хранится код символа *a* в кодировке, которую поддерживает данная реализация, то есть, если это ASCII, то код 48.
Также символ можно задать через код, записанный в шестнадцатеричном виде, чтобы было удобней. Например, '\0x30' будет тоже *a*.
Так как коды цифр последовательны, мы можем проверить, является ли *c* символом через простое сравнение: *c >= '0' && c <= '9'*.
Чтобы узнать, какая *c* цифра, из нее просто вычитается код 0, то есть *c - '0'*.

В коде ASCII (рис. 8.1) буквы верхнего и нижнего регистра составляют непрерывные последовательности: между *a* и *z* и соответственно между *A* и *Z* нет ничего кроме букв, расположенных в алфавитном порядке. Это же верно для цифр 0, 1, …, 9.

Рассмотрим преобразование строки цифр в число. Реальная функция будет сложнее, но основной цикл ровно такой же: у нас есть выборка символов, которые являются символами и обладают свойством, что их коды цифр идут подряд , и пока символ является цифрой происходит вычисление самого числа через код символа. Это верно для любой кодировки.

```
int atoi (char s[]) {
    int i, n;
    n=0
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
    n = 10*n + (s[i] - '0');
    return n;
}
```

Как уже говорилось ранее, любой язык программирования, по сути, представлен базовыми типами, а пользовательские типы представляют из себя разнообразные комбинации базовых типов. Так из набора элементов одного типа можно составить массив, а из элементов разного типа – структуры.

Код	0	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
0	NUL	DLE	SP	0	@	P	`	p	А	Р	а	▒	└	╨	р	Ё
1	SOH	DC1	!	1	A	Q	a	q	Б	С	б	▒	┴	╤	с	ё
2	STX	DC2	“	2	B	R	b	r	В	Т	в	▓	┬	╥	т	Є
3	ETX	DC3	#	3	C	S	c	s	Г	У	г	│	├	╙	у	є
4	EOT	DC4	$	4	D	T	d	t	Д	Ф	д	┤	─	╘	ф	Ї
5	ENQ	NAK	%	5	E	U	e	u	Е	Х	е	╡	┼	╒	х	ї
6	ACK	SYN	&	6	F	V	f	v	Ж	Ц	ж	╢	╞	╓	ц	Ў
7	BEL	ETB	´	7	G	W	g	w	З	Ч	з	╖	╟	╫	ч	ў
8	BS	CAN	(	8	H	X	h	x	И	Ш	и	╕	╚	╪	ш	°
9	HT	EM	)	9	I	Y	i	y	Й	Щ	й	╣	╔	┘	щ	•
A	LF	SUB	*	:	J	Z	j	z	К	Ъ	к	║	╩	┌	ъ	·
B	VT	ESC	+	;	K	[	k	{	Л	Ы	л	╗	╦	█	ы	√
C	FF	FS	,	<	L	\	l	|	М	Ь	м	╝	╠	▄	ь	№
D	CR	GS	-	=	M	]	m	}	Н	Э	н	╜	═	▌	э	¤
E	SO	RS	.	>	N	^	n	~	О	Ю	о	╛	╬	▐	ю	■
F	SI	US	/	?	O	_	o	DEL	П	Я	п	┐	╧	▀	я	

рис. 8.1 таблица ASCII

Массивы

Массивы представляют из себя набор элементов одного типа и позволяют организовывать непрерывные последовательности нескольких элементов и общаться с ними по номеру (индексу).

- Элементы массивов располагаются в памяти последовательно и индексируются с 0:

  ```
  int a[30]; /* элементы a[0], a[1], ..., a[29] */
  ```

- Все массивы одномерные, но элементом массива может быть массив

```
int b [3][3]; /* b[0][1], b[0][1], b[0][2],
                 b[1][0], b[1][1], b[2][2],
                 b[2][0], b[2][1], b[2][2] */
```

- Контроль правильности индекса массива не производится!

Чтобы задать массив, достаточно объявить тип элементов и размер массива. К массивам можно применять оператор sizeof(). Так, для int a[30] sizeof() выдаст 30 (кол-во элементов) *4 (размер типа int) = 120.

Рассмотри, как в Си, массивы устроены в памяти. В стандарте Pascal, например, представление элементов массива в памяти не специфицировано, то есть они могут идти не подряд. В Си элементы массива в памяти идут подряд. И если мы знаем адрес первого элемента (с индексом 0), то мы точно знаем и адреса последующих элементов. Например, если адрес первого элемента 2000, то адрес второго элемента 2004, то есть адрес первого + размер типа элемента (в нашем случае int).
Зная адрес начала массива (базы), можно узнать адрес i-го элемента. Для этого нужно к адресу базы прибавить i * sizeof(type). Такие режимы адресации встроены и в процессор. То есть в процессоре есть инструкция, которая не просто загружает значение из памяти, но которой можно сразу передать и базу, и адрес, и смещение (offset), чтобы она нашла адрес заданной этими параметрами ячейки памяти. Чтобы эта инструкция работала нумерация элементов должна начинаться с 0, что и происходит в С.
Второй особенностью является локальность памяти (причина работа КЭШ). КЭШи работают из-за верности следующего утверждения: если вы обращались в программе к каким-то данным, то скорее всего в будущем вы тоже будете к ним обращаться. Это утверждение представляет принцип *временной локальности*. Также бывает и *пространственная локальность*, как в массивах: если вы воспользовались одной ячейкой памяти, то скорее всего вы воспользуетесь и ближайшей ячейкой памяти. На основании этого работает КЭШ. КЭШ всегда загружает не один элемент, который попросили, а целую строку в памяти, потому что скорее всего вам потребуется следующий элемент. И это все может работать, только если элементы массива расположены в памяти подряд.

Вернемся к примеру int b [3][3]; /* b[0][1], b[0][1], b[0][2],
b[1][0], b[1][1], b[2][2],
b[2][0], b[2][1], b[2][2] */

Запись int b [3][3]; означает, что у нас массив из 3 элементов, а дальше указан тип элементов – массив из 3 элементов. Когда компилятор вычисляет адрес элемента a[i], он берет base(a) + i*sizeof(type(a)), то есть база + смещение. Когда элементами массива является массив, нужно i умножить на количество элементов массива, умноженного на sizeof() типа этого массива.

Рассмотрим более сложную программу, которая заводит массив из 10 элементов-символов и считает количество цифр, пробелов и остальных символов. В условии сравнения символа с числом мы снова используем факт, что в ASCII коды цифр идут по порядку и что индексы массивов начинаются с 0. Также в этой программе представлен типичный способ обхода массива из N элементов – по счетчику от 0 до N. Здесь обязательно использовать строгое неравенство. Если поставим <=, то мы обязательно захватим 10й элемент, которого нет, что является off-by-one ошибкой.

```
#include <stdio.h>

int main(void) {
        int c, i, nwhite = 0, nother = 0, ndigit[10];
        for (i=0; i<10; i++)
         ndigit[i] =;
        while (c = getchar() != EOF)
          if (c >= '0' && c <= '9')
            ++ndigit[c - '0'];
          else if (c == ' ' || c == '\n' || c =="\t")
            ++nwhite;
          else
            ++nother;
        printf("difits =");
        for (i=0; i < 10; ++i)
          printf("%d", ndigit[i]);
        printf(", white space = %d, other =%d\n", nwhite, nother);
        return 0;
}
```

Про операцию индексации a[i] нужно знать 2 вещи:

1) Оная является lvalue, так как представляет из себя новый способ занимания памяти.
2) Когда компилятор видит a[i], он автоматически генерирует это выражение base(a) + i*sizeof(type(a)). Соответственно, если будет указан несуществующий индекс, то компилятор все равно сгенерирует это выражение, что спровоцирует

undefined behavior, так как будет найдена область памяти, которая не была выделена. Сам компилятор не проверяет индексацию в этом выражение, так как это сильно его замедляет. Например, в Java, где эта проверка реализована, компилятор замедляется на порядок (программа может замедлится в 10 раз). Поэтому в C никак не проверяется, что значение сгенерированного выражения попало в границы индексов существующего массива. И когда мы получаем доступ к памяти за пределами выделенного участка, мы можем столкнуться с переполнение буфера. В лучшем случае программа завершится, в худшем – мы запросим чужую память, где хранятся совершенно другие данные. От такого типа ошибок стараются защититься. Для этого есть анализаторы, которые проверяют программу, стараясь понять и доказать, что таких ошибок нет. Есть анализаторы по типу Java, которые вставляют проверки в программу, и если такие ошибки есть, то программа ничего не выдаст.

Чтобы задать начальное значение в массиве, нужно разобраться с новой сущностью в C, которая потом используется и для структур и называется *списком инициализации.* Переменные мы инициализировали различными литералами. Чтобы инициализировать несколько элементов, будем использовать список.
type name[dim1]…[dimN] = {value list};
Можно не указывать размер массива – он будет вычислен по количеству элементов инициализатора:
int sqrs[] = {1, 4, 9, 16, 25}; /* 5 элементов*/

В C99 стала возможна инициализация лишь некоторых элементов (остальные инициализируются нулями)
int days[12] = {31, 28; [4] = 31, 30, 31, [1] =29};
Здесь нулевой элемент массива – 31, первый – 28, далее мы указываем, какой элемент инициализируем [4] – четвертый элемент и присваиваем ему значение 31, сразу после идет пятый элемент, шестой элемент. Потом мы переинициализируем первый элемент, теперь первый элемент равен 29.

При этом:

- При инициализации одного элемента дважды используется последнее значение
- После задания номера элемента дальнейшие инициализаторы присваиваются следующим по порядку элементам

Можно использовать модификаторы const, static итп.
Можно использовать любое константное целочисленное выражение для определенного размера массива. При этом const -переменная не является константным выражением!

Строки

Строки часто попадают в базовый тип языка, просто потому что это очень популярный тип данных при решении реальных кейсов.

- Строка - одномерный массив типа char. Объявляя массив, предназначенный для хранения строки, необходимо предусмотреть место для символа ‘\0’ (конец строки).
- Строковая константа записывается как string constant.
- В конце строковой константы компилятор добавляет ‘\0’
- Стандартная библиотека функций работы со строками

<string.h>, в частности, содержит такие функции, как:

- strcpy(s1, s2) - копирование s2 в s1
- strcat(s1, s2) – конкатенация s2 и s1
- strlen(s) – длина строки s
- strcmp(s1, s2) – сравнение s2 и s1 в *лексикографическом* порядке: 0 – если s1 и s2 совпадают, отрицательное значение, если s1 < s2, положительное значение, если s1 > s2
- strchr(s, ch) – указатель на первое вхождение символа ch в s
- strstr(s1, s2) – указатель на первое вхождение подстроки s2 в строку s1

Для того, чтобы узнать длину строки есть 2 основных способа. Первый заключается в том, что под строку выделяется некоторая память, в первом бите которой (аналогично знаковому биту) хранится длина этой строки. Такой способ использовался и в Pascal. В таком случае нахождение длины строки – операция константной сложности. Однако у этого подхода есть и свои минусы: необходимость постоянно обновлять информацию о длине строки при изменении этой строки и фиксированность возможной длины, например, в Pascal для записи длины строки был выделен 1 байт, соответственно строк с длиной большей, чем та, которая вмещается в байт, просто не было.

В Си ситуация следующая: под длину строки не выделяется отдельное поле. Конец строки помечается отдельным символом ‘\0’. Поиск длины строки strlen(s) – операция линейной сложности, так как приходится пройтись по всей строке. Если же искать длин строки через цикл for – сложность станет квадратичной, так как на каждой итерации будет вычисляться длина.

Лекция 9. Строки и указатели

Вспомним некоторые операции для работы со строками:

- strcat(s1, s2) – конкатенация s2 и s1

В Java и некоторых других языках, где символьный тип является базовым, операция конкатенации настолько частый случай, что для нее переопределена операция «+».

- strcpy(s1, s2) - копирование s2 в s1

В операции копирования всегда есть строка, которую копируют (src – source), и строка, в которую копируют (dst – destination). Функция strcpy(s1, s2) копирует байты из одной строки в другую, пока не будет достигнут ‘\0’. Эта функция также часто является источником проблемы переполнения буфера, так как не проверяет, хватит ли памяти в том месте, куда мы копируем.
Обычно названия функций, работающих со строками устроены следующим образом: str<название_функции>. Однако есть еще и функции формата str**n**<название_функции> - они означают, что в функции появляется третий аргумент, который задает максимальное количество копируемых или пересылаемых байт. Есть и более сложные функции strspn, strpbrk, stpcpy, которые возвращают несколько иной результат. Например, если strcpy возвращает обычно начало выходной строки, а stpcpy возвращает конец выходной строки.
Рассмотрим пример программы:

```
#include <stdio.h>
#include <string.h>

int main(void) {
        char str1[80], str2[80], smp[3] = "ВМК";
        fgets(str1, 80, stdin); str1[strlen(str1) - 1] = '0\';
        fgets(str2, 80, stdin); str2[strlen(str2) - 1] = '0\';
        printf("Строки имеют длину: первая %d, вторая %d\n",
             strlen(str1), strlen(str2));
        if (!strcmp(str1, str2))
          printf("строки равны\n");
        strncat(str1, str2, 80 - strlen(str1) -1);
        printf("%s\n", str1);
        sprintf(str1, "Привет, %s, smp");
        puts(str1);
        return 0;
}
```

В этой программе используются отдельные функции gets(s) и puts(s). Функция puts(s) вводит строку на стандартный поток вывода и является некоторым аналогом printf(“%s, s). Функция gets(s) вводит строку до конца в буфер. С функцией puts(s) никогда не возникает проблем, но функция gets(s) гарантирует переполнения буфера, так как она считывает символы до конца строки, а мы не знаем, сколько памяти нужно выделить под эту строку. В какой-то момент эту функцию удалили из стандартной библиотеки С. Вместо нее есть fgets(s, length, stdin). Когда к функциям ввода-вывода добавляется префикс f, это означает, что можно указать, из какого потока ее читать. У таких функций появляется возможность указать, из какого конкретно файла мы читаем. В них используется константа stdin, которая при возвращении показывает, что достигнут конец стандартного потока ввода. Stdin – это указатель, который позволяет задать тип, который полностью не прозрачен для пользователя.
Однако у этой функции есть неприятные особенности. Первое, например, в нашем примере, если в конце строки есть “\n” и он поместился в заданные 80 символов, то он оказывается в этой строке, то есть строка “Hello\n\0” будет выглядеть не как “Hello\0” из 6 символов, а как “Hello\n\0” из 7 символов. 'n\' нам не нужен, поэтому str1[strlen(str1) - 1] = '0\'; его просто перетирает в '0\'.

В этой программе есть баг, который заключается в следующем: если мы считали 80 символов и в ней не оказалось символа перехода строки, то наши страхи были необоснованными и '0\' там окажется, а 'n\' – нет и str1[strlen(str1) - 1] = '0\' в таком случае просто затрет последний символ строки.
Функция strncat(s1, s2, n) на вход принимает 3 параметра, где третий параметр n – количество символов, которые будут скопированы. Для этого обычно производятся следующие вычисление количество свободных байт минус длина первой строки минус 1, в нашем примере это strncat(str1, str2, 80 - strlen(str1) -1);.
Также в программе используется очень удобная функция sprintf. Если f обозначает возможность указать файл для печати, то префикс s означает, что все можно вывести в строку.
sprintf(str1, "Привет, %s, smp");
Опять же sprintf не гарантирует, что вы не выйдете за границу строки, с чем связано большое количество багов.

В этой программе помимо проблемы с get есть и другая проблема – проблема off-by-one. Вот у нас есть массив из 3 символов и строка. Массив символов, который, по сути, является строкой, разрешено инициализировать строковым литералом, котором в данном случае работает как список инициализаций, то есть вычисляется, сколько нужно места и копируются байты константной строки из статической памяти в динамическую, которая представляет из себя этот массив, поскольку он локальная переменная этой функции.

char str1[80], str2[80], smp[3] = "ВМК";
Но здесь выделено 3 байта, а нужно 4, поскольку есть еще нулевой байт. Это стандартная ошибка off-by-one. Либо нужно писать так char str1[80], str2[80], smp[4] = "ВМК";, либо ничего не писать в [] и разрешить компилятору вычислить размер массива самостоятельно.

Оператор sizeof

Оператор sizeof в дальнейшем нам потребуется для динамической памяти. Вспомним, что он может применять как к переменной, так и к типу.
Одноместная операция sizeof позволяет определить длину операнда в байтах.
Операнды – это типы, либо переменные. Результат имеет тип size_t.
Операция sizeof выполняется во время компиляции, ее результат представляет собой константу. Sizeof помогает улучшить переносимость программ.

Для определения объема памяти в байтах, нужного для двумерного массива, используется следующее:
number_of_bytes = d1 * d2 * sizeof(element_type),
где d1 – количество элементов по первому измерению,
d2 – количество элементов п второму измерению,
element_type – тип элемента массива.
Можно поступить проще:
number_of_bytes = sizeof(array_name)

Однако, sizeof можно применять только к «полностью» определенным типам. Для массивов это означает, что

- размерности массива должны присутствовать в его объявлении
- тип элементов массива должен быть полностью определен

Пример. Если объявление массива имеет вид: **extern int** arr[ ];,
то операция sizeof(arr) ошибочна, так как у компилятора нет возможности узнать, сколько элементов содержит массив arr.

Компилятор, когда видит глобальную переменную, выделяет под нее статическую память, предварительно посчитав, сколько. Если есть, например, 3 файла, где используется эта глобальная переменная, получится 3 различный места в памяти. Для того, чтобы под нее выделялось только одно место, нужно сделать следующее: выделить память только в одном файле, а в остальных просто указать с помощью **extern**, что под эту переменную уже выделено место. То есть в **extern int** arr[ ]; мы не указываем количество элементов, так как память выделена в другом файле, соответственно и sizeof(arr) использовать нельзя.

Рассмотрим другую программу:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
        char buffer[10];

        /* копирование 9 символов из argv[1] в buffer;
        sizeof(char) равно 1, число элементов масссива
        buffer равно его размеру в байтах */
        strncpy(buffer, argv[1],
              sizeof(buffer) - sizeof(char));
        buffer[sizeof(buffer) -1] = '0\';

        return 0;
}
```

По интерфейсу функции main в C есть ряд договоренностей. Одна из таких договоренностей – функция main должна возвращать код возврата (0 – хорошо, не 0 – плохо). В более сложном случае в функции main появляются параметры – аргументы командной строки или переменные окружения, к которым программа может получить доступ. Соответственно, могут быть ситуации, когда не во все операционных системах поддерживаются все варианты прототипов функции main, которые могут быть в стандарте. Главное, что может быть вариант функции main, которая получает количество аргументов, которые есть в программе при запуске и сами эти аргументы как массив строк. Если есть буфферы, то можно использовать sizeof(buffer) как заменитель длины буффера, что позволяет не писать константы.
В нашей программе хорошим тоном считается вместо константы 10 в buffer[10]; в дальнейшем использовать либо sizeof, либо создать и использовать другую константу, которую можно будет изменить.

Указатели

Считается, что в низкоуровневых языках обязательно нужно понимать, что такое рекурсия и указатели. Вспомним, что в концепции машины фон Неймана ячейки памяти кроме адреса ничем друг от друга не отличались, и в них можно было хранить как программы, так и данные. В C появились данные разных типов, но тем не менее адресами этих данных иногда удобно манипулировать, чтобы расширить способ именования памяти. Указатели позволяют неявно именовать память.

& - операция адресации
* - операция разыменовывания

```
int a =1;
int *p;
p = &a;
printf("Значение переменной a = %d\n", *p);
printf("Адрес переменной a = %p\n", p);
```

В результате выполнения фрагмента будет напечатано:
Значение переменно a = 2
Адрес переменной = 0xbffff7a4

&foo является константой, указатель переменной, при этом:

- foo должен быть l-значным (lvalue)
- печать адреса модификатор %p
- нулевой указатель (никуда не указывающий) – NULL (константа в stdlib.h может не иметь нулевого значения)

Невозможно сконструировать объект, который будет расположен в памяти по адресу NULL

- размер памяти, выделяемой под указатели, определяется многообразием адресов возможных на данной архитектуре

Рассмотрим, что происходит в памяти при компиляции следующей программы:

```
int x;
int *p;
p = &x;
```

Допустим переменная x типа int хранится по адресу 100 (рис. 9.1). Возникает вопрос: сколько байт нам нужно взять по этому адресу? На него отвечает тип самого указателя. Так, указатель типа short дал бы нам 2 байта, начиная со 100, а указатель типа int – 4 байта.
С точки зрения процессора, у нас лежит число 100 в памяти, которое может быть целым числом как заведенная константа, так и как адрес. Его интерпретация зависит от компилятора. Таким образом, указатель знает адрес и тип объекта, расположенного по этому адресу.

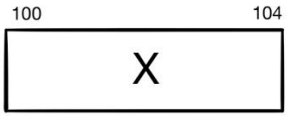


рис. 9.1 ячейка, в которой хранится переменная x

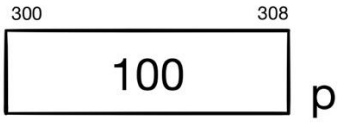


рис. 9.2 адрес, на который указывает указатель p

Таким образом, указатель – это, по сути, типизированный адрес переменной, который позволяет по этому адресу обратиться, получить объект искомого типа и с ним работать. Гибкость такого подхода заключается в том, что, меняя значения адреса, мы можем обрабатывать разные участки памяти, избегая дублирования кода.

Выражение int **q; обозначает, что q – это переменная, которая является указательным типом и указывает на *int (базовый тип указателя). Компилятор, видя такую переменную, Язык с устроен таким образом, что размеры всех указателей одинаковы, но бывают случаи, когда размер указателей на функции отличен от размера указателей на данные, потому что так устроена машина.
Адрес на рис. 9.3 означает адрес объекта int *, а адрес на рис. 9.4 объекта int.

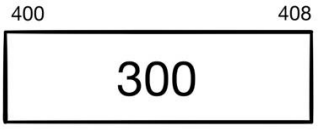


*рис. 9.3 адрес int **

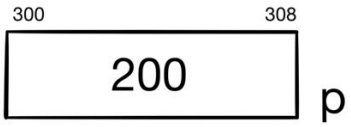


рис. 9.4 адрес int

p = &q; - в запишется 300
**q; - нужно посмотреть, где в памяти лежит указатель q (это ячейка 400), загрузить по этому адресу 8 байт, потому что в ней хранится int *, это число 300, которое нужно

интерпретировать как место, в котором хранится указатель int *. И если мы это место хотим разыменовать, то мы пойдем по этому адресу 300, снова загрузим 8 байт, и то, что в них хранится интерпретировать как адрес объекта, который является целым числом. Поэтому можно писать так: **q + 8, что эквивалентно следующему ассемблерному коду:
t1 = load q, 8 - здесь процессор пойдет по адресу 400 и загрузит 8 байт
t2 = load t1, 8 - t1 = 300
t3 = load t2, 4 - t2 = 200, в t3 будет храниться то, что лежит по адресу 200
t4 = t3 + 8

Адресная арифметика

В языке Си допустимы следующие операции над указателями:

- сложение указателя с целым числом
- вычитание целого числа из указателя
- вычитание указателей
- операции отношения и сравнения

Пример. Пусть sizeof(int) == 4 и пусть текущее значение int* p1 равно 2016 = 0x7E0.
После операции p1++ значение p1 будет 2020 = 0x7E4 (а не 2017 = 0x7E1),
После операции p1 - 3 значение 2004 = 0x7D4.
При увеличении (уменьшении) на целое число i указатель будет перемещаться на i ячеек соответствующего типа в сторону увеличения (уменьшения) их адресов.
Рассмотрим массив из 5 элементов типа short (рис. 9.5).

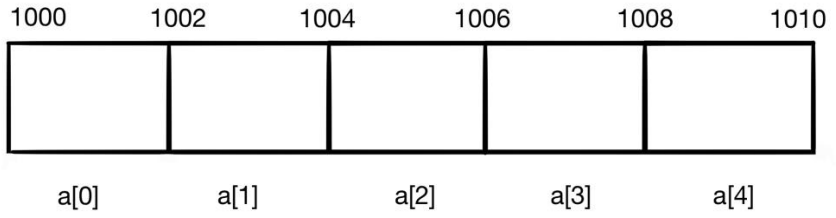


рис. 9.5 массив из 5 элементов

short a[5];
Теперь у нас есть указатель (рис. 9.6):
short *p;
p = &a[0];

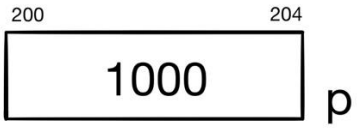


рис. 9.6 указатель

Адресная арифметика позволяет нам записать следующее выражение:
p + expr
p + 3, что означает 1000 + 3*2 = 1006 и все это выражение будет типа short*
*(p + 3) означает значение по адресу 1006, то есть a[3]

Указатель можно преобразовать к другому типу, но такое преобразование типов *обязательно* должно быть явным.
Условие: исходный указатель правильно выравнен для целевого типа. Значение указателя сохраняется.
Иногда такое преобразование типов может вызвать непредсказуемое поведение программы:

```
#include <stdio.h>

int main(void) {
        double x = 200.35, y;
        int *p;
        p = (int *)&x; /* &x - double, a p имеет тип int* */
        y = *p;          /* ,будет ли присвоено значение 200.35? */

        printf("значение x равно %f\n", x);
        printf("значение н равно %f\n", y);
        return 0;
}
```

В данной программе переменная x типа double занимает 8 байт. Пусть, например, переменная x лежит по адресу от 64 до 72 (рис. 9.7). В локальной памяти y будет лежат от 72 до 80, и указатель p размером 4 байта от 80 до 84. Адрес x – указатель типа double* содержит значение 64. Мы хотим сделать следующее: интерпретировать 64 как число типа int, присвоить его p, загрузить что-т по указателю p и присвоить это в y. Это означает, что мы должны взять 4 байта, начиная с 64 и трактовать это как int. После чего эти 4 байта int нужно присвоить y типа double.

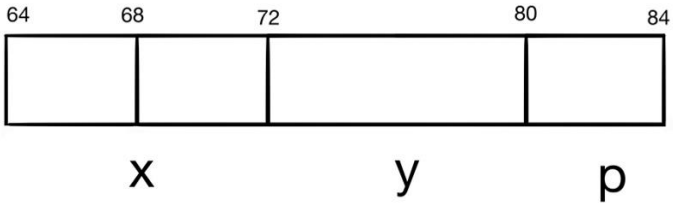


рис. 9.7 расположение переменных в памяти

Но байты с 64 до 68 и с 72 по 80 будут разными, потому что загрузка из p (*p) имеет тип int, так как операции с указателями выполняются всегда в соответствии с его базовым типом. Таким образом, типичный вывод результатов (GCC, Linux) следующий:
Значение x равно 200.350000
Значение y равно 858993459.000000
Так как в присваивании y =*p; загрузка *p считывает только первые 4 байта области памяти с адресом &x (т.к sizeof(int) в данном случае равен 4).
В представлении 200.35 в формате числа double первые четыре байта соответствуют целому числу 858993459.
Таким образом, необходимо учитывать, что операции с указателем выполняются в соответствии с базовым типом указателя.

Говоря о выравнивании, нужно понимать, что если загрузки не выравнены, то есть, например double лежит по адресу 70, то тогда, чтобы сконструировать этот double в архитектуре, которая позволяет загрузки только кратные 8, нужно выдать 2 загрузки, загрузив кусочек по адресу 70-72 и кусочек по адрес 72-76, и сконструировать их вместе. Это работает (не всегда работает вообще) медленно, поэтому компилятор всегда следит за выравнивание базовых типов (aligned). Как правило выравнивание кратно 2, четырёхбайтные классические int хранятся по адресу кратному 4. То есть выравнивание 4 означает, что адрес должен на это делиться в корректном случае.

Лекция 10. Функции

Преобразование типа указателя

При преобразовании типов указателей, как мы обсуждали в прошлой лекции, часто возникает проблема с выравниванием, что приводит к некорректному указателю или к некорректному содержанию, на которое указывает указатель.
Разрешено преобразование целого типа в указатель и наоборот (поведение определяется реализацией). Однако пользоваться этим нужно очень осторожно.
aux = (void *) -1;

Допускается неявное преобразование *менее* квалифицированного типа void * указателю любого другого типа (и наоборот) без явного преобразования типа указателя. Это позволяет использовать void *, когда тип объекта неизвестен, что особенно полезно для функций выделения динамической памяти. Использование типа void * в качестве параметра функции позволяет передавать в функцию указатель на объект любого типа.

Допускается неявное преобразование *менее* квалифицированного типа указателя к более квалифицированному типу (int * → const int *).
const int x =42;

Далее варианты:

1) const int *p; в этом случае можно менять адрес, но при его разыменовывании получится объект с const, который меняться уже не должен

2) int *const p; в этом случае по указателю можно менять память, но сам указатель менять нельзя (его сразу нужно инициализировать), зато в *p можно что-то записать

Второй вариант более редкий, а первый – это просто любая строковая функция, которая просто, как, например strlen, читает элементы строки, но ничего не записывает. В эту функцию обычно передается указатель с квалификатором const.
Константа NULL неявно преобразуется к любому другому типу указателя type *.

В массивах указатель на первый элемент массива можно создать, присвоив переменной типа «указатель на тип элемента массива» имя массива без индекса:
int array[15];
int *p, *q;
p = array;
q = &array[0];

p и q указывают на начало массива array[15], причем значение array изменить нельзя, а значение p – можно, так как array н является l-значением, а p – является.

- array = p; array++ - ошибка
- p = array; p++ - верно, так как и указатель, и адрес начала массива – это какое-то число.

Если увеличить значение указателя в момент указания на последний элемент массива, он увеличится и будет указывать на область памяти, следующую за массивом, в таком случае указатель разыменовывать нельзя, поэтому стоит делать проверку.

Индексирование указателей происходит следующим образом:

```
int *p, a[10]; /* два способа присвоить 100 6-му элемента массива a[10] */
p = a;
```

1) *(p+5) = 100; /* адресная арифметика*/
2) p[5] = 100; /* индексирование указателя */

Сравнение указателей происходит по следующим правилам:

- Если p и q являются указателями на элементы одного и того же массива и p < q, то q – p + 1 равно количеству элементов массива от p до q включительно. Оба указателя должны указывать на одну область памяти!

Можно написать:

```
if (p < q)
  printf(“p ссылается на меньший адрес, чем q”)
```

Так как каждый раз появляются новые способы именования памяти и комбинирования типов, стоит уточнить, что *указатели могут быть собраны в массив.*

1) int *mu[27]; /* это массив из 27 указателей на int */
2) int (*um)[27]; /* это указатель на массив из 27 int */

um – это обычный указатель, под который выделяется 8 байт и разыменовывание которого приводит к большей области памяти, в которой расположен массив из 27 элементов

Первый случай часто используется в таком виде:

```
static void error (int errno) {
  static char *errmsg[] = {
    “переменная уже существует”,
    “нет такой переменной”,
    <…>
```

```
   “нужно использовать переменную-указатель”
  };
printf (“Ошибка: %s\n”, errmsg[errno]);
}
```

Имя массива указателей – пример многоуровневого указателя. Массив errmsg можно представить как char ** errmsg.

В прошлых лекциях мы рассматривали функцию main с argv. argv – это тоже пример многоуровневого указателя, а именно – массив указателей на строки, то есть каждый элемент массива — это char* и в конце там есть NULL, который указывает на окончание входных элементов.

Функции

Мы уже много раз пользовались функциями и хорошо пониманием, что у функции есть имя, набор параметров, у каждого из которого есть тип, и возвращаемое значение. Если возвращаемого значения нет, то по договоренности это тип void.

Объявление функции: return-type func(type1 arg1, type2 arg2, …, typen argn);
int atoi (char s[ ]);
void QuickSort (char *items, int count);
Тип возвращаемого значения void означает, что функция не возвращает значения.
В Си можно пропускать имена параметров, как в нашем примере.

Определение функции: func-decl { body }

В определении функции уже нельзя пропускать имена параметров.

Областью действия функции является весь программный файл, в котором она объявлена, начиная со строки, содержащей ее объявление. Если в программном файле вызывается какая-либо функция, она обязательно должна быть объявлена в этом программном файле до ее вызова.

Директива препроцессора #include <имя_библиотеки.h> вставляет в программу объявления всех функций соответствующей библиотеки.
Если функция возвращает значение, то ее результатом можно пользоваться в выражениях: v = f(); a = f(y) +2;
Если функция не возвращает значений, то вызов выглядит просто как f(args);

В языке Си все аргументы передаются по значению (то есть передаются только значения аргументов, и эти значения копируются в локальную область памяти функции).
Если аргументом является указатель, его значением может быть адрес объекта вызывающей функции, что обеспечивает вызываемой функции доступ к объекту.

Вызов функции и указатели функций

Теперь разберемся, как происходит вызов функции.
Рассмотрим функцию и разберем, что происходит при вызове функции:

```
int f(int x, double d) {
        char c;
        …
        x+d;
        ...
        return x;
}
z = f(y, 2.0)
```

У нас есть автоматическая память, которая представляет из себя большой массив (по сути, стек), от которого компилятор по необходимости «откусывает» кусочки (рис. 10.1). Будем считать, что адреса этого массива растут вниз. Есть некий адрес, который обычно называется stack pointer (sp), который обозначает, что выше него все занято, а ниже – свободно. При вызове z = f(y, 2.0) сначала сопоставляются формальные и фактические параметры. Далее нужно выделить память под параметры, локальные переменные и возвращаемое значение. В нашем случае это 4 (int) + 8 (double) + 1 (char) + 4 (int возвращаемое значение) = 17 байт. Если во время работы функции f нужно будет обратиться к локальной переменной, то в качестве адреса этой переменной подставится выделенное значение. Где-то в памяти хранится также ячейка с y и есть число 2, которое нигде не хранится, просто константа. При вызове функции перед началом работы компилятор перемещает значение из ячейки под именем y в область памяти (в стек) функции, в ячейку, именуемую x. В d запишется 2.0 (рис. 10.2). В c мы ничего не запишем, так как локальные переменные не инициализируются.

Это важно, потому что без инициализации переменной *c* при начале работы функции мы получим 13й байт от начала автоматической памяти на момент начала работы функции, и что там в этот момент было, то и будет (мусор).
У нас также есть ячейка памяти z (рис. 10.3). Например, функция вычислила значение ret = 50, нам нужно организовать присваивание из ячейки ret в области памяти функции в ячейку z. После выполнения функции память, выделенная под нее нам больше не нужна. Поэтому мы возьмем верхушку стека, которая теперь sp + 17 и уменьшим ее,

вернув к старому значению sp и таким образом освободим память, выделенную под функцию.

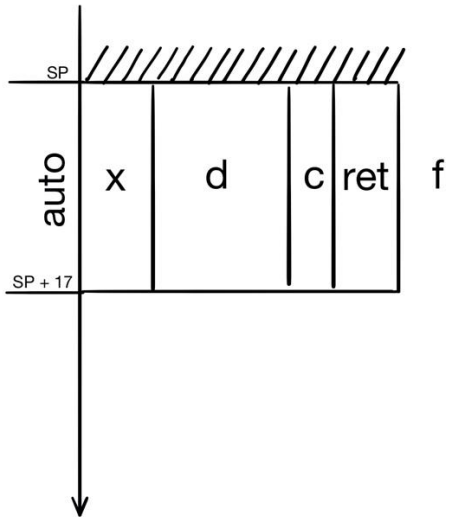


рис. 10.1 область памяти функции

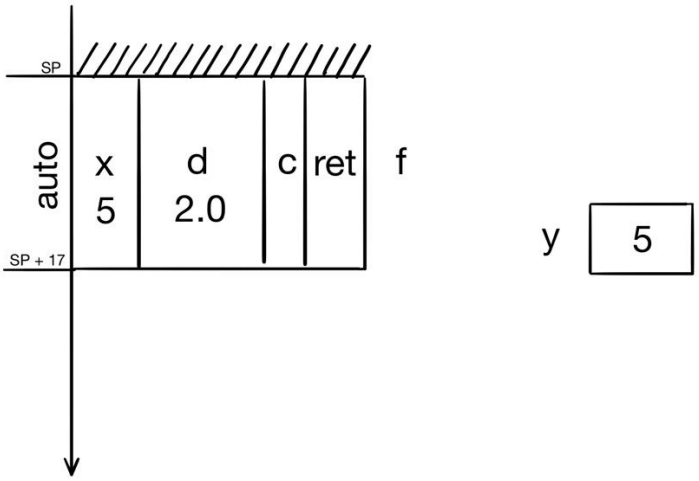


рис. 10.2 перенос значений

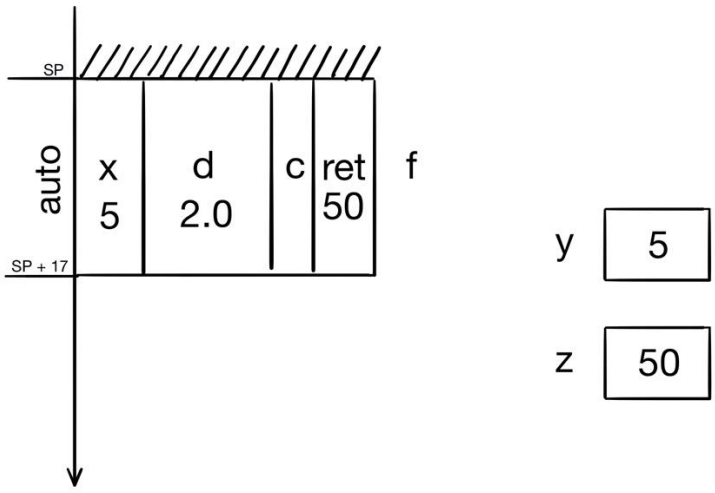


рис. 10.3 в конце работы функции

Если у нас есть сложна функция $f \to g() \to h() \to p()$
то есть функция *f*, которая вызывает функцию *g*, которая вызывает функцию *h*. И по окончании работы функции *f* вызывается функция *p*, то со стеком происходит следующее (рис. 10.4):

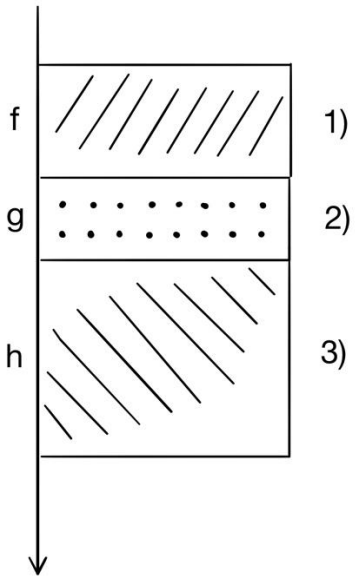


рис. 10.4 вызов сложной функции

По окончании работы функции *h* освобождается область памяти, выделенная под эту функцию. Путем уменьшения значения указателя sp. Однако значение функции *h*, переданное в функцию g сохранится (рис.10.5).

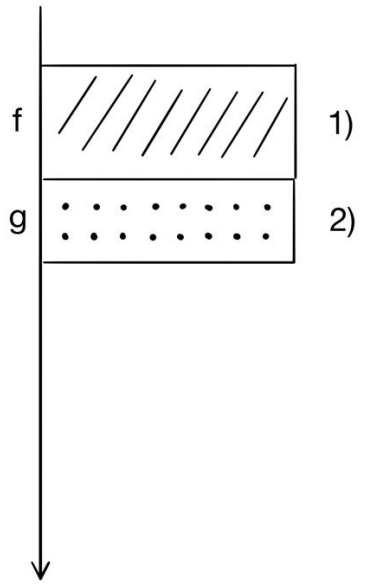


рис. 10.5 завершение работы функции h

Далее вызывается функции *p*, которая, допустим, использует меньше памяти, чем *f* (рис. 10.6).

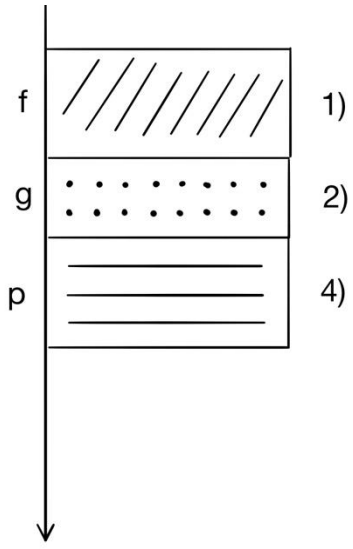


рис. 10.6 вызов функции p

Часть памяти, с которой работает функция *p* (возможно, она захватила больше, возможно, меньше), — это память функции *h*. Поэтому неинициализированные переменные *p* получат мусор, который останется после функции *h*. После завершения работы функции *p* освободится область памяти, выделенная под нее. Аналогично с функцией *g*.

Таким образом, каждый раз при вызове функции в стеке автоматической памяти выделяется область памяти под функцию: ее параметры и локальные переменные. По завершении работы память освобождается. Выделение и освобождение памяти заключается в увеличении и соответственно уменьшении значения верхушки стека sp - места, которое разграничивает занятую и свободную память стека.

Использование автоматической памяти дает нам следующее:

- автоматическая память очень дешевая, чтобы ее выделить достаточно увеличить указатель
- автоматическая память сама никак не инициализируется – ее обязательно нужно инициализировать!

Используя аргументы-указатели, функция может обращаться к объектам вызвавшей ее функции. Использование указателей позволяет избежать копирования сложных структур данных: вместо этого передаются указатели на эти структуры.
Важно запомнить, что параметры в функции, как мы рассмотрели выше, передаются *по значению*, чтобы менять внешние значения в функции нужно передавать параметры *по адресам*, то есть передать адрес параметров.

Пример. Функция void swap(int x, int y); меняет местами значения переменных x и y

Неправильно:	Правильно:

void swap(int x, int y)
{
int tmp;
tmp = x;
x = y;
y = tmp;
}
	void swap(int *px, int *py)
	{
	int tmp;
	tmp = *px;
	*px = *py;
	*py = tmp;
	}

В неправильном подходе модификация всех переменных (на самом деле копий внешний переменных) происходит в автоматической памяти, которая освобождается после завершения функции, соответственно не сохранив значения и не поменяв значения внешних переменных (рис. 10.7).
В правильном подходе мы используем указатели (традиционно 8-байтовые), которые указывают на области памяти в функции main и впоследствии меняют эти переменные.

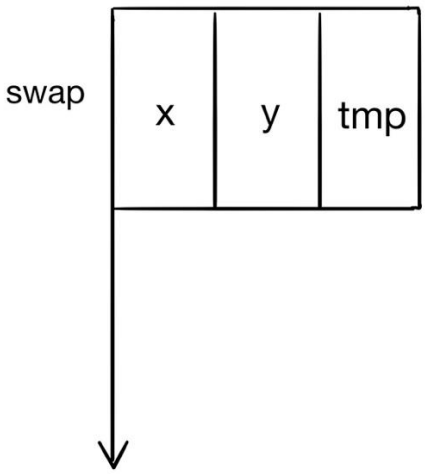


рис. 10.7 неправильный подход

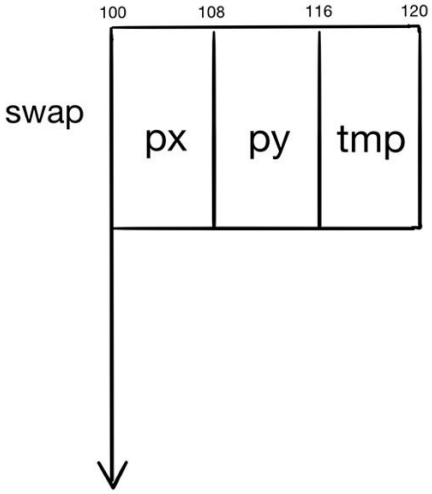


рис. 10.8 правильный подход

Массив всегда передается с помощью указателя на его первый элемент.

```
int asum1d (int a[], int n) {
        int s = 0;
        for (int i = 0; i < n, i++)
          s += a[i];
        return s;
}
```

Чтобы знать количество элементов массива, передаваемого как параметр функции необходимо либо передать количество как параметр функции, либо использовать строку с конечным нулевым элементом.
Можно объявить массив *a* в списке как *const a[ ]*.

Функции с переменным числом параметров объявляются так:
int scanf(const char *, …);
Всегда должен быть явно задан хотя бы один параметр. После многоточия не должно быть других явных параметров. Для обработки переменных параметров используется файл stdarg.h и макросы va_start/va_arg/va_end.

Однако такой подход 1) не очень дешевый, 20 реализуется по-разному на разных архитектурах.

Лекция 11. Рекурсия и указатели на функции

Возврат из функции и результат выполнения

Рассмотрим пример:

```
#include <ctype.h>
int atoi (char *s)
{
        int n, sign;
        for (; isspace (*s); s++);
        sign = (*s == '-') ? -1 : 1;
        if (*s == '+' || *s == ' -')
          s++;
        for (n=0; is digit (*s); s++)
          n = 10 * (*s - '0');
          return sign * n;
}
```

В файле ctype.h хранятся разнообразные функции или макросы для работы с символами. На вход поступает строка, а строка – это массив символов, а в случае массива передается указатель. Первый цикл пропускает все пробельные символы с помощью макроса isspace из файла ctype.h. Далее учитывается и пропускается знак. Следом идет цикл, выполненный через адресную арифметику, который меняет указатель.

Возврат из функции в точку вызывавшей ее функции, следующей за точкой вызова функции, осуществляется:

- либо выполнении оператора **return**
- либо после выполнения последнего оператора функции, если она не содержит оператора **return**

```
#include <string.h>
#include <stdio.h>

void print_str_reverse (char *s) {
        register int i;
        for (i = strlen(s) - 1; i >= 0; i --)
          putchar(s[i]);
}
```

Если тип не void, то в ее теле на каждом пути выполнения должен быть оператор **return**, возврат осуществляется немедленно по тому из них, который будет выполнен первым.
Все функции, кроме тех, которые относятся к типу void, возвращают значение, которое определяется выражением в операторе **return.**
Помимо вычисления возвращаемого значения, функция может изменить значения переменных вызывающей функции (по указателю), а также изменять значения глобальных переменных. Результаты вызова функции, не связанные непосредственно с вычислением возвращаемых значений, составляют *побочный эффект* функции.

Выделяют функции следующих видов:

- функции, которые на сленге компиляторщиков называются purer. Они выполняют операции над своими аргументами с единственной целью – вычислить возвращаемое значение. Они никак не влияют на внешнюю память. В основном это математические функции.
- функции, которые обрабатывают данные и возвращают значение, которое показывает, успешно ли была выполнена эта обработка. Например, функция main
- функции, возвращающие несколько значений (через указатели аргументы и через возвращаемое значение
- функции, не возвращающие значений – все такие функции имеют тип void (в других языках это называется процедурой)

Возвращаемым значением может быть указатель. Требуется, чтобы в объявлении такой функции тип возвращаемого указателя был объявлен точно: нельзя объявлять возвращаемый тип как int *, если функция возвращает указатель типа char *.
Пример функции, возвращающей указатель: поиск первого вхождения символа *c* в строку *s*:

```
char *match (char c, char *s)
{
        while (c != *s && *s)
          s++;
        return s;
}
```

Цикл работает до тех пор, пока присваиваемое значение не станет нулевым, то есть пока не будет достигнут конец строки. Из-за приоритетности сначала произойдет разыменование, а потом сдвинется указатель. Так как операция ++ постфиксная, значение возьмется старое.

Рекурсия

Функция может быть *рекурсивной*, то есть вызывать саму себя.
Локальная память функции foo никак не влияет на локальную память функции bar (рис. 11.1). При окончании вызова функции bar верхушка стека sp переместится наверх, освободив локальную память, занимаемую функцией bar. Если вместо bar будет написано foo, абсолютно ничего не изменится, просто код возврата вернется к началу.

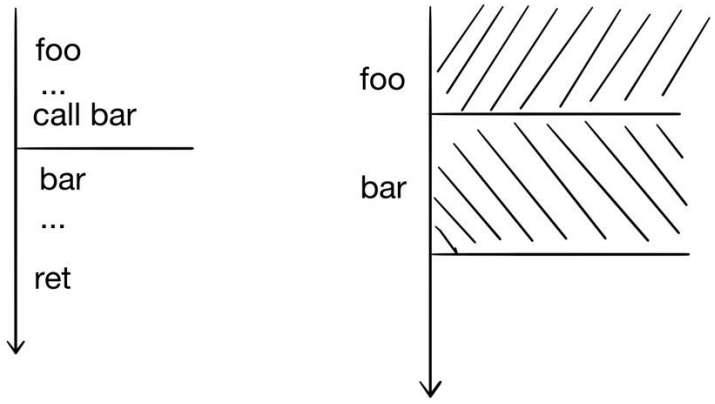


рис. 11.1 область памяти при вызове функции foo

Рассмотрим пример такой функции – числа Фибоначчи.

```
int fib(int n) {
        if (n == 1 || n == 2)
          return 1;
        else
          return fib(n-2) + fib(n-1);

}
```

При вызове fib(n-2) + fib(n-1) первым будет посчитан любое из слагаемых, так как тут нет точек следования. Чтобы рекурсия остановилась в функции должен быть не рекурсивный путь. Чтобы понять, как ведет себя функция и сколько вычислений будет

произведено, рассмотрим конкретный пример с достаточно небольшим числом – 5 (рис. 11.2). При вызове fib(5) сначала вызывается fib(3), который в свою очередь вызывает fib(1) и fib(2).

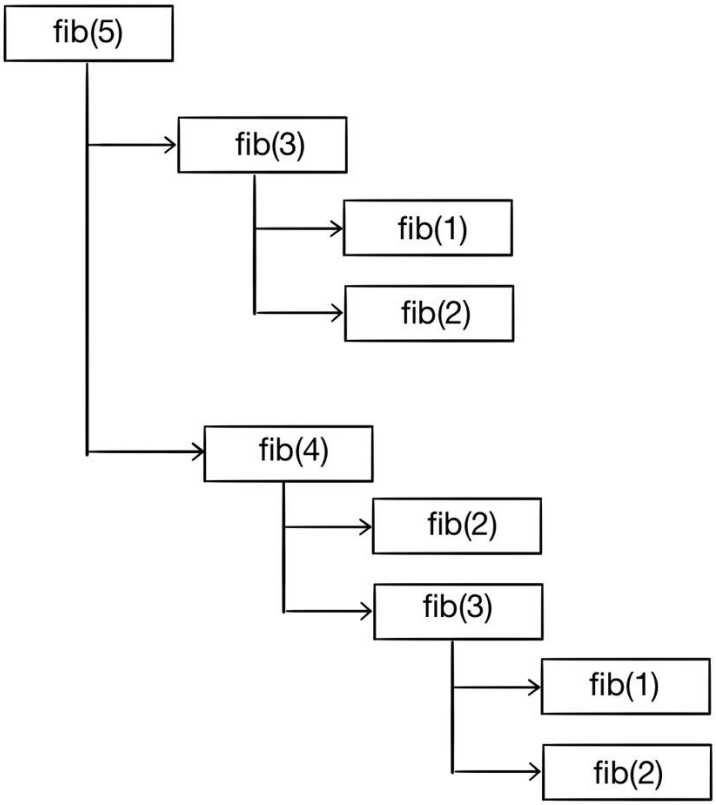


рис. 11.2 рекурсивный вызов fib

При вызове этой функции автоматическая память будет выглядеть следующим образом (рис. 11.3). После окончания вызова fib(1), память, выделенная под 1 освобождается.

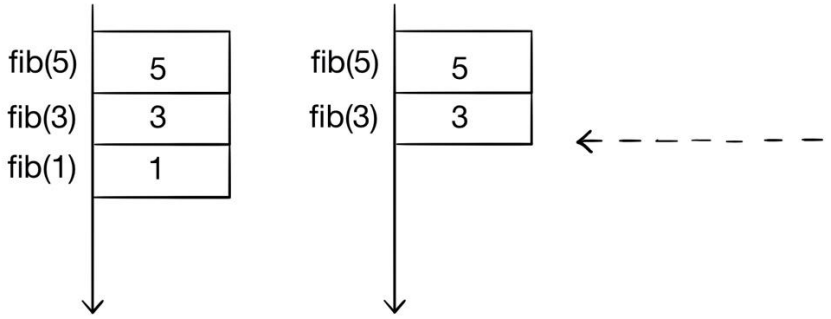


рис. 11.3 вызов fib(5)

Далее происходит вызов fib(2), под 2 будет использована область памяти, ранее выделенная под 1. Аналогично будут происходить следующие вызовы. По рис. 11.2 можно заметить, что fib(2) считается 3 раза, хотя было бы достаточно посчитать 1 раз. Поэтому использование памяти нелинейно и зависит от глубины вызовов. Кроме того, есть затраты на каждый вызов. Часто при большом возрастании объема данных рекурсию переписывают на итерации, чтобы избежать исчерпывания автоматической памяти. Для этого руками заводится структура очень похожая на стек и в нее руками вписываются эти 5, 3, 2,1 итп.

На самом деле числа Фибоначчи никто не вычисляет так, как это было в примере выше. Обычно используют нерекусивный вариант:

```
int fibn (int n) {
        int i, g, h, fb;
        if (n == 1 || n == 2)
          return 1;
        else
          for (i = 1, g = h = 1; i < n; i++) {
                fb = g + h;
                h = g;
                g = fb;
          }
        return fb;
}
```

Функция fib работает за экспоненциальное время и линейную память, функция fibn – за линейное время и константную память.

Более сложным вариантом рекурсии является *хвостовая рекурсия*.
Хвостовая рекурсия (tail recursion) – рекурсивный вызов в самом конце функции. Как правило, этот вызов может быть оптимизирован компилятором в цикл.

```
int fact (int n) {
        if ( n == 0)
          return 1;
        else
          return n*fact(n-1);
}
```

```
int fact(int n) {
        return tfact(n, 1);
}
int tfact(int n, int acc) {
        if (n == 0)
          return acc;
        return tfact(n-1, n*acc)
}
```

Чтобы отличить рекурсию от хвостовой рекурсии нужно, посмотреть на название: в хвостовой рекурсии рекурсивный вызов должен быть в хвосте, то есть в самом конце. При возвращении из хвостового вызова память, занимаемая этой функцией больше не нужна, потому что больше ничего не произойдет. Таким образом, при вызове хвостовых функций (рис. 11.4) можно переиспользовать память. Как только tale1(int x) закончит работу с x=10, мы можем вызвать tale1(x-2), то есть от 8, записав значение в ту же область памяти.

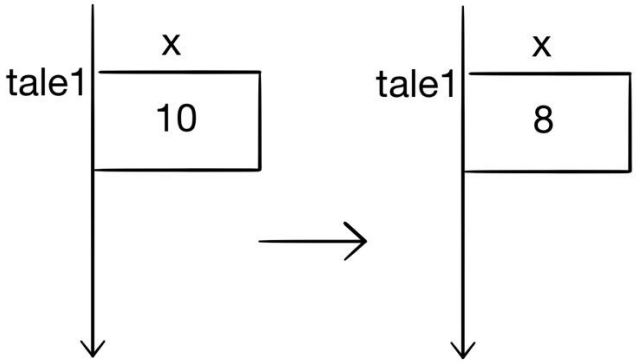


рис. 11.4 вызов tale1

На практике это работает следующим образом:
Возьмем, например, функцию факториал из примера выше. Чтобы превратить ее в хвостовую рекурсию нужно создать «оберточную» функцию fact, которая явно заведет переменную, аккумулирующую результат вычисления факториала. В самой функции tfact при n = 0 возвращается параметр, если n <> 0, то вызовется эта же функция с первым параметром n-1 и вторым параметром n*acc. Это пример хвостовой рекурсии, так как здесь происходит вычисление фактических параметров и вызов функции, после которого уже ничего не происходит.
Генерацию хвостовой рекурсии можно представить в виде следующего цикла:

```
int fact(int n) {
        int t_n = n, t_acc =1;
        /* tfact встроена в fact
        и оптимизирована в цикл */
start:
  if (t_n == 0)
    return t_acc;
  t_acc = t_n * t_acc;
  t_n = t_n - 1;
  goto start;
}
```

Это очень удобно, когда нужно убрать затраты на рекурсивный вызов. В любом языке в том или ином виде есть tail recursion elimination.
Более подробно поговорим о встраиваемых функциях.

Встраиваемые функции. Inline

Встраивание – это альтернативный способ (напряду с хвостовой рекурсией) избавиться от затрат на рекурсивный вызов.
Иногда вызываемая функция обладает несколькими интересными свойствами, например, она вызывается только один раз или она слишком маленькая и затраты на ее вызов превзойдут затраты на ее выполнение. В таких случаях мы хотим избавиться от затрат на вызов операциями подстановки или встраиванием (inline).

```
#include <stdio.h>
inline static int max(int a, int b) {
        return a > b ? a ^ b;
}
int main(void) {
        int x = 5, y = 17;
        printf("Maximum of %d and %d is %d\n",
              x, y, max(x,y));
        return 0;
}
```

Вместо того, чтобы вычислять максимум, можно прямо в точку использования этой функции подставить тело этой функции, заменив формальные параметры на фактические и получив копию тела, которое не универсально и вычисляет только то, что нам нужно. При типичной реализации inline программа будет преобразовываться как

```
#include <stdio.h>
inline static int max(int a, int b) {
        return a > b ? a ^ b;
}
int main(void) {
        int x = 5, y = 17;
        printf("Maximum of %d and %d is %d\n",
              x, y, (x > y ? x : y);
        return 0;
}
```

Такой поход позволяет избавиться от затрат на вызов и передать знание контекста в точку использования функции о происходящем. Однако из-за раздельной компиляции некоторые функции вставить просто не получится, потому что мы не знаем их тело. Кроме того, при встраивании функции в разные точки с разными параметрами в рамках одного файла существенно увеличивается объем. Необходимо находить оптимальные точки для встраивания. Слово **inline** показывает компилятору желание (а не приказ) встроить функцию, однако и без него компилятор может сам догадаться о встраивании. Ключевое слово **inline** появилось в C99.

Указатели на функцию

Каждая функция располагается в памяти по определенному адресу. Адресом функции является точка ее входа (при вызове функции управление передается именно на эту точку). Присвоив значение адреса функции переменной типа указатель, прлучим указатель на функцию.
Указатель функции можно использовать вместо ее имени при вызове этой функции. Указатель «лучше» имени тем, что его можно передавать другим фугкциям в качес тве их аргумента. Имя функции f без скобок и аргументов по определению являет указателем на функциюf() (по аналогии с массивом).

```
int (*pf) (const char*, const char*);
char *s1, *s2;
int x = (*pf) (s1, s2);
int y = pf(s2, "string constant");
```

Чтобы вызвать функцию, на которую указывает указатель, нужно его разыменовать. Однако указателем можно пользоваться так же, как и именем, не разыменовывая его. Рассмотрим пример. В функцию check передается 3 параметра: 2 строчки и параметр pf, указатель на функцию). То есть функция check – это обертка над pf, которая проверяет строки на совпадение.

```
#include <stdio.h>
#include <string.h>
static void check (char *a, char *b, int *pf) (const char*, const char*)) {
        printf("Проверка на совпадение: ");
        if (!pf(a,b))
          printf("равны\n");
        else
          printf("не равны\n");

}
```

Рассмотрим теперь пример использования этой функции.

```
int main(void) {
        char s1[80], s2[80];
        printf("Введите две строки\n");
        fgets(s1, sizeof(s1), stdin); s1[strlen(s1)-1] = 0;
        fgets(s2, sizeof(s2), stdin); s1[strlen(s2)-1] = 0;
        check(s1, s2, strcmp);
        return 0;
}
```

Объявление int (*p) (const char*, const char*); сообщает компилятору, что p – указатель на функцию, имеющую 2 параметра типа const char* и возвращающую значение типа int. Скобки вокруг *p нужны, так как операция * имеет более низкий приоритет, чем (). Если написать int *p(. . .), получится, что объявлен не указатель на функцию, а функция p, которая возвращает указатель на целое.

(*cmp)(a, b) эквивалентно cmp(a,b)

Указатель pf и функция strcmp имеют одинаковый формат, что позволяет использовать имя функции в качестве аргумента, соответствующего параметру pf.

В данном случае использование указателя на функцию позволяет не менять программу сравнения, и тем самым получается более общий алгоритм.

```
int compvalues(const char *f, const char *b) {
        return atoi(a) != atoi(b);
}
```

Массивы указателей на функции подходят для гибкой обработки событий оконных менеджеров. При определённом событии общий код по указателем вызовет все нужные обработчики событий по очереди.

Лекция 12. Структуры и объединения.

Побитовые операции

Самая суть системного программирования заключается в побитовый операциях. Манипуляция битами отдельных чисел позволяет работать с данными на уровне «железа».

- & - поразрядовое И
- | - поразрядовое включающее ИЛИ
- ^ - поразрядовое исключающее ИЛИ
- << - сдвиг влево (слева битовая маска, которую сдвигаем, справа – число, насколько сдвигаем. Возможно возникновение неопределённого поведения при сдвиге на большее число битов, чем есть в левом числе)
- >> - сдвиг вправо

Беззнаковое число – заполнение нулями
Знаковое число – заполнение значением знакового разряда (арифметический сдвиг) или нулями (логический сдвиг)

Обычно все побитовые операции производятся с беззнаковыми числами!

~ - дополнение до 1 или инверсия*

*Некоторые путают ~ с !, но ! – это логическое отрицание, то есть все, что было не 0, станет 0 и наоборот. А побитовое отрицание заключается в замене 0 на 1 (и наоборот) в побитовой записи числа.

hackersdelight.org – сайт, на котором можно посмотреть разные побитовые трюки

x & 1 x | 1 x | (1 << 5) x & (x-1)
x ^= y, y ^= x, x ^= y ~x + 1 x | (x + 1)
Большое количество приемов ускорения вычислений связано с побитовыми операциями, потому что процессор быстро оперирует битами.

Структуры

Структура – это совокупность нескольких переменных, часто разных типов, сгруппированных под одним именем для удобства.

Переменные, перечисленные в объявлении структуры называются ее полями, элементами, или членами.

Объявление структуры:

```
struct point
{
        int x;
        int y;
} f, g;
struct point h, center = {32, 32};
```

До окончания {} структурный тип н является законченным, поэтому нельзя написать в списке полей структуры поле типа struct point. Внутри {} происходит объявление полей структуры. Имя структурного типа – это все выражение вместе с ключевым словом struct – в нашем случае struct point. Поэтому в Си можно иметь локальную переменную point, входящую в struct point, но в C++ это будет ошибкой.

После} могут быть еще несколько переменных, записанных сразу. Это означает, что f и g - \то глобальные переменные типа struct point. Это опционально, после} может стоять просто ;. Слово point тоже опционально и может отсутствовать, тогда структура будет анонимной. У анонимных структур есть одно неприятное свойство: например, если есть две анонимные структуры с одинаковыми по типу, названию и порядку полей, это все равно 2 разные структуры.

Поля структуры могут иметь любой тип, например, тип массива или тип другой структуры.

```
struct rect
{
        struct point pt1;
        struct point pt2;
};
```

Инициализация структуры:

```
struct rect r = {.pt1 = {4, 4},
                 .pt2 = {7, 6}};
/* Остальные элементы --- нулевые */
struct rect r2 = {.pt2.x = 5};
```

Размер структуры в общем случае не равен сумме размеров ее элементов (выравнивание, как правило кратность степени 2).

Рассмотрим пример:

```
 struct foo {
     int i;         // 4
     double d;  // 8
};
```

Чтобы организовать эту структуру в памяти (рис. 12.1), нужно выровнять как int, так и double по адресам, кратным их размерности.

Из-за необходимости выравнивания в памяти будут «дырки». Поэтому размер структуры часто не равен сумме размеров ее элементов.

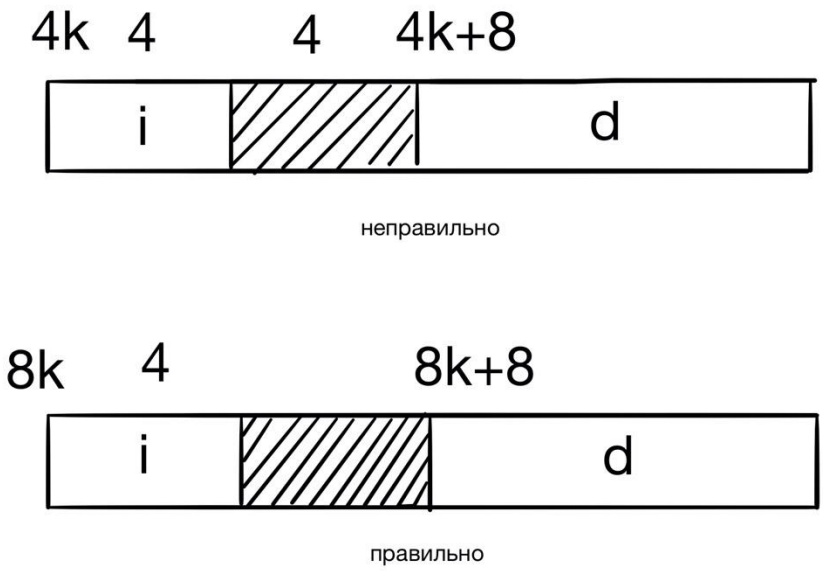


рис. 12.1 расположение структуры foo в памяти

В структурах основной операцией является операция, дающая доступ к полю структуры – операция точка **.** .
f.x, g.y, r.pt1.x

Также возможно присваивание структур целиком: f =g;
Массивы структур с частичной инициализацией:

```
#define NRECT 15
/* Первый прямоугольник вокруг 0, 0 */
struct rect rectangles[NRECT] = {{-1, -1, 1, 1}};
/* Последний прямоугольник ---- большой */
#define BOUND 1024
struct rect bounded_rectangles[NRECT]
      = {[NRECT - 1] = {-BOUND, -BOUND,
                        BOUND, BOUND}};
```

Указатели на структуры

```
struct rect r = {.pt1 = {4,4},
                 .pt2 = {7,6}};
struct rect *pr =&r;
```

Доступ к полям структуры через указатель:

pr -> (= (*pr).pt1) //важно ставить (), так как приоритет **.** выше, что не очень удобно
Поэтому для этого придумали синтаксический сахар, который позволяет записывать это короче pr -> pt1.x

Адресная арифметика:

```
struct rect *pr = &bounded_rectangles[0];
while (pr -> pt1.x != -BOUND)
  pr++;
```

Составные инициализаторы структур (C99)

```
struct rect r;
r = (struct rect) { {4,4}, {7,6} };
```

Составной инициализатор генерирует lvalue!, то есть можно передавать и указатель:

```
double area (struct rect *r) {
          return (r->pt1.x - r->pt2.x)
               *(r->pt1.y - r->pt2.y);
```

```
}
double da = area (&(struct rect) {{4,4}, {7,6}});
```

Приоритет операций

Приоритетность	Ассоциативность
( ) [ ] -> .	Слева направо
! ++ -- + - sizeof(type)	Справа налево
* / %	Слева направо
+ -	Слева направо
<< >>	Слева направо
< <= > >=	Слева направо
== !=	Слева направо
&	Слева направо
.	Слева направо
|	Слева направо
&&	Слева направо
||	Слева направо
? :	Справа налево
= += -= *= /= %=	Справа налево
‘	Слева направо

Объединения

Объединение – это объект, который может содержать значения различных типов (но не одновременно - только одно в каждый момент).

```
struct constant
{
        int ctype;
        union
        {
                int i;
                float f;
                char *s;
        } u;
} sc;
```

```
switch (sc.ctype)
{
        case CI:
         printf("%d", sc.u.i);
         break;
        case CF:
         printf("%f", sc.u.f);
         break;
        case CS: puts(sc.u.s);
}
```

Размер объединения достаточно велик, чтобы содержать максимальный по размеру элемент. С ними можно выполнять те же операции, что и со структурами.
Для вложенных структур и объединений разрешено опускать тег для повышения читаемости.

```
struct constant
{
        int ctype;
        union
        {
                int i;
                float f;
                char *s;
        } /* нет имени! */;
} sc;
```

```
switch (sc.ctype)
{
        case CI:
         printf("%d", sc.u.i);
         break;
        case CF:
         printf("%f", sc.u.f);
         break;
        case CS: puts(sc.u.s);
}
```

Поля анонимной структуры считаются принадлежащими родительской структуре (если родительская также анонимна, то следующей родительской структуре и т.п)

Битовые поля

Для экономии памяти можно точно задать размер поля в битах (например, флагов).

```
struct tree_base {
        unsigned code : 16;
        unsigned side_effects_flag : 1;
        unsigned constant_flag : 1;
        <...>
        unsigned lang_flag_0 : 1;
        unsigned lang_flag_1 : 1;
        <...>
        unsigned spare : 12;
}
```

Здесь представлен большой набор структур, который отвечает за деревья, которые строит компилятор для сбора программы. В общем в языке есть много выражений, для

каждого и них строится дерево, которое показывает взаимосвязь операндов выражения друг с другом. Для разных выражений строятся разные деревья, но есть базовая информация, которая есть во всех деревьях. Эту базовую информацию хорошо «упаковывают». Нужно в начале каждого типа, например, вставить 4 байта, порезать их на кусочки и работать с ними. Если так делается постоянно, то в конце концов захочется, чтобы конкретному биту в памяти предать конкретный смысл, то есть имя, для чего есть битовые поля.
Набор из 64 битов как-то распределяется между базовой информацией, которую нужно хранить во всех деревьях. Например, есть поле code под которое отводится число бит, ратное 8, и так как 8 бит, то есть 256 явно недостаточно, выделяют 16 бит. Также есть набор флажков, под которые выделяются по 1 биту. Также есть поле spare, показывающее, сколько свободного места осталось от 64 бит.

Но возникают свои нюансы: адрес битового поля брать запрещено. Можно объявить анонимные поля для выравнивания и поле шириной 0 для перехода на следующий байт.

Перечисления

Перечисления – целочисленные типы данных, определяемые программистом.
Перечисление определяется следующим образом:

```
enum typename { name[=value], ...};
```

Пример:

```
enum colors {red, orange, yellow, green, azure,
        blue, violet};
```

Значение перечисления нумеруются с 0, но можно присваивать свои значения.

```
enum colors {red, orange = 23, yellow = 23, green, cyan = 75,
        blue = 75, violet};
```

Для перечислений доступны операции над целочисленными типами и объявление указателей на переменные перечислимых типов.
Проверка корректности присваиваемых значений не производится.

Лекция 13. Препроцессор и распределение динамической памяти

Схема раздельной компиляции

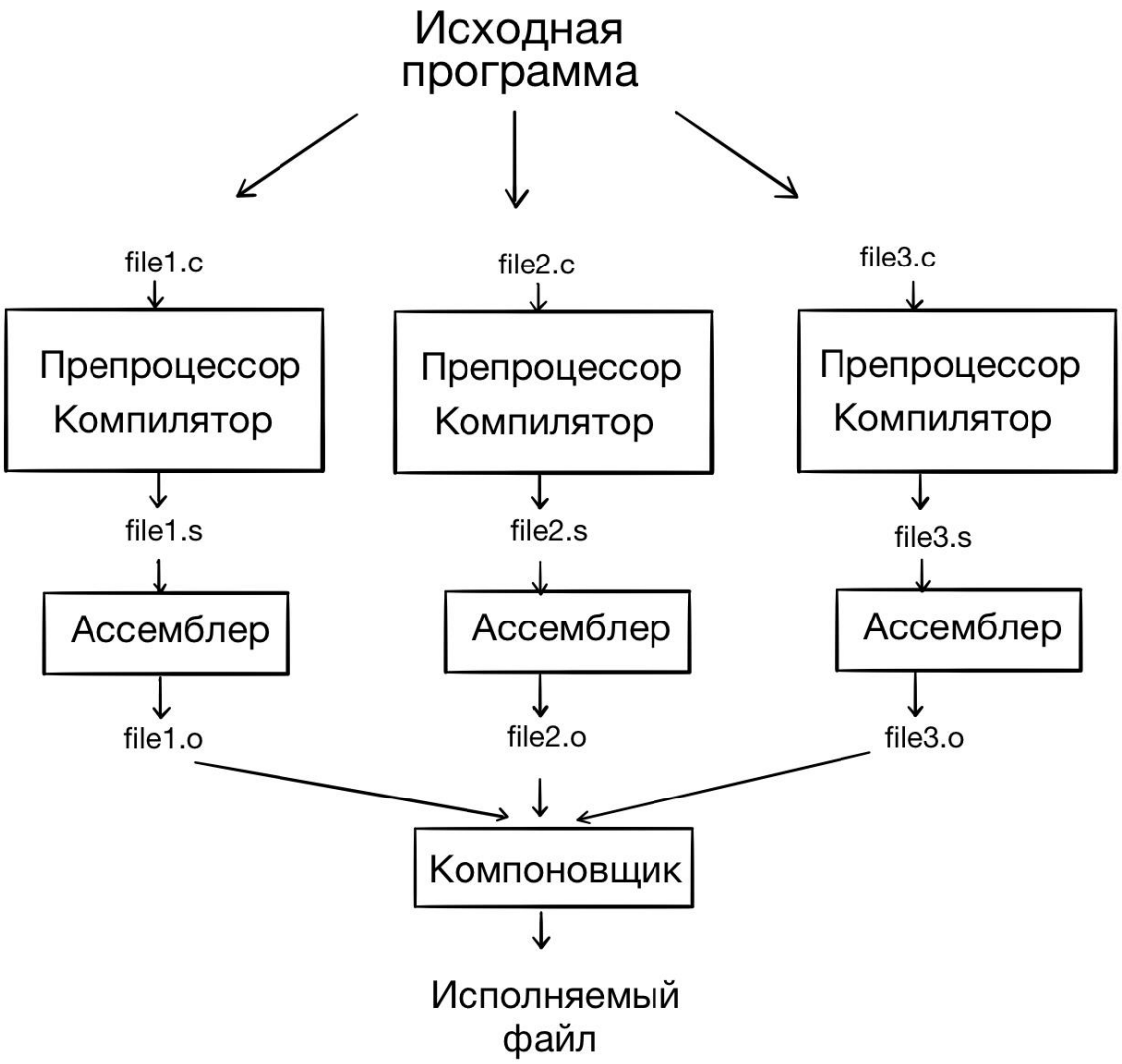


рис. 13.1 схема раздельной компиляции

Компилятор превращает файл на Си в файл на ассемблере. На втором этапе получается бинарный файл. Далее компоновщик (linker) преобразует ассемблерные файлы в исполняемы файл. Помимо компилятора, ассемблера и компоновщика в toolchain выделяют еще отладчики и инструменты, позволяющие работать с бинарным кодом.

Препроцессор

Перед компиляцией выполняется этап *препроцессирования.* Это обработка программного модуля для получения его окончательного текста, который отдается компилятору. Управление препроцессированием выполняется с помощью *директив препроцессора.*

```
#include <...>  // системные библиотеки
#include "..."  // пользовательские файлы
#define name(parametrs) text
#undef name
```

Это работает так: в компиляторе есть уже вшитый набор путей в файловой системе, в которых он обязан искать системные заголовочные файлы. Далее написано имя файла или конца пути, например, stdio.h, которое ищется в различных каталогах, вшитых в компилятор. Для пользовательских файлов есть отдельный набор каталогов, который ищет пользовательские файлы, он обычно пустой и в него можно добавить свои каталоги.

Кроме этого, есть макросы и условная компиляция. Макрос – способ некоторый кусок текста программы соотнести с именем и там, где есть текст этой программы использовать имя. Чтобы это процесс конфигурировать, у макросов, как и у функции могут быть некоторые параметры, и при подстановке формальные параметры будут становиться фактическими.
Например, #define MAX 128 с именем MAX соотнесли число 128, причем для препроцессора 128 – это токен, кусок текста. И каждый раз, когда вы программе будет встречаться MAX, он будет заменяться на число 128. В более сложном случае у тела макроса есть параметр, который обязательно нужно выделить в ().

```
#define ABS(x)  ((x) >= 0 ? (x) : -(x))
```

```
x -> y - 7
ABS(x) -> ((y-7) >= 0 ? (y-7) : -(y-7))
```

Часто тело макросов «приезжает» из #include или начала программы, поэтому не всегда понятно, что именно происходит в теле макроса. Например, что будет, если написать x -> a -- ?
Если у фактического параметра макроса есть побочный эффект, то побочный эффект произойдет 2 раза, так как формальный параметр употребляется в теле макроса дважды. От этого нельзя никаким образом защититься, потому что препроцессор выполняет исключительно текстовую подстановку, не волную о правильности кода.

Условная компиляция важна для многих вещей, в частности для конфигурирования. Часто бывают те или иные объявления или определения, которые должны варьироваться в зависимости от архитектуры. Для этого можно написать несколько кусков текста и в зависимости от условия выбрать или иной кусок. Для этого есть условная компиляция, которая дает разные способы реализации условия для включения того или иного текста в программу.
То есть препроцессор позволяет организовать условное включение фрагментов в программу.
Первое, для чего это используется - include guard. Если в маленьком проекте есть 2 заголовочных файла, то скорее всего вы помните, какие из них включает какие-то другие, но если проект большой и в нем примерно 500 заголовочных файлов, то несложно догадаться, что для объявление функций из одних файлов нужны функции из других файлов и так образуется целое дерево или лес. В результате может появится файл, в котором 10 include. Например, в третьем include включается файл foo.h, и в седьмом include включается файл foo.h. Файл foo.h содержит какое-то количество вещей, которые, например, могут быть повторно включены. Но если в нем содержится какая-нибудь константа, то с точки зрения компилятора вы определяете константу 2 раза в одной и той же области видимости, что делать запрещено. Поэтому делают так: Берут файл (в примере ниже реальный кусок stdio.h), на каждый файл есть уникальное имя и в начале файла вы определяете, определено ли оно. Если оно не определено, то этот файл еще не включался и его нужно тут же определить. Дальше идет текст файла. После определения препроцессор вставляет текст файла в нужное место.
При втором включении этого файла имя уже существует, значит между #ifndef и #endif вообще ничего не нужно вставлять.

#ifdef name /# endif

```
#ifndef _STDIO_H
#define _STDIO_H
<... текст файла ...>
#endif
```

Include guardian обеспечивает, чтобы тело заголовочного файла попало в программу только 1 раз (не более 1 раза).

Более сложное условие препроцессора позволяет считать, связав константу связать ее с числами и записать более сложное выражение. Например, рассмотрим часть кода из gfortran (части gcc, понимающей Fortran). В fortran есть символьный тип, который требует 32-битного int под собой, но в зависимости от архитектуры может быть разный целый тип под 32-битный int).

#if/#if defined /#elif/#else/#endif --- общие проверки условий

```
#if HOST_BITS_PER_INT >= 32
typedef unsigned int gfc_char_t;
#elif HOST_BITS_PER_INT >= 32
typedef unsigned long gfc_char_t;
#elif defined(HAVE_LONG_LONG)
        && (HOST_BITS_PER_LONGLONG >= 32)
typedef unsigned long long gfc_char_t;
#else
#error "Cannot find an integer type with at least 32 bits"
#endif
```

Константа HOST_BITS_PER_INT, по сути, задает число sizeof(int). Если говорить точнее, то в Си есть несколько важных терминов, входящих в toolchain: host – система, на которой работает компилятор, и target – система, на которой будет работать код, который генерирует компилятор. В обычной жизни для нас одно равно другому, но как только вы захотите собрать программу под Android на desktop, одно перестает равняться другому. Таким образом, HOST_BITS_PER_INT – количество бит в целом числе на host, то есть там, где работает компилятор. Например, у нас может быть Linux сервер, на котором стоит gc, который компилирует файлы для Windows.

Вернемся к примеру. Если у нас есть тип, в котором 32 бита, например int, скажем, что это unsigned int. Оператор typedef позволяет определить пользовательский тип. Смысл всего этого if, чтобы после него в gfc_char_t был достаточно большой целый тип для того, чтобы Фортрану было удобно с ним работать, поэтому здесь используется условная компиляция по данным, собранным о хосте. Есть конфигурируемая программа (конфигурирует программу на C или C++ для конкретной хостовой системы), которая после отработки устанавливает в своем заголовочном файле набор констант, которым можно пользоваться. Например, если константа HOST_BITS_PER_INT больше или равна 32, то можно просто этот тип определить как unsigned int. Если это не так, то посмотрим, может, у нас есть LONG, который >= 32, что странно, потому что он не может быть меньше 32 бит. А дальше более сложное условие: может быть LONG_LONG больше 32 бит? Но LONG_LONG появился не сразу и поэтому сначала проверяется переменная HAVE_LONG_LONG, которая просто говорит о том, что тип LONG_LONG определен, и когда он уже есть, проверяется вторая переменная - HOST_BITS_PER_LONGLONG, определяющая количество бит. Если эти оба условия выполнены, тогда можно сделать unsigned long long как gfc_char_t, но если ни одно из выше перечисленных условий не выполнено, то есть #error, который устанавливает ошибку компиляции с последующим сообщением.

У препроцессора есть операции, позволяющие производить интересные операции над строками. Например, операция # позволяет получить строковое представление аргумента.

```
#define FAIL(op)                                    \
  do {                                              \
        fprintf(stderr, "Operation " #op "failed: "          \
              "at file $s, line %d\n", __FILE__, __LINE__);     \
        abort();                                    \
  } while(0)
  int foo(int x, int y) {
        if (y==0)
          FAIL(division):
        return x/y;
  }
```

Макрос раскроется так:

```
  do {
        fprintf(stderr, "Operation" "division" "failed: "
              "at file %s, line %d\n", "fail.c", 13);
        abort();
  }
  while(0);
  }
```

В этом примере # позволяет из параметра макроса сделать константу. Макрос не может быть многострочным в классическом понимании, но его можно написать на несколько физический строк в файле, потому что не очень удобно иметь строку из 1000 символов. Чтобы это сделать, нужно в конце каждой строки ставить \. Главное не ставить после него пробел, иначе макрос превращается в непонятное тело и препроцессор перестает его понимать и выдает очень смешные ошибки.
Тело макроса в нашем примере завернуто в цикл do while(0). Зачем?
На самом деле можно было бы написать просто fprintf и abort(), завершающий программу с сообщением об ошибке. Но тело макроса должно быть написано таким образом, чтобы после него можно было поставить ; и с точки зрения языка он был statement, а do while(0) является прекрасным примером statement и заведомо выполнится 1 раз.

Операция ## позволяет объединить фактические аргументы макроса в одну строку. Ниже приведен кусок gcc для java, который уже не поддерживается. Этот макрос описывает какую-то и операция на байт-коде машины. Этот макрос превращается в одно описание одной констатнты.

java-opcodes.h:

```
 enum java_opcode {
 #define JAVAOP(NAME, CODE, KIND, TYPE, VALUE) \
       OPCODE_##NAME = CODE,
 #include "javaop.def"
 #undef JAVAOP
 LAST_AND_UNUSED_JAVA_OPCODE
 };
```

javaop def:

```
 JAVAOP (nop,                0, STACK,  POP,    0)
 JAVAOP (aconst_null,        1, PUSHC,  PTR,    0)
 JAVAOP (iconst_m1,          2, PUSHC,  INT,   -1)
 <...>
 JAVAOP (ret_w,            209, RET,    RETURN, VAR_INDEX_2)
 JAVAOP (impdep1,          254, IMPL,   ANY,    1)
```

При разворачивании макроса, например, берется макрос JAVAOP (nop…) и заводится строка JAVAOP_nop = 0.
Этот макрос может раскрываться по-разному в зависимости от того, что нам нужно. Сам файл "javaop.def" может включаться несколько раз, если, конечно, не использовать include guard, с разными определениями макроса JAVAOP. Тогда тело его файла будет представлять из себя набор констант, который мы не будет трогать, но его смысл будет каждый раз меняться, позволяя нам заводить разный тип констант.
В итоге получится следующий список кодов java-машины:

gcc -E java-opcodes.h:

```
 enum java_opcode {
 OPCODE_nop = 0,
 OPCODE_aconst_null = 1,
 OPCODE_iconst_m1 = 2,
 OPCODE_iconst_0 = 3,
 <...>
 OPCODE_impdep2 = 255,
 LAST_AND_UNUSED_JAVA_OPCODE
 };
```

Константа LAST_AND_UNUSED_JAVA_OPCODE в конце позволяет завести массив, который достаточно велик, чтобы хранить все члены перечисления, потому что нам гарантируется, что счетчик, который становится константой значений очередного перечислимого типа увеличивается на 1. Поэтому часто, чтобы узнать количество перечислений, заводят подобную константу или заводят циклы/ массивы нужного размера.

Компоновка и классы памяти

У всех переменных есть классы памяти (storage), время жизни (lifetime), области видимости, компоновка (linkage).

Класс памяти	*Время жизни*	*Видимость*	*Компоновка*	*Определена*
автоматический	автоматическое	блок	нет	в блоке
регистровый	автоматическое	блок	нет	в блоке как register
статический	статическое	файл	внешняя	вне функций
статический	статическое	файл	внутренняя	вне функций как static
статический	статическое	блок	нет	в блоке как static

Квалификатор extern: переменная определена и память под нее выделена в другом файле.
Функции, помеченные static, нельзя вызвать извне, они не участвуют в процессе компоновки. Функция main именно поэтому не static, потому что ее кто-то должен вызвать снаружи.

У функций выделяют следующие классы памяти:

- *статическая* (квалификатор static)
- *внешняя* (extern), по умолчанию
- *встраиваемая* (inline, C99)

Объявление *внешних* функций в заголовочных файлах происходит следующим образом:
**extern void ***realloc (void *ptr, size_t size);

Компоновщик

Рассмотрим весь процесс компоновки.

1. Компоновщик организовывает слияние нескольких объектных файлов в одну программу
2. Он разрешает неизвестные символы (внешние переменные и функции). При этом:
   – Глобальные переменные с одним именем получают одну область памяти
   - Происходят ошибки, если необходимых имен нет или есть несколько объектов с одним именем
   - Есть опции для указания места поиска
3. Компоновщик собирает исполняемый файл или библиотеку (статическую или динамическую).
   Обычная библиотека — это статическая библиотека. Если этой библиотекой пользуются несколько программ, то возникает несколько копий этой библиотеки, что приводит к значительному потери памяти и скорости обработки. Поэтому если библиотека популярна и используется несколькими программами, ее делают динамической. Она загружается в память только один раз, не создавая лишних копий. При этом на момент запуска еще нет полностью готовой программы, потому что из нее выходят вызовы наружу, к «воображаемой» динамической библиотеке. Чтобы не загружать копию, мы должны понять, что нужная нам библиотека уже загружена, найти, где она загружена и адреса, которые уже есть в памяти, положить в нашу программу. То есть дополнительная работа должна произойти не на момент компиляции, а на момент запуска. Таким образом, все, что происходит до запуска -это статика, после - динамика. В случае динамики используется динамический компоновщик.

Хорошим стилем программирования является экспорт лишь тех объектов, которые используются в других файлах (интерфейс модуля).

Динамическое выделение памяти

Динамическое выделение памяти происходит с помощью функции
void *malloc (size_t size);
которая выделяет область памяти из кучи размером size байтов и возвращает указатель на выделенную область памяти.
Если память не выделена (например, в системе не осталось свободной памяти требуемого размера), возвращаемый указатель имеет значение NULL.
Поскольку результат операции sizeof имеет тип size_t и равен длине операнда в байтах, в качестве size можно использовать результат операции sizeof.

```
char *p;
p = (char*) malloc(1000 * sizeof(char));

int *p;
p = malloc (50* sizeof(int));
```

Управление динамической памятью происходит через malloc и free. Функция
void free(void *p);
возвращает системе выделенный ранее участок памяти с указателем p.

Важно! Аргументом функции free() обязательно должен быть указатель p на участок памяти, выделенный ранее функцией malloc()

- Вызов функции free() с неправильным указателем не определён и может привести к разрушению системы распределения памяти.
- Вызов функции free() с указателем NULL не приводит ни к каким действиям
- Обращение к освобожденному указателю не определено

Функции malloc() и free() объявлены в stdlib.h

Служебные данные менеджера памяти лежат до выделенной области памяти, на них указывает указатель q (рис. 13.2). Функция free() получает указатель p и переводит его назад на фиксированную длину (длину служебных данных). Если передать в функцию free() указатель, не выделенный malloc(), то сотрется середина библиотеки или какой-нибудь заголовок, так как функция не обнаружит то, что что туда было изначально положено malloc().

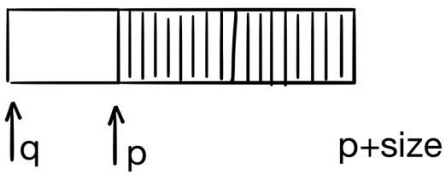


рис. 13.2 указатель p и область памяти

Пример.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main (void) {
```

```
        int t;
        char *s = malloc(80*sizeof(char));
        if (!s) {
                fprintf(stderr, "требуемая память не выделена\n");
                return 1; /* исключительная ситуация */
        }
        fgets(s, 80, stdin); s[strlen(s)-1] = '0\';
        // посимвольный вывод перевернутой строки на экран
        for (t=strlen(s)-1; t>=0; t--)
          putchar(s[t]);
          dree(s);
          return 0;
}
```

Обязательно нужно обрабатывать ситуацию, когда malloc возвращает null. Чтобы не писать после каждого malloc проверки, обычно делают пользовательские надстройки над malloc(). В gcc и gnu есть функции xmalloc(), то есть extended malloc, который уже внутри себя содержит проверку. В конце память освобождается. Ленивые люди не используют free() в main, потому что пере возвратом управления ОС вся память автоматически будет освобождена. Новы не будьте ленивыми людьми, потому что в нормальной ситуации память всегда нужно освобождать!
После вызова free(s) указателем s уже нельзя пользоваться, так как это спровоцирует неопределенное поведение.

Лекция 14. Отладка программы

Malloc() выделяет необходимый объем памяти для последующего использования любым типом. Но для каждого типа необходимо свое выравнивание, как мы обсуждали ранее. Как malloc() обеспечивает выравнивание?

Вспомним, что ранее, обсуждая выравнивание в случае int и double, мы пришли к тому, что выравнивание происходит по большему типу - double и образуются дырки. Такая же стратегия применятся и malloc(). Выравнивание происходит по самому большому типу.

Динамическое выделение памяти для двумерного массива

Выделить память под двумерный массив можно разными способами. Например, функция, которая возводит число в степень:

```
#include <stdio.h>
#include <stdlib.h>

long pwr(int a, int b) {
        long t = 1;
        for (;b; b--)
          t *= a;
        return t;
}
```

На самом деле она заполняет массив степенями:

```
int main(void) {
        long *p[6]; int i, j;
        for (i = 0; i < 6; i++)
          if (!(p[i] = malloc(4*sizeof(long)))) {
                printf("out of memory...\n");
                exit(1);
          }
        for (i = 1; i < 7; i++ )
          for(j = 1; j < 5; j++)
            p[i-1][j-1] = pwr(i,j);
        for(i = 1; i < 7; i++ ) {
                for(j = 1; j < 5; j++)
                  printf("%10ld ", p[i-1][j-1]);
```

```
        printf("\n");
    }
    <...>
```

Массив long *p[6] - это 6 указателей на long. Далее под каждый из этих указателей выделяется память. Если после этого malloc вернет null, то память кончилась. Далее происходит работа с массивом и освобождение памяти. Это самый простой способ. Чем он плох?
На самом деле в памяти происходит следующее (рис.14.1):

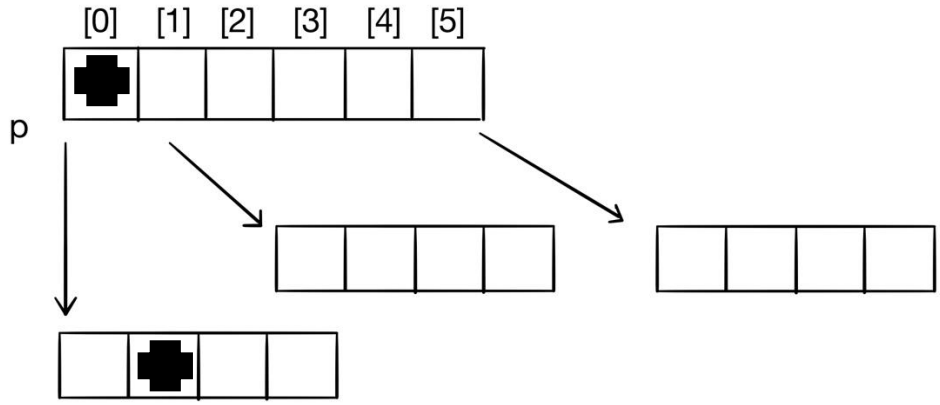


рис. 14.1 двумерный массив в памяти

При записи p[i][j] происходит следующее:
Возьмем, например, p[1]. Для этого берем адрес массива и считаем смещение. Ячейка p[1] сама является указателем. Перейдя по указателю p[1] и найдя нужное смещение, получим указатель на нужную область памяти p[1][2]. В итоге получается 2 разыменования указателей, что не очень хорошо, потому что любое разыменование - это загрузка из памяти и чего-то стоит. Кроме того, нарушается локальность пространства памяти: вместо того, чтобы иметь один кусок памяти, у нас получается 6 маленьких кусочков памяти, которые не обязаны лежать рядом.
Как это можно улучшить?
Первый способ: выделить один большой кусок памяти, в котором будет нужное количество строчек и столбцов. У нас снова будет массив p, указатели которого вручную поставим в соответствие определенным строкам.

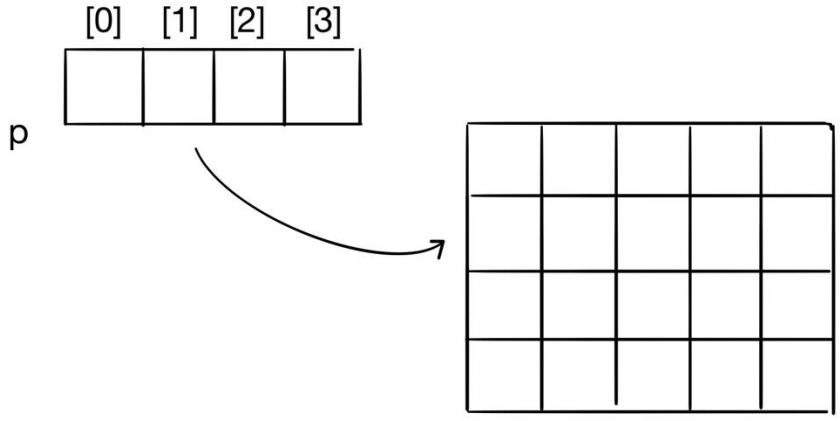


рис. 14.2 двумерный массив в памяти реализация 2

Такой подход решает проблему локальности памяти, но проблема повторного разыменовывания остается.
Второй вариант: сделать один указатель long *p, вызвать снова большой блок памяти в виде массива и вручную сделать все то, что делал компилятор при p[i][j] (рис. 14.2). В этот раз во избежание второго разыменовывания смещение придется считать самим. То есть *p будет указывать на начало массива, а перемещение внутри массива организовываем вручную. Этот подход решает 2 проблемы, но с точки зрения кода является корявым.

VLA-массивы

В C-89 размер массива обязан являться константной, что неудобно при передаче массивов (многомерных) в функции.
Чтобы написать функцию, которая суммирует элементы одномерного массива, передаем массив в эту функцию и его размер.

```
/* можно передать int a[5]; int a[42]; ... */
int asum1d(int a[], int n) {
        int s = 0;
        for (int i = 0; i<n; i++)
          s+= a[i]
         return s;
}
```

Чтобы просуммировать элементы двумерного массива, нужно сгенерировать выражение a[i][j], для этого надо знать первый индекс, чтобы далее умножить его на размер второй части. В C89, чтобы передать такое в функцию, нужно знать все

измерения кроме последнего или первого. Например, int a[][5], int n, иначе sizeof работать не будет. В этом случае s += a[i][j]; это не двойное разыменование, а подсчет индексов в массиве из массивов.

```
/* можно передать только int a[???][5] */
int asum2d(int a[][5], int n) {
        int s = 0;
        for (int i = 0; i<n; i++)
        for (int j = 0; j<5; j++)
          s += a[i][j];
        returns;
}
```

Это неудобно, потому что для массива другого измерения придется писать другую функцию с другим циклом. Поэтому в C99 размер массива автоматического класса памяти уже может задаваться во время выполнения программы (C11 делал VLA необязательными, проверка через макрос __STDC_NO_VLA__). То есть размер массива определяется локальными переменными или параметрами.

```
int foo(int n) {
        int a[n];
        // можно обрабатывать a[i]...
}
```

Из опасностей здесь можно выделить следующее:

- Размеры перестают быть константными, адресное выражение, генерируемое компилятором, значительно усложняется, потому что вместо выражения: база + константа появляется выражение: вычисленное значение + база, которая зависит от входных параметров и не может быть вычислена заранее.
- Размер стека ограничен, миллионные массивы - это уже грань.

Теперь можно использовать это в функции asum2d. Передаем 2 параметра, которые являются размерностью массива, и третьим параметром передаем VLA массив с этой размерностью.

```
// можно передавать int a[???][???]
int asum2d(int m, int n, int a[m][n]){
        int s = 0;
        for (int i = 0; i < m, i++)
          for (int j = 0, j < n; j++)
            s+= a[i][j];
```

```
        return s;
}
int asum2d (int m, int n, int a[m][n]);
```

В случае безымянного оформления для VLA массива необходимо указывать третьим параметром [*][*].

```
int asum2d (int, int, int [*][*]);
```

Функция asum2d может использоваться с VLA-массивами, но они всегда выделяются в автоматической памяти.

```
int foo(int m, int n) {
        int a[m][n]; int s;
        <...Считаем a[i][j]...>
        s = asum2d(m,n,a)
        return s;
}
```

Можно выделить VLA-массив в динамической памяти. Для этого заводится указатель на VLA-массив в n штук int (*pa)[n]; и выделяется блок памяти. После преобразуем это к указателю на VLA-массив:

```
int main(void){
        int m, n;
        scanf("%d%d", &m, &n);

        int (*pa)[n];
        pa = (int (*)[n]) malloc(m*n*sizeof(int));
        <...считаем pa[i][j]...>
        s = asum2d(m,n,pa);
        free(pa);
        return 0;
}
```

VLA-массивы часто нужны для математики, когда необходимо иметь дву-или-трехмерные массивы. Мы генерируем типы, похожие на эти массивы, и для них автоматическая память приведет к генерации ровно таких же адресных выражений, как те, что мы писали вручную выше.

Функция calloc

Функция calloc борется с тем, что функция malloc() не инициализирует память. malloc() не инициализирует память, потому что C спроектирован так, чтобы ничто не стояло между производительностью и вами. Заполнять выделенный гигабайт нулями - трата времени программы.
Состав функций динамического распределения памяти (заголовочный файл <stdlib.h>).

```
void *malloc(size_t size);
void free(void *p);
void *realloc(void *p, size_t size);
void *calloc(size_t num, size_t size);
```

Функция calloc работает аналогично функции malloc(size1), где size1 = num * size (т.е. выделяемая память для размещения массива из num объектов размера size).
Выделенная память инициализируется нулевыми значениями. Возврат null означает, что свободной памяти нет.

Функция
void *realloc(void *p, size_t size)
сначала выполняет free(p), а потом p = malloc(size), возвращая новое значение указателя p. При этом значения первый size байтов новой и старой областей совпадают.
Она нужна, когда мы не знаем объем и время жизни.
Параметр p должен был раньше прийти из системы управления памятью, второй параметр - размер, который мы хотим, чтобы блок памяти занимал теперь (это размер может быть как больше, так и меньше). С точки зрения стандарта эта функция ведет себя следующим образом: освобождает переданную память, выделяет память нового размера, при этом перенося то, что было в старой памяти, в новую, но не все, а только первый size байт.
Важно понимать и помнить, что realloc() тоже возвращает указатель.
Рассмотрим пример:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
        int *p = (int*)malloc(sizeof(int));
        int *q = (int*)realloc(p, sizeof(int));
        *p=1;
        *q =2;
        if (p == q)
          printf("%d %d\n", *p, *q);
```

```
        return 0;
}
```

Под p выделена память в один int. Под q с помощью realloc выделана память в другой int, то есть указатель p переделан так, чтобы он занимал int, и присвоен q. После этого вызова доступ к p запрещен. А этот код нарушает это правило, записывая в p 1, то есть мы пользуемся p, сделав free(p).
В итоге программа выведет следующее:
$ clang -02 realloc.c && ./a.out
1 2
Как «думает» компилятор: память *p «протухла», поэтому к ней обращаться больше не буду. Раз так, то можно все ссылки на эту память заменить на записанную константу 1. Условие p == q выполнилось, потому что realloc() ничего не поменял: было выделено 4 байта и снова выделили 4 байта - ничего менять не надо.

Но так if (p == q)

```
        printf("%d %d\n", *p, *q);
```

записывать тоже правильно, потому что если q = realloc(p, …); вернет null, то есть свободной памяти больше нет, то значение p «не протухнет». Здесь по-хорошему следует сделать так:

```
if (!a) {
  free)p);
  }
```

Кстати, realloc(null) тоже самое, что malloc(). Обычно эти проверки заворачивают в xrealloc(). В принципе если вы уверены, что realloc() никогда не вернет null, то можно писать p = realloc(p, size), но в реальной жизни так делать не стоит.

Массив переменного размера в структуре (C99)

Основной тип данных в C — это структуры, а не массивы. Допустим, у нас есть структура с полями базовых типов и не базовых (особенно если это какой-то пакет, например, сетевой или кусок видеоряда). Под эту структуру как-то расчерчена память (рис. 14.3), а дальше идет массив, размер которого мы не знаем и поэтому используем указатель int *a. Под *a* выделяем указатель размера n. Далее есть указатель struct foo*pf, который указывает, например, на поле x. Чтобы залезть в середину массива, нужно записать pf -> a[i];. Поэтому указателю сделаем одно разыменование, найдем смещение и перейдем в поле, где хранится адрес начала этого массива, перейдем по этому адресу, снова прибавим смещение и получим нужный элемент. Получается, что у нас происходит 2 разыменования.

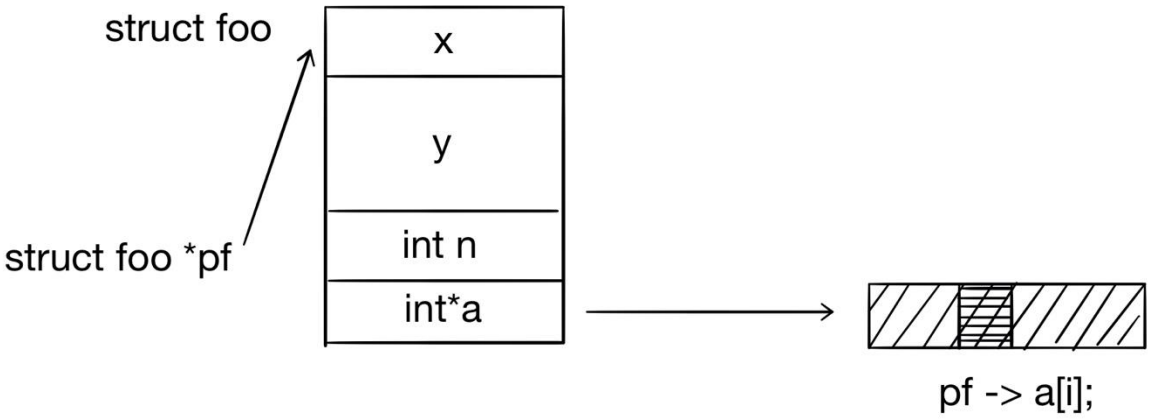


рис. 14.3 массив переменного размера в структуре

Если этот массив будет лежать в хвосте структуры, то можно в поле массива сделать массив *a* сделать из одного элемента, выделить память под эту структуру (с коротким массивом из 1 элемента) pf = malloc(sizeof(struct foo)); pf -> n=…;
Чтобы выделить память под массив из n элементов, нужно написать следующее: pf = malloc(sizeof(struct foo) + n*sizeof(int)). Вообще говоря, выделенная дополнительно память будет лежать вне структуры, после нее (рис. 14.4).

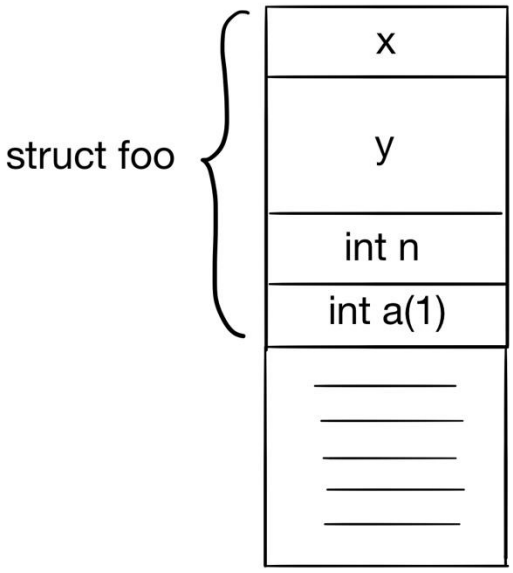


рис. 14.4 массив в другой реализации

Так как начало массива и все остальные элементы выравнены, можно дальше работать с этой памятью. В этом случае мы избавились от двойного разыменовывания, но с

такой структурой ничего нельзя сделать нормально, например, скопировать простым методом. В то время каждый писал по-своему: int a[1], int a[0]. Так было до C99.

В C99 стало можно писать int a[].

Flexible array member - последнее поле структуры.

```
struct polygon {
        int np;    /* число вершин */
        struct point points[];
}
```

Варьирование размера переменного массива:

```
int np; struct polygon *pp;
scanf("%d", &np);
pp = malloc(sizeof(struct polygon)
        + np* sixeof(struct point));
pp ->np = np;
for (int i=0l i < np; i++)
  scanf("%d%d", &pp -> points[i].x,
       &pp -> point[i].y);
```

Отладка программ

Все программы содержат ошибки.
Отладка — это процесс поиска и удаления (некоторых) ошибок.
Существуют другие методы обнаружения ошибок (тестирование, верификация, статические и динамические анализаторы кода), но их применение не гарантирует отсутствие ошибок.
Для отладки используют инструменты, позволяющие получить информацию о поведении программы на некоторых входных данных, не изменяя их поведения.

Есть программы, которые отладчиком не отлаживаются. Самый базовый прием - вставить в программу отладочную функцию, которая покажет, что происходит в массиве в каждой точке программы. Это не очень удобно, потому что нужно делать вручную, но

```
static void debug_array(int *a, int n) {
        fprintf(stderr, "Array (%d)", n);
        for (int i = 0; i < n; i++)
```

```
        fprintf(stderr, "%d", a[i]);
    fprintf(stderr, "\n");
}
```

Проверка инвариантов программы производится с помощью макроса assert (контролируется макросом NDEBUG). Например, если вы построили классическое дерево, логично проверить, что нет ссылок, ведущих от родителей к детям. Если они есть, то дерево построено неправильно. Нежелательно использовать выражения с побочным эффектом.
Есть 2 варианта работы assert: отладочный и боевой. Переключение между вариантами осуществляется макросом debug(). Если он есть, то assert ничего не делает. Если его нет, то выражение вычисляется и, если есть проблемы, программа останавливается с сообщением об ошибке.

```
#include <assert.h>
int foo(int *a, int n) {
    assert(n > 0);
    <...>
    debug_array(a, n);
}
```

Отладчик позволяет:

- Запустить программу для заданных входных данных
- Останавливать выполнение по достижении заданных точек программы *безусловно* или при выполнении некоторого *условия* на значения переменных (breakpoints)
- Останавливать выполнение, когда некоторая переменная изменяет свое значение (watchpoints)
- Выполнить текущую строку исходного кода программы и снова остановить выполнение
- Посмотреть/изменить значения переменных, памяти
- Посмотреть текущий стек вызовов

Компиляция с отладочной информацией: gcc -g. Команды gdb:

- **gdb <file> --args <args>** -загрузить программу с заданными параметрами командной строки
- **run/continue** - запустить/продолжить выполнение
- **break <function name/file:line number>** - завести безусловную точку останова
- **cond <bp#> condition** - задать условие остановки выполнения для некоторой точки останова

- **watch <variable/address>** - задать точку наблюдения (остановка выполнения при изменении значения переменной или памяти по адресу)
- **next/step** - выполнить текущую строку исходного кода программы без захода/с заходом в вызываемые функции
- **print <var>/set <var> = expression** - посмотреть/изменить текущие значения переменных, памяти
- **bt** - посмотреть текущий стек вызовов

Установка точек останова (можно использовать '.' вместо '->'):

```
b fancy_abort
b 7199
b sel-sched.c:7199
cond 2 insn.u.fld.rt_int == 112
cound 3 x_rtl -> emit.x_cur_insn_uid == 1396
```

Просмотр и изменение значений переменных:

```
p orig_ops.u.expr.history_of_changes.base
p bb -> index
set shed_verbose = 5
call debug_vinsn(0x4744640)
```

Установка точек наблюдения:

```
wa ca_issue_more
wa ((basic_block) 0xyffff58b5680) -> preds.base.prefix.nim
```

Частые ошибки работы с динамической памятью тяжело отлаживать (даже в небольших программах). Это:

- ошибки доступа за границы буфера
- ошибки использования неинициализированного или уже освобожденного указателя
- недостаточный размер буфера

Разработан ряд инструментов анализа, которые облегчают жизнь программисту*:

- valgrind: динамический двойной транслятор
  htttp://valgrind.org
  Предупреждает, есть ли доступ к массиву за пределами к массиву, есть ли доступ к указателю, который был освобожден

- sanitizers: компиляторная инструментация от Google
  https://github.com/google/sanitizers/wiki

*Linux only
На Windows работает Dr.Memory https://drmemory.org/

Лекция 15. Вещественные числа в Си

Вернемся к поиску ошибок работы с памятью. Valgrind представляет из себя динамический двоичный транслятор (плюс набор инструментов, наш - memcheck)

```
#include <stdlib.h>
void f(void) {
        int* x - malloc(10*sizeof(int));
        x[10]=0;          //problem 1: heap block overrun
}
int main(void) {
        f();
        return 0;
}
```

Программа выше выделяет память для массива из 10 int и потом совершает ошибку off-by-one

```
$ gcc -Og -g -j me && vakgrind ./me
<...>
==27164== invalid write of size 4
==27164==  at 0x400554^ f(me.c:7)
==27164==  by 0x400568: main(me.c:4)
==27164==  address 0x51da068 is 0 bytes after a block of size 40 alloc'd
==27164==  at 0x4C2C12F: malloc(in /usr/lib64/valgrind/bgpreload_memcheck
==27164==  by 0x400553: f(me.c:3)
==27164==  by 0x400568: main(me.c:7)
```

Sanitizers - встроенная в gcc/clang инструментация. Рассмотрим address sanitizer для программы выше:

```
$ gcc -Og -g -fsanitize-adress -o mesa && ./mesa
 ==27179==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60400000dff8
 WRITE of size 4 at 0x60400000dff8 thread To
   #0 0x4007c0 in f/home/bonzo/tmp/me.c:4
   #1 0x4007d5 in main/home/bonzo/tmp/me.c:7
   #2 0x7fba219d870f in __libc_start_main (/lib64/libc.so.6+0x2070f)
   #3 0x4006b8 in _start(home/bonzo/tmp/mesa+0x4006b8)
0x60400000dff8 is located 0 butes to the right of 40-byte region
allocated by thread To here:
   #0 0x7fba21df074a in malloc(/usr/lib64/libasan.so.2+0x9674a)
   #1 0x400793 in f /home/bonzo/tmp/me.c:3
```

SUMMARY: AddressSanitizer: heeap-buffer overflow /home/bonzo/tmp/me.c:4 f

Для того, чтобы понимать, что происходит со всем доступом к памяти, заводится теневая память, где ставятся флаги на все возможные адреса, отмечая, что с ними произошло. Код, который все проверяет вставляется не в бинарный, а в компилятор, вовремя компиляции, поэтому нужно передавать флаг.
Санитайзеров очень много на разные проблемы.

Вычисления с плавающей точкой

Что такое 1011.101_2?

$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 11\frac{5}{8} = 11.625$

$0.1111 \ldots_2 = 1.0 - \varepsilon(\varepsilon \to 0)$, т.к. $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{2^n} \to 1$ при n$\to \infty$

Точно можно представить только числа вида $\frac{x}{2^R}$. Остальные рациональные числа представляются периодическими двоичными дробями: $\frac{1}{5} = 0.(0011)_2$.

Иррациональные числа представляются апериоичсекими двоичными дробями и могут быть представлены только приближенно.

Числа с плавающей точкой представляются в *нормализованной* форме:

$$(-1)^s M 2^e$$

- s - код знака числа (он же знак мантиссы)
- M - мантисса (1≤M<2)
- e - (двоичный) порядок

Первая цифра мантиссы в нормализованном представлении всегда 1. В стандарте принято решение не записывать в представление числа эту единицу, тем самым мантисса как бы увеличивается на разряд).
В представление числа записывается не M, a frac = M-1.
Чтобы не записывать отрицательных чисел в поле порядка, вводится смещение $bias = 2^{k-1}$, где *k* - количество бит в поле записи порядка, и вместо порядка *e* записывается *код* порядка *exp*, связанные с e соотношением: *e = exp - bias.*
Нормализованное число $(-1)^s M 2^e$ упаковывается в машинное слово с полями *s, frac, exp.*

S	*exp*(код порядка)	*frac(*код мантиссы)

Ширина поля s всегда равна 1. Ширина полей *frac* и *exp* зависит от точности числа.

Типы плавающей арифметики (точность)

1. Одинарная точность - int (32 бита): exp - 8 бит, frac - 23 бита
   bias = 127, -126 ≤ e ≤ 127, 1≤ exp ≤ 254
2. Двойная точность - double (64 бита): exp - 11 бит, frac - 52 бита
   bias = 1023, -1022 ≤ e ≤ 1023, 1≤ exp ≤ 2046
3. Повышенная точность (80 бит): exp - 15 бит, frac - 64 бита

Рассмотрим пример:
float f = 15213.0;
$15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$
M = 1.1101101101101_2
frac = 11011011011010000000000
e = 13, bias = 127, exp = 140 = 10001100_2
Результат:

0	10001100	11011011011010000000000
s	exp	frac

Для типа float порядка exp изменится от 00000001 до 11111110 (значению 00000001 соответствует e = -126, значению 11111110 - порядок e = 127).
Код exp = 00000000, frac = 000…0 представляет нулевое значение, в зависимости от значения знакового разряда sэто либо +0, либо -0.
В случае, если exp = 00000000, frac ≠ 000…0, exp = 11111111 - это не число (NaN), то есть ошибка. При это если x = NaN, то условие if (x== !x) не выполнится.

Денормализованные числа

Это числа, представляемые кодами exp = 000…0, frac ≠ 000…0.
exp вносит в значение такого числа постоянный вклад 2^{-k-2}, frac меняется от 000…1 до 111…1 и рассматривается уже не как мантисса, а как значение, умножаемое на exp.

Рассмотрим как пример 8-разрядные числа:

s	exp	frac
1 бит	4 бита	3 бита

	s	exp	frac	E	значение	
Денормализованные числа	0	0000	000	-6	0	
	0	0000	001	-6	$^1/_8 \times {}^1/_{64} = {}^1/_{512}$	Близкие к 0
	0	0000	010	-6	$^2/_8 \times {}^1/_{64} = {}^2/_{512}$	
	0	0000	110	-6	$^6/_8 \times {}^1/_{64} = {}^6/_{512}$	Наибольшие денормализированные
	0	0000	111	-6	$^7/_8 \times {}^1/_{64} = {}^7/_{512}$	

	s	exp	frac	E	значение	
Нормализированные числа	0	0001	000	-6	$^8/_8 \times {}^1/_{64} = {}^8/_{512}$	Наименьшее нормализованное
	0	0001	001	-6	$^9/_8 \times {}^1/_{64} = {}^9/_{512}$	
	…					
	0	0110	110	-1	$^{14}/_8 \times {}^1/_{64} = {}^{14}/_{512}$	Ближайшее к 1 снизу
	0	0110	111	-1	$^{15}/_8 \times {}^1/_{64} = {}^{15}/_{512}$	
	0	0111	000	0	$^8/_8 \times 1 = 1$	Ближайшее к 1 сверху
	0	0111	001	0	$^9/_8 \times 1 = {}^9/_8$	
	…					
	0	1110	110	7	$^{14}/_8 \times 128 = 224$	Наибольшее нормализованное
	0	1110	111	7	$^{15}/_8 \times 128 = 240$	
	0	1111	000	$+\infty$		

Важные частные случаи:

	exp	frac	float	double
Ноль	00…00	00…00	0.0	
Наименьшее положительное денормадизованное	00…00	00…00	$2^{-23} \times 2^{-126}$	$2^{-52} \times 2^{-1022}$
Наибольшее положительное денормализованное	00…00	11…11	$(1-\varepsilon) \times 2^{-126}$	$1-\varepsilon) \times 2^{-1022}$
Единица	01…11	00…00	1.0	
Наибольшее				

положительное денормализованное	11…10	11…11	$(2-\varepsilon)\times 2^{127}$	$(2-\varepsilon)\times 2^{1023}$

Операции над числами с плавающей точкой

$x+_{FP}y = Round(x+y)$
$x \times_{FP} y = Round(x+y)$
Где Round означает округление.

Порядок выполнения операции:

1. Сначала вычисляется точный результат (получается более длинная мантисса, чем запоминаемая, иногда в 2 раза)
2. Фиксируется исключение (например, переполнение)
3. Результат округляется, чтобы поместиться в поле frac

Умножение чисел с плавающей точкой происходит следующим образом:

$$(-1)^{s^1}M_1 2^{e^1} \times (-1)^{s^2}M_2 2^{e^2}$$

Точный результат: $(-1)^s M 2^e$, где

- $s = s_1 \wedge s_2$
- $M = M_1 \times M_2$
- $e = e_1 + e_2$

При этом преобразование по следующим правилам:

- Если M ≥ 2, сдвиг M вправо с одновременным увеличением e
- Если e не помещается в поле exp, то фиксируется переполнение
- Округление M так, чтобы оно поместилось в поле frac

Основные затраты на перемножение мантисс.

Сложение чисел с плавающей точкой

Мы не можем складывать числа разного порядка, поэтому прежде всего их нужно привести к одному порядку.
$(-1)^{s^1}M_1 2^{e^1} + (-1)^{s^2}M_2 2^{e^2}$, где $e_1 > e_2$

Точный результат: $(-1)^s M 2^e$

- Порядок суммы - e_1
- К мантиссе M_1 прибавляется $e_1 - e_2$ старших разрядов к M_2

При этом происходит преобразование:

- Если M ≥ 2, сдвиг M вправо с одновременным увеличением e
- Если M < 1, сдвиг M влево на k позиций с одновременным вычитанием k из e
- Если e не помещается в поле exp, то фиксируется переполнение
- Округление M, чтобы оно поместилось в frac

При вычислении суммы чисел с одинаковыми знаками необходимо, упорядочить слагаемые по возрастанию и складывать, начиная с наименьших слагаемых.
При вычислении суммы чисел с разными знаками необходимо сначала сложить все положительные числа, потом - все отрицательные числа и в конце выполнить вычитание.
Вычитание (сложение чисел с разными знаками) часто приводит к потери точности, которая у чисел с плавающей точкой определятся количеством значащих цифр в мантиссе (при вычитании двух близких чисел мантисса «исчезает», что ведет к резкой потере точности). Итак, чем меньше вычитаний, тем точнее результат.

В функциях с переменным числом параметров float автоматически преобразуется в double (в части переменных параметров).

Режимы gcc для работы с плавающей точкой

На https://gcc.gnu.org/wiki/FloatingPointMath можно найти детальное резюме того, что бывает в gcc, и таблицу преобразований, влияющих на результат вычислений.

- Флаг -ffast-math: считать максимально быстро, но, возможно, нарушать стандарт IEEE-754
  Полезно для тестирования, но не распространения финальной версии программы.

- Флаг -fno-math-erno: не устанавливать переменную errno как результат ошибочного выполнения математических функций
  Можно обойтись и без этого, но зависит от библиотеки Си. Компилятор может заменять вызовы функций инструкциями процессора (например, sqrt).

- Флаг -fno-trapping-math: считать, что вычисления с плавающей точкой не могут вызывать исключений процессора (traps).
  То есть вы гарантируете отсутствие в своем коде ситуаций, вызывающих деление на ноль, переполнения, некорректные операции. В таком случае компилятор может более свободно комбинировать, переставлять, удалять операции с плавающей точкой.

Также статья David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23,1 (March 1991)

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldenberg.html

Лекция 16. Введение в алгоритмы. Сложность алгоритмов и алгоритм Кнута-Морриса-Пратта

Сложность алгоритмов

Сложность алгоритмов формализует из нескольких понятий.

Размер входа - числовая величина, характеризующая количество входных данных (например, длина битовой записи чисел-параметров алгоритма).

Сложность в наихудшем случае: функция размера входа, отражающая максимум затрат на выполнение алгоритма для данного размера:

- Временная сложность
- Пространственная сложность (затраты памяти)
- Часто оценивают не все затраты, а самые «дорогие» операции

Сложность в среднем: функция размера входа, отражающая средние затраты на выполнение алгоритма для входа данного размера (учет вероятностей входа).

Асимптотические оценки: O-нотация (оценка сверху), точная O-оценка, Θ-оценка.

O -оценка сверху, которую нельзя улучшить до степени константы.
o - оценка снизу
Θ - оценка сверху и снизу

Формальная постановка задачи поиска по образцу

Даны текст -массив T[N] длины N и образец - массив P[m] длины $m \le N$, где значениями элементов массивов T и P являются символы некоторого алфавита A. Говорят, что образец P входит в текст T со сдвигом s, если $0 \le s \le N-m$ для всех i = 1,2…m T[s+i] = P[i] (рис. 16.1).

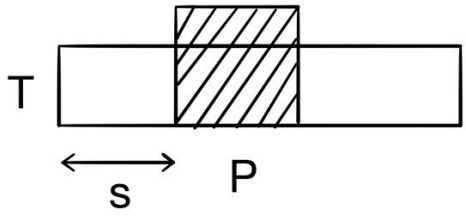


рис. 16.1 сдвиг s

Сдвиг s(T,P) называется *допустимым*, если P входит в T со сдвигом s(T,P), и *недоступным* в противном случае.
Задача поиска подстрок состоит в нахождении множества допустимых сдвигов s(T,P) для заданного текста T и образца P.

Пусть строки x,y,w ∈ A*, ε ∈ A* - пустая строка. Тогда
|x| - длина строки x;
xy - конкатенация строк x и y; |xy| =|x| + |y|;
если x = wy, то w - префикс (начало) x, обозначение w≺ x;
если x = yw, то w - суффикс (конец) x, обозначение w ≻ xx;
если w - префикс или суффикс x, то |w| ≤ |x|, при этом отношения префикса и суффикса транзитивны.

Для любых x,y ∈ A* и любого a ∈ A соотношения x ≻ y и xa ≻ ya равносильны.
Если S = S[r] - строка длины r, то ее префикс длины k, k ≤ r будет обозначен $S_k = S[k]$.
Ясно, что $S_0 = \varepsilon$, $S_r = S$.

Лемма о двух суффиксах (рис. 16.2):
Пусть x, y и z - строки, для которых x ≻ y и y ≻ z. Тогда:
Если |x| ≤ |y|, то x ≻ y,
Если |x| ≥ |y|, то y ≻ x,
Если |x| = |y|, то x = y

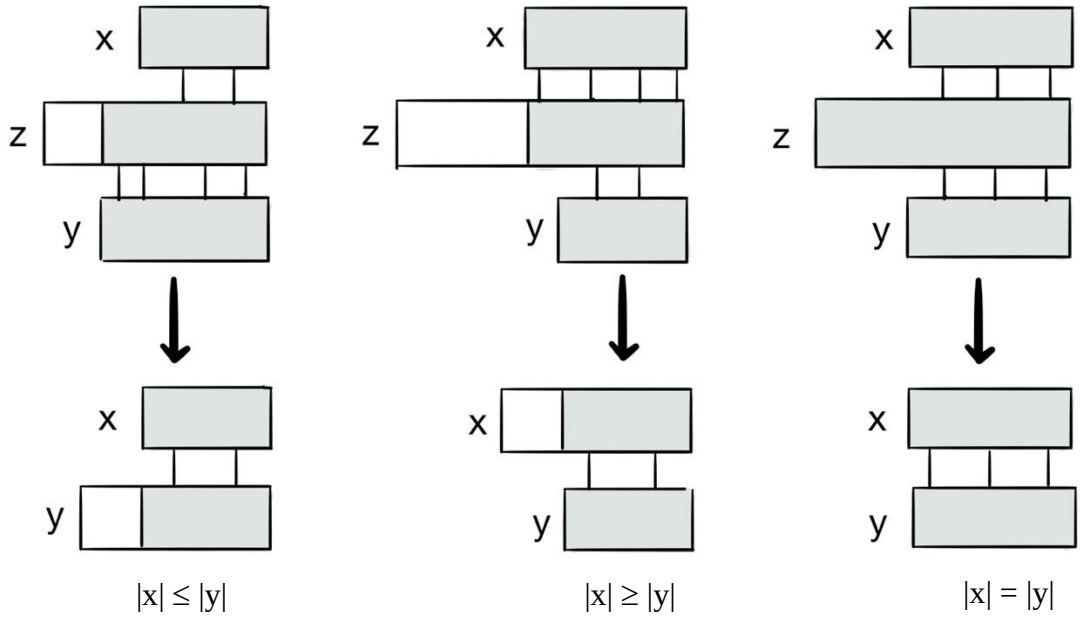


рис. 16.2 лемма о двух суффиксах

Очевидно, что то же самое можно сказать о префиксах.

Простой алгоритм

Проверка совмещения двух строк: посимвольное сравнение слева направо, которое превращается (с отрицательным результатом) при первом же расхождении.
Оценка скорости сравнения строк x и y - Θ(t + 1), где t - длина наибольшего общего префикса строк x и y.

```
for (s = 0; s <= n - m; s++){
        for (i = 0; i < m && P[i] == T[s+i]; i++);
        if (i == m)
          printf("%d\n", s);
}
```

Время работы в худшем случае Θ((nm+ 1)m) ~ Θ(nm), потому что каждый раз работа начинается сначала и информация о тексте T, полученная при проверке сдвига s, никак не используется при проверке следующих сдвигов. Например, если для образца dddc сдвиг s = 0 допустим, то сдвиги s =1, 2, 3 недопустимы, так как T[3] == c (рис. 16.3).

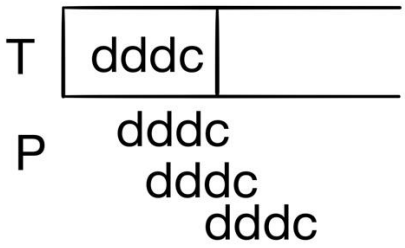


рис. 16.3 пример

Алгоритм Кнута-Морриса-Пратта

Идея алгоритма Кнута-Морриса-Пратта заключается в том, что префикс-функция, ассоциированная с образцом P, показывает, где в строке P повторно встречаются различные префиксы этой строки. Если это известно, можно не проверять заведомо недопустимые сдвиги.

Пример. Пусть ищутся вхождения образца P = ababaca в текст T. Пусть для некоторого сдвига s оказалось, что первые q символов образца совпадают с символами текста.

Значит, символы текста от T[s + 1] до T[s + q] известны, что позволяет заключить, что некоторые сдвиги заведомо недопустимы (рис. 16.4).

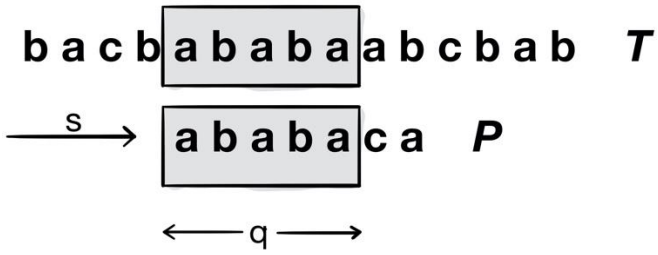


рис. 16.4 пример для алгоритма Кнутта - Морриса-Пратта

Пусть P[1..q] = T[s+1..q+1]; каково минимальное значение сдвига s’ > s, для которого P[1..k] = T[s’+1..s’+k], где s’ + k = s + q?

- Число s’ - минимальное значение сдвига, большего s, которое совместимо с тем, что T[s+1..s+q] = P[1..q]. Следовательно, значения сдвигов, меньшие s’, проверять не нужно
- Лучше всего, когда s’= s + q? так как в этом случае не нужно рассматривать сдвиги s + q - 1, s + q - 2, …, s + 1
- Кроме того, при проверке нового сдвига s’ можно не рассматривать первые его k символов образца: они заведомо не совпадут.

Чтобы найти s’. Достаточно знать образец P и число q:
R[s’+1…s’+k] - суффикс P_q, поэтому k - это наибольшее число, для которого P_k является суффиксом P_q.
Зная k (число символов, заведомо совпадающих при проверке нового сдвига s’), можно вычислить по формуле s’ = s + (q-k).

Таким образом, мы прикладываем образец не к тексту, а сдвигаем образец относительно куска самого себя, то есть прикладываем образец к самому себе. Поэтому понять, при каких сдвигах будет совпадение, можно не глядя на текст, а глядя только на то, как устроен образец.

Рассмотрим префикс-функцию на Си:

```
void prefix_func(char *pat, int *pi, int m) {
        int k, q;
        /* считаем, что pat и pi нумеруются от 1 */
        pi[1] = 0; k = 0;
        for (q = 2; q <= m; q++) {
```

```
            while (k > 0 && pat[k + 1] != pat[q])
              k = pi[k];
            if (pat[k + 1] == pat[q])
              k++;
            pi[q] = k;
        }
}
```

Для того, чтобы понять, как и почему работает эта программа, нужно понять несколько фактов.

Лемма1. Обозначим $\pi^*[q] = \{q, \pi[q], \pi^2[q], \dots, \pi^t[q]\}$, где $\pi^t[q]$ есть i-я итерация префикс-функции, $\pi^t[q] = 0$.Префикс-функция отображает числа от 1 до m в числа от 0 до m - 1. Пусть P - строка длины m с префикс-функцией π . Тогда для всех q = 1,2, …, m имеем $\pi^*[q] = \{ k: P_i \succ P_q\}$.
Лемма показывает, что при помощи итерирования префикс-функции можно для данного q найти все такие k, что P_k является суффиксом P_q.

Доказательство. 1) Покажем, что если $i \in \pi^*[q]$, то P_i является суффиксом P_q. Действительно, $P_{\pi[i]} \succ P_i$ по определению префикс-функции, так что каждый следующий член последовательности $P_i, P_{\pi[i]}, P_{\pi[\pi[i]]}$ ... является суффиксом всех предыдущих. 2) Покажем, что наоборот, если P_i является суффиксом, то $i \in \pi^*[q]$. Расположим все P_i, являющиеся суффиксами P_q, в порядке уменьшения i (длины): P_{i_1}, P_{i_2}, ... Покажем по индукции, что $P_{i_k} = \pi^k[q]$:

- *База индукции*: k=1
  Для максимального префикса P_i, являющегося суффиксом P_q, по определению $i = \pi[q]$.
- *Шаг индукции:* если $P_{i_k} = \pi^k[q]$, то по определению j = $\pi[\pi^k[q]]$ соответствует максимальный префикс P_j, который является суффиксом P_{i_k}. Обе строки P_j и P_{i_k} есть суффиксы P_q по построению. Таким максимальным префиксом из оставшихся $P_{i_{k+1}}, P_{i_{k+2}}$, ... по построению является префикс $P_{i_{k+1}}$, то есть j = i_{k+1}.

Префикс-функция от ababa очевидно равна 3. То есть q = 5, а из возможных k =1 и k =3 выбираем максимум: k =3. То есть мы к 5 применили один раз префикс-функцию и получили 3. К 3 применили префикс-функцию и получили 1. Таким образом находим все префиксы ababa, которые также являются суффиксами ababa.

Рассмотрим пример (рис. 16.5). Первые 8 символов повторяются и 2 символа в конце. $\pi^*[8]$ означает, что надо найти подстроку из 8 символов из большого образца, к ней применить префикс-функцию, (то есть это одна итерация сдвига на 2 символа вправо),

получится значение 6. $\pi[6] = 4$ и т.д. В итоге мы получили все возможные префиксы от $\pi[8]$ такие, что они являются суффиксом $\pi[8]$.

Лемма 2. Пусть P - строка длины m с префикс-функцией π. Тогда для всех q = 1, 2, …, m, для которых $\pi[q] > 0$, имеем $\pi[q] - 1 \in \pi^*[q-1]$.

Доказательство. Если k = $\pi[q] > 0$, то P_k является суффиксом P_q по определению префикс-функции. Следовательно, P_{k-1} является суффиксом P_{q-1}.
Тогда по Лемме 1 $k - 1 \in \pi^*[q-1]$, то есть $\pi[q] - 1 \in \pi^*[q-1]$.
Определим множества $E_{q-1} = \{k : k \in \pi^*[q-1] \wedge P[k+1] = P[q]\}$.
Множество E_{q-1} состоит из таких k, что P_k является суффиксом P_{k-1}, и за ними идут одинаковые буквы P_{k+1} есть суффикс P_q.

Следствие 1. Пусть P - строка длины m с префикс-функцией π. Тогда для всех q = 2, 3, …, m

$$\pi[q] = \begin{cases} 0, & \text{если } E_{q-1} \text{ пусто} \\ 1 + \max\{k \in E_{q-1}, & \text{если } E_{q-1} \text{ не пусто} \end{cases}$$

Доказательство. Если r = $\pi[q] \geq 1$, то P[r] = P[q] и по Лемме 2 r -1 = $\pi[q] - 1 \in \pi^*[q-1]$. Раз P[r] = P[q], то P[(r-1)+1] = P[q]. Поэтому r-1$\in E_{q-1}$ из определения E_{q-1}.
Следовательно, если E_{q-1} пусто, то $\pi[q] = 0$ (от противного).
Если же k $\in E_{q-1}$, то P_{k+1} есть суффикс P_q (по определению), тем самым $\pi[q] \geq k + 1$ и $\pi[q] \geq 1 + \max\{k \in E_{q-1}\}$. То есть, если E_{q-1} не пусто, то префикс-функция положительна.
Но тогда $\pi[q] - 1 \in E_{q-1}$, и $\pi[q] - 1$ не больше максимума из E_{q-1}, то есть $\pi[q] \leq 1 + \max\{k \in E_{q-1}\}$.

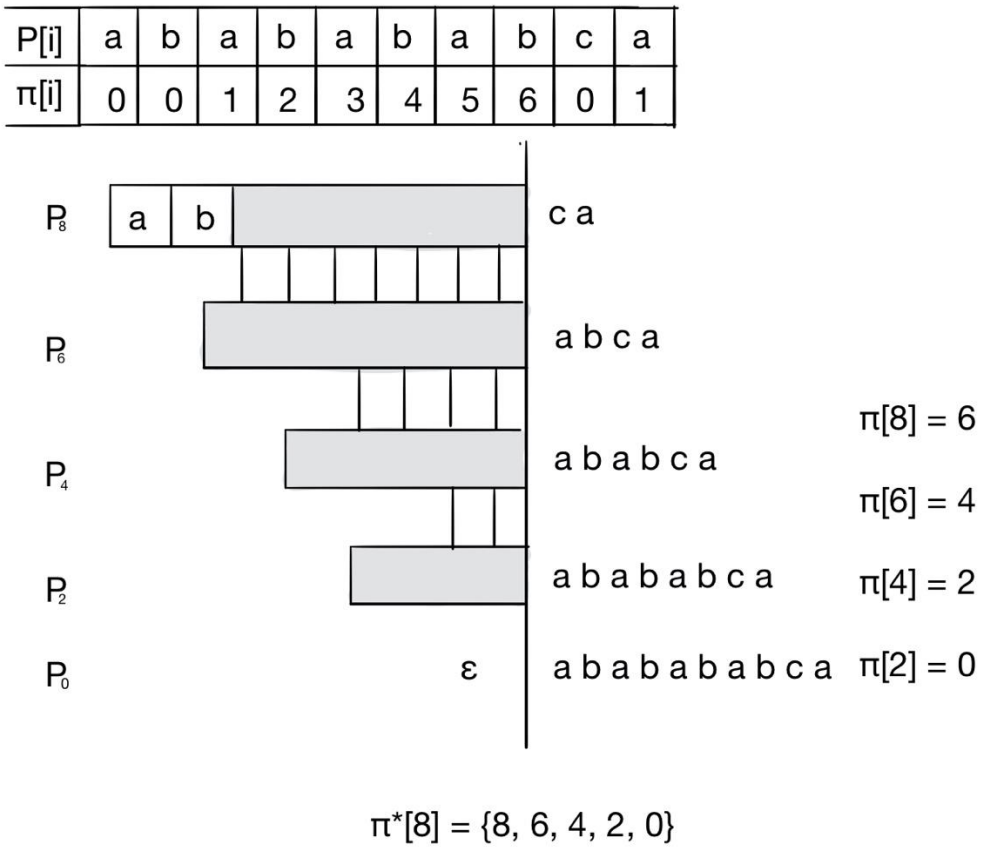


рис. 16.5 пример из доказательства

Вернемся к программе:

```
void prefix_func(char *pat, int *pi, int m) {
        int k, q;
        /* считаем, что pat и pi нумеруются от 1 */
        pi[1] = 0; k = 0;
        for (q = 2; q <= m; q++) {
                while (k > 0 && pat[k + 1] != pat[q])
                  k = pi[k];
                if (pat[k + 1] == pat[q])
                  k++;
                pi[q] = k;
        }
}
```

Цикл от 2 до m считает значение префикс-функции. Внутренний цикл по мере уменьшения k перебирает все потенциальные элементы множества E_{q-1}.

Элемент из множества E_{q-1} должен обладать 2 свойствами: 1) принадлежать π^*, 2) следующий символ должен совпадать. k = pi[k]; - последовательное итерирование по множеству π^*, и так как мы всегда начинаем с большего k, то мы еще и в правильном порядке перебираем от большего к меньшему, потому что нам нужен максимум.
При выходе из цикла используется конъюнкция, таким образом из цикла мы могли выйти в случае, когда 1) нарушилась левая часть, то есть k = 0, 2) нарушилась правая часть, то есть совпали значения. Если совпали значения, то был какой-то k> 0, максимальный k из множества E_{q-1}. Его нужно увеличить на 1, и это будет значение префикс-функции. А если элементы не совпали, то мы перебрали всевозможные k, при которых значения не совпали, значит, k = 0? Следовательно множество пусто и значение префикс-функции также равно 0.

Теорема 1. Функция prefix_func правильно вычисляет префикс-функцию π.
Доказательство. покажем, что при входе в цикл функции k = $\pi[q-1]$.
База индукции: При q = 2 k=0, pi[q-1] = pi[1]=0.
Шаг индукции: пусть при входе в цикл функции k = $\pi[q-1]$.
Код на строках 7-8: while (k > 0 && pat[k + 1] != pat[q])
k = pi[k];
находит наибольший элемент E_{q-1}, так как цикл перебирает в порядке убывания элементы из $\pi^*[q-1]$ и для каждого проверяет условие pat[k + 1] != pat[q].

Рассмотрим теперь саму kmp функцию:

```
void kmp(char *text, char 8pat, int m, int n) {
        int q;
        int pi[m+1]; /* VLA-массив*/
        /* считаем, что pat и pi нумеруются от 1 */
        prefic_func(pat, pi, m);
        q = 0;
        for (i = 1, 1 <= n; i++){
                while (q > 0 && pat[q+1] != text[i])
                  q = pi[q];
                if (pat[q+1] == text[i])
                  q++;
                if (q==m){
                        printf("образец входит со сдвигом %d\n", i-m);
                        q = pi[q];
                }
        }
```

Алгоритм КМП для подстроки P и текста эквивалентен вычислению префикс-функции для строки Q = P#T, где # - символ, заведомо не встречающийся в обеих строках.
Длина максимального префикса Q, являющегося ее суффиксом (т.е. значение префикс-функции), не превосходит длины P.
Допустимый сдвиг обнаруживается в тот момент, когда очередное вычисленное значение префикс-функции совпадает с длиной подстроки P (условие if (q==m)). В явном виде объединенная строка не строится!

Теорема 2. Функция kmp работает правильно.
Формальное доказательство осуществляется по аналогии с доказательством Теоремы 1, где множества, подобные E_{q-1}, строятся для строки-текста, а не строки-образца.

Свойства префикс-функции часто используются и в других задачах (кроме поиска подстроки в строке). Полезной оказывается Лемма 1: итерированием префикс-функции можно найти все префиксы строки, являющиеся ее суффиксами.

Функция выполняет ≤ (m-1) итерация цикла for. Стоимость каждой итерации можно считать равной O(1), а стоимость всей процедуры O(m).
Каждая итерация цикла while (строка 7-8) уменьшает k. Увеличивается k только в строке 10 не более одного раза на итерацию цикла for (6-11).
Следовательно, операций уменьшения не больше, чем итераций цикла for, то есть ≤ (m-1) на весь цикл и O(1) на итерацию в среднем.

Аналогично функция kmp выполняет ≤ (n-1) итераций, и ее стоимость (без учета вызова prefix_func) есть O(n). Следовательно, время выполнения всех процедуры - O(m+n).

Лекция 17. Стек. Очередь. Список.

Стек

Стек (stack) — это динамическая последовательность элементов, количество которых изменяется, причем как добавление, так и удаление элементов возможно только с одной стороны последовательности (вершины стека).

Работа со стеком осуществляется с помощью функций:
push(x) - затолкать элементы x в стек;
x = pop() - вытолкнуть элемент из стека.

Стек можно организовать на базе (примеры):

- Фиксированного массива stack[MAX], где константа MAX задает максимальную глубину стека;
- Динамического массива, текущий размер которого хранится отдельно.

В обоих случаях необходимо хранить позицию текущей вершины стека. Можно использовать и другие структуры данных (список).
Работа со стеком ведется по принципу LIFO (Last In First Out). При запуске программы вызываемые функции превращаются в LIFO последовательность. Поэтому логично иметь для этого память, организованную стеком. Как правило, во всех машинах есть аппаратный стек - регистр, выделенный под stack pointer, инструкция работы с ними, например, push и pop, инструкции вызова.
Рассмотрим стек из символов на базе динамического массива.

```
struct stack {
  int sp;    /* Текущая вершина стека */
  int sz;    /* Размер массива */
  char *stack;
} stack = { .sp = -1, .sz = 0, .stack = NULL};

static void push(char c) {
  if (stack.sz == stack.sp + 1) {
    stack.sz = 2*stack.sz + 1;
    stack.stack = (char *) realloc(stack.stack,
                    stack.sz*sizeof(char));
  }
  stack.stack[++stack.sp] = c;
}
```

Можно было воспользоваться частичной инициализацией и вместо stack = { .sp = -1, .sz = 0, .stack = NULL}; записать только stack = { .sp = -1};
Чтобы что-то положить в стек, нужно по индекс первого занятого элемента увеличить и положить по нему то, что передали. Поэтому при инициализации sp = -1: сначала его нужно увеличить до 0, а потом что-то положить. Если места больше нет, то нужно сделать realloc() массива - он гарантирует, что те элементы, которые уже были положены, останутся в памяти.
В реальном коде нельзя писать p = realloc(p). Нужно писать обертку: q = realloc(p), проверить не равен ли p нулю, если равен, то сделать free(p).

Важно запомнить! Если место в динамическом массиве закончилась, нельзя увеличивать его константными кусками памяти. Так ка функция выделения памяти дорогая, затраты на ее выполнение будут занимать существенную долю всех остальных затрат. Память всегда нужно увеличивать кратно, умножив размер, запрошенный ранее на некоторую константу, например, на 2. Важно добавить 1, чтобы в случае стека = 0 realloc() не выдал нам снова 0.

Функция pop() тоже простая:

```
struct stack {
  int sp;    /* Текущая вершина стека */
  int sz;    /* Размер массива */
  char *stack;
} stack = { .sp = -1, .sz = 0, .stack = NULL};

static char pop(void) {
  if (stack.sp < 0) {
    fprintf(stderr, "Cannot pop: stack is empty\n");
    return 0;
  }
  return stack.stack[stack.sp--];
}
```

В этой функции делается проверка, на пустоту стека.
Чтобы проверить пуст стек или нет, заведем еще одну функцию:

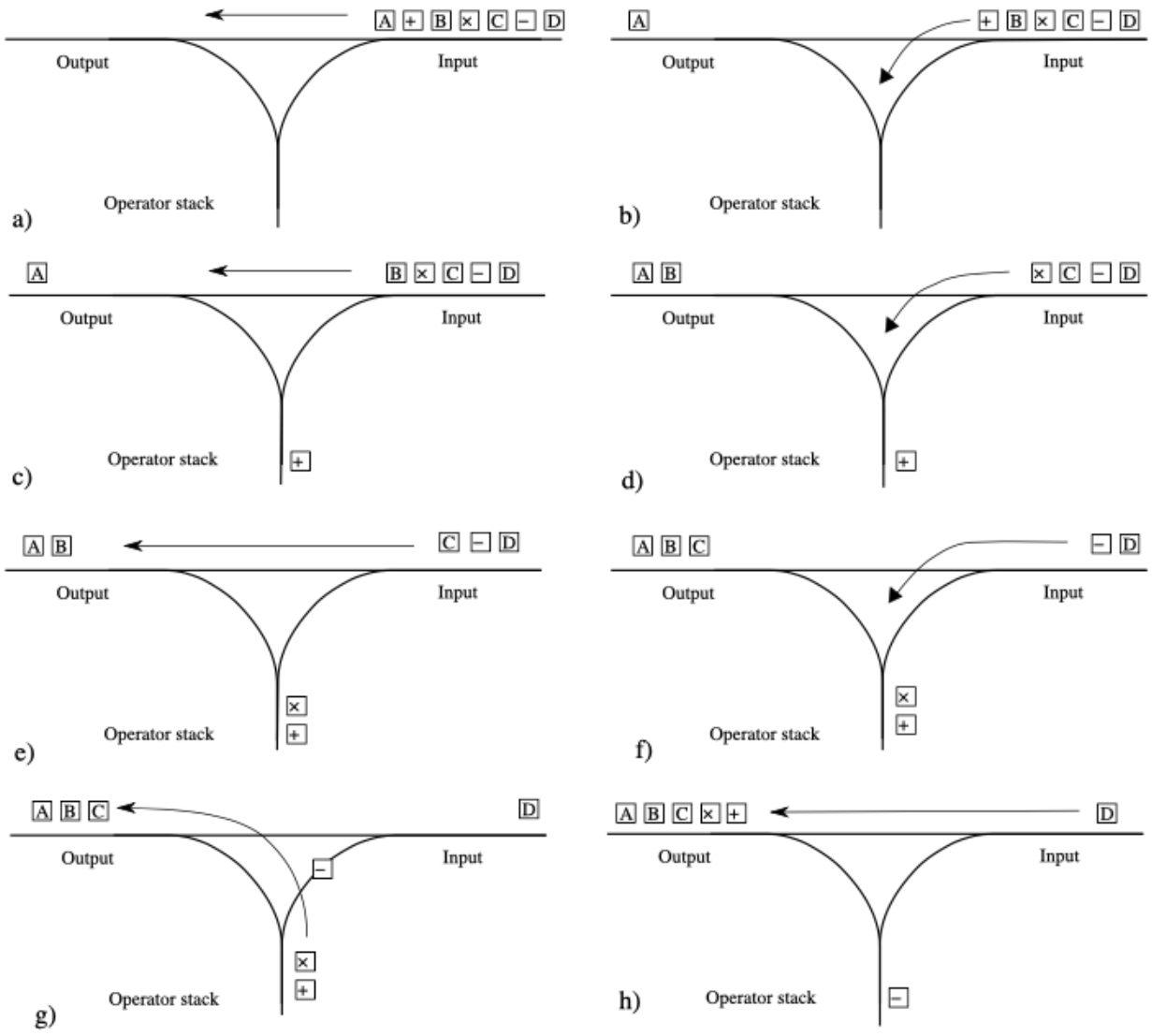
```
static int isempty(void) {
  return stack.sp == -1;
}
```

Она сравнивает указатель на первый элемент стека с -1.

Алгоритм перевода в обратную польскую запись

Стек часто объясняют на алгоритме перевода арифметического выражения в обратную польскую запись (постфиксную) (рис. 17.1), которая служит основой калькуляторов.

$a + b \times c - d \quad \rightarrow \quad abc \times + d -$
$c \times (a + b) - (d + e) \,/\, f \quad \rightarrow \quad cab + \times de + f / -$



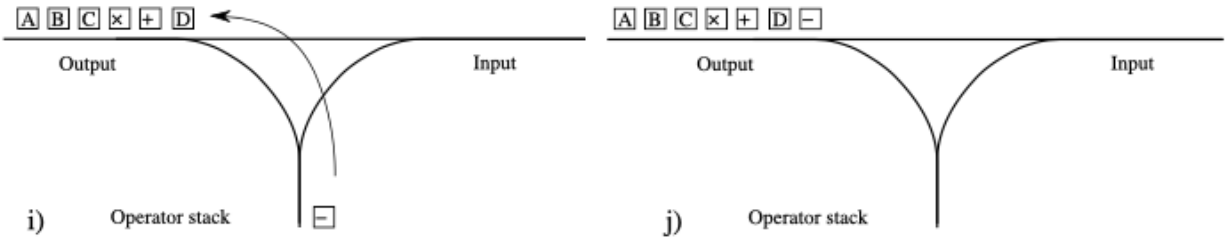


рис. 17.1 стек и польская запись

Суть очень проста: есть выражение, которое нужно посчитать, и в нем есть константы операции с различными приоритетами и скобки, которые заменяют приоритет. Чтобы посчитать выражение с операциями различных приоритетов, проще всего привести его к виду записи, в которой не будет скобок и операций. Такая запись и есть обратная польская запись.
В стековой машине есть неявный стек операндов. Например, при сложении выталкиваются 2 операнда, которые надо сложить и положить результат в стек. Так c = a + b происходит через:
push a
push b
add
pop c

Так же происходит вычисление в польской обратной записи: идем по выражению (уже без скобок) слева направо, встречающиеся операнды кладем в стек, достаем нужное количество операндов из стека, производим операцию и результат кладем снова в стек. В конце процесса на верхушке стека лежит результат.
Но пользователь не будет вводить выражение в таком виде, поэтому нужен алгоритм перевода арифметического выражения в польскую обратную запись. Этот алгоритм обычно сравнивают с сортировкой вагонов. Есть тупик, куда загоняют вагоны и оттуда выгоняют обратно.
Принцип очень простой: вход может быть 3 типов: переменная (константа), знак операции и скобочки. Если справа налево в тупик приезжает буква, то она сразу уезжает на выход. Если приезжает операция, нужно операции, которые приехали раньше и находятся на верхушке стека, выполнить раньше. Например, если приехал минус, а до этого было умножение, сначала должно выполниться умножение. Если операции равного приоритета, то операцию, пришедшую раньше, нужно выполнить раньше, потому что порядок обработки слева направо. Если сначала приехало умножение, а потом плюс, то сначала нужно выполнить умножение. Поэтому правило такое: приезжает знак операции, мы смотрим на верхушку стека, в зависимости от приоритета выталкиваем операцию, как только операции большего или равного приоритета с приехавшим знаком закончили, кладем эту операцию в стек.

В случае скобок сначала нужно вытолкнуть все правильные с точки зрения скобок скобки, то есть ().

Промоделируем $c \times (a + b) - (d + e) \,/\, f \quad \rightarrow \quad cab + \times de + f/-$ таким образом: *c* сразу пропускаем, умножение кладем в стек, скобочку тоже кладем в стек, *a* пропускаем, плюс кладем в стек, предварительно посмотрев, что находится на верхушке стека - скобка с меньшим приоритетом), b тоже пропускаем, скобку кладем в стек, видим, что образуется правильная скобка и выталкиваем все, что находится между скобок (рис. 17.2).

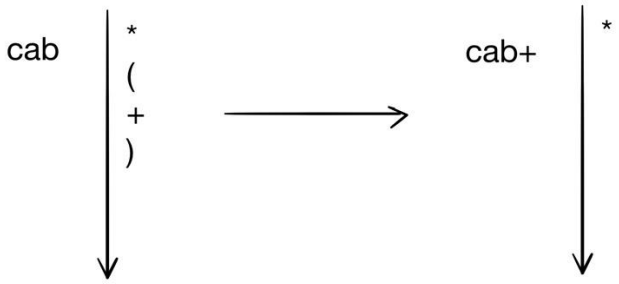


рис. 17.2 стек операций

Следующий минус должен вытолкнуть операции на верхушки стека с большим приоритетом, у нас там лежит умножение. Выталкиваем умножение, кладем минус в стек. Далее идет скобочка, которую мы тоже кладем в стек, d сразу пропускаем, плюс кладем в стек, так как скобочка меньшего приоритета, e пропускаем. Следующая скобочка выталкивает все до открывающей скобочки включительно, в стеке остается только минус. Следующая операция - деление, она выше приоритетом, чем минус, кладем ее в стек. Далее пропускаем f, выталкиваем деление и затем минус.
В коде на Си это выглядит следующим образом:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#include "stack.c"  (так писать нельзя, далее разберем альтернативу написания кода стека)
/*Считывание символа-операции или переменной*/
static char getop(void) {
  in c;
  while ((c=getchar()) != EOF && isblank(c))
    ;
  return c == EOF || c == '\n' ? 0 : c;
}
```

```
/* является ли символ операцией*/

static in isop(char c){
  return (c == '+') || (c == '-') || (c == '*')
        || (c == '/');
}
/* каков приоритет символа-операции */

static int prio(char c){
  if (c =='(')
    return 0;
  if (c == '+' || c == '-')
    return 1;
  if (c == '*' || c == '/')
    return 2;
  return -1;
}
```

Функция считывает символы из стандартного потока ввода, пропуская пробелы. В качестве знака окончания ввода возвращаем 0, иначе возвращаем символ. Проверяем, является ли считанный символ операцией и приоритет символа-операций отдельными функциями.
Рассмотрим теперь основную функцию:

```
int main(void) {
  char c, op;
  while (c = getop()){
    /*переменная-буква выводится сразу*/
    if (isalpha(c))
      putchar(c);
    /*скобка заносится в стек операций */
    else if (c =='(')
      push(c);
   /*операция заносится в стек в зависимости от приоритета*/
    /*скобка заносится в стек операций */
    else if (c =='(')
      push(c);
    else if (isop(c)){
      while (!isempty()){
        op = pop();
        /*заносим в стек, если больший приоритет*/
```

```
      if (prio(c) > prio(op)){
        push(op);
        break;
      } else
      /*иначе выталкиваем операцию из стека*/
      putchar(op);
    }
    push(c);
    /*скобка выталкивает операции до парной скобки*/
  }else if (c == ')')
    while ((op=pop()) != '(')
      putchar(op);
 }
 /*вывод остатка операций из стека*/
 while (!isempty())
   putchar(pop());
 putchar('\n');
 return 0;
}
```

Чтобы посмотреть на верхушку стека, мы должны вытолкнуть верхний в стеке операнд и потом сравнить его с текущим. В случае, если приоритет текущей операции выше приоритета только что вытолкнутой операции, надо сначала вернуть в стек вытолкнутую операцию, а затем положить в стек текущую. Цикл закончится, когда стек останется пустым. При этом знак текущей операции необходимо занести в стек, потому что еще непонятно, когда он должен быть выдан.

Теперь учтем оговорку в заголовке файла: **#include "stack.c".**
При таком include библиотека , которая должна давать описания для внешних интерфейсов и внутри содержать их реализацию, не скрывает реализацию, что плохо, потому что можно нечаянно ее испортить, также ограничены имена переменных.

stack.h:

```
extern void push(char);
extern char pop(void);
extern int isempty(void);
```

stack.c:

```
#include "stack.h"
struct stack{
  <...>
```

```
};
static struct stack stack = {<...>};

main.c:
#include "stack.h"
int main(void){
  <...push(c), pop(), ...>
}

$gcc main.c stack.c -o main
```

Для того, чтобы удобно и надежно было пользоваться, нужно использовать ключевое слово static, которое убирает функцию из процесса связывания, не делая ее внешней. Также хорошо бы отдавать пользователю (программисту) только то, что он должен видеть и не больше. Поэтому обычно есть заголовочный файл, в котором описаны прототипы тех функций, которые позволяют работать со стеком.
Файл stack.c содержит реализацию этих функций. Далее пользовательский код main.c включает stack.h, получая доступ к функциям над стеком. Пользовательский код теперь никак не сломает стек, потому что он вообще не видит этих переменных.
Обычно исходника библиотеки у вас не будет, она уже за вас где-то скомпилирована. Нужно будет только включить этот файл и передать gcc ключик, что нужно скомпоновать с этой библиотекой.

В самом простом варианте компилируем все эти файлы вместе и все.
Мы должны ввести в функции spush, pp, isempty параметры, с которыми они работают. Дальше возникает проблема: мы не хотим показывать пользователю как устроен стек, но если он становится параметром, то мы не можем этого не показать. Для этой проблемы есть решение с помощью pack pointers. Будем использовать forward declaration, которое говорит примерно следующее: «где-то у меня дальше есть стек, а как устроен он я тебе сейчас не скажу». Такой подход делает структуру не полностью определенным типом, но нее можно сделать указатель. Примерно так и устроено большинство библиотек.

```
stack.h:
struct stack; //forward declaration
extern void push(struct stack*, char);
extern char pop(struct stack *);
extern int isempty(struct stack*);
extern struct stack* new_stack(void);
```

```
extern void free_stack(struct stack*);
```

stack.c:

```
#include "stack.h"
struct stack{
  <...>
};
void push(struct stack *stack, char c){
  if (stack -> sz == stack -> sp+1) <...>
}
<...>
```

Важно определить функции, которые создают и очищают стек, чтобы иметь собственно возможность работать со стеком.
stack.c:

```
struct stack* new_stack(void){
  struct stack *s = malloc(sizeof(struct stack));
  *s = (struct stack){ .sp=-1, .sz = 0,  .stack = NULL};
  return s;
}
void free_stack(struct stack 8s){
  free(s->stack);
  free(s);
}
```

main.c:

```
#include "stack.h"
int main(void) {
  struct stack *s = new_stack();
  <...push(s,c), pop(s),...>
  free_stack(s);
  <....>
}
```

Очередь

Очередь(queue) - это линейный список информации, работа с которой происходит по принципу FIFO (First In First Out).
Для списка можно использовать статический массив: количество элементов массива (MAX) - наибольший допустимой длине очереди.
Работа с очередью осуществляется с помощью двух функций:

- qstore() -поместить элемент в конец очереди
- qretrieve()- удалить элемент из начала очереди

и двух глобальных переменных:

- spos - индекс первого свободного элемента очереди, его значение < MAX
- rpos - индекс очередного элемента, подлежащего удалению: «кто первый?»

```
int queue[MAX]; //заведите enum
int spos = 0; rspos = 0;
int qstore(int q){
  if (spos == MAX){
    /* можно расширить очередь, см. реализацию стека */
    printf("Очередь переполнена\n");
    return 0;
  }
  queue[spos++] = q;
  return 1;
}
int qretrieve(void){
  if (rpos == spos) { // очередь пуста
    return -1;
  }
  return queue[rspos++];
}
```

Функции, описанные выше, чреваты неприятностями. Например, если постоянно добавлять и удалять элементы из очереди, сама очередь останется того же размера, но она будет «ползти» по массиву вправо до тех пор, пока не дойдет до конца и в нее уже нельзя будет ничего добавить (рис. 17.3).

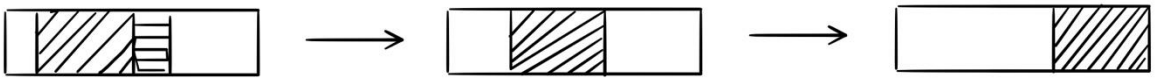


рис. 17.3 «ползущая» очередь

Как это можно исправить? Никто не говорит, что хвост должен быть правее начала. Можно закольцевать очередь (рис. 17.4).

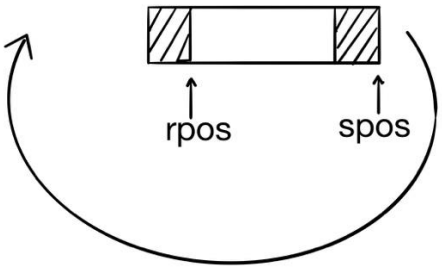


рис. 17.4 «закольцованная» очередь

```
int queue[MAX]; //заведите enum
int spos = 0; rspos = 0;
int qstore(int q){
  if (spos + 1 == rpos || (spos +1 == MAX && !rpos))  {printf("Очередь переполнена\n")
    return 0;
  }
  queue[spos++] = q;
  if (spos == MAX)
    spos = 0;
  return 1;
}
```

Если далее мы захотим расширить динамический массив, необходимо будет хвост, расположенный слева, перенести вправо от начала массива в сторону продолжения массива.
В случае, когда что-то достается из очереди:

```
int queue[MAX]; //заведите enum
int spos = 0; rspos = 0;
int qretrieve(void){
  if (rpos == spos) { // очередь пуста
    return -1;
  }
  if (rpos == MAX - 1){
    rpos = 0;
    return queue[MAX - 1];
  }
  return queue[rspos++];
}
```

Зацикленная очередь переполняется, когда spos находится непосредственно перед rpos, так как в этом случае запись приведет к rpos == spos, то есть к пустой очереди.

Списки

Связный список является простейшим типом данных динамической структуры, состоящей из элементов (***узлов***). Каждый узел включает в себя в классическом варианте два поля:

- данные (в качестве данных может выступать переменная, объект класса или структуры и т. д.)
- указатель на следующий узел в списке.

Элементы связанного списка можно помещать и исключать произвольным образом.

Доступ к списку осуществляется через указатель (рис. 17.5), который содержит адрес первого элемента списка, называемый *головным элементом или корнем списка.*

Связный список, содержащий только один указатель на следующий элемент, называется *односвязным*.

Связный список, содержащий два поля указателя – на следующий элемент и на предыдущий, называется *двусвязным*.

рис. 17.5 односвязный список

Рассмотрим добавление элемента в начало списка:

```
struct list *phead = NULL;
struct list*add_element(struct list *phead,
  struct data *elem) {
    struct list *new = malloc(sizeof(struct list));
    new -> info = *elem;
    new -> next = phead;
    return new;
  }
```

Такая операция имеет константную сложность. Выделяем память под новый элемент списка в начало, передаем значение этого элемента, связываем этот элемент со следующим элементом, обновляем указатель на голову.

Лекция 18. Графы. Топологическая сортировка.

Вернемся к прошлой лекции и спискам.

Односвязный список — это динамическая структура данных, каждый элемент которой содержит ссылку на следующий элемент (либо NULL, если следующего элемента нет). Доступ к списку осуществляется с помощью указателя на его первый элемент.

```
structlist {
        struct data info; /* Данные */
        struct list *next; /* Ссылка на след. элемент */
};
```

Выделение элемента:

```
struct list *phead = NULL;
phead = (struct list *) malloc (sizeof(struct list));
```

Если phead = 0, то список пустой. Поэтому при работе со списком нужно проверять головной указатель. Иногда хочется избежать этой проверки. Для этого есть список с заголовочным звеном, то есть его головной указатель никогда не равен NULL. Он указывает на какой-то буферный элемент, даже если список пустой. Остается только понять, является ли этот буферный элемент настоящим элементом списка.
Функция добавления нового элемента в начало списка, рассмотренная в прошлой лекции:

```
struct list *phead = NULL;
struct list*add_element(struct list *phead,
  struct data *elem) {
    struct list *new = malloc(sizeof(struct list));
    new -> info = *elem;
    new -> next = phead;
    return new;
  }
```

Одинаково хорошо работает и при phead == NULL, и при phead != NULL.

При добавлении элемента в конец списка (рис. 18.1) :

```
struct list *phead = NULL;
struct list *add_element(struct list *phead,
```

```
struct data *elem) {
    if (! phead) {
            phead = malloc(sizeof(struct list));
            phead -> info = *elem;
            phead -> next = NULL;
            return phead;
    }
  struct list *ph = phead; // сохраним голову
  while (phead -> next != NULL)
   phead = phead -> next;
  phead -> next = malloc(sizeof(struct list));
  phead -> next -> info = *elem;
  phead -> next -> next = NULL;
  return ph; //phead затерт, вернем сохраненный указатель
}
```

Сначала нужно дойти до этого элемента по ссылкам и к нему привязать созданный элемент. При проходе по элементам списка нужно учесть, что список может быть пустым. Если список пустой, нам нужно просто создать список из одного элемента. Если же список непустой, то мы проходим до конца через цикл. При этом используем переменную ph, указывающую на начало списка. Это необходимо только для случая пустого списка, но входит в общую функцию для универсальности.
Когда phead указывает на последний элемент, мы добавляем новый элемент, выделяя под него память malloc, и делаем от него указатель на следующий элемент NULL:

```
phead -> next = malloc(sizeof(struct list));
    phead -> next -> info = *elem;
    phead -> next -> next = NULL;
```

Сложность этой функции перестала быть константной за счет того, что нам нужно проходить по всем элементам списка.

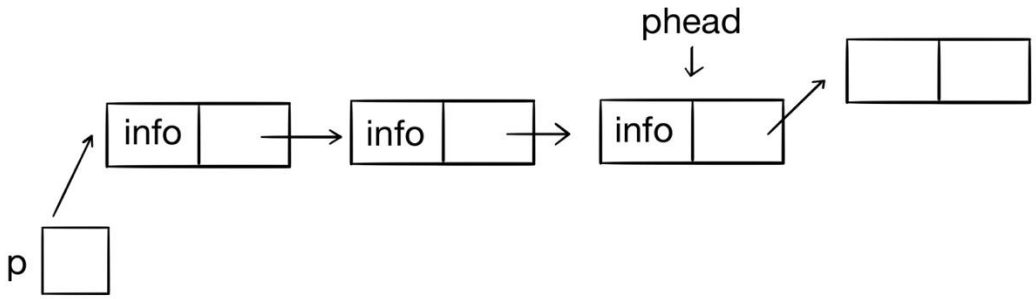


рис. 18.1 добавление элемента в конец списка

Рассмотрим теперь поиск элемента в списке:

```
struct list *phead;

    int equals(struct data*, struct data*);
    struct list *search(struct list *phead,
          struct data *elem) {
              while (phead && !equals($phead -> info, elem))
                phead = phead -> next;
            return phead;
          }
```

Мы двигаемся по указателям списка до тех пор, пока 1) список не закончится, 2) критерии поиска не будут выполнены.

Удаление элемента из списка сложнее, чем добавление:

```
struct list *remove(struct l;it *phead,
 struct data *elem){
              struct list *prev = NULL, *ph = phead;
              while (phead && !equals(&phead -> info, elem)){
                  prev = phead;
                  phead = phead -> next;
              }
              if (!phead)
                return ph;
              if (prev)
                prev -> next = phead -> next;
              else
                ph = phead -> next;
              free(phead);
              return ph;
 }
```

Чтобы удалить элемент, его сначала нужно найти, то есть он должен удовлетворять некоторым критериям. Поэтому первая часть функции удаления очень похожа на функцию поиска. Функция перебирает все элементы списка и ждет завершение поиска. Если она ничего не найдет, то ничего удалять не нужно. Если мы нашли необходимый элемент, то происходит удаление. В односвязных списках есть нюанс: необходим указатель prev. Если бы его не было, и функция работала, как функция поиска,

рассмотренная ранее, и указатель phead, допустим, указывал бы на нужный элемент, то при удалении этого элемента free() связность списка безвозвратно нарушалась бы (рис. 18.2).

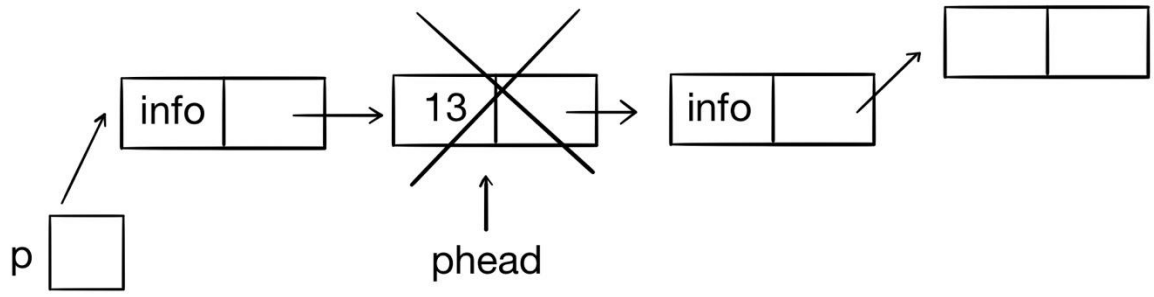


рис. 18.2 неправильное удаление элемента из списка

Поэтому необходим указатель prev, который бы указывал на предыдущий элемент списка и позволял бы соединить предыдущий и следующий за удаленным элементом элемент списка (рис. 18.3).

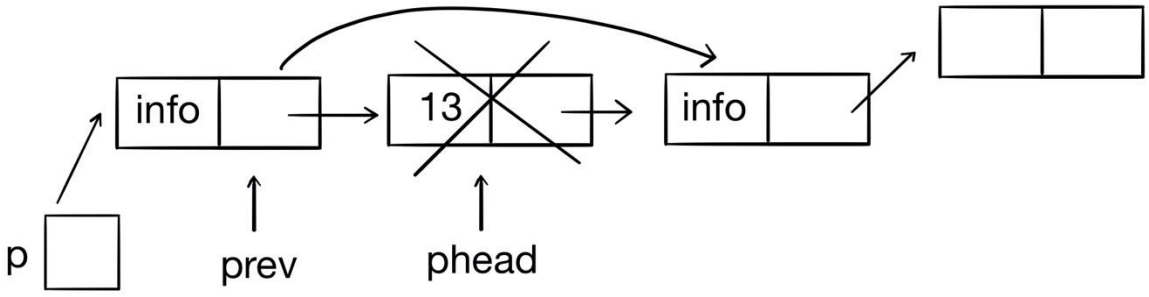


рис. 18.3 правильное удаление элемента из списка

При движении по циклу мы синхронно передвигаем эти 2 указателя, причем сначала prev, а потом phead. Нужно иметь в виду, что prev может быть равен NULL. Это случай удаления первого элемента из списка. Поэтому мы делаем следующую проверку:

```
if (prev)
 prev -> next = phead -> next;
```

Если в цикле не выполнилась ни одна итерация, это означает, что мы удаляем первый элемент списка. В таком случае нам нужно обновить указатель на голову:

```
 ph = phead -> next;
  free(phead);
```

Если в итоге мы ничего не удаляем, то мы просто возвращаем указатель на голову.

Есть второй вид интерфейсов, когда мы забираем ответственность у пользователя обновлять указатель на голову и обновляем его сами. В таком случае мы передаем двойной указатель, то есть не просто указатель на голову списка, а указатель на то место в памяти, где хранится указатель на голову списка.

```
void remove(struct list **pphead, struct data *elem){
        struct list *prev = NULL, *phead = *pphead;
        while (phead && !equals (&phead -> info, elem)){
            prev = phead;
            phead = phead -> next;
        }
        if (!phead)
          return;
        if (prev)
          prev -> next = phead -> next;
        else
          *pphead = phead -> next;
        free(phead);
}
```

Графы

V = {a, b, c}, E = {(a,b), (b,c)}, G = {V, E} (рис. 18.4)

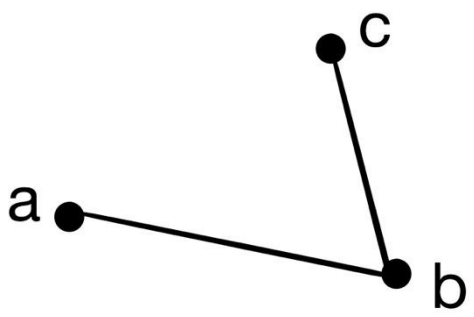


рис. 18.4 граф с 3 вершинами

Ациклические связные графы часто используют как метафору частичного порядка. Топологический порядок не единственен из- за частичности отношений, которая дает лаг.
Блок схемы с if очень похожи на ациклический связный граф.

Топологическая сортировка узлов ациклического ориентированного графа

Требуется привести рассматриваемый на рис. 18.5 граф к линейному графу.

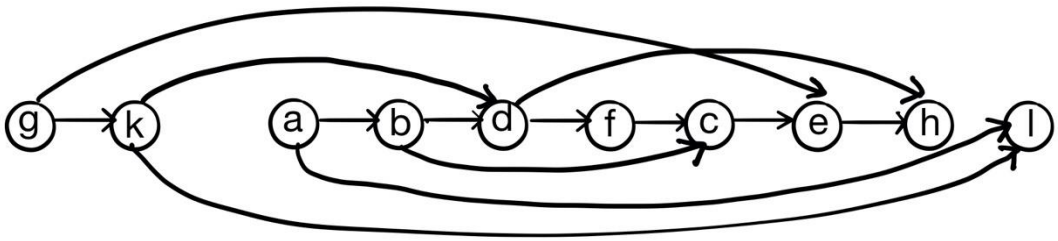


рис. 18.5 граф

На этом графе ключи расположены в следующем порядке: g, k, a, b, d, f, c, r, h, l. (поскольку топологическая сортировка неоднозначна, это один из топологических порядков). Последовательная обработка полученного линейного списка узлов графа эквивалентная их обработке в порядке обхода графа.
Частичный порядок (<) задается следующим набором отношений:
a<b, b<d , d<f, b<l, d<h, f<c, a<c, c<e, e<h, g< e, g<k, k<d, k<l
Его можно представить в виде такого графа:

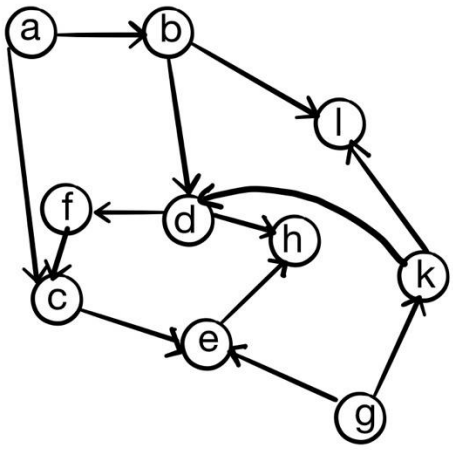


рис. 18.6 частичный порядок (<) графа

Так как граф конечен, всегда найдется вершина, в которую не входит ни одна дуга, то есть в отношении частичного порядка они стоят всегда в левой стороне. Это означает, что если мы формируем такой массив, где раньше стоящие элементы всегда состоят в отношения с большими числами слева, то мы можем перенести их в начало массива, не нарушив при этом свойства порядка. В нашем примере нет ничего меньше a и g, то есть они являются минимальными элементами, которые мы можем сразу выдать и удалить, потому что для них задачу мы решили. Если удалить вершину из ациклического ориентированного графа, то он останется ациклическим ориентированным графом.

Каждый узел исходного графа представляется с помощью дескриптора узла, который имеет вид (рис. 18.7):

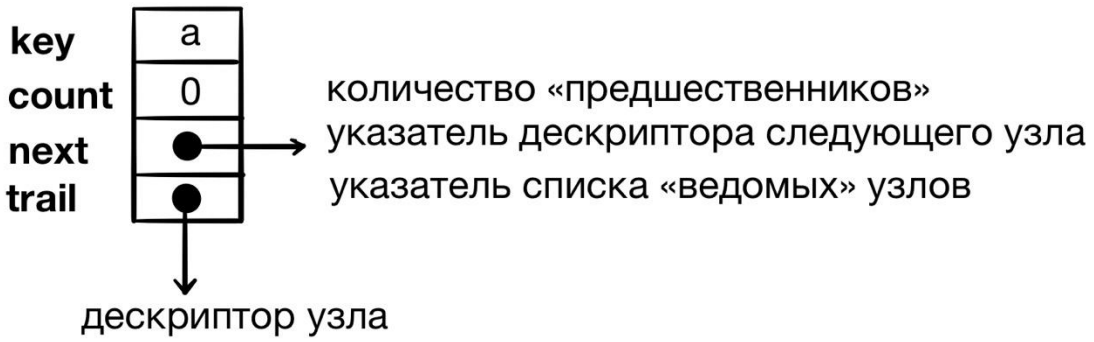


рис. 18.7 дескриптор узла

Ведомыми для узла n будут узлы, для которых n является предшественником. Каждый узел графа (не только ведущий) может иметь один или несколько ведомых узлов.

Если граф не очень плотный (между не всем вершинами есть ребра), в этой матрице будет много нулей, то есть они будут разреженными. И что бы мы не делали с такими матрицами, мы делаем это в основном нули, соответственно ответ знаем заранее и вычисления бесполезны.

Дескриптор каждого узла содержит ссылки на ведомые узлы. Так как заранее неясно, сколько, сколько у узла будет ведомых узлов, эти ссылки помещаются в список. На рис. 18.8 представлен элемент списка ссылок.

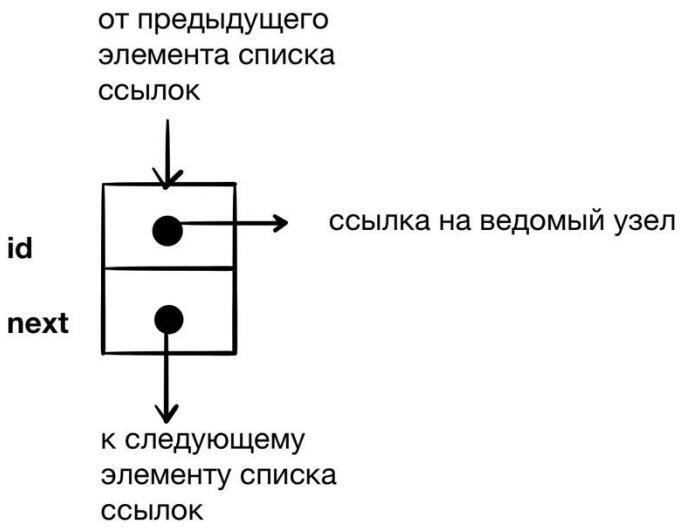


рис. 18.8 элемент списка ссылок

На схеме рис. 18.7, если мы хотим узнать, есть ли между 2 элементами дуга, нам нужно найти в списке всех вершин элемент, соответствующий вершине *a*, взять указатель на список ведомых элементов, его тоже перебрать полностью и найти, есть ли там нужная вершина. На схеме рис.18.8 более хитро: вместо того, чтобы иметь имя, сразу есть ссылка узел из этого большого списка. То есть хранится не b, а указатель на место, где хранится b. Поэтому мы сразу можем получить и имя, и count, и все, что хотим? То есть последующий список нужной вершины сильно упрощается.

На этой фазе вводятся пары ключей и из них формируется представление ациклического графа через дескрипторы узлов и списки ведомых узлов.

Первая фаза алгоритма: ввод исходного графа

- Исходные данные представлены в виде множества пар ключей (*), которые вводятся в произвольном порядке.
- После ввода очередной пары x < y ключи x и y ищутся в списке «ведущих» и в случае отсутствия добавляются к нему.
- В список ведомых узлов узла x добавляется ссылка на y, а счетчик предшественников y увеличивается на 1 (начальные значения всех счетчиков равны 0).

На рис. 18.9 представлена схема связи структуры в памяти.

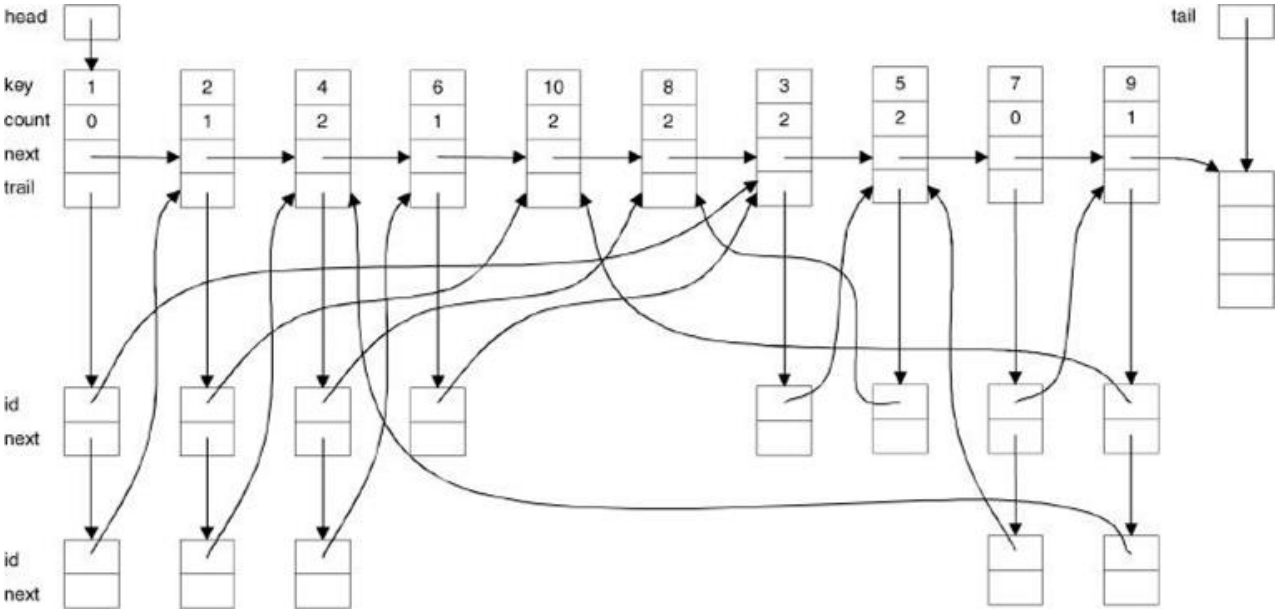


рис. 18.9 схема структуры в памяти

Большие блоки сверху — это большой список из вершин, маленькие блоки - много маленьких списков из «ведомых» узлов. Элемент ведомого узла — это не какая-то странная идентификация вершины, а ссылка на нее.
На самом деле это список с заголовочным звеном. Поэтому есть head и tail. Кроме того, чтобы все время не ходит до конца, тут сохраняется указатель на конец списка. Тогда пустой список — это список из одного буферного элемента, в котором нет данных. Проверяется, что head == tail (не забывайте, что указатели можно сравнивать на равенство). Когда в список добавляется новых элемент, память, которая уже есть сразу используется, а новая память заводится под новый буферный элемент.

Вторая фаза алгоритма: сортировка

- В списке «ведущих» элементов находим дескриптор узла z, у которого значение поля count равно 0.
- Включаем узел z в результирующую цепочку.
- Если у узла z есть «ведомые» узлы (значение поля trail не NULL):
  - просматриваем очередной элемент списка «ведомых» узлов;
  - корректируем поле count дескриптора соответствующего «ведомого» узла.

- Переходим к шагу 1.

Так как с каждой коррекцией поля count его значение уменьшается на 1, постепенно все узлы включаются в результирующую цепочку.

Топологическая сортировка на Си

Рассмотрим топологическую сортировку на Си:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ldr { /* дескрпитор ведущего узла*/
  char key;
  int count;
  struct ldr *next;
  struct trl *trail;
} leader;
typedefstruct trl { /*дескриптор ведомого узла */
  struct ldr *id;
  struct trl *next;
 } trailer;

 leader *head, *trail; /* два вспомогательных узла */
 int lnum; /*счетчик ведущих узлов */
```

Для того, чтобы понять, что сортировка закончилась успешно, есть поле lnum, которое уменьшается на 1 каждый раз, когда обрабатывается узел, то есть включается в итгговую цепочку, и уменьшается на 1 каждый раз, когда считывается новый узел. После успешной обработки поле lnum станет равным 0, то есть все узлы найдены.

Рассмотрим, как строится эта структура:

```
leader *find(char w) {
        leader 8h = head;
        /* барьер на случай отсутствия w */
        tail -> key = w;
        while (h -> key != w)
          h = h -> next;
        if (h == tail) {
                /* генерация нового ведущего узла*/
                tail = malloc(sizeof(leader));
```

```
            /* старый tail становится новым элементом списка */
            lnum++;
            h -> count = 0;
            h -> trail = NULL;
            h -> next = tail;
        }
        return h;
}
```

Функция find устроена так, что она знает об указателях head и tail на элементы списка с заголовочным звеном. Рассмотрим пример структуры из 2 элементов с заглушкой tail (рис. 18.10):

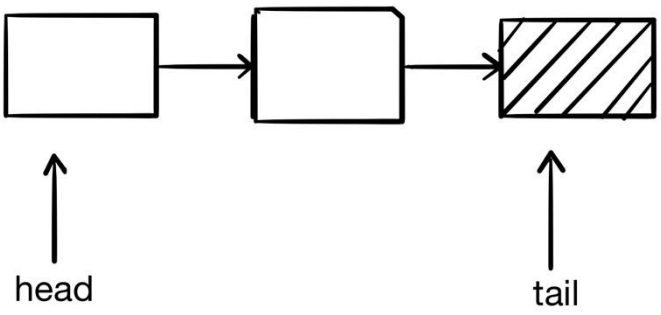


рис. 18.10 пример

Нам нужно понять, есть в этом списке ключ w. Для этого в буферный элемент заглушки, на который указывает tail, записываем w. До тех пор, пока ключ не равен w, идем дальше по next. Прелесть в том, что не нужно делать никаких проверок, так как есть буферный элемент, соответственно всегда есть next и head <> NULL. Если при этом мы остановимся tail, то очевидно, что элемента с ключом w не было. В таком случае нужно завести этот элемент, а для этого достаточно проинициализировать остатки буферного поля и создать новый буферный элемент. Если h <> w, то h указывает в середину списка, и мы возвращаем этот элемент.

Теперь рассмотрим, как это, собственно, используется:

```
void init_list() {
        leader *p, *q;
        trailer *t;
        char x, y;
```

```
        head = (leader *)malloc(sizeof(leader));
        tail = head;
        lnum = 0; /* начальная установка*/
        while(1){
                if sacnf("%c %c", &x, &y) != 2)
                 break;
           /* включение пары в список */
           p = find(x);
           q = find(y);
           /* включение пары в список */
           p = find(x);
           q = find(y);
           /* коррекция списка */
           t = malloc(sizeof(trailer));
           t -> id = q;
           t -> next = p -> trail;
           p -> trail = t;
           q -> count += 1;
        }
}
```

Читаем из входа очередную пара элементов, 2 раза вызываем функцию find, которая заодно добавляет в список x и y, если их не было. Чтобы это работало, сначала нужно завести буферный элемент с помощью malloc() и написать, что tail = head;
Так как пустой список — это список из одного буферного элемента в нашем случае. Потом нужно завести новый элемент в узде ведомых для x и записать его в p -> trail.

Это работает, только если мы уверены, что в списке нет повторяющихся элементов. Если такие элементы есть, то вместо 3 присваиваний должен быть цикл по p -> trail, который бы проверял, а нет ли в id чего-то похожего на q.

Лекция 19. Сортировки в Си.

Топологическая сортировка на Си: новый список

Рассмотрим сортировку уже готового списка:

```
void sort_list() {
        leader *p, *q;
        trailer *t;
        /* В выходной список включаются все узлы с count == 0 */
        p = head;
        head = NULL; /* голова выходного списка */
        while (p != tail) {
                q = p;
                p = q -> next;
                if (q -> count == 0) {
                        /* включение q в выходной список */
                        q -> next = head;
                        head = q;
                }
        }

        q = head; /* есть ведущий узел -> head != NULL */
        while (q ! = NULL) {
                printf ("%c\n", q -> key);
                lnum--;
                t = q -> trail;
                q = q -> next;
                while (t != NULL) {
                        p = t -> id;
                        p -> count -= 1;
                        if (p -> count == 0) {
                                p -> next = q; // достаточно для
                                q = p;         // правильной сортировки
                        }
                }
                t = t -> next;
        }
}
/* lnum == 0 */
```

Первый проход по списку (рис. 19.1) связывает в новый список все узлы с count == 0. Этот новый список начинается с head, поэтому, чтобы head не пропал, он записывается в p.

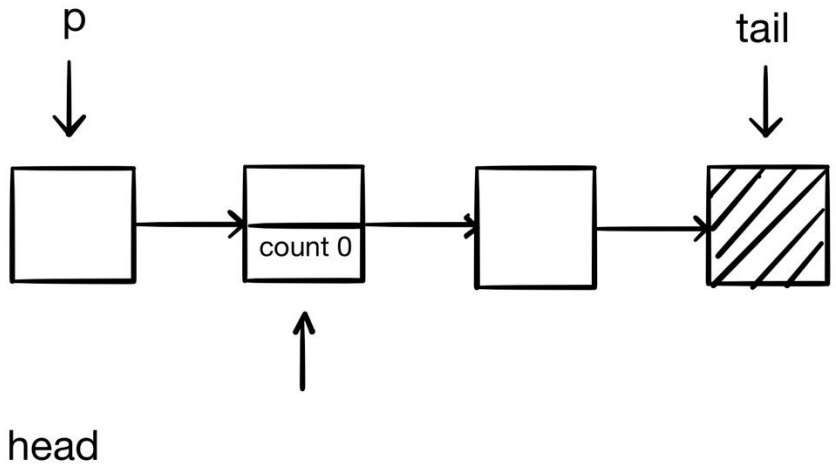


рис. 19.1 проход по списку

Далее происходит включения элемента q в голову списка, на которую указывает head. Теперь q указывает на следующий элемент, а q остается на месте (рис. 19.2).

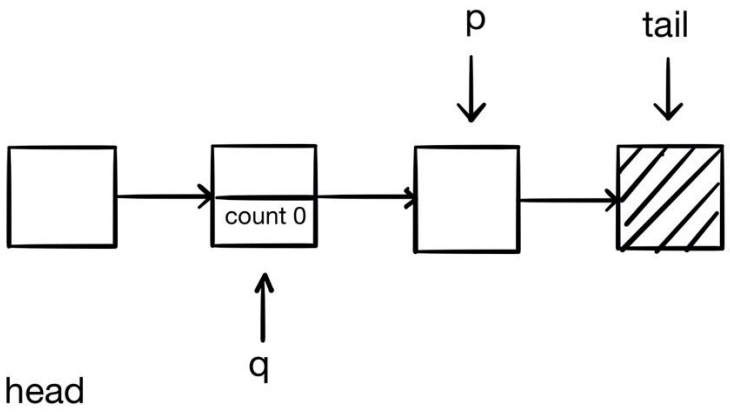


рис. 19.2 после первого прохода по списку

Теперь head становится новой головой, а поле next, на которое указывает q, указывает на старую голову. Но в нашем случае она никуда не указывает, потому что старой головы не было. Получается список из 1 элемента.

Когда q будет указывать на последний перед буферным элементом, а поле next элемента, на который указывает q, будет указывать на старую голову (рис. 19.3).

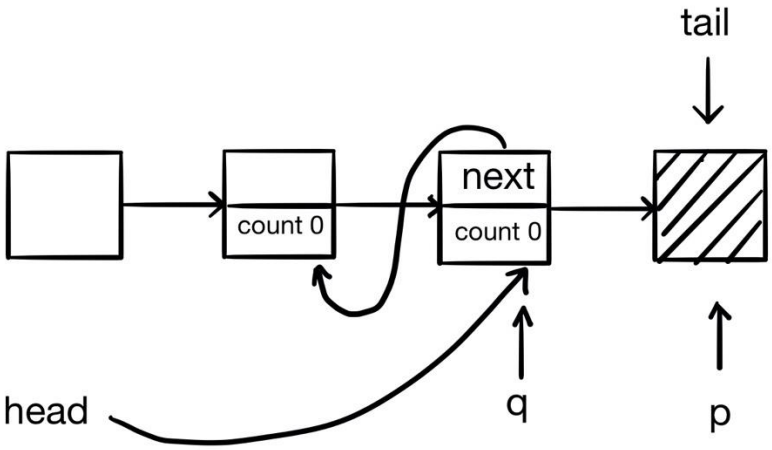


рис. 19.3 предпоследний проход по списку

После окончания цикла по списку ходить бесполезно.

q -> trail показывает на список всех узлов, в которые есть дуги из q. Далее проходимся по списку через цикл while (q ! = NULL).
Указатель p указывает туда, куда ведет дуга. Аналогом удаления дуги является уменьшение счетчика count на 1. Мы уменьшаем countна 1, если оказалось. Что это была последняя дуга, которая вела в p, то p -> next = q;, то есть q == 0.

Когда этот цикл заканчивается, lnum == 0. Lnum, как вы помните, глобальная переменная, которую мы увеличивали на 1, если заводили новую вершину. Если lnum != 0, то топологическая сортировка не удалась.

Основная функция топологической сортировки выглядит так:

```
int main(void) {
                        init_list();
                        sort_list();
                        return 0;
}
```

Сортировка

Теперь мы можем перейти к обычной сортировке.
Сортировка — это упорядочение наборов однотипных данных, для которых определено отношение линейного порядка (например, <, «меньше») по возрастанию или убыванию.
Здесь будут рассматриваться целочисленные данные и отношения порядка <.
Различают *внешнюю* и *внутреннюю* сортировку. Рассматривается только внутренняя сортировка: сортируемый массив находится в основной памяти компьютера. Внешняя сортировка применятся к записям на внешних файлах.

Сортировка слиянием (merge sort)

Идея сортировки слиянием заключается в следующем: у нас есть огромный массив, который не помещается в память. Мы разрезаем его на кусочки, которые уже помещаются в память. Берем первый кусочек и сортируем его, берем второй и тоже сортируем. Теперь у нас есть 2 отсортированных кусочка. Которые все еще помещаются в память. Теперь нужно выполнить операцию слияния (рис. 19.4). Мы хотим получить 1 упорядоченный массив из этих 2 кусочков за линейное время и так далее.

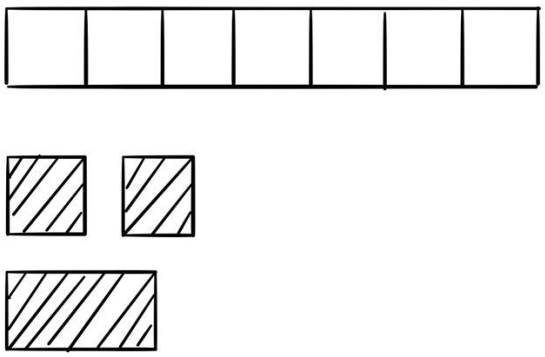


рис. 19.4 сортировка слиянием

Прежде, чем рассматривать внешние сортировки, посмотрим, что есть в стандартной библиотеке Си.

Сортировка: что есть в Си

#include <stdlib.h>

```
void qsort(void *buf, size_t num, size_t size,
        int(*compare)(const void *, const void *));
```

Функция qsort сортирует (по возрастанию) массив с указателем buf, используя алгоритм быстрой сортировки Ч.Э.Р. Хоара, который считается одним из лучших алгоритмов сортировки общего назначения.

Параметр num задает количество элементов массива buf, параметр size - размер (в байтах) элемента массива buf.
Параметр int(*compare)(const void *, const void *) задает правило сравнения элементов массива num. Функция сравнивает аргументы и возвращает:

- Целое < 0, если arg1 < arg2
- Целое = 0, если arg1 = arg2
- Целое > 0, если arg1 > arg2

Простейший алгоритм сортировки

Сведение сортировки к задаче нахождения максимального (минимального) из n чисел. Нахождение максимум из n чисел (n сравнений). Числа содержатся в массиве a[n];

```
max = a[0];
for (i =1; i < n; i++)
  if (a[i] > msx)
  max = a[i];
```

Алгоритм сортировки: находим максимальное из n чисел, получаем последний элемент отсортированного массива (n сравнений); находим максимальное из n-1 чисел, получаем предпоследний элемент отсортированного массива (еще n-1 сравнение), и так далее.

Общее количество сравнений: 1+ 2+ … + n-1 + n = n(n-1)/2.

Сложность алгоритма O(n^2).

Три общих метода внутренней сортировки

- *Сортировка обменами*: рассматриваются соседние элементы сортируемого массива и при необходимости меняются местами;
- *Сортировка выборкой*: идея описана выше;

- *Сортировка вставками*: сначала сортируются 2 элемента массива, потом выбирается третий элемент и вставляется в нужную позицию относительно первых двух и т. д.

Сортировка обменами (пузырьком)

Общее количество сравнений (действий): n(n-1)/2, так как внешний цикл выполняется (n-1) раз, а внутренний - в среднем n/2 раза.

```
void bubble_sort(int *a, int n) {
        int i, j, tmp;
        for (j = 1; j < n; j++)
          for (i = n-1; i >=j; --i){
                if (a[i-1]> a[i]){
                        tmp = a[i-1];
                        a[i-1]=a[i];
                        a[i] = tmp;
                }
          }
 }
```

Смысл внешнего цикла: минимальный элемент от 0 до j опустить вниз. Для этого рассматривается каждая пара элементов a[i-1], a[i], если они стоят не в том порядке, то они меняются местами. После первой итерации мы точно знаем, что элементы от a-1 до a-i стоят в правильном порядке. Сортировка называется так, потому что неотсортированный элемент, как пузырек, всплывает наверх до своего места (рис. 19.5).

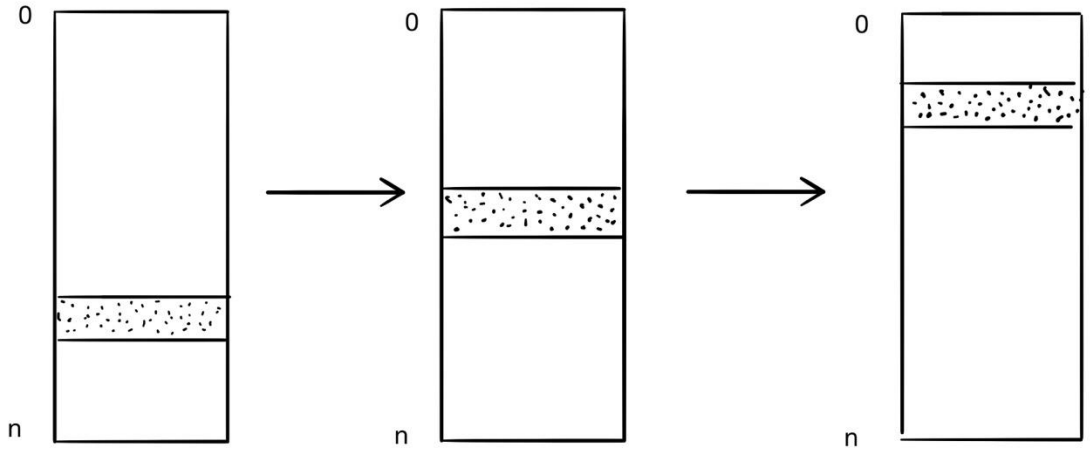


рис. 19.5 сортировка пузырьком

Сортировка вставками

Количество сравнений зависит от степени перемешанности массива *a*. Если массив *a* уже отсортирован, количество сравнений равно n-1. Если массив *a* отсортирован в обратном порядке (наихудший случай), количество сравнений имеет порядок n^2 (рис. 19.6).

```
void insert_sort(int *a, int n) {
        int i, j, tmp;

        for (j = 1; j < n; ++j) {
                tmp = a[j];
                for (i=j-1; i >= 0 && tmp < a[i]; i--)
                  a[i+1] = a[i];
                a[i+1] = tmp;
        }
 }
```

Задача внешнего цикла - вставить j-й элемент в уже отсортированный кусок от 0 до j-1.

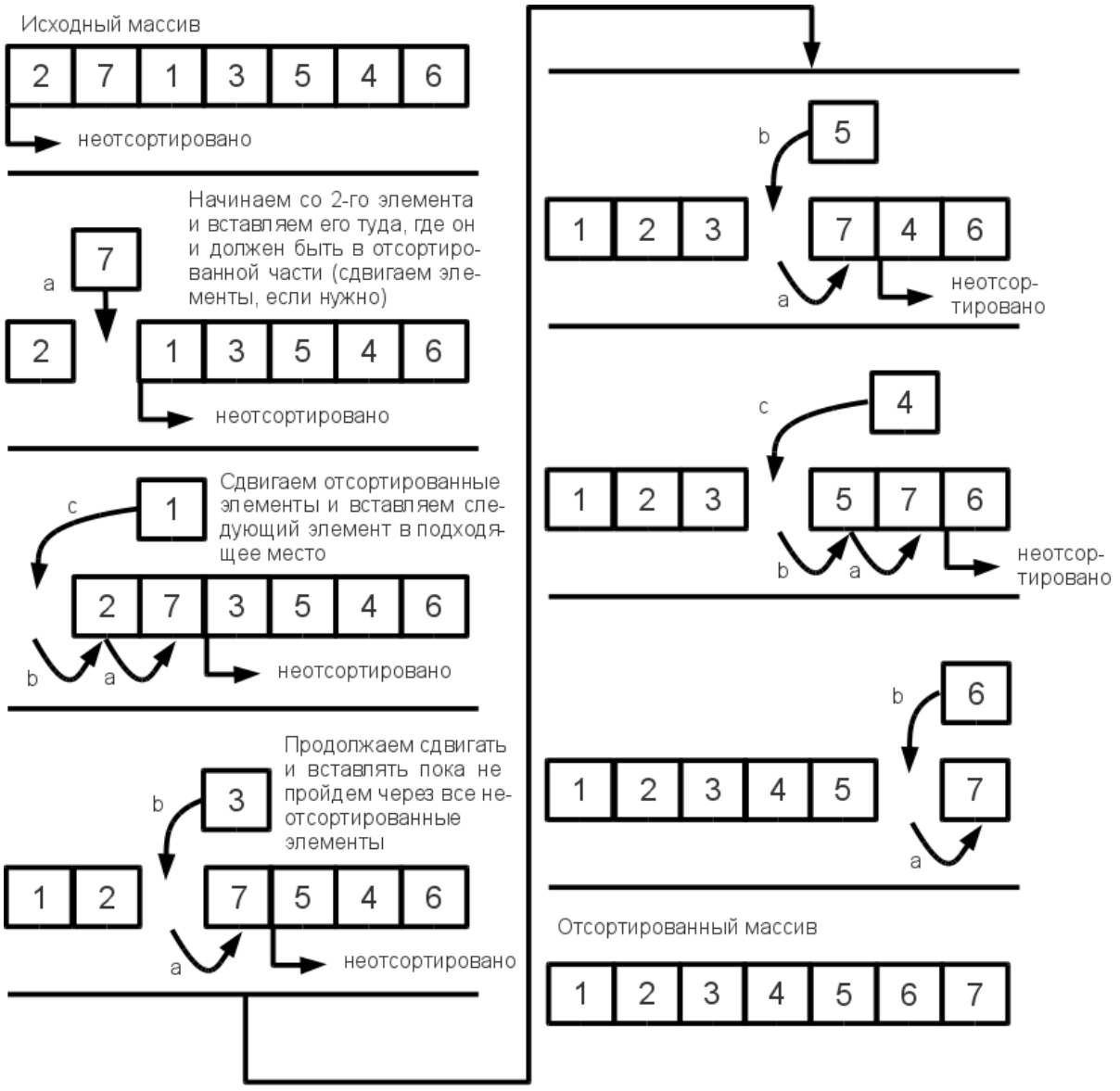


рис. 19.6 сортировка вставками

Оценка сложности алгоритмов

Скорость сортировки определяется количеством сравнений и количеством обменов (обмены занимают больше времени). Эти показатели интересны для худшего и лучшего случаев, а также интересно их среднее значение.

Кроме скорости, оценивается «естественность» алгоритма сортировки.

Естественным считается алгоритм, который уже на отсортированном массиве работает минимальное время, а на не отсортированном работает тем дольше, чем больше степень его неупорядоченности.

Важным показателем является и объем дополнительной памяти для хранения промежуточных данных алгоритма. Для рекурсивных алгоритмов расход памяти связан с необходимостью сохранять в автоматической памяти (стеке) локальные переменные и параметры.

Теорема. Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений $C_s \geq O(n\ log_2 n)$.

Доказательство. Сначала покажем, что
$C_s \geq log_2(n!)$ (1)
Алгоритм S можно представить в виде двоичного дерева сравнений. Так как любая перестановка индексов рассматриваемого массива может быть ответом в алгоритме, она должна быть приписан хотя бы одному листу дерева сравнений. Таким образом, дерево сравнений будет иметь не менее n! листьев.
Для высоты h_m двоичного дерева с m листьями имеет место оценка: $h_m \geq\ log_2 m$.
Любое двоичное дерево высоты h можно достроить до полного двоичного дерева высоты h, а у полного двоичного дерева высоты h 2^h листьев. Применив полученную оценку к дереву сравнений, получим искомую оценку (1).

Далее, применим к $log_2(n!)$ формулу Стирлинга
$n! = \sqrt{2\pi n} n^n e^{-n} e^{\theta(n)}$,
где $|\theta(n)| \leq \frac{1}{12n}$. Подставляя и логарифмируя, имеем
$log_2 n! = \frac{1}{2} log_2 2\pi n + n log_2 n - n +\ \theta(n)$,
$log_2 n! \geq\ O(n\ log_2 n)$.

Быстрая сортировка

Самая плохая с точки зрения наихудшей сложности сортировка — это сортировка пузырьком. Альтернативный метод, впоследствии получивший название «быстрая сортировка», изобрел информатик Чарльз Хоар в 1960. Он предполагает деление массива на две части, в одной из которых находятся элементы меньше определённого значения, в другой – больше или равные.

```
static void QuickSort(int *a, int ledt, int right) {
        /* comp -- компарад? i, j -- значения индексов */
```

```
        int comp, tmp, i, j;
        i = left; j = right;
        comp = a[(left + right)/2];
        do {
                while (a[i] < comp && i < right)
                  i++;
                while (comp < a[j] && j > left)
                  j--;
                if (i <= j) {
                        tmp = a[i];
                        a[i] = a[j];
                        a[j] = tmp;
                        i++, j--;
                }
                while (i <= j);

        }
}
```

Идея, реализованная во внешнем цикле — это процедура, которая называется partition. Изобразим числовую прямую (рис. 19.7), на которой изобразим точки нашего массива. Если мы найдем точку, которая называется компарандом (comparand), через которую проведем прямую, то она поделит все наше множество на 2 части: левое подмножество — это все элементы, которые будут меньше компаранда, а правое множество - элементы, которые больше компаранда.

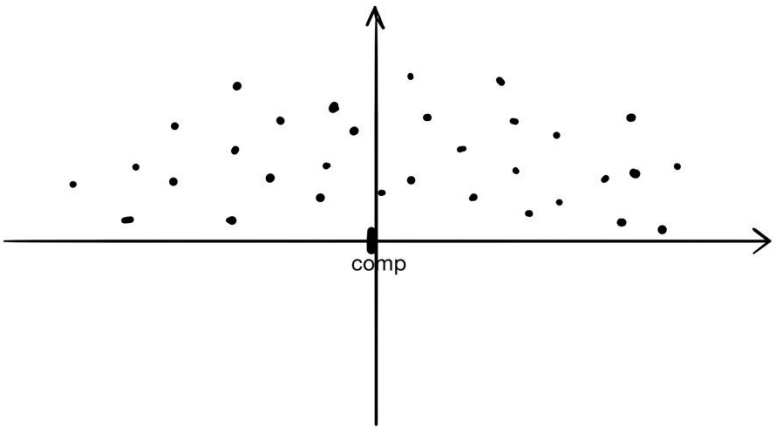


рис. 19.7 числовая прямая с компарандом

Для нашего массива используем процедуру, которая находит точку компаранда, разделяющую массив на 2 множества, элементы, одного из которых меньше компаранда, а другого - больше (рис. 19.8).

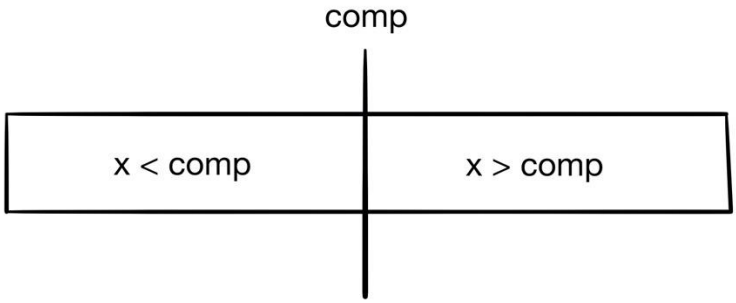


рис. 19.8 массив с компарандой

Если компаранд выбран так, что левое и правое множество не пусты, то мы можем перейти к следующему шагу и сортировать каждую из этих частей отдельно таким же образом, потому что сортировка частей не нарушит корректности общей сортировки. Далее, сливаем эти 2 множества и получаем отсортированный массив.
Теперь наша задача оформить процедуру выбора компаранда и следить за тем, чтобы ни одна из этих частей не была пустой. Каждый шаг разделения должен сводить задачу к меньшей.

Рассмотрим, как работает код, представленный выше. В результате первых двух циклов while массив разбился на 3 части: левая часть меньше компаранда, правая - больше, а посередине все плохо, так как тут те 2 элемента, на которых мы остановились и которые стоят не на своем месте (рис. 19.9).

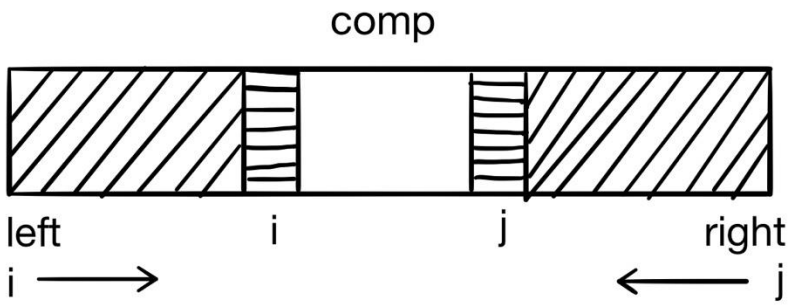


рис. 19.9 разделение массива

Элементы i и j могут схлопнуться в один. Но если, например i< j, то это означает, что это те элементы, на которых нужный порядок нарушился. Поменяем их местами.

Теперь массив хорошо разбит на 3 части: меньшую, большую и terra incognito, которую мы еще не прошли. Оборачиваем все во внешний цикл, задача которого неизведанную часть свести к 0. По завершении большого цикла i и j встретятся и получится, что массив разбился на 2 части (рис. 19.10).

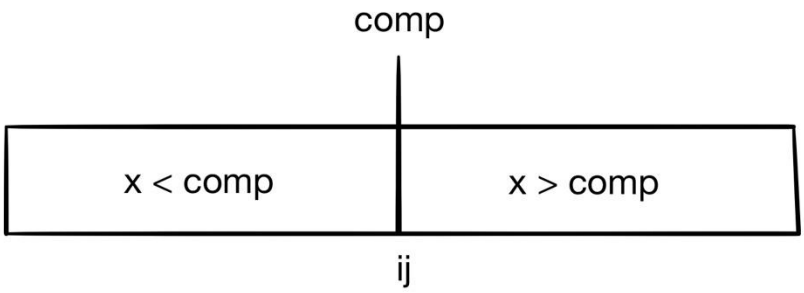


рис. 19.10 массив по завершении внешнего цикла

```
static void QuickSort(int *a, int ledt, int right) {
                                        ...
                                        if (left < j)
                                          QuickSort(a, left, j);
                                        if (i < right)
                                          QuickSort(a, i, right);
```

Программа быстрой сортировки:

```
void qsort(int *a, int n) {
  QuickSort(a, 0, n-1);
}
```

Нужно, чтобы значение компаранда было таким, чтобы он попал в середину результирующей последовательности. Мы пытаемся угадать, какой из элементов массива имеет такое значение. Чем лучше мы угадаем, тем быстрее выполнится алгоритм.

Лекция 20. Двоичные деревья

Вернемся к рассмотрению QuickSort.

```
static void QuickSort(int *a, int ledt, int right) {
        /* comp -- компарад? i, j -- значения индексов */
        int comp, tmp, i, j;
        i = left; j = right;
        comp = a[(left + right)/2];
        do {
                while (a[i] < comp && i < right)
                  i++;
                while (comp < a[j] && j > left)
                  j--;
                if (i <= j) {
                        tmp = a[i];
                        a[i] = a[j];
                        a[j] = tmp;
                        i++, j--;
                }
                while (i <= j);

        }
}
```

Покажем, что цикл **do-while** действительно строит нужное нам разбиение массива a[ ]:

- В процессе работы цикла i и j не выходят за пределы отрезка [left, right], так как в циклах **while** выполняются соответствующие проверки
- В момент окончания работы цикла **do-while** $j \leq right$, так как части разбиения не могут быть пустыми: хотя бы один элемент массива a[ ] (в крайнем случае a[right]) содержится в правой части разбиения.
- Аналогично, в момент окончания работы цикла **do-while** $i \geq left$
- В момент окончания работы цикла **do-while** элемент подмассива a[left…j] не больше любого элемента подмассива a[i…right], что очевидно.

Рассмотрим пример цикла **do-while** на примере: 5 3 2 6 4 1 3 7

1) Пусть в качестве первого компаранда выбран первый элемент массива - 5 (a[left]). Во время первого прохода цикла **do-while** после выполнения обоих циклов while получим: (5) 3 2 6 4 1 {3} 7 (в круглых скобках элемент с индексом j).

2) Поскольку i < j, элементы, выделенные скобками нужно поменять местами: 3 (3) 2 6 4 {1} 5 7.
3) В результате второго прохода цикла **do-while** получим: до обмена - 3 3 2 (6) 4 {1} 5 7;
   После обмена - 3 3 2 1 ({4}) 6 5 7.
   Третий проход лишь увеличивает i.

Теперь массив a состоит из двух подмассивов 3 3 2 1 4 и 6 5 7, причем i = 5, j = 4. Нужно рекурсивно применить метод к этим подмассивам.

При выборе компаранда можно брать первый элемент, значение которого больше значения следующего элемента. Для результирующих подмассивов из примера компаранды заключены в квадратные скобки:
3 [3] 2 1 4 и [6] 5 7.

Оценка времени работы быстрой сортировки (⊖- нотация). Если f(n) и g(n) - некоторые функции, то запись g(n) = ⊖ $(f(n))$ означает, что найдутся такие константы $c_1, c_2 > 0$ и такое n_0, что для всех n ≥ n_0 выполняются соотношения $0 \le c_1 f(n) \le g(n) \le c_2 f(n)$, то есть при болших n f(n) хорошо описывает произведение g(n).

Время выполнения цикла do-while - ⊖ (n), где n = right - left + 1. Для алгоритма QuickSort максимальное (наихудшее) время выполнения $T_{max}(n) = T_{max}(n-1) + \ominus(n)$. Очевидно, что $T_{max}(1) =$ - ⊖ (1). Следовательно,
$T_{max}(n) = T_{max}(n-1) + \ominus(n) = \frac{n(n-1)}{2} = \ominus(n^2)$.

Если исходный массив *a* отсортирован в порядке убывания, время его сортировки в порядке возрастания с помощью алгоритма QuickSort будет = ⊖ (n^2).
Минимальное и среднее время выполнения алгоритма QuickSort

$$T_{mean} = \ominus(n \log n)$$

С разными константами. Чем ближе разбиение на подмассивы к сбалансированному, тем константы меньше.

Доказательство использует теорему о рекуррентных оценках из Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 1999 ISBN 5-900916-37-5, с. 66-77
Рекуррентное соотношение для минимального (наилучшего) времени сортировки $T_{min}(n)$ имеет вид:
$T_{min}(n) = 2T_{min}(n/2) + \ominus(n)$,
Так как минимальное время получается тогда, когда на каждом шаге удается выбрать компаранд, который делит массив на два подмассива одинаковой длины [n/2].
Применяя ту же теорему, получаем $T_{min}(n) = \ominus(n \log n)$.

Рекуррентное соотношение для T(n) в общем случае, когда на каждом шаге массив делится в отношении q: (n-q), причем q равновероятно распределено между 1 и n, также можно решить и установить, что $T(n) = \ominus(n \log n)$ (та же книга с. 160-164).

Двоичное дерево

Двоичное дерево - набор узлов, который:

- Либо пуст (пустое дерево),
- Либо разбит на три непересекающиеся части:
  - узел, называемый корнем
  - двоичное дерево, называемое левым поддеревом
  - двоичное дерево, называемое правым поддеревом

Двоичное дерево не является частным случаем обычного дерева, хотя у этих структур много общего.
Основные отличия:

- Пустое дерево является двоичным деревом, но не является обычным деревом;
- Двоичные деревья (A(B, NULL)) и (A(NULL, B)) различны, а обычные деревья - одинаковы (рис. 20.1).

Термины:

- Ветви - дуга, связывающая два узла;
- Корень - главный узел;
- Листья - вершины, в которые входит одна *ветвь* и не выходит ни одной ветви;
- Высота - длина пути из вершины лист, при этом высота пустого дерева равна нулю, высота дерева из одного корня – единице. На первом уровне дерева может быть только одна вершина – корень дерева, на втором – потомки корня дерева, на третьем – потомки потомков корня дерева и т.д.

рис. 20.1 различные двоичные деревья

Однако, графы на рис. 20.1 являются *изоморфными*.

Узел двоичного дерева на Си описывается следующим образом:

```
typedef struct bin_tree {
        char info;
        struct bin_tree *left;
        struct bin_tree *right;
} node;
```

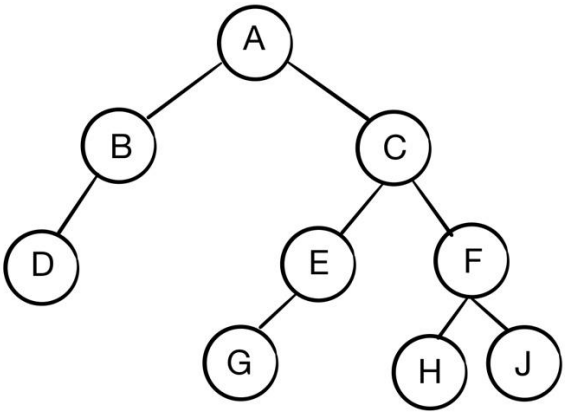


рис. 20.2 двоичное дерево

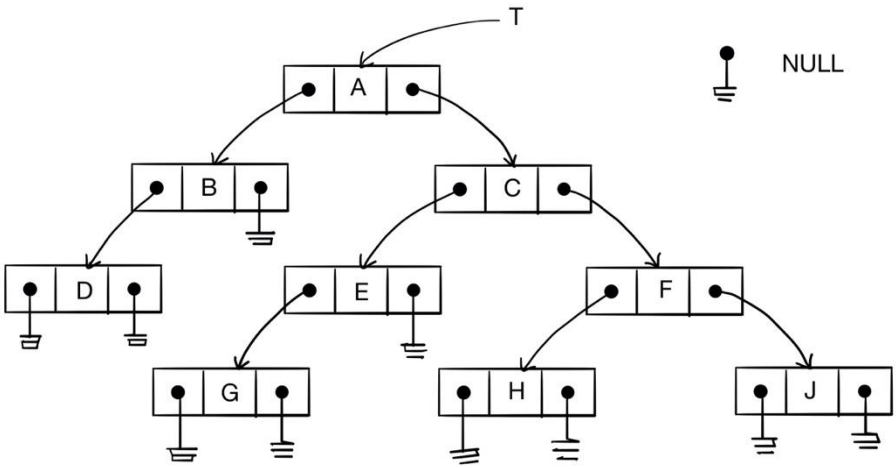


рис. 20.3 представление дерева с рис. 20.2 в компьютерном виде

Работа с деревом ведется точно так же, как со списком, то есть через указатель на корень дерева.

Способы обхода двоичного дерева

Имея один указатель, дерево, как и список, можно обойти единственным способом - в прямом порядке.
Обход в глубину в прямом порядке происходит следующим образом:

1) Обработать корень
2) Обойти левое поддерево
3) Обойти правое поддерево

Порядок обработки узлов дерева на рис. 20.2 такой: A B D E G F H J.
Линейная последовательность узлов, полученная при прямом обходе, отражает «спуск» информации от корня дерева к листьям.

Обход в глубину в обратном порядке осуществляется так:

1) Обойти левое поддерево
2) Обойти правое поддерево
3) Обработать корень

В таком случае порядок обработки узлов дерева на рис. 20.2: D B G E H J F C A.
Линейная последовательность узлов, полученная при обратном обходе, отражает «подъем» информации от листьев к корню дерева.

Кроме того, есть симметричный обход дерева (обход в симметричном порядке):

1) Обойти левое поддерево
2) Обработать корень
3) Обойти правое поддерево

Порядок обработки узлов дерева на рис. 20.2: D B A E G C H F J.
Чаще всего такой подход применяется в деревьях поиска, потому что в деревьях поиска нужно дополнительное ограничение на данные, которое заключается в том, что все что, находится слева, меньше всего того, что находится справа. То есть корень в таком случае является идеальным компарандом. Поэтому если пойти сначала налево, потом направо, получится обойти дерево в порядке возрастания.

Обход двоичного дерева *в ширину* заключается в обработке узлов дерева «по уровням» (уровень составляют все узлы, находящиеся на одинаковом расстоянии от корня).
В этом случае порядок обработки узлов дерева на рис. 20.2 такой: A B C D E F G H J.

Реализуем теперь прямой порядок обхода на Си:

```
void preorder(node *r) {
        if (r == NULL)
          return;
        if (r -> info)
          printf("%c", r -> info);
        preorder(r -> left);
        preorder(r -> right);
}
```

Если дерево пустое, то нужно вернуться, так как обходить нечего. Если дерево непустое, то с ним начинается работа: обрабатываем корень, обходим левое поддерево, обходим правое поддерево.
Функции inorder. postorder, как легко догадаться, будут отличаться только порядком вызова функции для правой и левой ветвей:

```
void postorder (node *r) {
        if (r == NULL)
          return;
        postorder(r -> left);
        postorder(r -> right);
        if (r -> info)
          printf("%c", r -> info);
}

void inorder(node *r) {
        if (r == NULL)
          return;
        inorder(r -> left);
        if (r -> info)
          printf("%c", r -> info);
        inorder(r -> right);
}
```

На этих примерах помимо того, что мы разобрали строение деревьев, можно научиться превращать рекурсивный вариант в нерекурсивный.
Для этого введем понятия:
r - указатель на корень дерева;
t - указатель на корень обрабатываемого (текущего) поддерева;
stack - массив, на котором моделируется стек;
depth - глубина стека;

top - указатель вершины стека.

Стек требуется для ручного сохранения параметров функции, локальных переменных и точки возврата 9если рекурсивных вызовов функции несколько). В функции inorder нет локальных переменных, а второй из двух рекурсивных вызовов хвостовой, что позволяет не сохранять его параметры в стеке. Поэтому сохраняется только параметр функции.

Рассмотрим нерекурсивный алгоритм функции симметричного обхода:

1) *Инициализация.*
   Сделать стек пустым, то есть затолкнуть NULL на дно стека:
   stack[0] =NULL;
   установить указатель t на корень дерева: t = r.
2) *Конец ветви.*
   Если t == NULL, перейти к пункту 4.
3) *Продолжение ветви.*
   Затолкнуть t в стек:
   stack[++top] = t;
   установить t = t -> left и вернуться к шагу 2.
4) *К обработке первой ветви.*
   Вытолкнуть верхний элемент стека в t:
   t = stack[top];
   top--;
   Если t == NULL, выполнение алгоритма прекращается, иначе обработать данные узла, на который указывает t, и перейти к шагу 5.
5) *Начало обработки правой ветви.*
   Установить t = t -> right и вернуться к шагу 2.

Посмотрим, как это будет выглядеть в коде:

```
int inorder( node *r, char *order){
        node *t = r, *stack[depth];     // depth = ?
        int top =0; i = 0;
        if (!t)
          return 0;
        stack[0] = NULL;                          // 1
        while (1) {
                while(t){                         // 2
                        stack[++top] = t;         // 3
                        t = t -> left;
                }
```

```
            t = stack[top--];                    //4
            if (t) {
                    order[i++] = t -> info; // обработка
                    t = t -> right;             // 5
            } else                              // t = NULL
              break;                           //4
            }
            return i;
    }
```

Так как мы не знаем глубину дерева, создаем динамический стек. i нужно для того, чтобы складывать элементы в массив в порядке inorder. Сначала мы заталкиваем 0 в стек. Если t == NULL, мы выходим, если нет, заталкиваем его в стек. По сути, дальше пока левый указатель не равен NULL, идем до упора налево.
Мы делаем хвостовую рекурсию в t = t -> right; и возвращаемся в точку while (1) {…

Таким образом, когда мы переписали рекурсию в итерации, у нас появилась память, которую эта функция от нас скрывала. Теперь понятно, что функция потребляет количество памяти, пропорциональное высоте дерева.

Основная проблема этого кода — это то, что мы не знаем глубину дерева, то есть depth. Следовательно, в таком виде этот код не сработает. Нужно делать либо динамический стек, либо заводить массив в статической памяти и как-то понять, какая будет глубина.

Прошитое двоичное дерево

Рассмотрим двоичное дерево на рис. 20.4. У этого дерева нулевых указателей больше, чем ненулевых: 10 против 8. Это -типичный случай. Будем записывать вместо нулевых указателей указатели на родителей (или более далеких предков) соответствующих узлов (такие указатели называются нитями). Это позволит при обходе дерева (рис.20.5) не использовать стек.

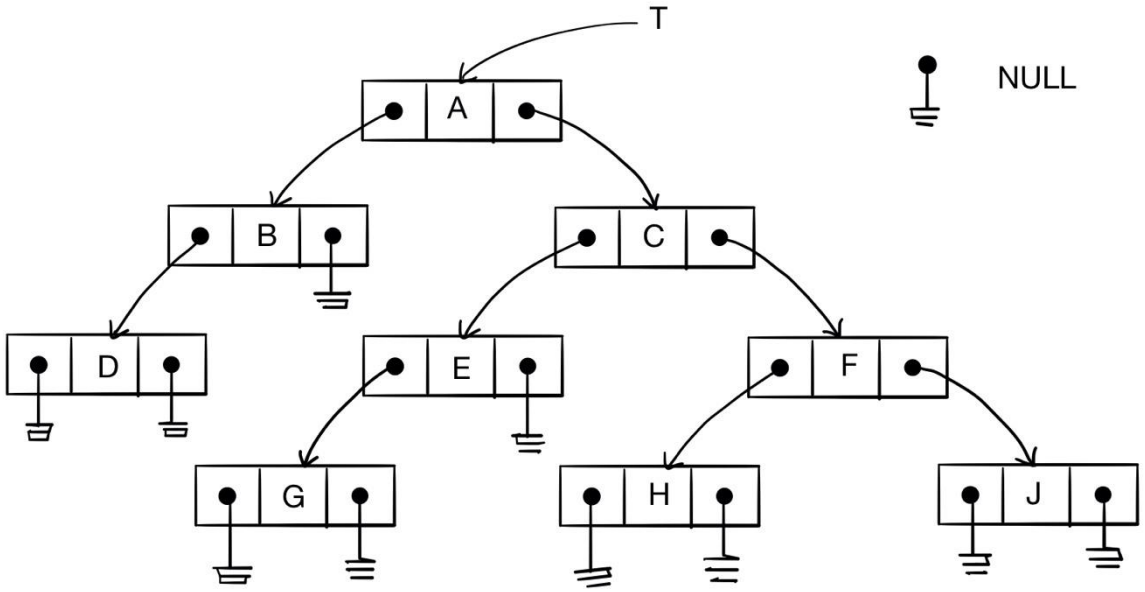


рис. 20.4 представление дерева в компьютерном виде

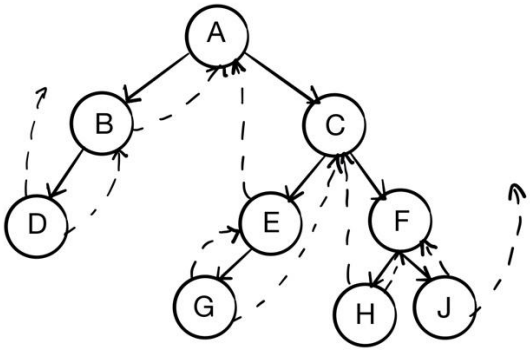


рис. 20.5 обход дерева

Эти указатели называются нитями, потому что они «прошивают» дерево через несколько уровней. На рис. 20.5 они отмечены пунктирной линией.

Лекция 21. Дерево поиска

Описание узла прошитого двоичного дерева на Си выглядит так:

```
typedef struct bin_tree {
        char info;
        struct bin_tree *left;
        struct bin_tree *right;
        char left_tag;
        char right_tag;
} threaded_node;
```

Нити устанавливаются таким образом, чтобы указывать на предшественников (левые нити) или последователей (правые нити) текущего узла при соответствующем обходе дерева.

Если сравнивать структуру обычного и прошитого дерева, то

Обычное дерево	Прошитое дерево
p -> left == NULL	p -> left_tag == 1, p -> left == P_pred_in
p -> left == Q	p -> left_tag == 0, p -> left == Q
p -> right == NULL	p -> right_tag == 1, p -> right == P_next_in
p -> right == Q	p -> right_tag == 0, p -> right == Q

Структура, рассмотренная в предыдущей лекции нам, не подходит, потому что мы не можем различить происходящее. Просто завести еще 2 указателя — это не вариант, потому что тогда память всегда пропадает.
Для этого в прошитом дереве добавляются теги. Раз 2 типа указателей, нужно понять, какого типа указатель левый, какого типа указатель правый. Для этого нам нужно хранить 2 бита информации -один на левый и один на правый указатель. Выше представлен самый неэкономный вариант. В зависимости от того, NULL или не NULL в левых указателях, если tag == 0, то обычный указатель, если tag == 1, то P_pred_in, то есть указатель на предыдущий элемент в симметричном порядке обхода. То же самое для правого: если указатель изначально ненулевой, то tag = 0, и этот указатель обычно правый сын; если указатель == 0, то tag = 1 и указатель указывает на следующий элемент в симметричном порядке обхода.

На самом деле скрыть эти теги, воспользовавшись утверждением: если нужно хранить k бит информации и адреса в структуре выравнены как минимум по k, то все корректные адреса заканчиваются на k нулевых бит. Поэтому можно необходимый бит

спрятать в младший бит указателя. Указатель в таких сложных структурах выравнен как минимум по 2, поэтому младший бит всегда будет равен 0. Следовательно можно взять адрес и младший бит использовать как тег. Главное не забыть, что если этот тег равен 1, то сначала нужно этот бит снять перед тем, как все разыменовывать. Правда из-за проверки и снятия в таком случае придется заплатить скоростью доступа к указателю, получая взамен отсутствие тегов. Важно прежде проверить, что это действительно уменьшает размер структуры.

```
void inorder(node *r) {
        if (r == NULL)
          return;
        inorder(r -> left);
        if (r -> info)
          printf("%c", r -> info);
        inorder(r -> right);
}
```

Рассмотрим на реальном дереве, как работает эта функция (рис. 21. 1). Пусть мы находимся в узле, на который указывает стрелочка.

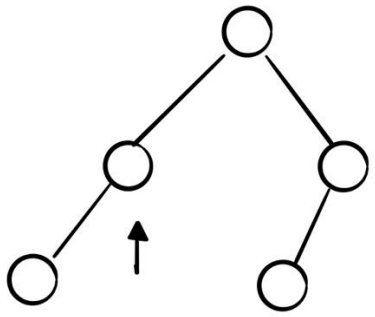


рис. 21.1 пример дерева

Сначала эта функция рекурсивно вызовет сама себя с левым сыном. То есть мы идем вниз налево до упора, потом начинаем возвращаться. Снова возвращаемся в эту точку. Теперь нужно понять. Когда мы в следующий раз вернемся в эту точку, каким будет t. Каким будет t — это следующий элемент, в ней мы должны пойти направо и обработать правого сына. Если правого сына нет, то мы возвращаемся в точку, где правый сын есть (в нашем примере в корень). То есть мы возвращаемся, пока сын лежит слева от родителя. Когда он доходит до точки корня, это означает, что мы вышли из верхней ветви рекурсии и следующим элементом будет корень. Если у него есть правый сын, то есть мы находимся в корне, идем направо И в этом сыне история

повторяется - идем до упора налево. Когда мы слева доходим до элемента, у которого нет левого сына, это означает, что мы дошли до следующего элемента. То есть если нумеровать элементы, то получится рис. 21.2:

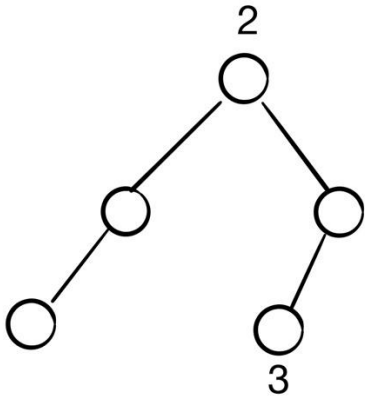


рис. 21.2 нумерация элементов после обхода

```
threaded_node *next_in(threaded_node *p) {
        threaded_node * = p -> right;
        if (p -> right_tag == 1)
          return q;
        while (q -> left_tag == 0) // q != NULL
          q = q -> left;              // q -> left != NULL
        return q;
}
```

Логика функции выше ровно такая же: действие с вершиной зависит от того, есть ли у этой вершины правый сын.
Функция next_in фактически реализует симметричный обход дерева, так как позволяет для произвольного узла дерева P найти следующий элемент P_next_in. Многократно применяя эту функцию, можно вычислить топологический порядок узлов двоичного дерева, соответствующий симметричному обходу.
Аналогичным образом можно построить функции, вычисляющие следующий узел дерева в прямом или обратном порядке обхода.

Важно обратить внимание на то, если мы идем влево до упора, последний элемент необязательно будет листом. Например, в ситуации на рис. 21.3 для выделенного стрелочкой указателя следующим элементом будет элемент, выделенный квадратом.

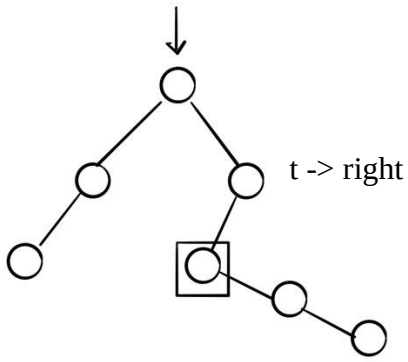


рис. 21.3 стандартная ситуация, когда следующий элемент не лист

Таким образом, с помощью обычного представления невозможно для произвольного узла P вычислить P_next_in, не вычисляя всей последовательности узлов.
Функции next_in не требуется стек ни в явной, ни в неявной (рекурсия) форме.
Если p -произвольно выбранный узел дерева, то следующий фрагмент функции next_in:

```
q = p -> right;
if (p -> right_tag == 1)
  return q;
```

выполняется только 1 раз.
Обход прошитого дерева выполняется быстрее, так как для него не нужны операции со стеком.
Для inorder требуется больше памяти, чем для next_in, из-за массива stack[depth] (т.к пропорционально высоте). Кроме того, нельзя допускать переполнения стека деревьев (массив выделяется с запасом, либо используется реализация стека с динамическим выделением памяти).

В функции inorder используется указатель r на корень двоичного дерева. Желательно, применив функцию next_in к корню r, получить указатель на самый первый узел дерева для выбранного порядка обхода (рис. 21.4). Для этого к дереву добавляется еще один узел - заголовок дерева (header).

```
header -> left_tag = 0;
header -> right_tag = 0;
header -> left = r;
header -> right = header;
```

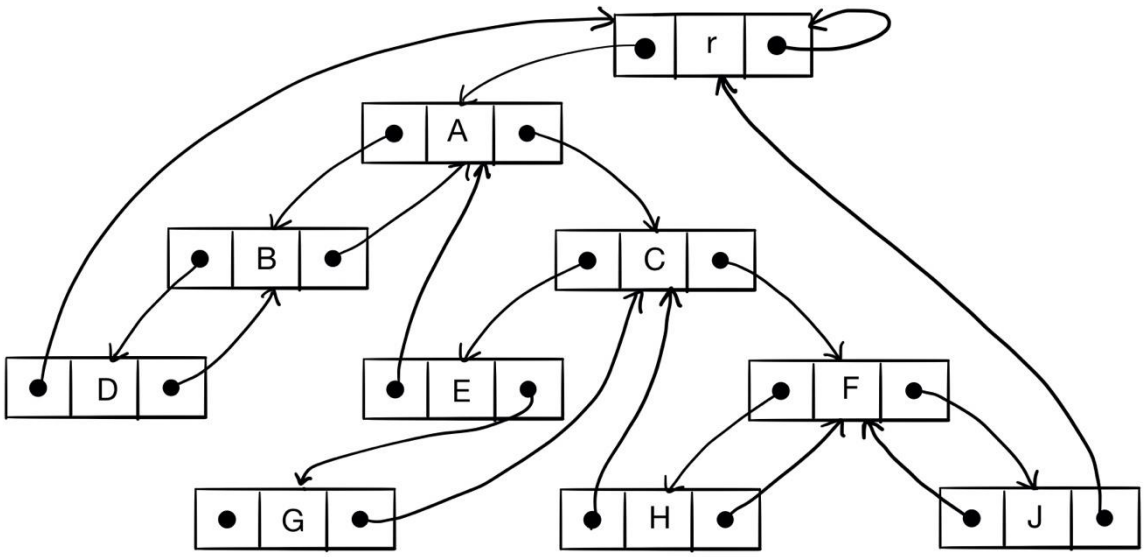


рис. 21.4 указатель на корень двоичного дерева

Двоичные деревья поиска

Проблема: организовать хранилище данных, которое позволяет хранить большие объемы данных и предоставляет возможность быстро находить и модифицировать данные.
Хранилище данных обеспечивает пользователю интерфейс, в котором определены словарные операции:

- Search (fetch) - найти
- Insert - вставить
- Delete - удалить

Также предоставляется один или несколько вариантов обхода хранилища (посещение всех данных).

Варианты решения: деревья поиска, хеширование.

Если у нас нет вставки и удаления, то можно использовать массив, в который занесены элементы. Поиск по отсортированному массиву логарифмический, по неотсортированному - линейный, то есть пропорционален количеству элементов. Если также есть дополнительные ограничения на удаления, как, например, в стеке, то ставка и удаление становятся проблемой. Чтобы поиск был логарифмическим, нужна сортировка, а в отсортированный массив новые элементы нужно будет вставлять в определенные места, чтобы не нарушать отсортированности.

Если бы у нас был список, то вставка элементов превратилась бы в константную операцию. Но возникает вопрос: а что, если мы вставим тот же самый элемент? Тогда нам нужно бы убедиться, что этого элемента раньше не было, для этого нужно просмотреть весь массив, что представляет из себя линейную сложность. Таким образом, и вставка, и удаление будут иметь линейную сложность.
Если бы у нас было любое дерево (без ограничений на структуру), то поиск бы просматривал все дерево. При вставке элемента сначала нужно проверить все дерево на наличие этого элемента, а потом решить, куда именно вставить. Поэтому нужно задать ограничение на структуру данных, которое будет ускорять все процессы.

При выполнении инварианта операции делать проще, однако надо поддерживать непроизвольную структуру данных, а это стоит чего-то. Поэтому нужно смотреть, оправдывает ли такая структура ресурсы, выделенные на ее поддержку.

В обычном двоичном дереве неудобно, что мы не знаем, где хранится элемент. Для того, чтобы это знать, будем использовать двоичное дерево, у которого для каждой вершины не более двух сыновей. Кроме того, еще из сортировок мы знаем, что если есть частичный порядок элементов, то поиск из линейной сложности превращается в логарифмическую. Объединив эти 2 идеи, получим, что, если у нас есть множество с заданным порядком и любые два элемента состоят в отношении частичного порядка, то есть два сына и два варианта: меньше или больше (при условии, что мы не храним элементов одинакового размера).

```
struct BT_node {
        int key;
        struct BT_node *left;
        struct BT_node *right;
        struct BT_node *parent;
}
```

Ключи в двоичном дереве поиска хранятся с соблюдением свойства упорядоченности.
Пусть x -произвольный узел двоичного дерева поиска.
Если узел y принадлежит левому поддереву, то key[y] < key[x];
Если узел y находится в правом поддереве узла x, то key[y] > key[x].
Возможно хранение дублирующихся ключей (не строгие неравенства), не рассматривающихся в этом курсе.
Например, дерево на рис. 21.5:

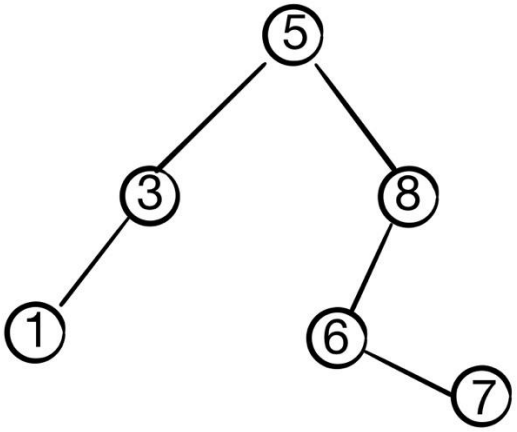


рис. 21.5 пример двоичного дерева поиска

Чтобы найти число 6 в этом дереве, смотрим на корень, он равен 5, значит, 6 может быть только в правом поддереве. Правый сын = 8, сравниваем 8 с 6, если 6 есть в дереве, то она должна быть в левом поддереве и т. д.

Двоичные деревья: поиск узла

На входе: искомый ключ k и указатель root на корень поддерева, в котором производится поиск.
На выходе: указатель на узел с ключом key == k (если такой узел есть), либо пустой указатель NULL.

```
struct BT_node *Btserach (struct BT_node *root, int k)
{
        if (! root || root -> key == k)
          return root;
        if (k < root -> key)
          return Btsearch(root -> left, k);
        else
          return Btsearch(root -> right, k);
}
```

Продвинутый компилятор может превратить код, написанный выше, в следующий код:

```
struct BT_node *Btserach (struct BT_node *root, int k)
{
```

```
        struct BT_node *p = root;
        while (p && p -> key != k)
          if (k < p -> key)
            p = p -> left;
          else
            p = p -> right;
        return p;
}
```

Время поиска O(h), где h -высота дерева

Двоичные деревья: минимум и максимум

На входе: указатель root на корень поддерева.
На выходе: указатель на узел с минимальным ключом key.

```
struct BT_node *Btmin(struct BT_node *root)
{
        struct BT_node *p = root;
        while (p -> left)
          p = p -> left;
        return p;
}
```

Время поиска O(h), где h -высота дерева. Функция для поиска максимума аналогична.
Теперь рассмотрим более сложное - обход дерева.
Приведем код. Аналогичный next_in в прошитом дереве.

На входе: указатель node на узел поддерева.
На выходе: указатель на следующий за node узел дерева.

```
struct BT_node *Btsucc(struct BT_node *node){
        struct BT_node *p = node, *q;
        /* 1 случай: правое подерево узла не пусто */
        if (p -> right)
          return Btmin(p -> right);
   /* 2 случай: правое поддерево узла пусто, идем по родителям до тех пор,
     пока не найдем родителя, для которого наше поддерево левое */
   q = p -> parent;
   while (q && p == q -> right) {
        p =q;
```

```
        q = q -> parent;
   }
   return q;
}
```

Здесь прослеживается связь с симметричным порядком. Время выполнения данной программы O(h), где h -высота дерева.

Двоичные деревья поиска: вставка

На входе: указатель root на корень дерева и указатель node на новый узел, у которого есть значение ключа, а все поля с указателями имеют значение NULL.

```
struct BT_node *Btinsert(struct BT_node * root, struct BT_node){
        struct BT_node *p, *q;
        p = root, q = NULL;
        while (p) {
                q = p;
                p = (node -> key < p -> key) ? p -> left : p -> right;
        }
        node -> parent = q;
        if (q == NULL)
          root = node;
        else if (node -> key < q -> key)
          q -> left = node;
        else
         q -> right = node;
        else
         q -> right = node;
        return root;
}
```

Всегда есть только 1 место, куда можно вставить новый элемент и это место задается процессом поиска. Например, мы хотим вставить 4, расположенное в node, в дерево, представленное на рис. 21.5. В этом конкретном дереве 4 может быть только слева от 5. В дереве с корнем 3 4 может быть только справа от 3, а там ничего нет, значит, подвешиваем туда (рис. 21.6).
Эта функция состоит из поиска и какой-то константной сложности, следовательно ее сложность тоже O(h), где h -высота дерева.

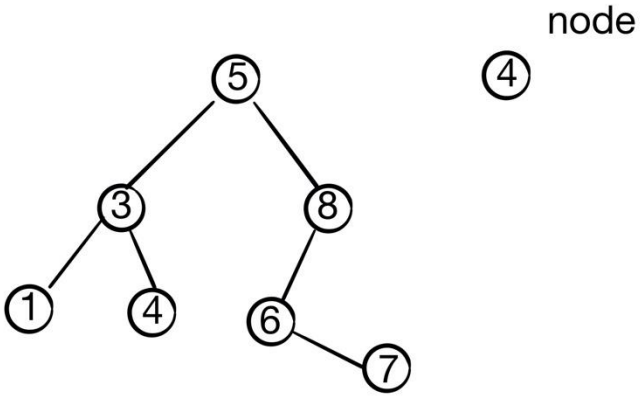


рис. 21.6 добавление 4 в двоичное дерево поиска

Двоичные деревья поиска: удаление

На входе: указатель на корень root дерева T и указатель на узел n дерева T.
На выходе: двоичное дерево T с удаленным узлом n (ключи нового дерева по-прежнему упорядочены).

Необходимо рассмотреть три случая:

1) У узла n нет детей (листовой узел.
   Тогда мы просто удаляем узел n;
2) У узла n только 1 ребенок.
   Вырезаем узел n, соединив единственного ребенка узла n с родителем узла n.
3) У узла n два ребенка.
   Находим succ(n) и удаляем его, поместив ключ succ(n) в узел n.

Таким образом, в самом сложном случае, когда мы хотим удалить узел, у которого есть 2 сына, мы берем следующий по размеру элемент в дереве (минимальный элемент в дереве, который больше, чем текущий). Про этот элемент известно: 1) он гарантированно есть, 2) у него нет левого сына. Поэтому в примере на рис. 21.7, где мы хотим удалить узел с ключом 7, физически мы удаляем 9. Но мы то должны были удалить 7, а не 9. Это проблема решается таким способом: мы можем на место 7 поставить 9, потому что 9 — это следующий элемент (минимальный элемент в дереве,

который больше, чем 7). Таким образом, мы не нарушаем никакие свойства двоичного дерева поиска.

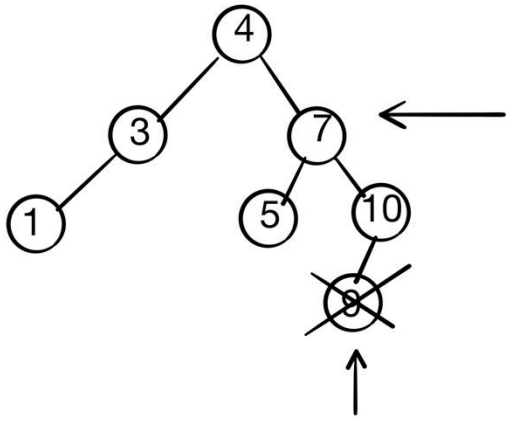


рис. 21.7 удаление узла с ключом 7

```
struct BT_node *BTdelete(struct BT_node**root ,struct BT_node *n) {
        struct BT_node *x, *y;
        /* Шаг 1: y -- указатель на удаляемый узел n */
        y = (!n -> left || ! n -> right) ? n : BT_succ(n);
        /* Шаг 2: x -- указатель на ребенка y либо NULL */
        x = y -> left ? y -> left : y -> right;
        /* Шаг 3: если x есть, вырезаем y из родителей */
        if x
          x -> parent = y -> parent;
        /* Шаг 3: если у y нет родителя, x -- новый корень */
        if (!y -> parent)
          *root = x;
        else {
                /* Шаг 3: ч присоединяется к y -> parent */
                if (y == y -> parent -> left)
                  y -> parent -> left = x;
                else
                  y -> parent -> right = x;
                /*Шаг 4: если удаляется не узел n, а succ(n), необходимо
                изменить данные узла n на данные узла succ(n) */
                if (y != n)
                  n -> key = y -> key;

                /* функция возвращает указатель удаленного узла,
```

```
                что дает возможность использовать этот узел в других
                структурах, либо очистить занимаемую им память */
                return y;
        }
```

Время выполнения O(h), где h - высота дерева.

Лекция 22. АВЛ деревья

Построение двоичного дерева поиска

Постановка задачи. Пусть имеется множество K из m ключей: K = $\{k_0, k_1, \dots, k_{m-1}\}$.
Разобьем K на три подмножества K_1, K_2, K_3 такие, что $|K_2| = 1, |K_1| \geq 0, |K_3| \geq 0$.
$K_2 = \{k\}$ и $\forall l \in K_1 : l < k, \forall r \in K_3 : k < r$.

Далее аналогично разбиваем множества K_1, K_2, K_3 пока не кончатся ключи.
Пример. K = {15, 10, 1, 3, 8, 12, 4}.
Первое разбиение: {1, 3, 4}, {8}, {15, 10, 12}.
Второе разбиение: {{1} {3} {4}} {8} {{10} {12} {15}}.
Получилось полностью сбалансированное двоичное дерево.

Определение. Дерево называется полностью сбалансированным (совершенным), если длина пути от корня до любой листовой вершины одинакова.

Пусть h - высота полностью сбалансированного двоичного дерева. Тогда число вершин m должно быть равно:

$$m = 1 + 2 + 2^2 + \cdots + 2^h = 2^h - 1$$

Откуда h = $log_2(m + 1)$.
Если все m ключей известны заранее, их можно отсортировать за O(mlog_2m), после чего построение сбалансированного дерева будет тривиальной задачей.

Если дерево строится по мере поступления ключей, то возможны все варианты: от линейного дерева с высотой O(m) до полностью сбалансированного дерева с высотой O($log_2 m$).

Пусть T = root, left, right - двоичное дерево; тогда $h_T = \max(h_{left}, h_{right}) + 1$.

Деревья Фибоначчи

Числа Фибоначчи возникли в решении задачи о кроликах, предложенном в XIII веке Леонардо из Пизы, известным как Фибоначчи.
Задача о кроликах заключалась в следующем: пара новорожденных кроликов помещена на остров. Каждый месяц любая пара дает приплод - также пару кроликов. Пара начинает давать приплод в возрасте двух месяцев. Сколько кроликов будет на острове в конце n-го месяца?

В конце первого и второго месяцев на острове будет одна пара кроликов: $f_1 = 1, f_2 = 1$.
В конце третьего месяца родится новая пара, так что $f_3 = f_2 + f_1 = 2$.
По индукции можно доказать, что для $n \geq 3$ $f_n = f_{n-1} + f_{n-2}$.

Определение дерева Фибоначчи:

- Пустое дерево — это дерево Фибоначчи с высотой h = 0
- Двоичное дерево, левое и правое поддерево которого есть деревья Фибоначчи с высотами соответственно h-1 и h-2 (либо h-2 и h-1), есть дерево Фибоначчи с высотой h.

Из определения следует, что в дереве Фибоначчи значения высот левого и правого поддерева отличаются ровно на 1.
Пример. Дерево Фибоначчи с h = 6 выглядит так (рис. 22.1):

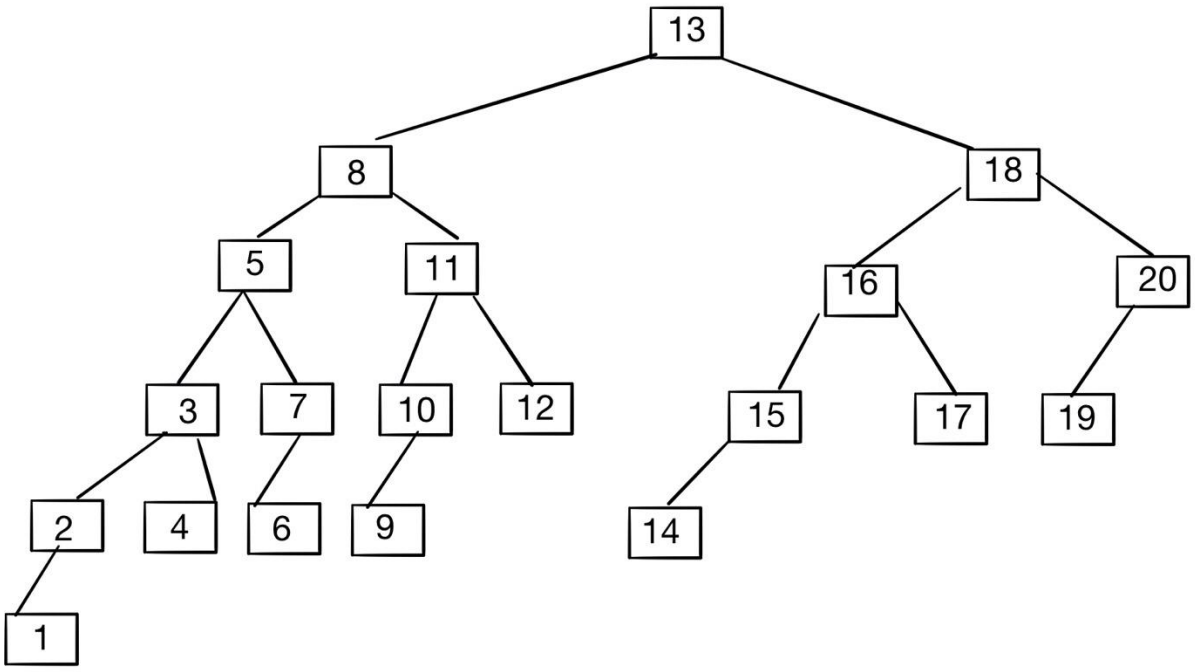


рис. 22.1 дерево Фибоначчи с h =6

Теорема. Число вершин в дереве Фибоначчи F_h высоты h равно $m(h) = f_{h+2} - 1$.
Доказательство проводится по индукции:

1) При h = 0: m(0) = $f_2 - 1 = 0, m(1) = f_3 - 1 = 1$.
2) По определению m(h) = m(h-1) + m(h-2) + 1, значит,

$$m(h) = (f_{h+1} - 1) + (f_h - 1) + 1 = f_{h+2} - 1$$

Так как $f_h + f_{h+1} = f_{h+2}$.

Теорема. Пусть C_1 и C_2 таковы, что уравнение

$$r^2 - C_1 r - C_2 = 0$$

Имеет два различных корня r_1 и r_2, $r_1 \neq r_2$.
Тогда для $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ выполняется соотношение $a_n = C_1 a_{n-1} + C_2 a_{n-2}$.
Доказательство: раз r_1 и r_2 - корни одного уравнения, то $r_1^2 = C_1 r_1 + C_2$,
$r_2^2 = C_1 r_2 + C_2$.
Поэтому $C_1 a_{n-1} + C_2 a_{n-2} = C_1(\alpha_1 r_1^{n-1} + \alpha_2 r_2^{n-1}) + C_2(\alpha_1 r_1^{n-2} + \alpha_2 r_2^{n-2})$
$= \alpha_1 r_1^{n-2}(C_1 r_1 + C_2) + \alpha_2 r_1^{n-2}(C_1 r_1 + C_2)$
$= \alpha_1 r_1^{n-2} r_1^2 + \alpha_2 r_2^{n-2} r_2^2$
$= \alpha_1 r_1^n + \alpha_2 r_2^n$
$= a_n$.

Теорема. Пусть C_1 и C_2 таковы, что уравнение

$$r^2 - C_1 r - C_2 = 0$$

Имеет два различных корня r_1 и r_2, $r_1 \neq r_2$.
Тогда из $a_n = C_1 a_{n-1} + C_2 a_{n-2}$ и начальных условий a_0, a_1 следует $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$.

Доказательство: нужно повторить в обратном порядке вывод предыдущей теоремы и подобрать такие α_1, α_2, чтобы $a_0 = \alpha_1 + \alpha_2$,
$a_1 = \alpha_1 r_1 + \alpha_2 r_2$.
Решая эту систему линейных уравнений, получим
$\alpha_1 = \frac{a_1 - a_0 r_1}{r_1 - r_2}$, $\alpha_2 = \frac{-a_1 + a_0 r_1}{r_1 - r_2}$.

Применим доказанные теоремы к числам Фибоначчи $f_n = f_{n-1} + f_{n-2}$.
Уравнение $r^2 - r - 1 = 0$ имеет корни $r_{1,2} = \frac{1 \pm \sqrt{5}}{2}$.
Следовательно, $f_n = \alpha_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + \alpha_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$,
$f_0 = \alpha_1 + \alpha_2 = 0$,
$f_1 = \alpha_1(\frac{1+\sqrt{5}}{2}) + \alpha_2(\frac{1-\sqrt{5}}{2}) = 1$.
Из описанной системы получаем $\alpha_1 = \frac{1}{\sqrt{5}}$, $\alpha_2 = -\frac{1}{\sqrt{5}}$.
Отсюда $m(h) = f_{h+2} - 1 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2} - 1$.

Заметим, что второе слагаемое по модулю не превосходит единицы, а, следовательно:

$m(h) + 1 > \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}$.

Обозначим $\gamma = \frac{1+\sqrt{5}}{2}$ и логарифмируем неравенство.

Тогда $h + 2 < \frac{log_2(m+1)}{log_2\gamma} + \frac{log_2\sqrt{5}}{log_2\gamma}$,

Откуда

$h < 1.44log_2(m+1) - 0.32.$

Таким образом, мы доказали, что для деревьев Фибоначчи с числом вершин m количество сравнений в худшем случае не превышает $1.44log_2(m+1)$.

АВЛ-деревья

В АВЛ-деревьях (Адельсон-Вельский, Ландис) оценка сложности не лучше, чем в совершенном дереве, но не хуже, чем в деревьях Фибоначчи для всех операций: поиск, исключение, занесение.

Определение. *АВЛ-деревом* (подравненным деревом) называется такое двоичное дерево, в котором для любой его вершины высоты левого и правого поддерева отличаются не более, чем на 1 (рис.22.2).

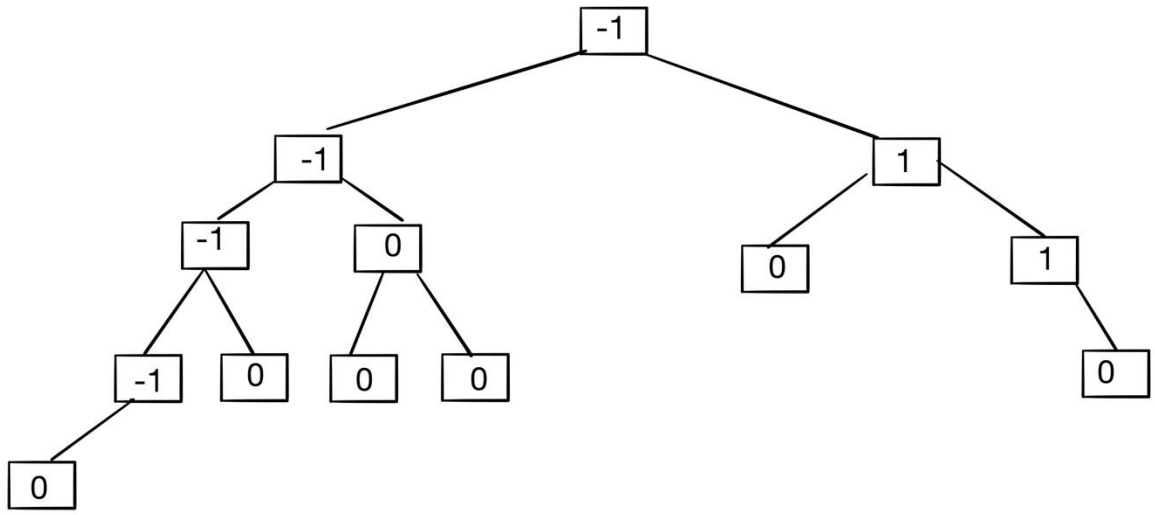


рис. 22.2 АВЛ-дерево

В узлах дерева записаны значения показателя сбалансированности (balance factor), равного разности высот левого и правого поддеревьев. Показатель сбалансированности может иметь одно из трех значений:
-1: Высота левого поддерева на 1 больше высоты правого поддерева.
0: Высоты обоих поддеревьев одинаковы.
+1: Высота правого поддерева на 1 больше высоты левого поддерева.

У совершенного дерева все узлы имеют показатель 0 (это самое «хорошее» АВЛ-дерево), а у дерева Фибоначчи все узлы имеют показатель баланса +1 (либо -1) (это самое «плохое» АВЛ-дерево).

Структура узла и операции в АВЛ-деревьях

Типичная структура узла и операции в АВЛ-деревьях выглядит так:

```
typedef struct avlone *avltree;
struct avlnode {
        key_tkey; //ключ
        avltree left;  //левое поддерево
        avltree right; // правое поддерево
        //int balance; показатель баланса
        int height;  // высота поддерева
};
 avltree makeempty(avltree t); // удалить дерево
 avltree find(key_t x, avltree t); //поиск по ключу
 avltree findmin(avltree t); // минимальный ключ
 avltree findmax(avltree t); // максимальный ключ
 avltree insert(key_t x, avltree t); // вставить узел
 avltree delete(key_t x, avltree t); // исключить узел
```

Показатель сбалансированности очень просто посчитать по высоте - одним вычитанием и 2 доступа к памяти. Поэтому хранят сразу высоты.
В принципе показатель сбалансированности (3 варианта), занимающий 2 бита, но непонятно, где его хранить, чтобы размер структуры сильно не возрос.
Поэтому лучше хранить разницу между текущей вершиной и сыном - там возможно только 2 варианта, которые нас интересуют и для них нужен 1 бит.

Рассмотрим теперь подробнее операции в АВЛ-деревьях.

Операция удаления дерева:

```
avltree makeempty(avltree t){
        if (t != NULL){
                makeempty(t -> left);
                makeempty(t -> right);
                free(t);
        }
        return NULL:
}
```

Если есть узел, который нужно удалить, нужно сначала очистить поддерево, а потом сам узел.

Операция поиска:

```
avltree find(key_t x, avltree t){
        if (t == NULL || x == t -> key)
          return t;
        if (x < t -> key)
          return t;
        if (x > t -> key)
          return find(x, t -> right);
}
```

АВЛ-дерево в первую очередь является двоичным деревом поиска, поэтому операция поиска для АВЛ-дерева просто использует свойства поиска двоичного дерева.

Операция поиска минимума:

```
avltree findmin(avltree t) {
        if (t == NULL)
          return NULL;
        else if (t -> left == NULL)
          return t;
        else
          return findmin(t -> left);
}
```

Операция поиска максимума:

```
avltree findmax(avltree t) {
        if (t != NULL)
```

```
        while (t -> right != NULL)
          t = t -> right;
        return t;
}
```

Включение узла в АВЛ-дерево

Рассматриваемое дерево состоит из корневой вершины r и левого (L) и правого (R) поддеревьев, имеющих высоты h_L и h_R соответственно. Для определенности будем считать, что новый ключ включается в поддерево L.

Если h_L не изменяется, то не изменяются и соотношения между h_L и h_R, и свойства АВЛ-дерева сохраняются.
Если h_L увеличивается на единицу, то возможны три случая:

- $h_L = h_R$, тогда после добавления вершины L и R станут разной высоты, но свойство сбалансированности сохранится;
- $h_L < h_R$, тогда после добавления новой вершины L и R станут равной высоты, то есть сбалансированность общего дерева даже улучшится;
- $h_L > h_R$, тогда после включения ключа сбалансированность нарушится, и потребуется перестройка дерева.

Новая вершина добавляется к левому поддереву поддерева L. В результате поддерево с корнем в узле B разбалансировалось: разность высот его левого и правого поддеревьев стала равной -2 (рис. 22.3).
Преобразование, разрешающее ситуацию - однократный поворот PR:
Делаем узел A корневым узлом поддерева, в результате правое поддерево с корнем в узле в «опускается» и разность высот становится равной 0 (рис. 22.4).

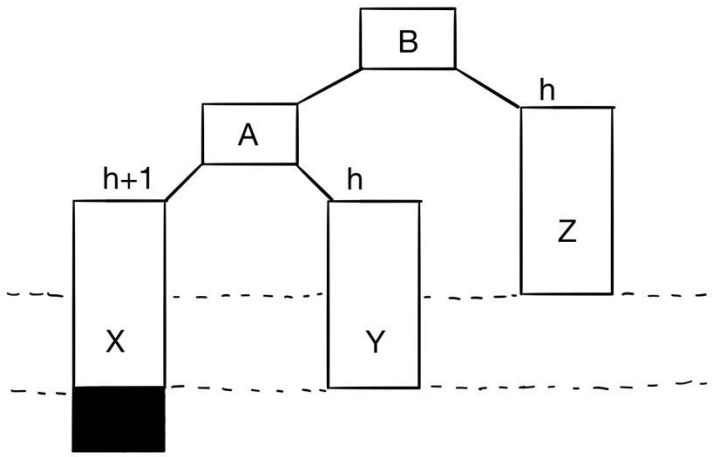


рис. 22.3 разбалансированное дерево

Размеры узлов A и B связаны соотношением A < B и A < Y. Таким образом, X < A < Y < B < Z - информация, которая должна остаться в перераспределенном дереве, чтобы двоичное дерево поиска осталось корректным. Так как мы добавляли новый узел в X, то он стал слишком большим и его надо перестроить. Чтобы перестроить, нужно эту картинку повернуть направо (однократный поворот).

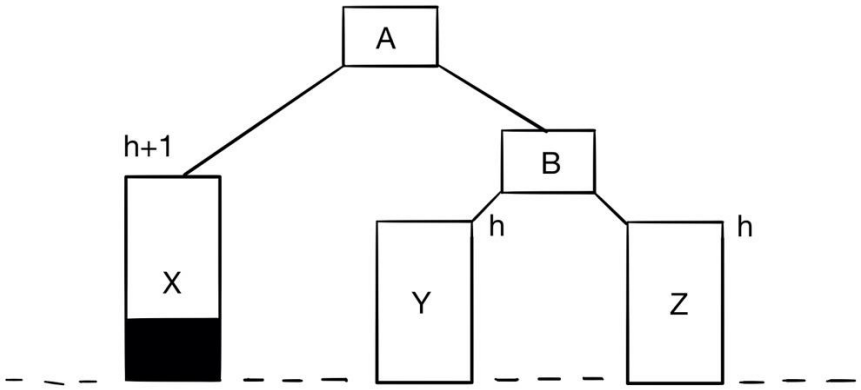


рис. 22.4 преобразованное дерево

Поддерево Y из правого сына A стало левым сыном B, а A стало корнем. Так как все, что в Y < B, к B можно подвесить Y слева. Таким образом в B получилась идеальная балансировка. Смысл поворота в том, чтобы вытаскивать длинные и неудобные поддеревья, а короткие опускать, тем самым балансируя дерево.

Рассмотрим случай, когда новая вершина добавляется к правому поддереву поддерева L. В результате поддерево с корнем в C разбалансировалось: разность высот его левого и правого поддеревьев стала равна -2 (рис. 22.5).

Преобразование, разрешающее ситуацию - двукратный поворот LR: «вытягиваем» узел B на самый верх, чтобы его поддеревья поднялись. Для этого сначала делаем левый поворот, меняя местами поддеревья с корневыми узлами A и B, а потом правый поворот, меняя местами поддеревья с корнями B и C (рис. 22.6).

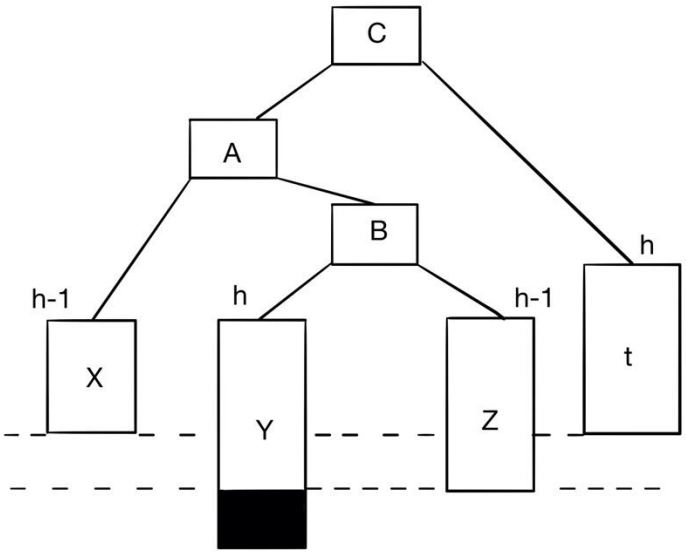


рис. 22.5 разбалансированное дерево после добавления к правому поддереву

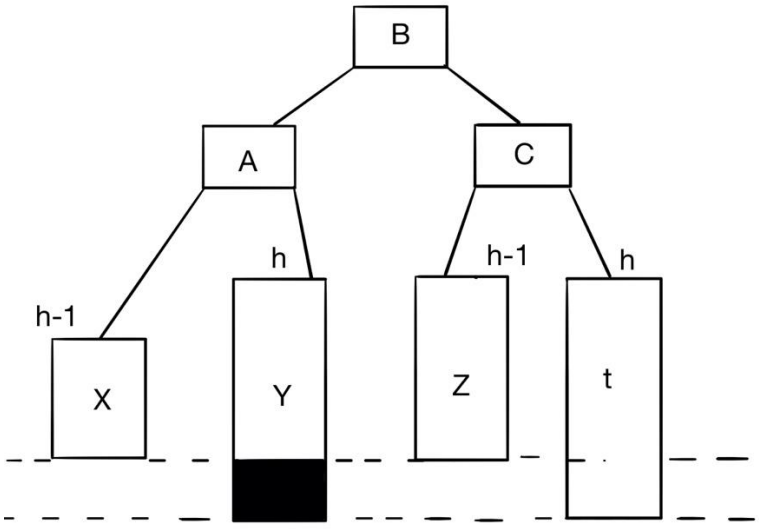


рис. 22.6 преобразованное дерево

В коде это все будет выглядеть так:

```
static inline int int height(avltree p){
        return p ? p -> height : 0;
}
```

Рассмотрим, что происходит с поворотом направо между узлом и его левым сыном.

Функция SingleRotateWithLeft вызывается только в том случае, когда у узла K2 есть левый сын. Функция выполняет поворот между узлом (K2) и его левым сыном, корректирует высоты поддеревьев, после чего возвращает новый корень.

```
static avltree SingleRotateWithLeft(avltree K2){
        avltree K1;
        /* выполнение поворота */
        k1 = k2 -> left;
        k2 -> left = k1 -> right; /* k1 != NULL */
        k1 -> right = k2;
        /*корректировка высот поставленных узлов */
        k2 -> height = max(height(k2 -> left),
                        height(k2 -> right)) + 1;
   k1 -> height = max(height(k1 -> left), k2 -> height)+1;
   return k1; /*новый корень*/
}
```

k2 -> left = k1 -> right - левый сын K2 берется из правого сына левого сына. Теперь нужно правого сына K1, который раньше был непонятно чем, сделать K2. Чтобы сделать K2 правым сыном K1 пишем k1 -> right = k2;
Это и есть поворот. Для корректировки высот новый корень нужно поставить на место, где раньше был K2.
Функция max пересчитывает высоты от листьев к корням, то есть сначала для K2, потом для K1.

Полностью симметричная функция SingleRotateWithRight делает все то же самое, только между узлом и его правым сыном. Она вызывается только в том случае, когда у узла K1 есть правый сын. Функция выполняет поворот между узлом K1 и его правым сыном, корректирует высоты поддеревьев, после чего возвращает новый корень.

```
static avltree SingleRotateWithRight(avltree K1){
        avltree K2;
        /* выполнение поворота */
        k2 = k1 -> right;
        k1 -> right = k2 -> left;
        k2 -> left = k1;
        /*корректировка высот поставленных узлов */
        k1 -> height = max(height(k1 -> left),
                        height(k1 -> right)) + 1;
   k2 -> height = max(height(k2 -> right), k1 -> height)+1;
   return k1; /*новый корень*/
```

```
}
```

Когда нужен двойной поворот, пользуемся функциями, которые делают два одинарных поворота. Функция для LR-поворота выполняется только тогда, когда у узла К3 есть левый сын, а у левого сына К3 есть правый сын. Функция выполняет двойной поворот LR, корректирует высоты, после чего возвращает новый корень.

```
static avltree DoubleRotateWithLeft(avltree k3){
        /*поворот между k1 и k2 */
        k3 -> left = SingleRotateWithRight(k3 -> left);
        /* поворот между k3 и k2 */
        return SingleRotateWithLeft(k3);
}
```

Вставка нового узла в АВЛ-дерево

Функция вставки должна вставить новый узел в двоичное дерево поиска так, чтобы все его свойства сохранились. При вставке могут нарушиться свойства АВЛ-дерева, тогда необходимо, возвращаясь обратно, постепенно эти свойства восстанавливать.
Функция avltree insert рекурсивна.

Если дерево было пустым, создаем узел. Если дерево не NULL, то то, что лежало в дереве, теперь должно перенестись в левое поддерево (симметричный случай в правое поддерево). После вставки обработаны все свойства двоичного дерева. Но могли быть нарушены свойства АВЛ-дерева. В случае, когда высота левого поддерева стала превосходить высоту правого на 2, нужна балансировка.

```
avltree insert(key_t x, avltree t) {
        if (t == NULL) {
                /*создание дерева с одним узлом */
                t = malloc(sizeof(struct avltree));
                if (!t)
                  abort();
                t -> key = x;
                t -> height =1;
                t -> left = t -> right = NULL;
        }
        else if (x < t -> key){
                t -> left = insert(x, t -> left);
```

```
            if(height(t -> left) - height(t -> right) == 2) {
                if (x < t -> left -> key)
                  t = SingleRotateWithLeft(t);
                else
                  t =DoubleRotateWithLeft(t);

            }
        }
        else if (x > t -> key){
            t -> right = insert(x, t -> right);
            if (height(t -> right) - height(t -> left) == 2){
                if (x > t -> right -> key)
                  t = SingleRotateWithRight(t);
                else
                  t = DoubleRotateWithRight(t);
            }
        }
        /* иначе x уже в дереве */
        t -> height = max(height(t -> left), height(t -> right))+1;
        return t;
}
```

Если на пути вверх находим точку, в которой нарушилась балансировка, и проводим необходимые повороты, то на пути наверх это может произойти только 1 раз. То есть если мы один раз зайдем в эти if, то больше уже в них не зайдем. Поэтому вставка в АВЛ-деревья требует всего 1 или 2 поворота.

Рассмотрим пример (рис. 22.7) последовательной вставки целых чисел 4, 5, 7, 2, 1, 3, 6.

Удаление узла из АВЛ-дерева требует балансировки дерева. Иными словами, в конец функции, выполняющей удаления узла, необходимо добавить вызовы функций:

- SingleRotateWithRight(T),
- SingleRotateWithLeft(T),
- DoubleRotateWithRight(T),
- DoubleRotateWithLeft(T).

Возможны случаи вращения, не встречавшиеся при вставке. Может оказаться необходимым выполнить несколько вращений.

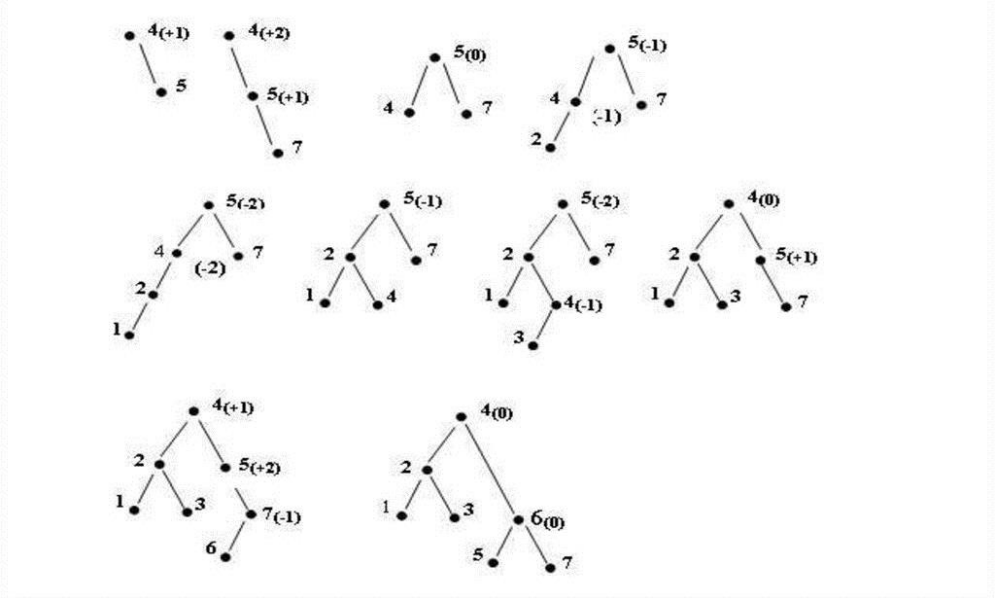


рис. 22.7 пример вставки

Лекция 23. Красно-черные деревья и самоперестраивающиеся деревья

Красно-черное дерево

Красно-черное дерево - двоичное дерево поиска, каждая вершина которого окрашена либо в красный, либо в черный цвет.
У него есть следующие поля: цвет, дети, родители.

```
typedef struct rbtree{
        int key;
        char color;
        struct rbtree *left, *right, *parent;
} rbtree, *prbtree;
```

Будем считать, что если left или right равны NULL, то это «указатели» на фиктивные листы, то есть все вершины внутренние.

Красно-черное дерево (рис. 23.1) обладает следующими свойствами:

1) Каждая вершина либо красная, либо черная;
2) Каждый лист (фиктивный) - черный;
3) Если вершина красная, то оба ее сына - черные;
4) Все пути, идущие от корня к любому листу, содержат одинаковое количество черных вершин.

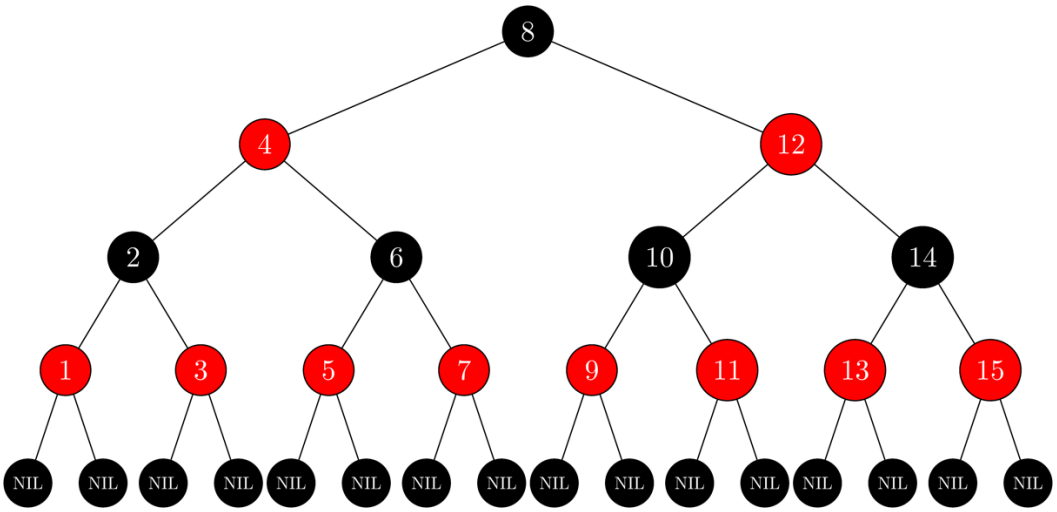


рис. 23.1 красно-черное дерево

Из пункта 2 и 3 следует, что красных вершин может быть не больше половины всех вершин.

Обозначим bh(x) - *«черную» высоту* поддерева с корнем x (саму вершину в число не включаем), то есть количество черных вершин от x до листа.
Черная высота дерева - черная высота его корня.

Лемма 1. Красно-черное дерево с n внутренними вершинами (без фиктивных листьев) имеет высоту не более $2log_2(n+1)$
Доказательство. Покажем вначале, что поддерево x содержит не меньше $2^{bh(x)}-1$ внутренних вершин. По индукции:

1) Для листьев bh = 0, то есть $2^{bh(x)}-1=2^0-1=0$.
2) Пусть теперь x не лист и имеет черную высоту k.
   Тогда каждый сын x имеет черную высоту не меньше k - 1 (красный сын имеет высоту k, черный имеет высоту k - 1).
3) По предположению индукции каждый сын имеет не меньше $2^{k-1}-1$ вершин.
   Поэтому поддерево x имеет не меньше $2^{k-1}-1+1=\ 2^k-1$ вершин.
4) Теперь пусть высота дерева равна h.
5) По свойству 3 черные вершины составляют не меньше половины всех вершин на пути от корня к листу.
   Поэтому черная высота древа bh ≥ h/2.
6) Тогда $n\ \geq 2^{\frac{h}{2}}-1$ и $h\ \leq 2log_2(n+1)$. Лемма доказана.

Следовательно, поиск по красно-черному дереву имеет сложность O($log_2 n$).

Красно-черные деревья: вставка вершины

- Сначала мы используем обычную процедуру занесения новой вершины в двоичное дерево поиска.
- Красим новую вершину в красный цвет.
- Если дерево было пустым, красим в черный цвет.
- Свойство 4 при вставке изначально не нарушено, так как новая вершина красная.
- Если родитель новой вершины черный (новая - красная), то свойство 3 также не нарушено, иначе (родитель красный) свойство 3 нарушено.

Случай 1: «дядя» (второй сын родителя родителя текущей вершины) тоже красный (как текущая вершина и родитель) (рис. 23.2).

Решение: возможно выполнить перекраску: родителя и дядю (вершина A и D соответственно) в черный цвет. В таком случае свойство 4 не нарушено (черные высот поддеревьев совпадают).

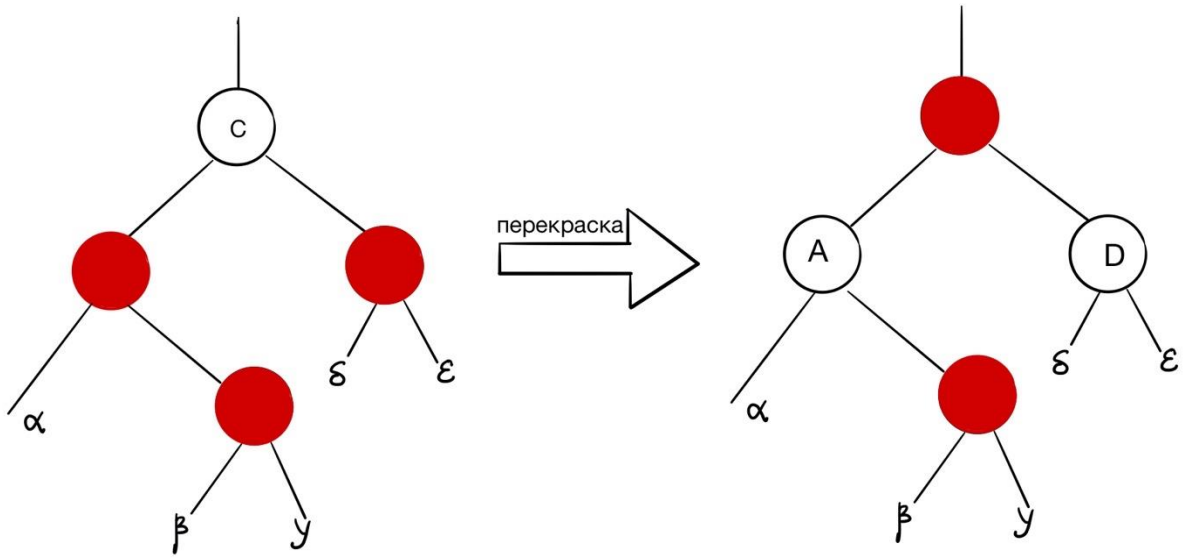


рис. 23.2 перекраска случай 1.

Случай 2: «дядя» (второй сын родителя родителей текущей вершины) черный.
Решение:

1) Необходимо выполнить левый поворот родителя текущей вершины (вершины A) (рис. 23.3)
2) Необходимо выполнить правый поворот вершины C (рис. 23.4)
3) Перекрасить вершины B и C.
   Теперь все поддеревья имеют черные корни и одинаковую высоту, поэтому свойства 3 и 4 верны.

Есть еще 4 случая, которые мы рассматривать не будем.

Таким образом, когда у нас есть корректная черная высота (для этого мы красим вершины в красный цвет и используем остальные признаки), может быть нарушено только чередование цветов, которое можно вытянуть либо перекраской, как в 1 случае, либо, покрутив дерево в окрестности той вершины, где все нарушилось, перекрасить, как в случае 2.

Как в случае вставки, так и в случае удаления, у красных вершин константное количество оборотов (нет такой проблемы, как у АВЛ-деревьев, что может потребоваться количество поворотов, сравнимое с высотой).

Обычно красно-черные деревья лежат в основе стандартных структур для словарных отображений в популярных языках программирования.

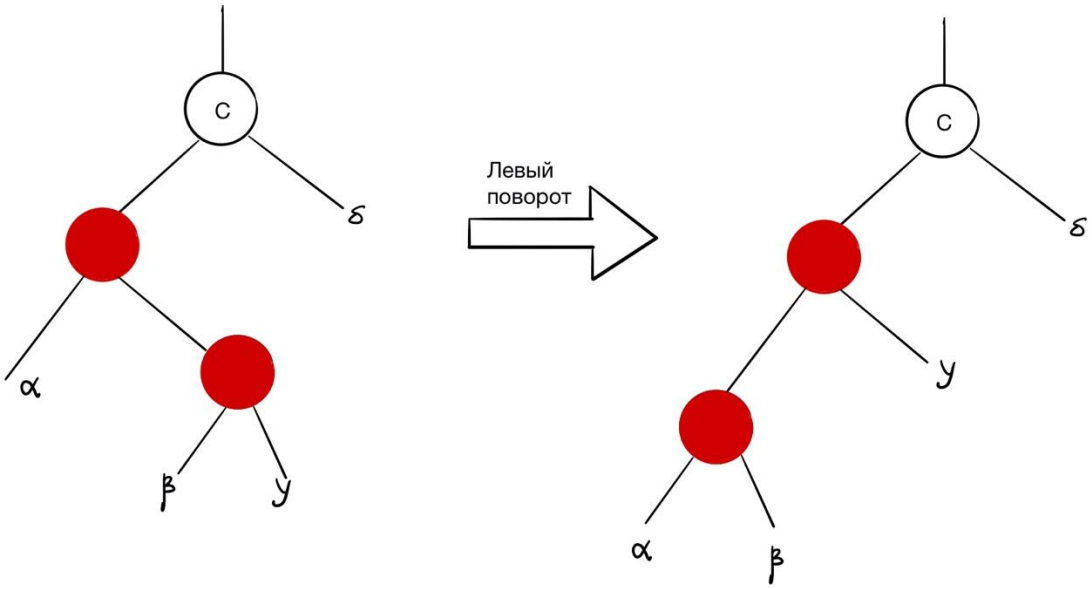


рис. 23.3 случай 2: шаг 1

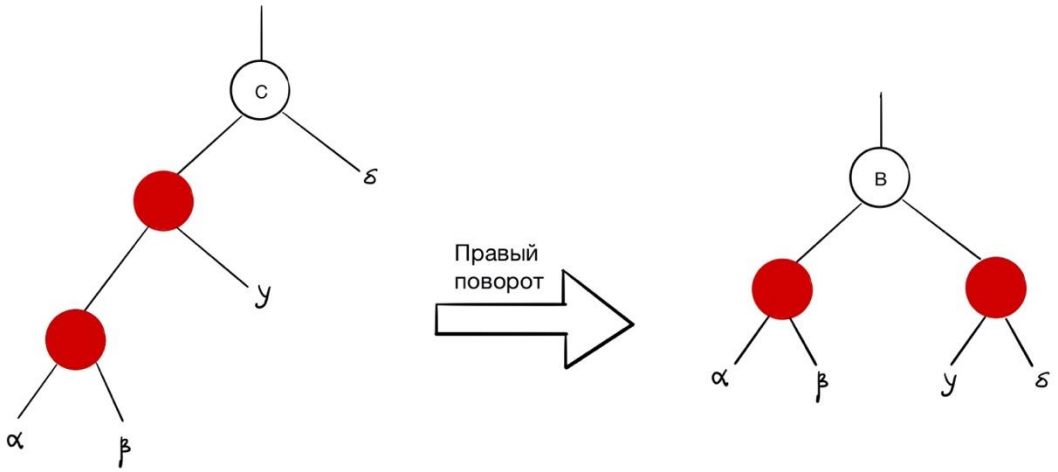


рис. 23.4 случай 2: шаг 2-3

Есть еще 4 случая, которые мы рассматривать не будем.

Самоперестраивающиеся деревья (splay trees)

Некоторые свойства самоперестраивающихся деревьев:

- Это двоичное дерево поиска, не содержащее дополнительных служебных полей в структуре данных (нет баланса, цвета и т.п.).
- Гарантируется не логарифмическая сложность в худшем случае, а *амортизированная* логарифмическая сложность:
  - Любая последовательность из m словарных операций (поиска, вставки, удаления) над n элементами, начиная с пустого дерева, имеет сложность O(m log n);
  - Средняя сложность одной операции O(log n)
  - Некоторые операции могут иметь сложность $\ominus$(n)
  - Не делается предположений о распределении вероятностей ключей дерева и словарных операций (то есть что некоторые операции выполняются чаще других).

Хорошее описание этих деревьев можно найти в:
Harry R. Lewis, Larry Denenberg. Data structures and Their Algorithms. HarperCollins, 1991. Глава 7.3.

Идея самоперестраивающихся деревьев заключается в эвристике Move-to-Front:

> При поиске элемента в списке будем перемещать найденный элемент в начало списка. Так, если он понадобится снова, найдется быстрее.

Рассмотрим Move-To-Front двоичного дерева, который реализуется операцией Splay(K, T) (подравнивание, перемешивание, расширение):

- После выполнения операции Splay дерево T перестраивается, оставаясь деревом поиска так, что
  - Если ключ K есть в дереве, то он становится корнем
  - Если ключа K нет в дереве, то в корне оказывается его предшественник или последователь в симметричном порядке обхода.

Реализация словарных операций через splay

Поиск Look Up - выполнение операции Splay(K, T) и проверка значения ключа в корне:

- Если значение равно K, то ключ найден (рис. 23.5)
- Если ключа K нет в корне, то ключа нет в дереве

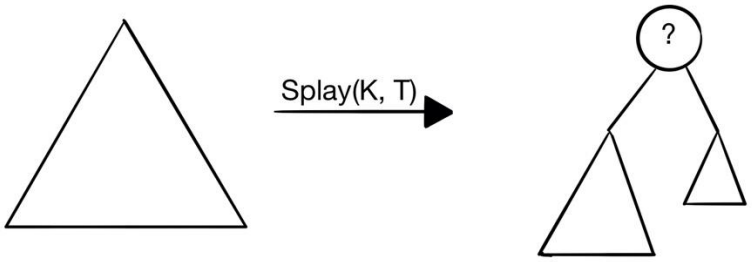


рис. 23.5 поиск ключа, проверка корня

Вставка (Insert) реализуется через выполнение операции Splay(K, T) и проверку значения ключа в корне:

- Если значение уже равно К, то обновим данные ключа
- Если значение другое, то вставим новый корень К и поместим старый корень J слева или справа в зависимости от значения J (рис. 23.6).

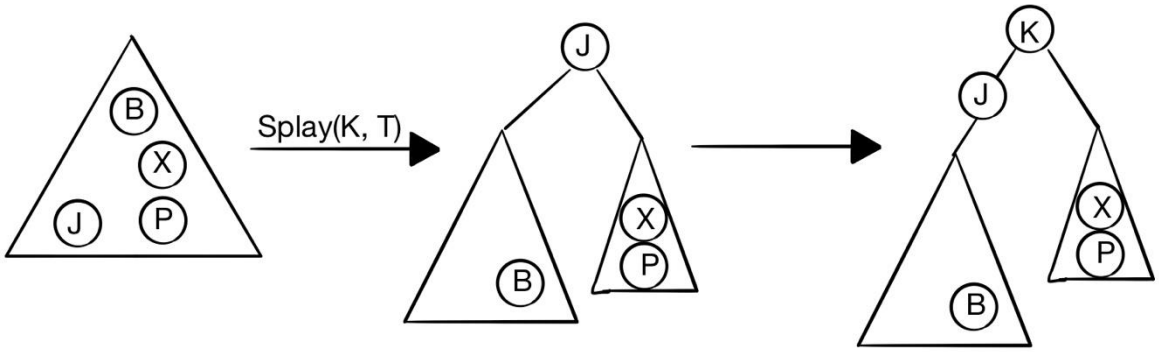


рис. 23.6 вставка ключа

Операция Concat(T_1, T_2) - слияние деревьев поиска T_1 и T_2 таких, что все ключи в дереве T_1 меньше, чем **все** ключи в дереве T_2 в одно дерева поиска.
Операция слияние реализуется путем выполнения операции Splay(+∞, T_1) со знанием ключа, заведомо больше любого другого в T_1.
После Splay(+∞, T_1) у корня дерева T_1 нет правого сына, присоединим дерево T_2 как правый сын корня T_1 (рис. 23.7).

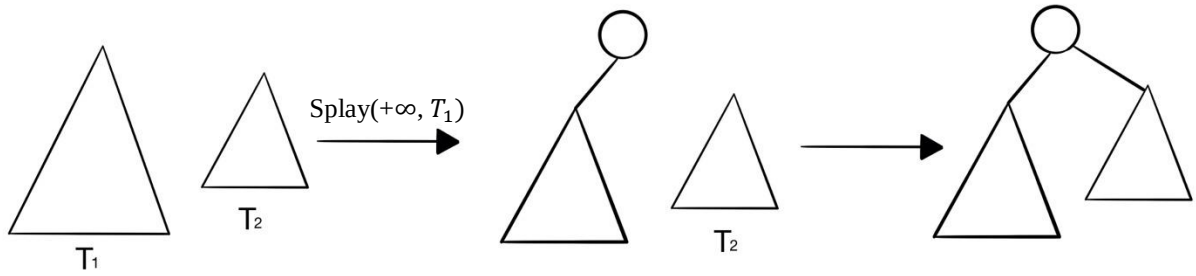


рис. 23.7 операция слияния

Операция удаления (delete) реализуется путем выполнения операции Splay(K, T) и проверки значения ключа в корне:

- Если значение **не равно** K, то ключа в дереве нет и удалять там нечего
- Иначе (ключ был найден) выполним операцию Concat над левым и правым сыновьями корня, а корень удалим (рис. 23.8)

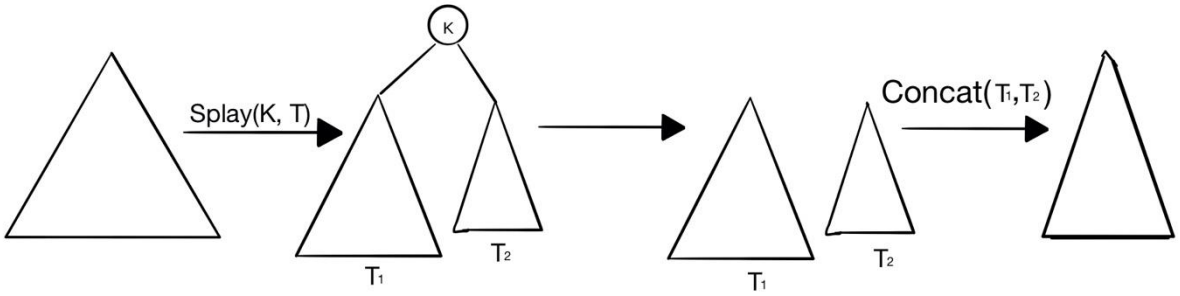


рис. 23.8 операция удаления

Реализация операции splay

1) Ищем ключ K в дереве обычным способом, запоминая пройденный путь по дереву.
   Может потребоваться память, линейная от количества узлов дерева.
   Для уменьшения количества памяти можно воспользоваться *инверсией ссылок* (link inversion), то есть перенаправить указатели на сына назад на родителя вдоль пути по дереву плюс 1 бит на обозначение направления.

2) Получаем указатель P на узел дерева либо с ключом K, либо с его соседом в симметричном порядке обхода, на котором закончился поиск (сосед имеет единственного сына).
3) Возвращаемся назад вдоль запомненного пути, перемещая узел P к корню (узел P будет новым корнем).
   А) Если отец узла P - корень дерева (или у P нет деда), то выполняем однократный поворот направо или налево (рис. 23.9)

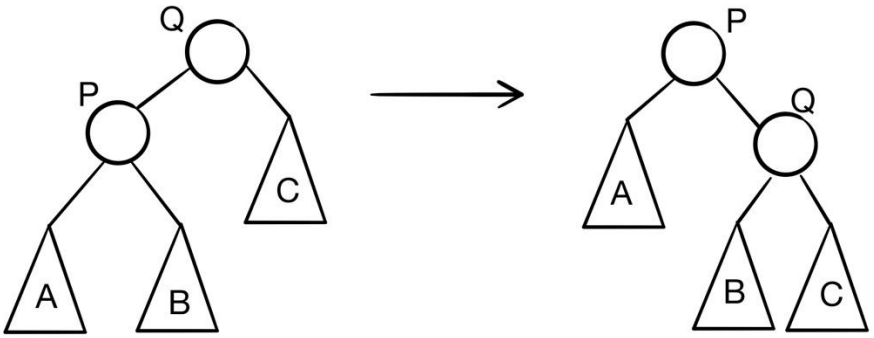


рис. 23.9 однократный поворот

Б) Если узел P и отец узла P оба левые или правые дети, то выполняем сразу два однократных поворота направо (налево), сначала вокруг деде P, потом вокруг отца P (рис. 23.10).

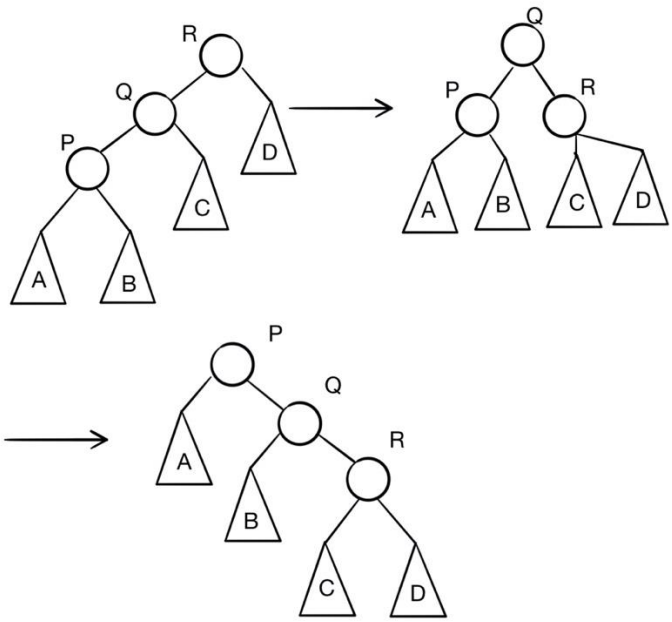


рис. 23.10 два однократных поворота

В) Если отец узла P -правый сын, а P - левый сын (или наоборот), то выполняем два однократных поворота в противоположных направлениях (сначала вокруг отца P, потом вокруг деда P налево) (рис. 23.11).

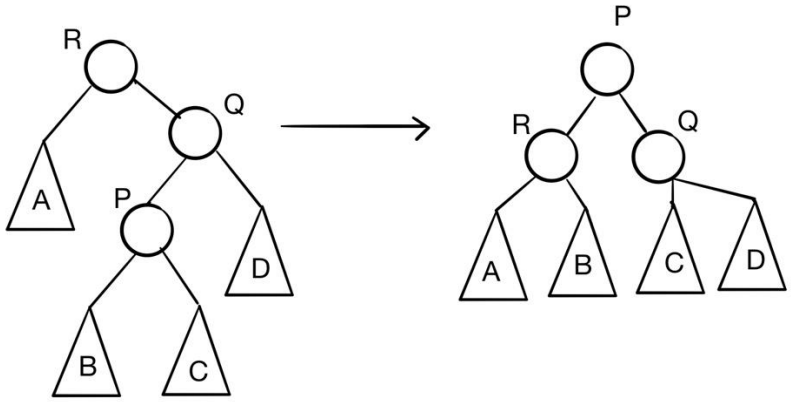


рис. 23.11 два однократных поворота в противоположные направления

Рассмотрим пример операции Slay над узлом D.
На рис. 23.12 представлен случай Б, когда отец узла D (E) и сам узел D -оба левые сыновья.

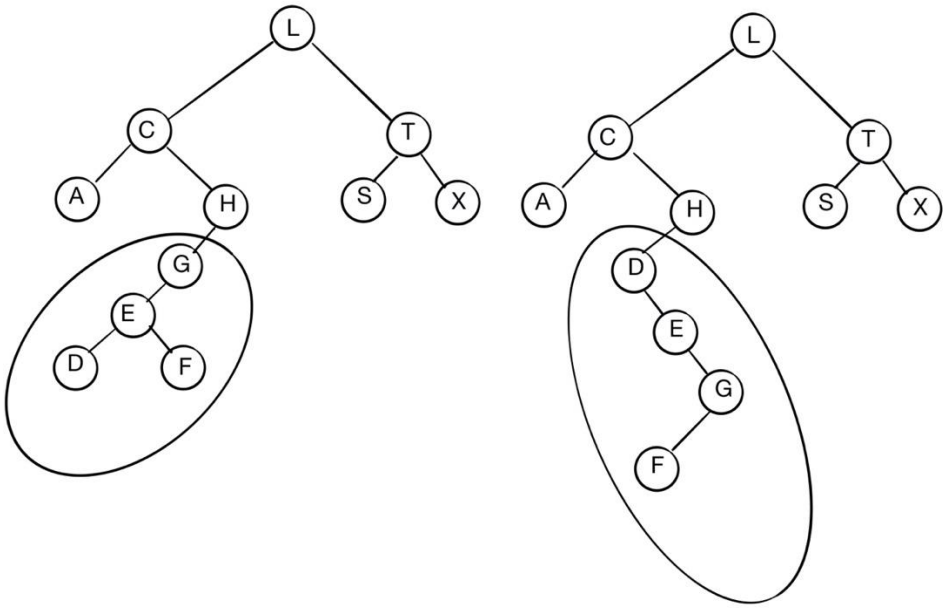


рис. 23.12 пример операции splay, случай Б

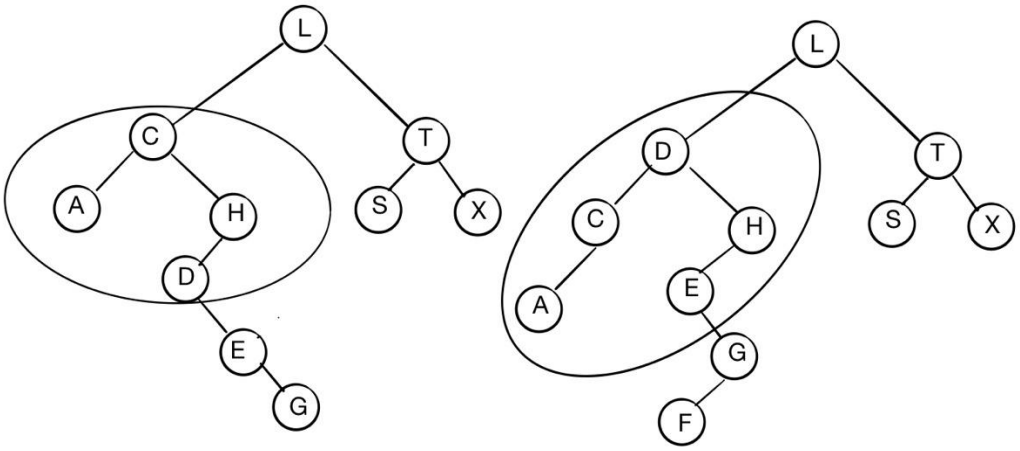


рис. 23.13 пример операции splay, случай B

Надо понимать, что для того, чтобы реализовать функцию splay нужно где-то явно сохранить путь от корня вниз (массив, рекурсивная функция или инверсия ссылок).

Сложность операции splay

Пусть каждый узел дерева содержит некоторую сумму денег.

- *Весом* узла является количество его потомков, включая сам узел
- *Рангом* узла r(N) называют логарифм его веса
- *Денежный инвариант:* во время всех операций с деревом каждый узел содержит r(N) рублей
- Каждая операция с деревом стоит фиксированную сумму за единицу времени

Лемма. Операция splay требует *инвестирования* не более, чем $3[\lg n] + 1$ рублей с **сохранением денежного инварианта.**

Теорема. Любая последовательность из m словарных операций на самоперестраивающемся дереве, которое было изначально пусто и на каждом шаге содержало не более n узлов, занимает не более $O(m \log n)$ времени.
При этом:

- Каждая операция требует не более $O(\log n)$ инвестиций, при этом может использовать деньги узла;
- По лемме инвестируется всего не более $m(3[\lg n] + 1)$ рублей.
  Сначала дерево содержит 0 рублей, в конце содержит ≥ 0 рублей - $O(m \log n)$.

Сбалансированные деревья: обобщение через ранги

Haeupler, Sen, Tarjan. Rank-balanced trees. ACM Transactions on Algorithms, 2015.

Обобщение разных видов сбалансированных деревьев через понятие ранга (rank) и ранговой разницы (rank difference): АВЛ, красно-черные деревья, 2-3 деревья, B-деревья. Также рассмотрен новый вид деревьев: слабые АВЛ-деревья (weak AVL), анализ слабых АВЛ-деревьев, анализ потенциалов.

Лекция 24. Сбалансированные деревья и хеш-функции

Сбалансированные деревья: понятие ранга

Ранг — это формализм, через который можно выразить все необходимые ограничения на деревья.

Ранг (rank) вершины r(x): неотрицательное число.
Ранг отсутствующей (null) вершины равен -1.
Ранг дерева — это ранг корня дерева.
Ранговая разница (rank difference) — это число r(p(x)) - r(x), если у вершины x есть родитель p(x). При этом у корня дерева нет ранговой разницы.

i-сын - вершина с ранговой разницей, равной i;
ij -вершина - это вершина, у которой левый сын - это i-сын, а правый сын - это j-сын. Один или оба сына могут отсутствовать. ij и ji вершины не различаются.

Конкретный вид сбалансированного дерева определяется *рангом* и *ранговым правилом*.
Ранговое правило должно гарантировать, что:

- Высота дерева (h) превосходит его ранг не более чем в константное выражение раз (плюс, возможно, O(1)).
- Ранг вершины (k) превосходит логарифм ее размера (n) не более чем в константное количество раз (плюс, возможно, O(1)).
  Размер вершины (n) - число ее потомков, включая себя, то есть размер поддерева с корнем в этой вершине.
- $h = O(k)$
  $k = O(\log n) \ \rightarrow h = O(\log n)$

У совершенного дерева ранг - его высота, а все вершины — 1, 1.

Сбалансированные деревья: ранговые правила

АВЛ-правило: каждая вершина 1,1 или 1,2.
Ранг -высота дерева или все ранги положительны, каждая вершина имеет хотя бы одного 1-сына. В таком случае можно хранить один бит, указывающий на ранговую разницу вершины.

Красно-черное правило: ранговая разница любой вершины равна 0 или 1, при этом родитель 0-сына не может быть 0-сыном.

0-сын - красная вершина; 1-сын черная вершина.

Ранг -черная высота. При этом корень не имеет цвета, так как не имеет ранговой разницы!

Слабое АВЛ-правило: ранговая разница любой вершины равна 1 или 2; все листья имеют ранг 0.
Вдобавок к АВЛ-деревьям разрешаются 2,2 - вершины.
Выделяется бит на узел для ранговой разницы или ее четности.
Для балансировки требуется не более двух поворотов и O(log n) изменений ранга для вставки/удаления, при этом амортизированно - лишь O(1) изменений.
Слабое АВЛ-дерево — это красно-черное дерево.

Пирамидальная сортировка: пирамида (двоичная куча)

Будем рассматривать массив *a* как двоичное дерево:

- элемент a[i] является узлом дерева;
- элемент a[i/2] является родителем узла a[i];
- элемент a[2*i] и a[2*i+1] являются детьми узла a[i].

Для всех элементов пирамиды выполняется соотношение (основное свойство кучи):

a[i] >= a[2*i] и a[i] >= a[2*i+1]
или
a[i/2] <= a[i]
Сравнение может быть как в большую, так и в меньшую сторону.

Замечание: определение предполагает нумерацию элементов массива от 1 до n.
Для нумерации от 0 до n-1:
a[i] >= a[2*i+1] и a[i] >= a[2*i+2].

Дерево довольно просто разложить в массив (рис. 24.1). Массив будем нумеровать с 1, как обычно нумеруют ключи, в коде нужно будет просто добавить -1, так как в Си массивы начинаются с 0. 5 записываем в первую ячейку массива, i = 1. Его дети будут идти под номером 2*i и 2*i+1, то есть 2 и 3. Соответственно в a[2] записываем 7, в a[5] 9. Далее у 7 (i = 2) детей нет, поэтому под номером 4 и 5 ничего нет, запишем NULL, аналогичным образом заполняются ячейки 6 и 7.

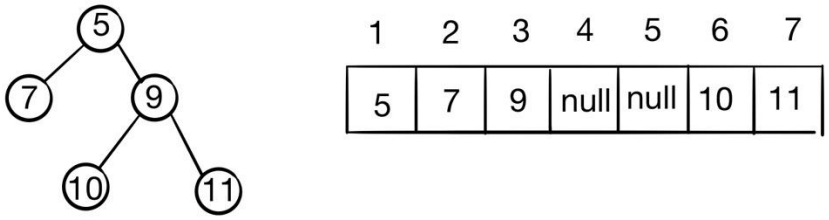


рис. 24.1 пример дерева

Рассмотрим теперь пример реальной пирамиды на рис. 24.2:

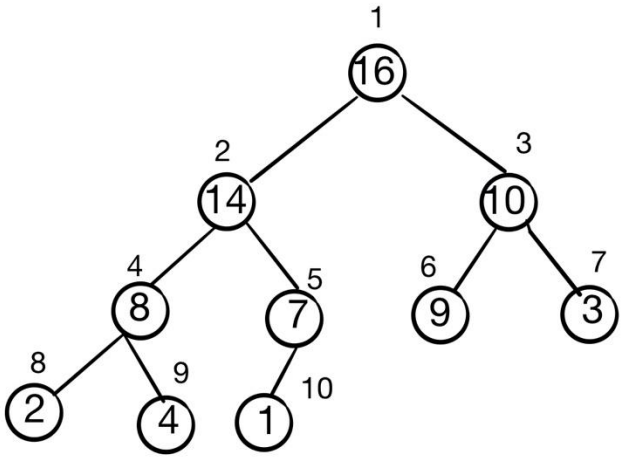


рис. 24.2 пример пирамиды

Здесь выполняется условие, что корень больше обоих своих сыновей.

Пирамидальная сортировка: просеивание элемента

Чтобы добавить элемент в уже *существующую* пирамиду, воспользуемся алгоритмом:

1. Поместим новый элемент в корень пирамиды
2. Если этот элемент меньше одного из сыновей:
   Элемент меньше наибольшего сына, обменяем элемент с наибольшим сыном, что позволит сохранить свойство пирамиды для другого сына.
3. Повторим процедуру для обмененного сына.

Код, реализующий этот алгоритм, очень простой:

```
static void sift(int *a, int l, int r) {
```

```
        int i, j, x;

        i = l; j = 2*l; x = a[l];
        /* j указывает на наибольшего сына */
        if (j < r && a[j]< a[j+1])
          j++;
        /*  i указывает на отца */
        while (j <= r && x < a[j]){
                /* обмен с наибольшим сыном: a[i] == x */
                a[i] = a[j];
                a[j] = x;
                /*продвижение индексов к следующему сыну */
                i = j;
                j = 2*j;
                /* выбор наибольшего сына */
                if (j < r && a[j] < a[j+1])
                  j++;
        }
}
```

В цикле i - индекс массива, который указывает на родителя (текущую обрабатываемую вершину), j - индекс, который указывает на наибольшего сына. Нужна итерация цикла для того, чтобы просеять элемент вниз, на нужное место.
Для начала нужно установить условия инварианта. Первый родитель - i = l, x - значение текущего родительского элемента. j должен указывать на наибольшего сына, поэтому j = 2*l. Далее идет непосредственно просеивание элемента.

Рассмотрим пример работы алгоритма sift(2,10) для левого поддерева (рис. 24.3):

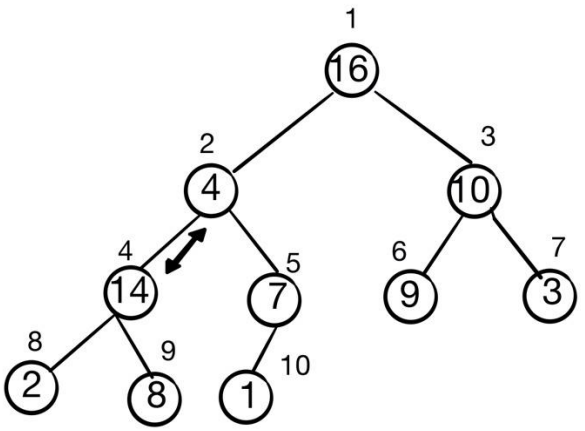


рис. 24.3 sift(2,10), первый обмен

Нужно выбрать из двух сыновей второго узла (4) наибольший: 14 или 7 и поменять местами узлы. Таким образом 14 и 4 меняются местами. В правой ветке все хорошо: 14 > 7 > 1, так как такой обмен изначально гарантировал, что в соседней ветви все останется корректно - мы поставили наверх наибольшего сына. Далее (рис. 24.4) свойство дерева нарушаются, нужно поменять 4 и 8 местами.

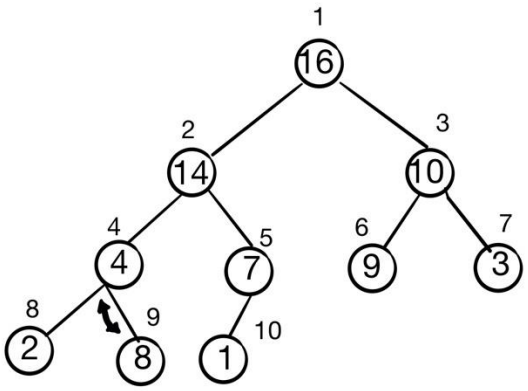


рис. 24.4 sift(2,10), второй обмен

Теперь мы спустились вниз, i = 9, j = 18. Так как 18 > 10 ничего делать не надо, так как мы вышли за границу r = 10, то есть находимся в листовой вершине, у которой нет детей и ограничений. В вершинах с 6 до 10 свойства пирамиды сохраняются просто потому, что его не нужно обеспечивать (детей нет). После добавления нового элемента область правильной пирамиды увеличатся на 1, то есть становится от 5 до 10. И так идя от n/2 -го элемента к началу, по очереди добавляя элемент к хвосту, получим за n/2 шагов правильную пирамиду.

Это первый шаг пирамидальной сортировки. Весь алгоритм пирамидальной сортировки заключается в следующем:

(1) Построим пирамиду по сортируемому массиву
Элементы массива от n/2 до n являются листьями дерева, а следовательно, правильными пирамидами из одного элемента. Для остальных элементов в порядке уменьшения индекса просеиваем их через правую часть массива.
(2) Отсортируем массив по пирамиде
Первый элемент массива максимален (корень пирамиды).
Поменяем первый элемент с последним (таким образом первый элемент отсортирован).
Теперь для первого элемента свойство кучи нарушено: повторим просеивание первого элемента в пирамиде от первого до предпоследнего.

Снова поменяем первый и предпоследний элемент и т. п.

Программа пирамидальной сортировки:

```
void heapsort(int *a, int n){
   int i, x;
/* Построим пирамиду по сортируемому массиву*/
/* Элементы нумеруются с 0 -> идем n/2-1 */
for (i = n/2 -1; i >= 0; i --)
  sift(a, i, n-1);
for (i = n-1; i>0; i--) {
   /*текущий максимальный элемент в конец */
   x = a[0]; a[0] = a[i]; a[i] =x;
   /* восстановим пирамиду в оставшемся массиве */
   sift(a, 0, i-1);
 }
}
```

Рассмотрим пример пирамидальной сортировки (рис. 24.5):
На рисунке изображена правильная пирамида. Жирной стрелкой показано, как наибольший элемент уходит вниз, а тонкими - как sift происходит по тому элементу, который мы подняли. Первая итерация цикла: 16 - самый большой элемент массива, ставим его на место 1, а 1 в корень. Смысла sift в том, что элемент всегда спускается в сторону наибольшего сына. Поэтому получится картина как на рис. 24.6.
16 - элемент, отмеченный зеленым — это отсортированный элемент, который мы больше не трогаем. Теперь мы идем по правой ветви и снова меняем элементы, как показано тонкими стрелочками.

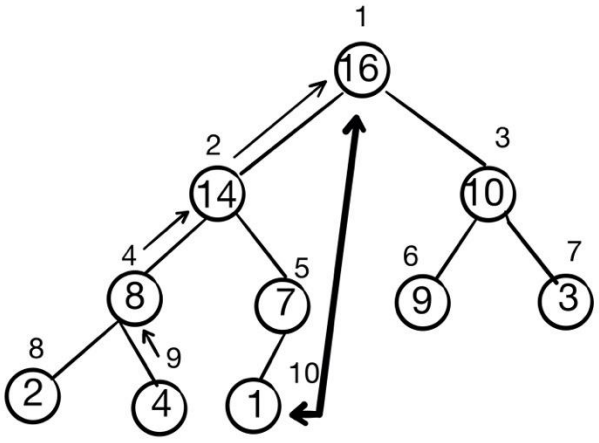


рис. 24.5 первая итерация алгоритма

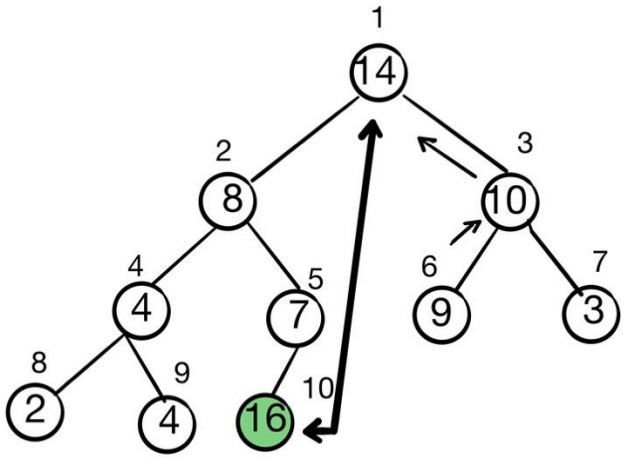


рис. 24.6 последующая итерация

В итоге получается 2 отсортированных элементы, и мы продолжаем работу с элементами с индексами от 1 до 8 (рис. 24.7). После этого у нас получается уже 3 отсортированных элемента (рис. 24.8). Дальше 8 становится корневым элементом (рис. 24.9) и так далее.

Таким образом:

- Сложность этапа построения пирамиды есть O(n)
- Сложность этапа сортировки есть O(n log n)
- Сложность в худшем случае O(n log n)
- Среднее количество обменов - n/2*log n

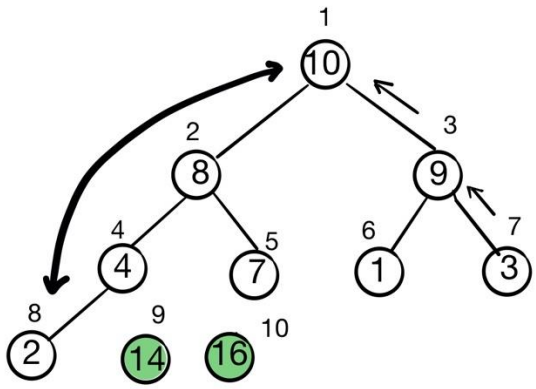


рис. 24.7

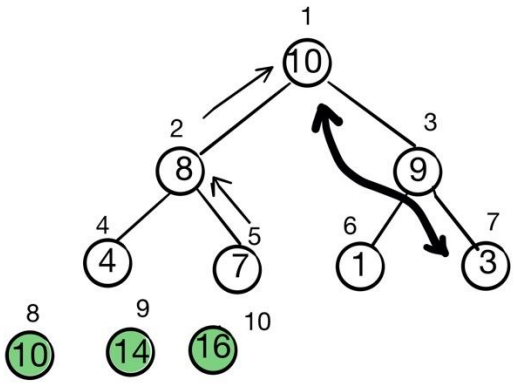


рис. 24.8

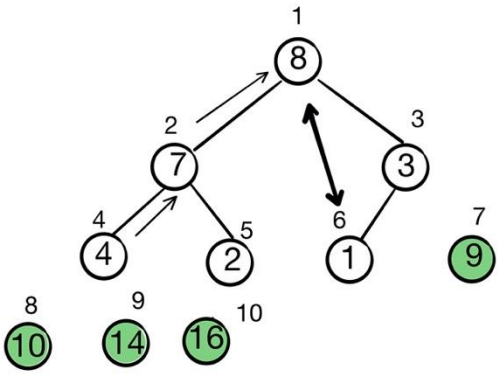


рис. 24.9

Хеш-таблицы

Организуется таблица ключей: массив index[m] длины m, элементы которого содержат значение ключа и указатель на данные (информацию), соответствующие этому ключу. При этом *прямая адресация* применяется, когда количество возможных ключей невелико: например, ключи пронумерованы целыми числами из множества U = {0, 1, 2,…, m-1}, где m не очень большое число. В случае прямой адресации ключ с номером k соответствует элементу index[k]. Этот ключ обычно не записывается в элемент массива, так как совпадает с индексом. Все три словарные операции (добавление, поиск и удаление элементов по их ключам) выполняются за время порядка O(1). Основной недостаток прямой адресации - таблица index занимает слишком много места, если множество всевозможных ключей U достаточно велико (m - большое целое число).

Хеширование тоже позволяет обеспечить среднее время операций с данными $T_{ср}(n) = O(1)$ и тоже за счет использования таблицы index.

Хэш-таблица использует память объемом $\Theta(|K|)$, где |K| - мощность множества использованных ключей (правда, это оценка в среднем, а не в худшем случае, да и то при определенных предположениях).
В случае хеш-адресации элементу с ключом key отводится строка таблицы с номером hash(key), где hash: U → {0, 1, 2, …, m-1} - хеш-функция.
Число hash(key) называется *хеш-значением* ключа key.
Если хеш-значения ключей key_1 и key_2 совпадают, то есть hash(key_1) == hash(key_2), говорят, что случилась *коллизия.*
Выбрать хеш-функцию, для которой коллизии исключены, возможно лишь тогда, когда все возможные значения ключей заранее известны. В общем же случае коллизии неизбежны, так как |U| > m.
Есть два основных способа разрешения коллизии: прямая адресации и хеширование цепочками.

Первая мысль, которая приходит - все, что попадет в одну ячейку массива, связать в однонаправленный (или двунаправленный) список (рис. 24.10).

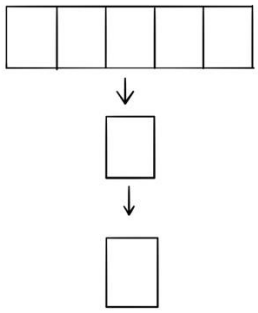


рис. 24.10 массив с однонаправленным списком

В таком случае все операции становятся двухэтапными: 1) вычисление хеш-функции и определение ячейки, 2) работа со списком.

В случае использования двусвязного списка среднее время выполнения каждой из трех операций будет иметь порядок O(1). Основная трудность - в поиске по списку, но коллизий не очень много и hash(key) можно выбрать так, чтобы списки были достаточно короткими. Примером хеш-таблицы с цепочками является записная книжка с алфавитом.

Устройство хеш-таблицы

Рассмотрим устройство хеш-таблицы при реализации хеширования с цепочками.

(1) Задается некоторое фиксированное число m (типичное значение m от 100 до 1,000,000).
(2) Создается массив index[m] указателей начал двусвязных списков (цепочек) который называется *индексом* хеш-таблицы. В начале работы все указатели имеют значения NULL.
(3) Задается хеш-функция hash(), которая получает на вход ключи и выдает значение от 0 до m-1.
(4) При добавлении пары (key, value) вычисляется h = hash(key) и пара добавляется в список index[h].
(5) При удалении либо поиске пары (hash, value) вычисляется h = hash(key) и происходит удаление либо поиск пары (key, value) в списке index[h].

Равномерность разброса ключей по ячейкам таблицы опирается на анализ хеширования с цепочками.
Пусть index[m] - хеш-таблицы с mпозициями, в которую занесено n пар (key, value).
Отношение $\alpha = n/m$ называется *коэффициентом* заполнения хеш-таблицы.
Коэффициент заполнения α позволяет судить о качестве хеш-функции:
пусть $M = \frac{1}{m}\sum_{i=0}^{m-1}|index[i]|$ - средняя длина списков.
Если hash(key) - «хорошая» хеш-функция, то дисперсия (насколько сильно данные отличается от среднего)

$$D = \frac{1}{m}\sum_{i=0}^{m-1}(|M - index[i]|)^2 \leq \alpha$$

Это условие исключает наихудший случай, когда хеш-значения всех ключей одинаковы, заполнен только один список в этом списке из n элементов имеет среднее время $\Theta(n)$.

Равномерное хеширование: хеш-функция подобрана таким образом, что каждый данный элемент может попасть в любую из m позиций хеш-таблицы с равной вероятностью, независимо от того, куда попали другие элементы.
Средняя длина каждого из m списков хеш-таблицы с коэффициентом заполнения α равна α.
Среднее время поиска элемента, отсутствующего в таблице, пропорционально средней длине списка α, так как поиск сводится к просмотру одного из списков.
Поскольку среднее время вычисления хеш-функции равно $\Theta(1)$, то среднее время выполнения каждой из функции равно $\Theta(1 + \alpha)$.

Теорема. Пусть T - хеш-таблица с цепочками, имеющая коэффициент заполнения α, причем хеширование равномерно. Тогда при поиске элемента, **отсутствующего** в таблице, будет просмотрено в среднем α элементов таблицы, а время поиска, включая время вычисления хеш-функции, будет равно $\Theta(1 + \alpha)$.

Теорема. При равномерном хешировании среднее время **успешного поиска** в хеш-таблице с коэффициентом заполнения α есть $\Theta(1+\alpha)$.

Замечание. Теорема не сводится к предыдущей, так как в предыдущей теореме оценивалось среднее число действий, необходимых для поиска случайного элемента, равновероятно попадающего в любую из ячеек таблицы.
В этой теореме сначала рассматривается случайно выбранная последовательность элементов, добавляемых в таблицу (на каждом шаге все значения ключа равновероятны и шаги независимы); потом в полученной таблице выбираем элемент для поиска, считая, что все элементы равновероятны.

Из теорем следует, что в случае равномерного хеширования среднее время выполнения любой словарной операции есть O(1).

Методы построений хеш-функций

Рассмотрим построение хеш-функции *методом деления с остатком*.

Хеш-функция hash(key) определяется соотношением
hash(key) = key % m
При правильном выборе m такая хеш-функция обеспечивает распределение, близкое равномерному. Для этого в качестве m выбирается достаточно большое простое число, далеко отстоящее от степени двойки.
Например, если устраивает средняя длина списков 3, а число записей, доступ к которым нужно обеспечить с помощью хеш-таблицы примерно равно 2000, то можно взять m = 2000/3 ≈ 701. Тогда hash(key) = key % 701.

Однако есть недостаток: в качестве m нельзя брать степень двойки, так ка если $m = 2^p$, то hash(key) — это просто p младших битов числа key.

Кроме этого, есть *метод умножения.*

Пусть количество хеш-функций равно m. Выберем и зафиксируем вещественную константу v, 0 < v < 1;
Положим $\mathrm{hash(key)} = \mathrm{m(frac(key} \cdot \mathrm{v))}$
$\mathrm{frac(key} \cdot \mathrm{v)}$ - дробная часть числа $\mathrm{key} \cdot \mathrm{v}$
Достоинство метода умножения в том, что качество хеш-функции слабо зависит от выбора m. Обычно в качестве m выбирают степень двойки, так как в этом сличая умножение на m сводится к сдвигу.

Пример. Пусть в используемом компьютере длина слова равна w битам и ключ key помещается в одно слово.
Если $m = 2^p$, то вычисление hash(key) можно выполнить следующим образом:
Умножим keyна w-битное числоr_0.
В качестве значения hash(key) возьмем старшие p битов «дробной» части числа

$$\frac{r_0}{2^w}$$

($r_0 \% 2^w$ или обнуление w старших разрядов, потом умножение на m = 2^p).
Согласно Д. Кнуту выбор $v = \frac{\sqrt{5}-1}{2} = 0{,}6180339887$ ... является удачным.

Хеш-функции: программы

Вычисление хеш-адреса и поиск по ключу k: если элемент с ключом kнайден, возвращаем значение true и указатель на найденный элемент; если элемент не найден, возвращаем значение false и указатель на последний элемент либо NULL, если цепочка пустая.

```
static bool serach_internal(int k, struct  htype **r){
        struct htype *p, *q;
        if ((p = index[hash(k)])) != NULL) {
                do {
                        if (p -> key == k) {
                                *r = p;
                                return true;
                        }
                        else
                          q = p, p = p -> next;
                } while(p);
                  *r = q;
        } else
          *r = NULL;
        return false;
}


/* добавление новой пары (key, value) */
void insert (int k, int v) {
        struct htype *p, *q;
        /*  если элемент с ключом k уже имеется в цепочке,
          изменяем его значение на v */
```

```
        if (search_internal(k, &p))
          p -> val =v;
        else {
                /* если элемента с ключом k в цепочке нет */
                /* порождение и инициализация нового элемента в цепочке */
                q = new();
                q-> key = k;
                q -> val = v;
                /*включение порожденного элемента в цепочку */
                if (p){
                        p -> next =q;
                        q -> prvs = p;
                } else
                  index[hash(k)] = q;
        }
}

/* исключение пары (key, value) */

void delete(int k, int v) {
        struct htype *p;
        if (search_internal(k, &p)){
                if (p -> prvs)
                  p -> prvs -> next = p -> next;
                else
                  index[hash(k)] = p -> next;
                if (p -> next)
                  p -> next -> prvs = p -. prvs;
                free(p);
        }
        /*иначе ничего не нашли, удалять не нужно */
}
```

Лекция 24. Цифровой поиск

Хеширование с открытой адресацией

Все записи хранятся в самой хеш-таблице: каждая ячейка таблицы (массива длины m) содержит либо хранимый элемент, либо NULL. Указатели вообще не используются, что приводит к сохранению места и ускорению поиска.
Таким образом, коэффициент заполнения $\alpha = n/m$ не больше 1.

Поиск (search): мы определенным образом просматриваем элементы таблицы, пока не найдем искомый или не убедимся, что искомый элемент отсутствует.
Просматриваются не все элементы (иначе это был бы последовательный поиск), а только некоторые согласно значению хеш-функции, которая в этом случае имеет два аргумента - ключ и «номер попытки»:

$$hash: U \times \{0, 1, \dots, m-1\} \to \{0, 1, \dots, m-1\}$$

Функцию hash можно выбрать такой, чтобы в последовательности проб (hash(k,0), hash(k,1),…,hash(k, m-1)) каждый номер ячейки $0, 1, \dots, m-1$ встречался только 1 раз.
Если при поиске мы добираемся до ячейки, содержащей NULL, можно быть уверенным, что элемент с данным ключом отсутствует (иначе он попал бы в эту ячейку).

```
#define m 1999
struct htype {
        int key; /*ключ */
        int val; /*значение элемента данных */
} *index[m];

/* Поиск элемента */
struct htype *search(int k) {
        int i = 0, j;

        do {
                j = hash(k,i);
                if (index[j] && index[j] -> key == k)
                  return index[j];
        } while (index[j] && ++i < m);
        return NULL;
}

/*Добавление элемента */
```

```
int insert(int k, int v) {
        int i = 0; j;

        do {
                hash(k, i);
                if (index[j] && index[j] -> key == k){
                        index[j] -> val =v;
                        return j;
                }
        } while (index[j] && ++i < m);
        /* таблица может оказаться заполненной */
        if (i == m)
        return -1; /* ил расширим index */
        index[j] = new();
        index[j] -> key = k, index[j] -> val = v;
        return j;
}

/* внутренний поиск */

static int search_internal(int k) {
        int i = 0, j;

        do {
                j = hash(k, i);
                while(index[j] && ++i < m);
                  return j;
        } while (index[j] && ++i < m);
        return -1;
}

/* внешний поиск легко реализуется через внутренний */

struct htype *search(int k) {
        int j = search_internal(k);
        return j >= 0 ? index[j] : NULL;
}

/*удаление элемента */
```

```
void delete(int k) {
        int j;

        j = search_internal(k);
        if (j < 0)
          return;
        /* нельзя писать index[j] = NULL!
        Будут потеряны ключи, возможно, находящиеся
        за удаленным ключом (с тем же хешем) */
        ???
}
```

Можно сказать, что адрес 1 в нашей машине никогда не будет валидным указателем (на самом деле почти всегда это будет правда).
Есть тип intptr_t - достаточно большой тип, чтобы держать любой указатель Его с (void *) будем использовать как маркер того, что у нас в ячейке что-то было, поэтому называется SHADOW (тень).
Теперь у нас есть 3 варианта значений ячейки:
(1) NULL
(2) SHADOW
(3) Не NULL и не SHADOW (заполненная ячейка)

Теперь удаление элемента выглядит так:

```
#define SHADOW((void *) (intptr_t) 1)

/* Удаление элемента */

void delete(int k) {
        int j;

        j = search_internal(k);
        if(j < 0)
          return;
        /* нельзя писать index[j] = NULL! */
        free(index[j]);
        index[j] = SHADOW;

}

#define SHADOW((void *) (intptr_t) 1)
```

```
#define ISEMPTY (el) ((!el) || (el) == SHADOW)

static int search_internal(int k) {
        int i = 0, j;
        do {
                j = hash(k, i);
                if (!ISEMPTY(index[j]) && index[j] -> key == k)
                  return j;
        } while (index[j] && ++i < m);
        return -1;
}
```

Вставка в каком-то смысле лучше, чем поиск.

```
int insert(int k, int v) {
        int i = 0; j;

        do {
                hash(k, i);
                if (! ISEMPTY(index[j]) && index[j] -> key == k){
                        index[j] -> val =v;
                        return j;
                }
        } while (!(ISEMPTY(index[j]) && ++i < m);
        /* таблица может оказаться заполненной (много вставок/удалений)*/
        if (i == m)
        return -1; /* ил расширим index */
        index[j] = new();
        index[j] -> key = k, index[j] -> val = v;
        return j;
}
```

Хеш-функции для открытой адресации

Линейная последовательность проб. Пусть
$hash: U \to \{0, 1, \dots, m-1\}$- обычная хеш-функция.
Функция
$hash(k, i) = (hash'(k) + i) \ mod \ m$
Определяет *линейную последовательность проб*.
При линейной последовательности проб начинают с ячейки index[h’(k)], а потом перебирают ячейки таблицы подряд:

index[h'(k)+1], index[h'(k)+2], … (после index[m-1] переходят к index[0]).
Существует лишь m различных последовательностей проб, так как каждая последовательность однозначно определяется своим первым элементом.

Серьезный недостаток - тенденция к образованию *кластеров* (длинных последовательностей занятых ячеек, идущих подряд), что удлиняет список:

- Если в таблице все четные ячейки заняты, а нечетные ячейки свободны, то среднее число проб при поиске отсутствующего элемента равно 1,5.
- Если же те же m/2 занятых ячеек идут подряд, то среднее число проб равно (m/2)/2 = m/4.

Причины образования кластеров: если k заполненных ячеек идут подряд, то

- Вероятность того, что при очередной вставке в таблицу будет использована ячейка, непосредственно следующая за ними, есть (k+1)/m (пропорционально «толщине слоя»),
- Вероятность использования конкретной ячейки, предшественница которой тоже свободна, всего лишь 1/m.

Таким образом, хеширование с использованием линейной последовательности проб далеко неравномерное.
Возможное улучшение: добавлять не 1, а константу *c*, взаимно простую с *m* (для полного обхода таблицы).

Так появляется квадратичная последовательность проб:
$hash(k,i) = (hash'(k) + c_1 \cdot i + c_2 \cdot i^2) mod\ m,$
c_1 и $c_2 \neq 0$ - константы/
Пробы начинаются с ячейки index[h'(k)], а потом ячейки просматриваются не подряд, а по более сложному закону.
Метода работает значительно лучше, чем линейный. Чтобы при просмотре таблице index использовались все ее ячейки, значения m, c_1 и c_2 следует брать не произвольными, а подбирать специально. Если обе константы равны единице:

- Находим $i \leftarrow hash'(k)$; полагаем $j \leftarrow 0$;
- Проверяем index[i]:
  Если она свободна, заносим в нее запись и выходим из алгоритма, если нет - полагаем $j \leftarrow (j+1) mod\ m,\ i \leftarrow mod\ m$ и повторяем текущий шаг.

Однако обычно используют не этот подход, а *двойное хеширование.*
Двойное хеширование - один из лучших методов открытой адресации.
$hash(k,i) = (h_1(k) + ih_2(k) mod\ m,$
где $h_1(k)$ и $h_2(k)$ - обычные хеш-функции. Дополнительная хеш-функция $h_2(k)$ генерирует хеши, взаимно простые с m.

Если основная и дополнительная функция существенно независимы (то есть вероятность совпадения их хешей обратно пропорционально квадрату m), то скучивания не происходит, а распределение ключей по таблице близко к случайному.

Оценки. Среднее число проб для равномерного хеширования оценивается при успешном поиске как $\frac{1}{\alpha} ln \frac{1}{1-\alpha}$.
При коэффициенте заполнения 50% среднее число проб для успешного поиска $\leq 1{,}387$, а при 90% - $\leq 2{,}559$.

При поиске отсутствующего элемента и при добавлении нового элемента оценка среднего числа проб $\frac{1}{1-\alpha}$.

При таком подходе нет списков и множества указателей и с точки зрения кода все несколько проще.

Хеширование других данных

Рассмотрим такую интересную вещь, как хеширование идентификаторов в компиляторе.

```
harshval_t
htab_hash_string(const PTR p)
{
        const unsigned char *str = (const unsigned char*) p;
        hashval_t r = 0;
        unsigned char c;

        while ((c = *str++) != 0)
          r = r*67+c-113;
        return r;

}
```

Эта функция хеширует идентификаторы (все названия функций, переменных) в компиляторе.

Цифровой поиск

Цифровой поиск - частный случай поиска заданной подстроки (образца) в длинной строке (тексте).

Примеры цифрового поиска: поиск в словаре, в библиотечном каталог и т.п., когда делается поиск по образцу в нескольких текстах (названиях книг, фамилиях авторов, текстах на вызванных сайтах и т.п.).

Хороший пример - словарь с высечками, то есть словарь, в котором обеспечен быстрый доступ к некоторым страницам (например, начальным страницам списков слов, начинающихся на очередную букву алфавита). Иногда используется многоуровневые высечки.

При цифровом поиске ключи рассматриваются как последовательности символов рассматриваемого алфавита (в частности цифр или букв). Ожидаемое число сравнений порядка $O(log_m N)$,
где m - число различных букв, используемых в словаре,
N - мощность словаря.
В худшем случае дерево содержит k уровней, где k - длина максимального слова.

Пример. Пусть множество используемых букв (алфавит) {A, B, C, D}. Мы добавим к алфавиту еду одну букву _ - пробел. По определению слова AA, AA_ , AA__ совпадают. Пусть {A, AA, AC, ADBD, BCA, BCD, CBA} - словарь (множество ключей).
Построим m-ичное дерево, где m = 5 = |_, A, B, C, D|.
Следующая небольшая хитрость позволит иногда сократить поиск: если в словаре есть слово $a_1 a_2 a_3 \dots a_k$ и первые I его букв (i < k) задают уникальное значение: комбинация $a_1 \dots a_i$ встречается в словаре только один раз, то не нужно строить дерево для j > i, так как слово можно идентифицировать по первым i буквам.

Очень важное обобщение цифрового поиска: таким же образом можно обрабатывать любые ключи, не привязываясь к байту (8 битам), который обычно используется для кодирования символов алфавита. Мы можем отсекать от ключа первые m бит, использовать 2^m-ичное разветвление, то есть строить 2^m-ичное дерево поиска (на двоичных деревьях для разветвления берется один бит: m = 1).

Прямоугольниками изображены вершины дерева, в овалах - значения слов (ключей) и связанная с ним информация. Тем самым любая вершина - массив из m элементов. Каждый элемент вершины содержит либо ссылку на другую вершину m-ичного дерева, либо на овал (ключ) (рис. 25.1).

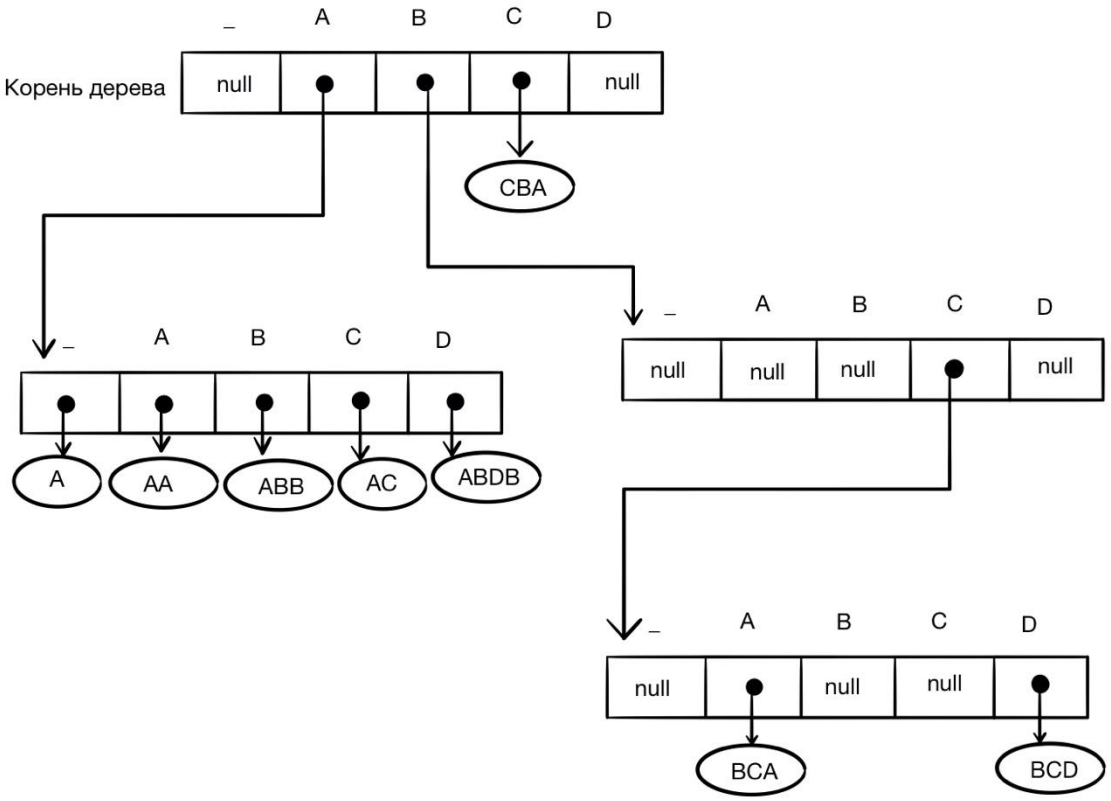


рис. 25.1

Иногда используют комбинации нескольких методов: цифровой поиск вначале, а затем переключение на поиск в последовательных таблицах.
Именно так мы работаем со словарем с высечками: вначале на высечку, а затем либо последовательный, либо дихотомический.

Обычно предлагается пользоваться цифровым поиском, пока количество различных слов не меньше некоторого k, а затем переключаться на последовательные таблицы.
Обобщением является поиск по неполным ключам и поиск по образцу.

Есть следующие варианты:

- Не строить промежуточных узлов из одного разветвления, вместо этого хранить индекс следующего символа с нетривиальным разветвлением
- Писать символы ключа на ребрах (“бор/сжатый бор”).

Реализация программы цифрового поиска:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define M 5

typedef enum {word, node} tag_t;
struct record {
        cahr *key;
        int value;
};

struct tree {
        tag_t tag;
        union {
                struct record *r;
                struct tree *nodes[M+1];
        }; /* анонимное объединение */
};
```

Чтобы искать элемент цифровым поиском нужно символ превратить в индекс массива.
Цифровой поиск:

```
static inline int ord(char c) {
        return c ? c - 'A'+ 1 : 0; // ASCII only! Другие кодировки не гарантируют букв
подряд
}
struct record *find(struct tree *t, char *key){
        int i =0;
        while (t) {
                switch (t -> tag){
                        case word:
                         for (key[i] != t -> r-> key[i])
                           return NULL:
                         return t -> r -> key[i] ? NULL : t -> r;
                        case node:
                          t = t -> nodes[ord(key[i])];
                          if (key[i])
                            i++;
                }
        }
        return NULL;
}
```

Вспомогательные функции для вставки:

```
struct record *make_record(char *keey, int value){
        struct record *r = malloc(sizeof(struct record));
        r -> key = strdup(key);
        r -> value = value;
        return r;
}
struct tree *make_from_record(struct record *r) {
        struct tree *t = malloc(sizeof(struct tree));
        t -> tag = word;
        t -> r = r;
        return t;
}

struct tree *make_word(char *key, int value) {
        return make_from_record(make_record(key, value));
}

struct tree *make_node(void) {
        struct tree *t = calloc(1, sizeof(struct tree));
        t -> tag = node;
        return t;
}
```

Вставка элемента:

```
struct tree *insert(struct tree *t, char *key, int value){
        if (!t)
          return make_word(key, value);
        int i = 0;
        struct tree *root = t;

        /* skip all nodes */
        while (t -> tag == node) {
                struct tree **p = &t -> nodes[ord(key[i++])];
                if (!*p) {
                        *p = make_word(key, value);
                        return root;
                }
                t = *p;
        }
```

```
        /* all words skipped -- key exists, update value */
        if (i && !key[i-1]){
                t -> r -> value = value;
                return root;
        }
        /* compare the remaining part */
        int j = i;
        for (; key[i]; i++)
          if (key[i] != t -> r -> key[i])
            break;
        /* key already exists -- update value */
         if (!key[i] && !t -> r ->key[i]){
                t ->r->value =  value;
                return root;
}
/* turn t into a node */
struct record *other = t->r;
t->tag = node;
memset(t->nodes, 0, sizeof(t->nodes));

/*make new nodes for remaining common prefix */
for (; j < i ; j++) {
        struct tree *p = make_node();
        t -> nodes[ord(key[j])] = p;
        t = p;
}

/* accommodate both other and new record */
t -> nodes[ord(other->key[i])] = make_from_record(other);
t->nodes[ord(key[i]})] = make_word(key, value);
return root;
}
```

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА

teach-in

ЛЕКЦИИ УЧЕНЫХ МГУ