Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Содержание

1. Вводная лекция. Цели и задачи курса. Характеристика разделов курса – введение в алгоритмы, язык программирования, структуры данных. Алгоритмы. Формализация понятия алгоритма.
2. Машина Тьюринга (МТ). Примеры МТ. Нормальные МТ. Диаграммы Тьюринга: диаграммы элементарных МТ, правила композиции диаграмм, примеры диаграмм.
3. Диаграммы Тьюринга: построение таблицы МТ по диаграмме. Построение языка описания МТ. Моделирование МТ. Универсальная МТ. Проблемы останова и самоприменимости.
4. Нормальные алгоритмы Маркова. Примеры других алгоритмических систем. Основная гипотеза теории алгоритмов. Тезис Тьюринга-Черча. Общая классификация и характеристики языков программирования. Место языка Си в современной индустрии ПО. Стандарты и роль стандартизации. Язык программирования Си. Первая программа на Си.
5. Описание Си-машины: процессор, классы памяти (регистровые, автоматические и статические переменные), структура программного файла. Системные библиотеки и их использование.Типы данных языка Си: целые, логические, символьные, с плавающей точкой, беззнаковые целые.
6. Арифметические и логические выражения. Старшинство операций. Операции присваивания. Форматный ввод-вывод. Приведение типов при вычислении выражений (явное и неявное). Операторы. Символьный тип и обработка символьных данных.
7. Массивы. Многомерные массивы. Строки. Обработка строк. Системная библиотека работы со строками.

Содержание

8. Указатели. Адресная арифметика. Массивы и указатели. Функции. Определение и объявление функции. Передача параметров. Рекурсия. Хвостовая рекурсия.
9. Представление данных с плавающей точкой. Стандарт IEEE 754. Вычисление сумм и произведений данных с плавающей точкой. Потеря точности при вычитании. Выбор правильной последовательности вычислений.
10. Побитовая обработка данных. Беззнаковые целые типы. Абстрактный тип данных «множество» и его реализация. Структуры. Указатели на структуры.
11. Составные инициализаторы структур. Объединения. Анонимные объединения и структуры. Битовые поля. Перечисления.
12. Динамическое выделение и освобождение памяти. VLA-массивы и их выделение в динамической памяти. Массив переменного размера в составе структуры.
13. Схема компиляции программ на языке Си. Препроцессор. Директивы препроцессора. Компоновка. Классы памяти и компоновки. Отладка программ. Понятие об отладчике gdb.
14. Организация типа данных «стек» на динамической памяти. Использование стека для построения обратной польской записи. Очередь.
15. Алгоритм топологической сортировки Вирта.
16. Алгоритм топологической сортировки Вирта (окончание). Понятие о сложности алгоритмов. Сортировка.
17. Простейшие алгоритмы сортировки. Оценка сложности алгоритмов сортировки. Быстрая сортировка.
18. Поиск подстроки по образцу. Простейший алгоритм. Алгоритм Кнута – Морриса – Пратта (КМП). Префикс-функция и ее вычисление. Сложность вычисления префикс-функции и алгоритма КМП.
19. Двоичные деревья. Обход двоичного дерева (сначала в глубину, сначала в ширину). Нерекурсивный обход двоичного дерева. «Прошитое» двоичное дерево и его обход.

Содержание

20. Двоичные деревья поиска и операции над ними (поиск элемента, минимальный и максимальный элементы, следующий элемент, вставка и удаление элемента). Деревья Фибоначчи. Оценка высоты дерева Фибоначчи. АВЛ-деревья. Базовые операции над АВЛ-деревьями. Балансировка АВЛ-дерева. Вставка и удаление элемента в/из АВЛ-дерева. Оценка высоты АВЛ-дерева по дереву Фибоначчи.
21. Красно-черные деревья. Структура данных «куча» (heap) и сортировка heapsort (пирамидальная сортировка).
22. Хеш-таблицы. Хеширование. Хеширование цепочками. Хеширование с открытой адресацией.
23. Цифровой поиск. Задача цифрового поиска. Деревья цифрового поиска. Вставка в дерево цифрового поиска.
    Самоперестраивающиеся деревья (splay trees). Операция splay (перестраивание). Реализация словарных операций через операцию splay. Реализация операции splay. Сложность словарных операций в splay-деревьях.
    Обобщение сбалансированных деревьев поиска: ранговые деревья, понятие ранга и ранговой разницы. Ранговые правила для АВЛ-деревьев и красно-черных деревьев. Слабые АВЛ-деревья.

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекции 1-2

Курс «Алгоритмы и алгоритмические языки»

Лектор Белеванцев Андрей Андреевич
кафедра СП

Лекции 2 раза в неделю: среда/суббота, 8.45

Практические и лабораторные занятия 2 раза в неделю

Структура курса:

- Элементы теории алгоритмов
- Язык Си
- Алгоритмы и структуры данных

В конце курса зачет с оценкой и письменный экзамен

Курс «Алгоритмы и алгоритмические языки»

Сайт курса: http://algcourse.cs.msu.su/

- Новости и объявления
- Материалы лекций
- Вопросы к экзамену
- Рекомендуемая литература
- Среда разработки программ и опции компилятора
- Стиль кодирования

Практические и лабораторные занятия

Контрольные и коллоквиумы
(по лекциям - 2, по семинарам - 3)

Анкета студента: заполните и верните лектору

Курс «Алгоритмы и алгоритмические языки»

Рекомендуемая литература:

По алгоритмам и машинам Тьюринга

1. Г. Эббинхауз, К. Якобс, Ф. Манн, Г. Хермес. Машины Тьюринга и рекурсивные функции. «Мир», М. – 1972
2. М.Минский. Вычисления и автоматы. «Мир», М. – 1971

По языку Си

1. Б. Керниган, Д. Ритчи. Язык программирования Си. Издание 2-е, «Вильямс» – 2013
2. Стандарт языка Си C99 + TC{1,2,3} http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf
3. Stephen Prata. C Primer Plus. Fifth Edition. Sams Publishing 2004. http://www.9wy.net/onlinebook/CPrimerPlus5/main.html
4. Г. Шилдт. Полный справочник по Си. Издание 4-е, «Вильямс» – 2010

По алгоритмам и структурам данных

1. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы. Построение и анализ. Издание 2-е, «Вильямс» – 2011
2. Harry R. Lewis, Larry Denenberg. Data Structures and Their Algorithms. HarperCollins, 1991.

Курс «Алгоритмы и алгоритмические языки»

Методические пособия:

1. А.А. Белеванцев, С.С. Гайсарян, Л.С. Корухова, Е.А. Кузьменкова, В.С. Махнычев.
Семинары по курсу “Алгоритмы и алгоритмические языки” (учебно-методическое пособие для студентов 1 курса).
М., 2012: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова.

2. А.А. Белеванцев, С.С. Гайсарян, В.П. Иванников, Л.С. Корухова, В.А. Падарян. Задачи экзаменов по вводному курсу программирования (учебно-методическое пособие).
М., 2012: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова.

3. К.А. Батузов, А.А. Белеванцев, Р.А. Жуйков, А.О. Кудрявцев, В.А. Падарян, М.А. Соловьев. Практические задачи по вводному курсу программирования (методическое пособие).
М., 2012: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова.

Неформальное (интуитивное) определение алгоритма.

Под ***алгоритмом*** (или эффективной процедурой) в математике понимают
точное предписание, задающее вычислительный процесс, ведущий от начальных данных*, которые могут варьироваться,* ***к искомому результату***.
Алгоритм должен обладать следующими свойствами:

(1) ***Конечность*** (результативность). *Алгоритм должен заканчиваться за конечное* (хотя и не ограниченное сверху) *число шагов*.

(2) ***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 приводить к одинаковому результату.

Неформальное (интуитивное) определение алгоритма.

Под ***алгоритмом*** (или эффективной процедурой) в математике понимают
точное предписание, задающее вычислительный процесс, ведущий от начальных данных*, которые могут варьироваться,* ***к искомому результату***.
Алгоритм должен обладать следующими свойствами:

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

(4) ***Массовость***. Алгоритм задает процесс вычисления для множества исходных данных (чисел, строк букв и т.п.), он представляет общий метод решения класса задач.

Неформальное (интуитивное) определение алгоритма.

Пример: ***Алгоритм Евклида*** нахождения наибольшего общего делителя двух целых положительных чисел $НОД(a, b)$
(в геометрической форме это алгоритм нахождения общей меры двух отрезков).

Даны два целых числа a и b.

Выполнить следующие шаги:

(0) Если $a < b$, то поменять их местами.

(1) Разделить нацело a на b; получить остаток r.

(2) Если $r = 0$, то $НОД(a, b) = b$

(3) Если $r \neq 0$, заменить: a на b, b на r и вернуться к шагу (1).

***Почему необходимо формальное определение алгоритма*.**

Не имея такого определения, **невозможно** доказать, что задача алгоритмически неразрешима, т.е. алгоритм ее решения никогда не удастся построить.

Тезис Тьюринга – Черча. Для любой интуитивно вычислимой функции существует вычисляющая её значения машина Тьюринга.

Тезис Тьюринга – Черча невозможно строго доказать или опровергнуть, так как он устанавливает эквивалентность между строго формализованным понятием частично вычислимой функции и неформальным понятием *вычислимости*.

Формализация понятия алгоритма

Алфавиты и отображения.

Алфавит $A_p = \{a_1, a_2, \ldots, a_p\}$.

Символы a_i

Слова $a_{i_1} a_{i_2} \ldots a_{i_m}$

Пустое слово ε

Множество всех слов над алфавитом A_p

$$A_p^* = \{\varepsilon\} \cup A_p \cup A_p^2 \cup \ldots \cup A_p^m \cup \ldots = \bigcup_{m=0}^{\infty} A_p^m$$

Длина слова w обозначается $|w|$, в частности $|\varepsilon| = 0$

Формализация понятия алгоритма

Кодирование

Утверждение. Для любой пары алфавитов A и B существует простой алгоритм, определяющий взаимно-однозначное отображение.

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

Формализация понятия алгоритма

Обработка информации.

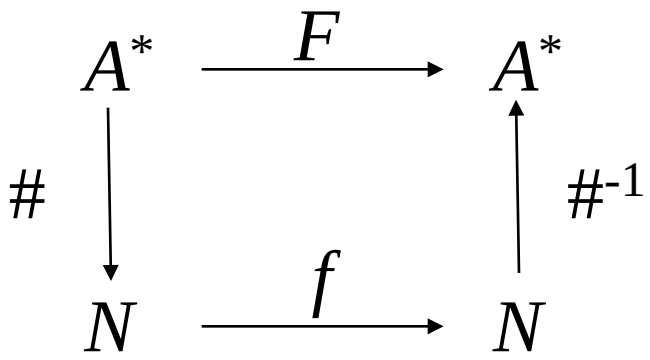
Задача обработки информации – это задача построения частичного отображения (функции)

$F : A^* \to A^*$

Утверждение. Существует взаимно-однозначное отображение (функция) $\#: A^* \to N$, где N – множество целых неотрицательных чисел, которое любому слову $w \in A^*$ ставит в соответствие его номер $n \in N$.

Формализация понятия алгоритма

Обработка информации.



Таким образом:

(1) Каждый алгоритм определяет частично вычислимую функцию;

(2) Каждая частично вычислимая функция определяет алгоритм.

Машина Тьюринга (МТ)

Вычислимость по Тьюрингу

Машина-автомат: предъявляется исходное слово $w \in A^*$, в результате обработки получается слово $v = F(w)$.

Каждая частичная функция F, для которой можно построить МТ, называется *вычислимой по Тьюрингу*.

Машина Тьюринга (МТ)

Алфавит состояний $Q = \{q_0, q_1, q_2, \ldots, q_n\}$

Рабочий алфавит $S = A \cup A'$:

A – алфавит входных символов,

A' – алфавит вспомогательных символов (маркеров).

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 (МТ)

```
...ΛΛΛ00101110010[1]1100000ΛΛΛΛΛΛ...
                  q
```

Конфигурация МТ: $\langle n, \mathcal{F}, q \rangle$, где $\mathcal{F}: Z \to S$

Такт работы МТ:

$\langle$состояние, символ$\rangle \to \langle$состояние, символ, направление$\rangle$

Машина Тьюринга (МТ)

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

Правильное скобочное выражение:

(1) число открывающих скобок равно числу закрывающих,

(2) каждая открывающая скобка предшествует парной ей закрывающей скобке.

(( ))( ) – правильное скобочное выражение

)(или (( ) – неправильные скобочные выражения

Машина Тьюринга (МТ)

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

Рабочий алфавит: $S = \{(,), 0, 1\} \cup \{\Lambda, X\}$ (X – маркер)

Алфавит состояний $Q = \{q_0, q_1, q_2, q_3, q_s\}$:

q_0 – начальное состояние МТ: поиск закрывающей скобки;

q_s – состояние останова;

q_1 – поиск парной открывающей скобки;

q_2 – стирание маркеров, запись результата 1 и переход в q_s;

q_3 – стирание маркеров, запись результата 0 и переход в q_s.

В начальном состоянии УГ обозревает самый левый символ входного слова

Машина Тьюринга (МТ)

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

Программа

$\boldsymbol{q_0}$, ($\rightarrow q_0, (, R;$ $\quad$ $\boldsymbol{q_0}$,) $\rightarrow q_1, X, L;$ $\quad$ $\boldsymbol{q_0}, \boldsymbol{X} \rightarrow q_0, X, R;$ $\quad$ $\boldsymbol{q_0}, \boldsymbol{\Lambda} \rightarrow q_2, \Lambda, L;$

$\boldsymbol{q_1}$, ($\rightarrow q_0, X, R;$ $\quad$ $\boldsymbol{q_1}$,) $\rightarrow q_1,), L;$ $\quad$ $\boldsymbol{q_1}, \boldsymbol{X} \rightarrow q_1, X, L;$ $\quad$ $\boldsymbol{q_1}, \boldsymbol{\Lambda} \rightarrow q_3, \Lambda, R;$

$\boldsymbol{q_2}$, ($\rightarrow q_3, \Lambda, L;$ $\quad$ $\boldsymbol{q_2}$,) невозможна; $\quad$ $\boldsymbol{q_2}, \boldsymbol{X} \rightarrow q_2, \Lambda, L;$ $\quad$ $\boldsymbol{q_2}, \boldsymbol{\Lambda} \rightarrow q_s, \boldsymbol{1}, H;$

$\boldsymbol{q_3}$, ($\rightarrow q_3, \Lambda, R;$ $\quad$ $\boldsymbol{q_3}$,) невозможна; $\quad$ $\boldsymbol{q_3}, \boldsymbol{X} \rightarrow q_3, \Lambda, R;$ $\quad$ $\boldsymbol{q_3}, \boldsymbol{\Lambda} \rightarrow q_s, 0, H;$

Машина Тьюринга (МТ)

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

Программа (другой способ записи)

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

Машина Тьюринга (МТ)

Нормальные МТ.

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

(1) в начальном состоянии (q_0) УГ установлена напротив пустой ячейки, которая следует за всеми исходными символами:

. . .ΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛ(()())((()))[Λ]ΛΛΛΛΛΛΛΛΛΛ. . .

q_0

(2) в состоянии останова (q_s) УГ установлена напротив пустой ячейки, которая следует за всеми символами результата:

. . .ΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛΛ1[Λ]ΛΛΛΛΛΛΛΛΛΛΛΛΛ. . .

q_s

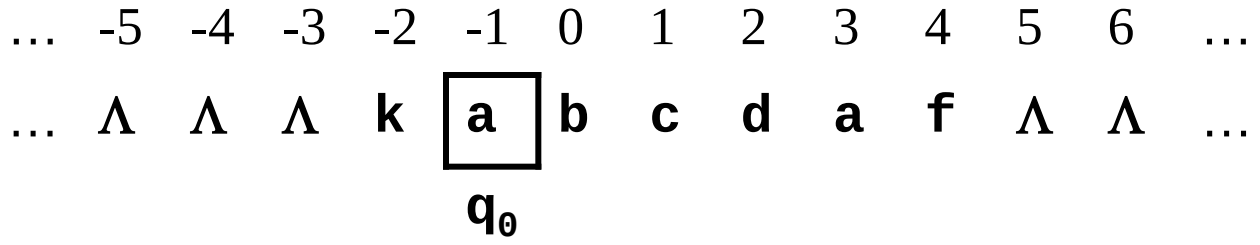
МТ, удовлетворяющая условиям (1) и (2), называется *нормальной* МТ.

Диаграммы Тьюринга (ДТ)

Перестройка МТ к виду, более удобному для ДТ

(1) МТ с лентой, ограниченной с левого конца

Для произвольной МТ ***T*** с неограниченной лентой построим МТ ***T'*** с лентой, ограниченной с левого конца, которая работает так же



(а) перегнем ленту по ячейке с номером 0;

(б) раздвинем ячейки правой части ленты, помещая содержимое ячейки с номером n в ячейку с номером $2 \cdot n$

(в) в освободившиеся ячейки с нечетными номерами поместим содержимое ячеек левой части ленты, помещая содержимое ячейки с номером n в ячейку с номером $2 \cdot |n| - 1$

Диаграммы Тьюринга (ДТ)

Перестройка МТ к виду, более удобному для ДТ

(1) МТ с лентой, ограниченной с левого конца

(а) перегнем ленту по ячейке с номером 0;

(б) раздвинем ячейки правой части ленты (с неотрицательными номерами), помещая содержимое ячейки с номером n в ячейку с номером $2 \cdot n$

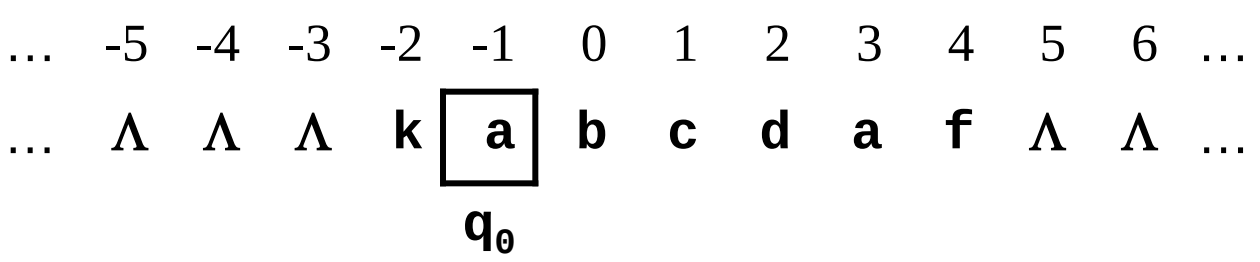
(в) в освободившиеся ячейки с нечетными номерами поместим содержимое ячеек левой части ленты (с отрицательными номерами), помещая содержимое ячейки с номером n в ячейку с номером $2 \cdot |n| - 1$

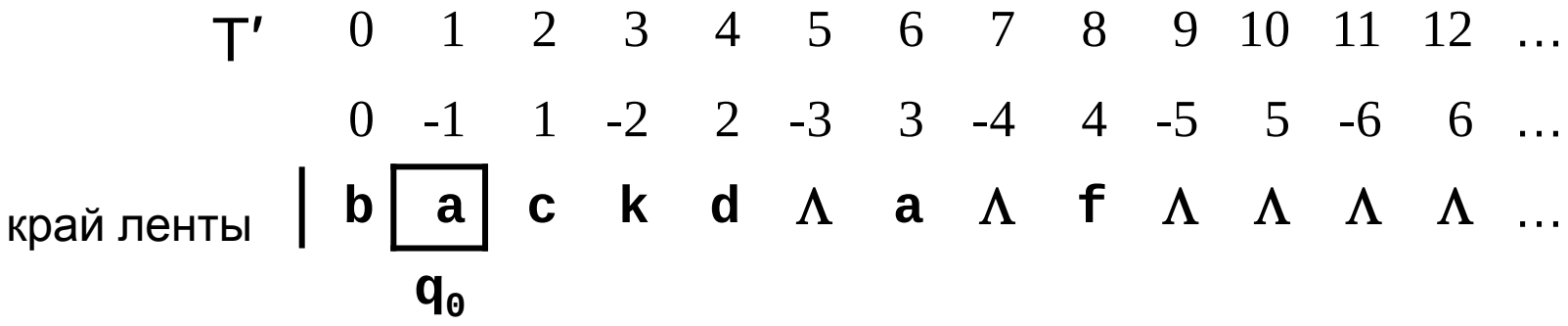
0	1	2	3	4	5	6	7	8	9	10	11	12	…
0	-1	1	-2	2	-3	3	-4	4	-5	5	-6	6	…
|	**b**	**a**	**c**	**k**	**d**	**Λ**	**a**	**Λ**	**f**	**Λ**	**Λ**	**Λ**	**Λ**	…
		q_0												

Диаграммы Тьюринга (ДТ)

Перестройка МТ к виду, более удобному для ДТ

(1) МТ с лентой, ограниченной с левого конца





T	T′ (четные)	T′ (нечетные)	T′ (ячейка 0)	T′ (ячейка 1)
вправо	на две вправо	на две влево	на две вправо	на одну влево
влево	на две влево	на две вправо	на одну вправо	на две вправо

Диаграммы Тьюринга (ДТ)

Перестройка МТ к виду, более удобному для ДТ

(2) МТ с укороченными инструкциями.

Рассмотрим произвольную инструкцию МТ **T**:

$q, a \to q', b, R;$

Разобьем ее на две инструкции:

(1) $q, a \to q'', b, H$ (только записывает символ в РЯ);

(2) $q'', b \to q', b, R$ (только сдвигает головку).

Можно доказать, что для любой МТ **T** можно построить МТ **T'**, каждая инструкция которой либо только сдвигает головку, либо только записывает символ в РЯ.

МТ **T'** и есть МТ с укороченными инструкциями.

Диаграммы Тьюринга (ДТ)

Перестройка МТ к виду, более удобному для ДТ

Далее будем рассматривать класс МТ, который содержит только МТ с укороченными инструкциями, лентой, ограниченной слева, выполняющие 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.***

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

Диаграммы Тьюринга (ДТ)

Построение диаграмм Тьюринга

Запись символа в РЯ или сдвиг УГ вправо или влево называются ***элементарными действиями.***

МТ, выполняющие элементарные действия

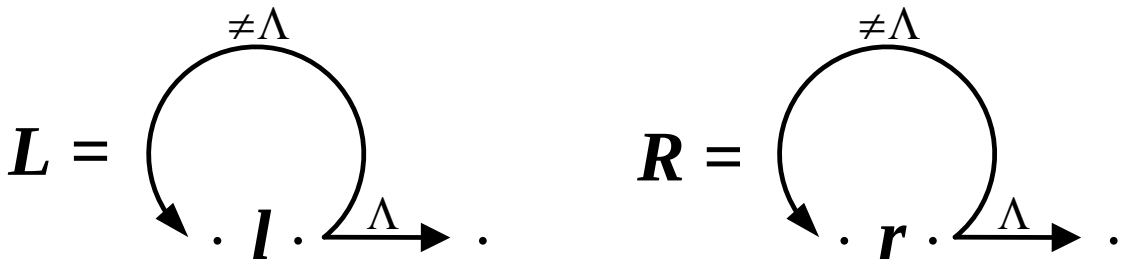
Элементарная МТ	Программа	Диаграмма
$\boldsymbol{l}$	$q_0\,\Lambda \to l\,q_1,\ q_0\,a_1 \to l\,q_1,\ \ldots,\ q_0\,a_p \to l\,q_1,$ q_1 – состояние останова*	$\cdot\boldsymbol{l}\cdot$
$\boldsymbol{r}$	$q_0\,\Lambda \to r\,q_1,\ldots,\ q_0\,a_p \to rq_1,$ q_1 – состояние останова*	$\cdot\boldsymbol{r}\cdot$
$\boldsymbol{a_i}$	$q_0\,\Lambda \to a_i\,q_1,\ q_0\,a_1 \to a_i\,q_1,\ \ldots,\ q_0\,a_p \to a_i\,q_1,$ q_1 – состояние останова*	$\cdot\boldsymbol{a_i}\cdot$

*Иногда пишут правила вида $q_1\,a_i \to hq_s$

Диаграммы Тьюринга (ДТ)

Построение диаграмм Тьюринга

Примеры не элементарных МТ.



МТ L переводит конфигурацию

$$[\Lambda\Lambda\ldots\Lambda a_1 a_2 a_3 \ldots a_n \underset{q_0}{\boxed{\Lambda}}\Lambda\ldots$$

в конфигурацию

$$[\Lambda\Lambda\ldots\underset{q_1}{\boxed{\Lambda}} a_1 a_2 a_3 \ldots a_n \Lambda\Lambda\ldots \qquad (1)$$

В дальнейшем слово на ленте $a_1 a_2 a_3 \ldots a_n$ будем обозначать w, т.е. (1) будем записывать в виде $[\Lambda\Lambda\ldots\underset{q_1}{\boxed{\Lambda}} w\Lambda\Lambda\ldots$

Диаграммы Тьюринга (ДТ)

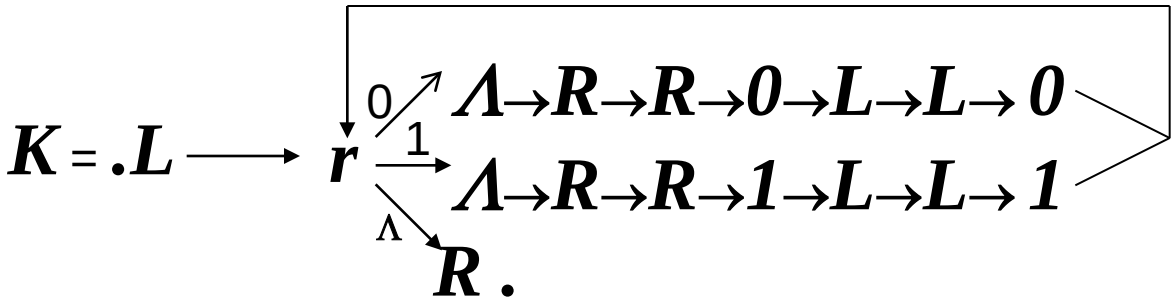
Построение диаграмм Тьюринга

МТ K переводит конфигурацию : в конфигурацию :

$[\Lambda\ \Lambda\ \dots\ \Lambda\ w\ \boxed{\Lambda}\ \Lambda\ \dots$ $\quad$ $[\Lambda\ \Lambda\ \dots\ \Lambda\ w\ \Lambda\ w\ \boxed{\Lambda}\ \Lambda\ \dots$

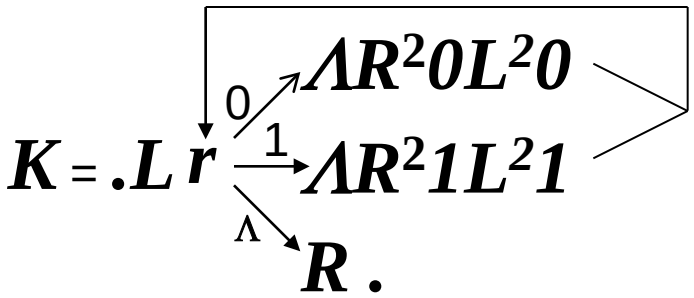
q_0 $\quad$ q_s

т.е. копирует слово w. Диаграмма МТ K (над алфавитом {0,1})



Соглашение: стрелочки, над которыми ничего не надписано, опускаются.

Получаем *упрощенную диаграмму*:



Левая точка соответствует состоянию q_0, правая – состоянию q_s

В дальнейшем при построении новых МТ можно использовать диаграмму МТ K

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

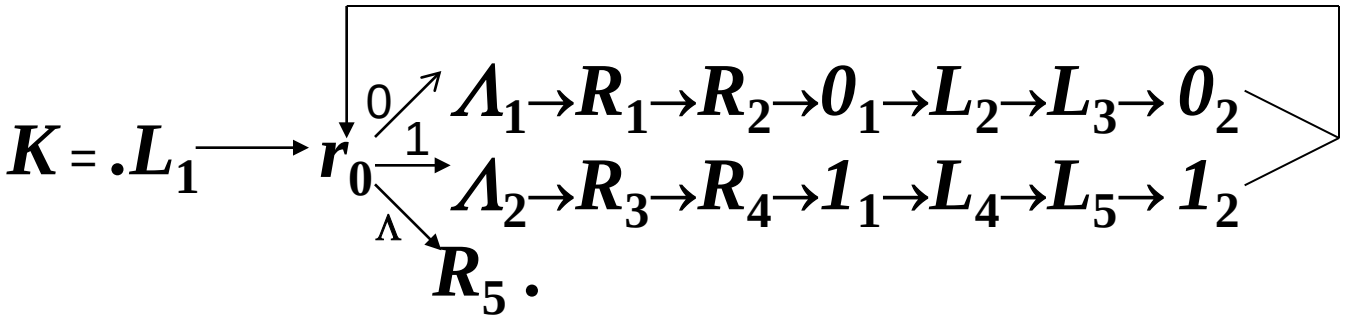
Лекция 3

Диаграммы Тьюринга (ДТ)

Построение таблиц по диаграммам

(1) Заменяем упрощенную диаграмму полной

(2) С помощью индексации добиваемся, чтобы каждый символ МТ входил в диаграмму только один раз.



(3) Сопоставим каждому символу МТ ее таблицу. Например, МТ $\boldsymbol{r_0}$ сопоставим таблицу: $q_{00}\Lambda\rightarrow \boldsymbol{r}\ q_{01}$; $q_{00}0\rightarrow \boldsymbol{r}\ q_{01}$; $q_{00}1\rightarrow \boldsymbol{r}\ q_{01}$; $q_{01}\Lambda\rightarrow\ \boldsymbol{h}\ q_{0s}$; $q_{01}0\rightarrow \boldsymbol{h}\ q_{0s}$; $q_{01}1\rightarrow \boldsymbol{h}\ q_{0s}$;

(4) Перепишем все таблицы одну за другой (в любой последовательности)

Диаграммы Тьюринга (ДТ)

Построение таблиц по диаграммам

Замечание. Диаграмма каждой МТ начинается и заканчивается точкой (начальное состояние и состояние останова). При композиции диаграмм конечная точка диаграммы сливается с начальной точкой следующей диаграммы и тем самым исключается. Следовательно, у каждой диаграммы остается только одна точка (начальная).

(5) Добавим в таблицу следующие строки:

(а) для каждого символа $\boldsymbol{A}$, которому соответствует стрелка, ведущая из точки снова к ней же, добавим строку $q_0 A \to A q_0$

(б) для каждого символа $\boldsymbol{A}$, которому соответствует стрелка, ведущая из точки к символу $\boldsymbol{M}$, добавим строку $q_0 A \to A q_{M0}$

(в) для каждого символа $\boldsymbol{A}$, которому не соответствует никакая стрелка, ведущая из точки, добавим строку $q_0 A \to h q_s$

Диаграммы Тьюринга (ДТ)

Построение таблиц по диаграммам

(5) (г) если два символа $\boldsymbol{M}$ и $\boldsymbol{M'}$ соединены стрелкой, над которой надписан символ a, то всякую строку $qa \to hq'$ для состояния останова q из части таблицы, соответствующей $\boldsymbol{M}$, заменяем строкой $qa \to aq_{M'0}$ (аналогично для стрелки в состояние останова)

В результате преобразований (1) – (5) получится таблица МТ, которая выполняет те же действия, что и МТ, заданная диаграммой.

Поэтому МТ, задаваемые диаграммами, эквивалентны МТ, задаваемыми таблицами.

Диаграммы Тьюринга (ДТ)

Моделирование МТ

Определение. МТ M моделирует МТ M', если выполнены следующие условия:

(1) Данная начальная конфигурация вызывает машинный останов МТ M после конечного числа шагов тогда и только тогда, когда указанная начальная конфигурация вызывает машинный останов МТ M' после конечного числа шагов

(2) Данная начальная конфигурация вызывает переход за край ленты у МТ M после конечного числа шагов тогда и только тогда, когда указанная начальная конфигурация вызывает переход за край ленты у МТ M' после конечного числа шагов

(3) Последовательность текущих конфигураций МТ M' для данной начальной конфигурации является подпоследовательностью последовательности текущих конфигураций МТ M для той же начальной конфигурации

Диаграммы Тьюринга (ДТ)

Универсальная машина Тьюринга

Определение. Универсальной машиной Тьюринга (УМТ) для алфавита A_p называется такая МТ $\boldsymbol{U}$, на которой может быть промоделирована любая МТ над алфавитом A_p.

Замечание. 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. 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, и выполняет его.

Диаграммы Тьюринга (ДТ)

Универсальная машина Тьюринга

Этапы построения УМТ.

(1) **Как представить программу моделируемой МТ на ленте УМТ?**

Рабочий алфавит A' УМТ является расширением алфавита $A_{p.}$

$$A' = A_p \cup \{b_1, b_2, \ldots, b_p\} \cup \{b_0\} \cup \{r, l, h, +, -, O, c, \S, *\}$$

Программа: $cw_0cw_1\ldots cw_s\S$, где w_i – слово-программа для q_i.

Правило $q_ia_j \rightarrow v_{ij}q_k$, слово-правило:
$$\begin{cases} b_j v_{ij} +^{k-i}, & \text{если } k > i \\ b_j v_{ij} O, & \text{если } k = i \\ b_j v_{ij} -^{i-k}, & \text{если } k < i \end{cases}$$

(2) **Как выглядит лента в исходном состоянии УМТ?**

$$[* w_0cw_1\ldots cw_s\S \mathbf{w}\Lambda\Lambda \ldots$$
$$\mathbf{q}$$

w – исходные данные моделируемой МТ. Звездочка отмечает q_0.

Диаграммы Тьюринга (ДТ)

Универсальная машина Тьюринга

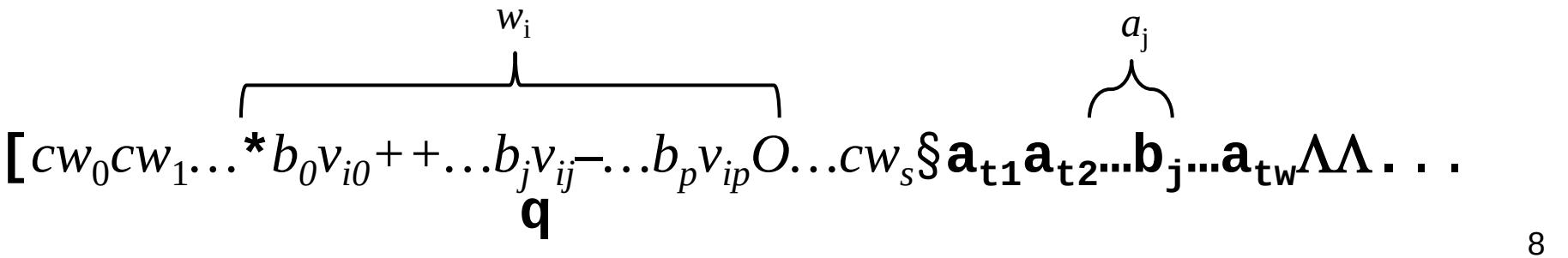
Этапы построения УМТ.

(3) **Как происходит интерпретация моделируемой МТ?**

(а) УМТ “запоминает” (размножением состояний) обозреваемый в ячейке символ a_j из A_p и заменяет его на b_j.

(б) УМТ ищет слово программы w_i, описывающее текущее состояние моделируемой МТ (отмечено звездочкой).

(в) УМТ ищет символ b_j, соответствующий “запомненному” на шаге а) символу a_j, и сдвигается вправо через действие v_{ij} до описания сдвига на следующее состояние моделируемой МТ.



Этапы построения УМТ.

(3) **Как происходит интерпретация моделируемой МТ?**

(г) УМТ передвигает символ обозначения текущего состояния моделируемой МТ (звездочку) по описанию сдвига на ее символ с и возвращается на описание действия v_{ij}.

(д) УМТ ищет записанный на шаге а) символ b_j из данных моделируемой МТ (после символа §) и выполняет считанное действие (запись или сдвиг).

Замечание. Если при сдвиге УГ попала на символ §, отделяющий программу моделируемой МТ от данных, это означает, что моделируемая МТ зашла за левый край ленты.

(е) Снова можно выполнять шаг а).

Диаграммы Тьюринга (ДТ)

Универсальная машина Тьюринга

Этапы построения УМТ.

(4) **Как происходит останов УМТ?**

Если на шаге 3 б) при поиске слова w_i был найден символ § (справа после звездочки), моделируемая МТ находится в состоянии останова.

(а) УМТ ищет символ b_j из данных моделируемой МТ (после символа §) и записывает запомненный символ a_j.

(б) После этого УГ УМТ указывает на ячейку, на которой должна остановиться УГ моделируемой МТ.

$[cw_0cw_1\ldots *w_s$§**MT(w)**$\Lambda\Lambda$**...**

q

Машина Тьюринга (МТ)

Проблема останова. Существует ли алгоритм, определяющий, произойдет ли когда-либо останов МТ $\boldsymbol{T}$ на входных данных w?

(другая формулировка) Остановится ли УМТ, моделирующая МТ $\boldsymbol{T}$ на входных данных w?

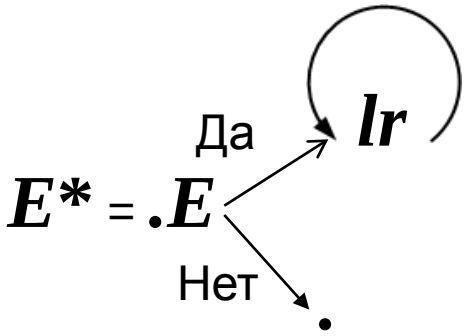
Утверждение. Проблема останова алгоритмически неразрешима.

Машина Тьюринга (МТ)

Проблема останова.

Пусть существует машина D, решающая проблему останова для всех МТ T и входных данных w. Построим машину E, которая по данной МТ T запускает машину D для МТ T и записи (описания) T на ленте.

Машина E^*:



Останавливается ли машина E^*, если ее применить к описанию самой себя?

Машина Тьюринга (МТ)

Проблема самоприменимости

- Рассмотрим МТ T для алфавита A_p. МТ T называется самоприменимой, если она останавливается, когда в качестве начальных данных используется описание T – слово над алфавитом $S \cup Q$.

  Существует ли МТ, которая по описанию МТ распознает, самоприменима ли она?
- Алгоритмическая неразрешимость проблемы самоприменимости следует из свойств МТ E и E^* с предыдущего слайда.

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 4

Нормальные алгоритмы Маркова

Определение нормального алгоритма Маркова (НАМ)

V – алфавит основных символов

V' – алфавит маркеров

$\sigma, \sigma' \in (V \cup V')^*$

Подстановка: $\sigma \to \sigma'$ переводит слово $\tau = \alpha\sigma\beta \in (V \cup V')^*$ в слово $\tau' = \alpha\sigma'\beta \in (V \cup V')^*$

подслова α и β могут быть пустыми (ε)

Помимо символов алфавита $V \cup V'$ в подстановках используются метасимволы «$\to$» (***стрелка***) отделяет левую часть подстановки от правой и «**.**» (***точка***) отмечает терминальную подстановку

Нормальные алгоритмы Маркова

Определение. ***Нормальный алгоритм Маркова*** **(*НАМ*)** задается конечной последовательностью подстановок $\{ p_1, p_2, \ldots p_n \}$.

При этом:

(1) если применимо несколько подстановок, применяется подстановка, которая встречается в описании алгоритма раньше других;

(2) если подстановка применима к нескольким подсловам обрабатываемого слова, выбирается самое левое подслово;

(3) после применения терминальной подстановки алгоритм завершается;

(4) если ни одна из подстановок неприменима, алгоритм завершается.

Нормальные алгоритмы Маркова

Пример НАМ. Шифр Юлия Цезаря

$V = \{a, b, c, \ldots, z\}$, $V' = \{*\}$.

j-ая буква латинского алфавита шифруется $j+h \pmod{26}$-ой буквой того же алфавита.

Например, для $h = 3$ подстановки имеют вид

(маркер * помечает текущую шифруемую букву, цифра в скобках – номер правила подстановки)*:*

(1) $*\boldsymbol{A} \to \boldsymbol{D}*$, (2) $*\boldsymbol{B} \to \boldsymbol{E}*$, (3) $*\boldsymbol{C} \to \boldsymbol{F}*$, …,

(23) $*\boldsymbol{W} \to \boldsymbol{Z}*$, (24) $*\boldsymbol{X} \to \boldsymbol{A}*$, (25) $*\boldsymbol{Y} \to \boldsymbol{B}*$,

(26) $*\boldsymbol{Z} \to \boldsymbol{C}*$, (27) $* \to \boldsymbol{.}$, (28) $\to *$

Применим построенный НАМ к слову ***CAESAR****:*

CAESAR (28)$\to$ ****CAESAR*** (3)$\to$ ***F*AESAR*** (1)$\to$ ***FD*ESAR*** (5)$\to$ ***FDH*SAR*** (13)$\to$ ***FDHV*AR*** (1)$\to$ ***FDHVD*R*** (12)$\to$ ***FDHVDU**** (27)$\to$ ***FDHVDU***

Нормальные алгоритмы Маркова

Процедура интерпретации НАМ

(1) Положить $i = 0$.

(2) Положить $j = 1$.

(3) Если правило p_j применимо к σ_i , перейти к шагу (5).

(4) Положить $j = j + 1$. Если $j \leq n$, то перейти к шагу (3).
В противном случае – остановка.

(5) Применить p_j к σ_i и найти σ_{i+1}. Положить $i = i + 1$.
Если p_j – нетерминальное правило, то перейти к 2.
В противном случае – остановка.

Говорят, что НАМ ***применим*** к слову σ_0, если в результате произойдет остановка.

Терминальное правило содержит метасимвол . (точка).

Иногда вместо $\rightarrow.$ используется метасимвол $\longmapsto$

Нормальные алгоритмы Маркова

Пример. Сложение чисел в единичной системе счисления:

$V = \{ +, | \}, V' = \{ \}.$

Правила подстановок: $\{ | + \to + |;\ \ + | \to |;\ \ | \to . | \}$

Пример применения алгоритма:

$$«|||+||+||» = «|||+||+||» \overset{(1)}{\Rightarrow} «||+|||+||»$$

$$\overset{(1)}{\Rightarrow} «|+||||+||» \overset{(1)}{\Rightarrow} «+|||||+||»$$

$$\overset{(1)}{\Rightarrow} «+|||||+|||» \overset{(1)}{\Rightarrow} «+|||||+|||» \overset{(1)}{\Rightarrow} \ldots$$

$$\overset{(1)}{\Rightarrow} «+ +|||||||» \overset{(2)}{\Rightarrow} «+ |||||||» \overset{(2)}{\Rightarrow} «|||||||» \overset{(3)}{\Rightarrow}$$

Первое правило «перегоняет» плюсы налево до упора, второе правило их «стирает», третье правило «убеждается», что плюсов не осталось.

Нормальные алгоритмы Маркова

Заключительные замечания

- ***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.

Заключительные замечания

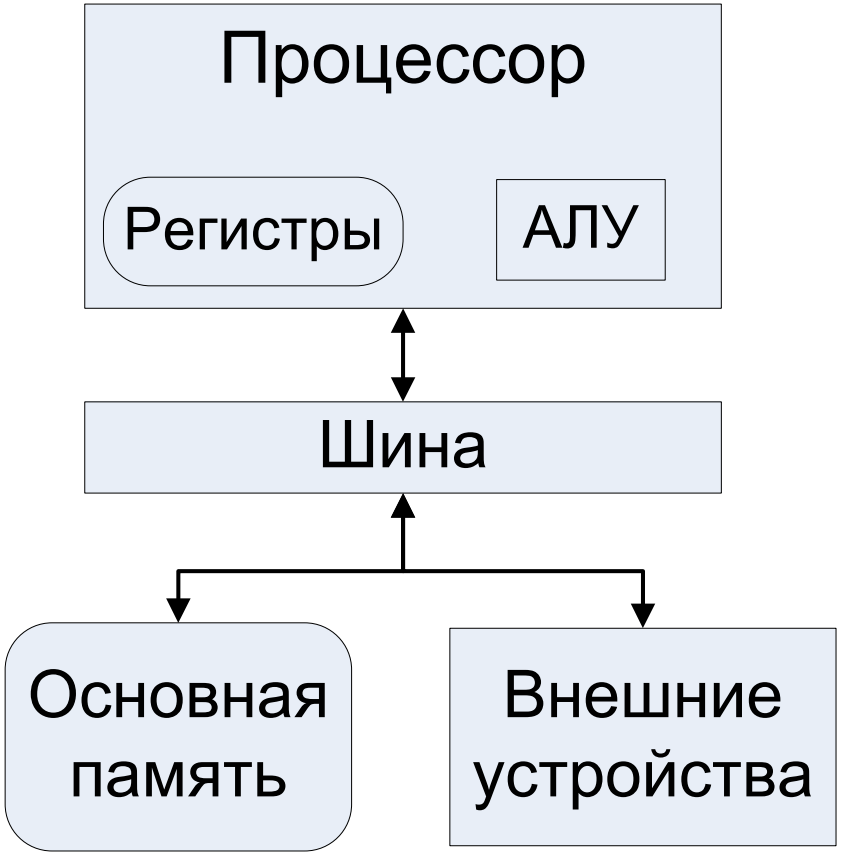
- Можно доказать эквивалентность двух формальных систем Тьюринга и Маркова конструктивным путем: построить универсальную МТ, которая могла бы интерпретировать любой НАМ и, наоборот, построить универсальный НАМ, который интерпретирует любую МТ.
- Существуют и другие формальные описания алгоритмов: машина Поста, λ-исчисление, рекурсивные функции и др.

  Для всех таких формальных систем доказана их эквивалентность МТ.
- МТ невозможно реализовать на *конечной* машине: МТ с лентой конечных размеров не обеспечивает реализации всех алгоритмов.
- ***Тезис Тьюринга – Черча*** (*основная гипотеза теории алгоритмов*). Для любой интуитивно вычислимой функции существует вычисляющая её значения МТ.

Критика модели вычислений Тьюринга

- Медленная (неускоряемая)
  - *частые копирования данных*: у нормальных МТ каждое неэлементарное действие выполняется над крайними правыми словами ленты
  - отказ от нормальных вычислений приведет к постоянному поиску данных и усложнит алгоритм
  - число состояний МТ часто зависит от числа символов в алфавите МТ

Схема простейшего компьютера



Язык программирования Си

- Си разрабатывался как язык для реализации первой в мире универсальной операционной системы UNIX
- 1973 – первая версия Си
- 1978 – выход книги Б. Кернигана и Д. Ритчи «Язык программирования Си» (K&R C). Русский перевод вышел в 1985 году.
- 1989 – первый стандарт ANSI C (C89)
- 1999 – стандарт C99
- 2011 – стандарт C11 (ранее назывался C1X)

Введение в язык программирования Си

Характеристики языка Си

- Императивный язык
- Удобный синтаксис
- Позволяет естественно оперировать «машинными» понятиями
- Переносимость на уровне исходного кода
  - Конфигурируемость
- Хорошие системные библиотеки
- Хорошие оптимизирующие компиляторы

Первая программа на Си

```
#include <stdio.h>

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

Программа:

- объявления переменных или функций
- определения функций

Первая программа на Си

```
#include <stdio.h>

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

Директивы препроцессора

Системные библиотеки

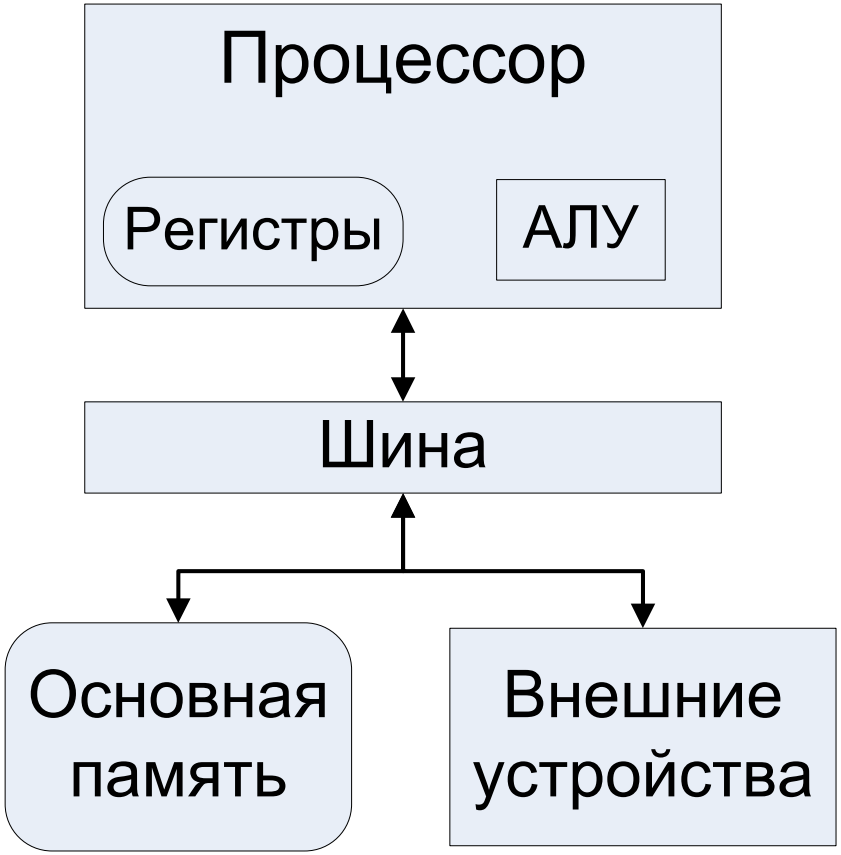
Строковые константы

Управляющие последовательности

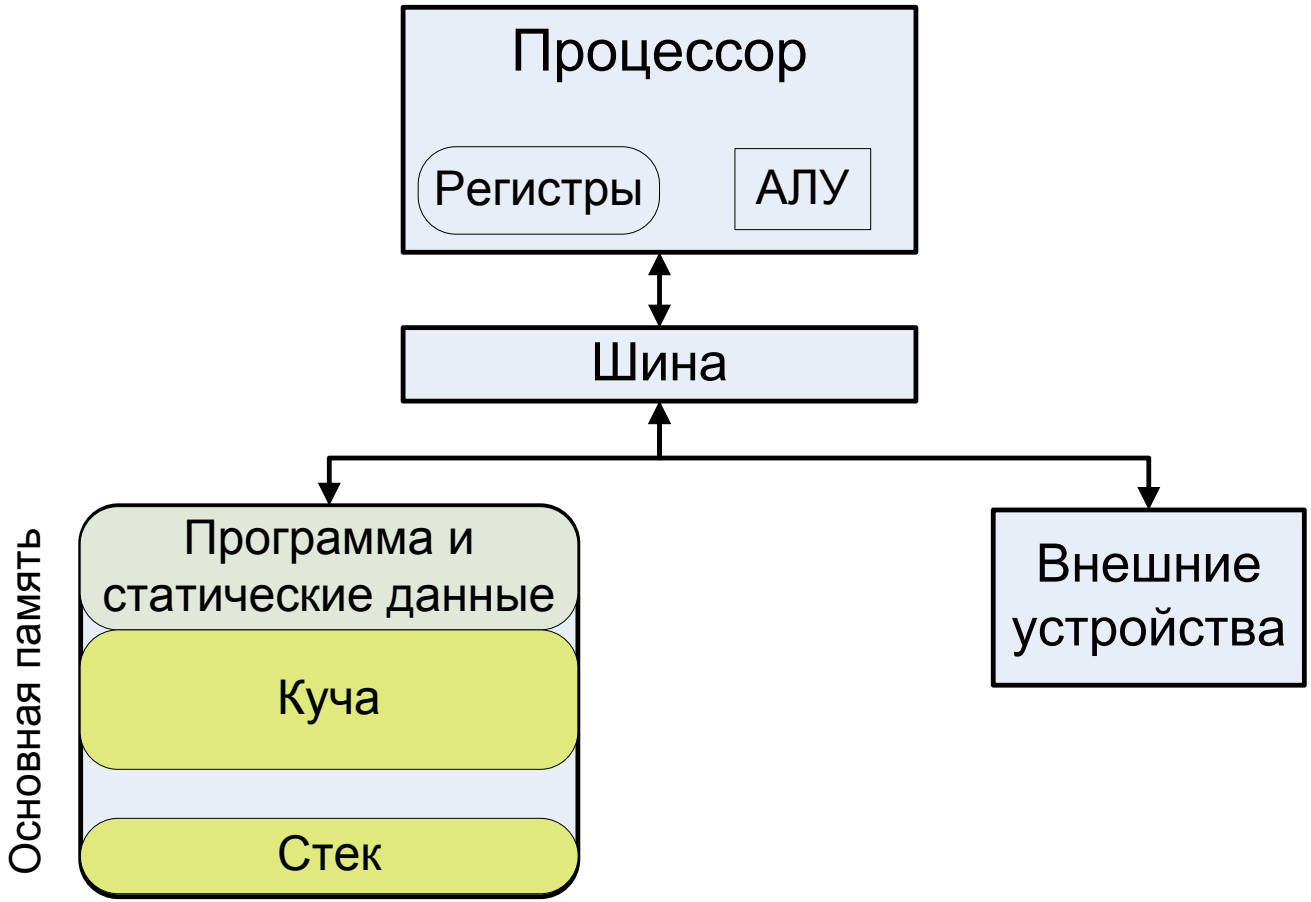
Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекции 5-6

Схема простейшего компьютера



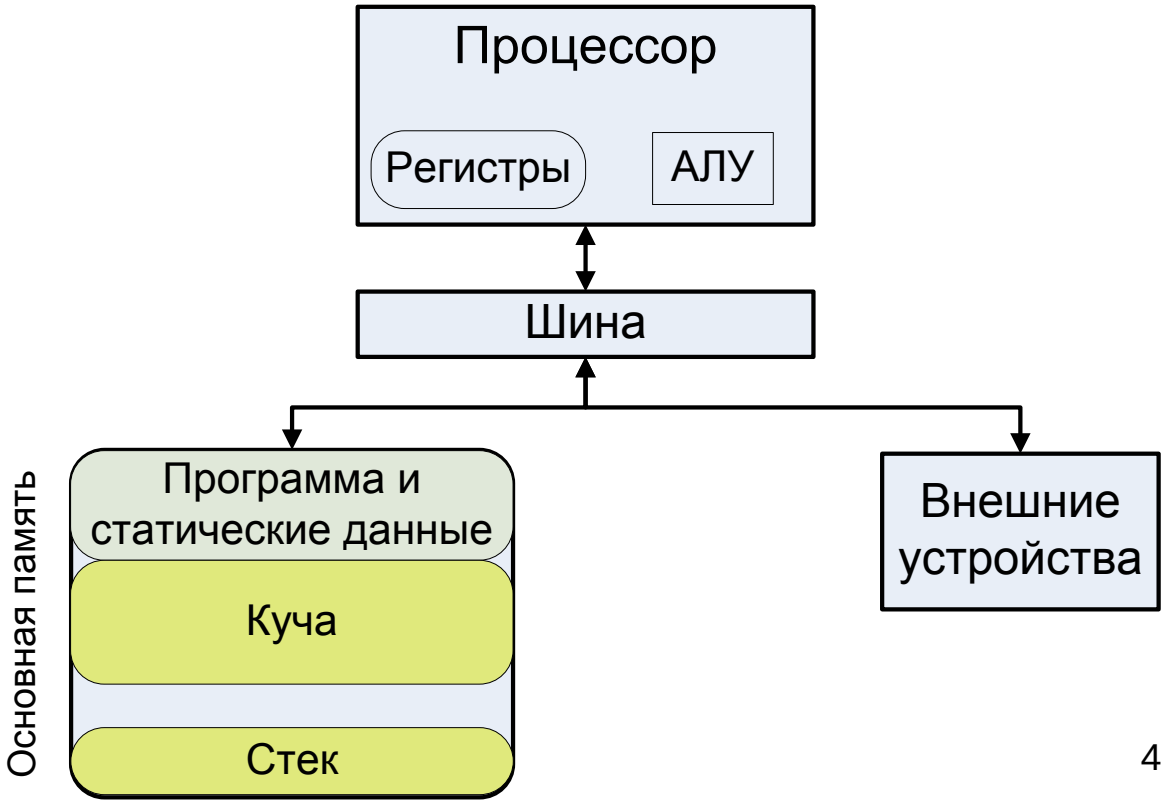
Си-машина


Процессор
Регистры
АЛУ
Шина
Основная память
Программа и статические данные
Куча
Стек
Внешние устройства

Си-машина

Классы памяти

- Регистровые переменные
- Автоматические переменные
- Статические переменные
- Глобальные переменные



Типы данных

- Базовые типы данных: **`char`** (*символьный*), **`int`** (*целый*), **`float`** (*с плавающей точкой*), **`double`** (*двойной точности*), **`_Complex`** (C99, *комплексный*)
- Тип **`void`** (*без значения*)

- Модификаторы базовых типов: **`signed`**, **`unsigned`**, **`long`**, **`short`**, **`long long`** (C99)
  - к типу **`int`** применимы все модификаторы
  - к типу **`char`** – только **`signed`** и **`unsigned`**
  - к типу **`double`** – только **`long`** (C99)

Типы данных

- Представление целых чисел: позиционная двоичная система
  - Байты в представлении числа идут подряд
  - Порядок байт 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 – значения

    - положительные значения – как обычно

    - отрицательные значения: $2^n - |x|$

Типы данных

- **`sizeof`** – размер типа (любого объекта типа)
  - **`int x -> sizeof(x) == sizeof(int)`**
  - Файл **`limits.h`** задает минимальные и максимальные значения целых типов
  - **`sizeof(char) == 1`**
    `∧`
    `sizeof(short)` $\geq$ **`2`**
    `∧`
    `sizeof(int)` $\geq$ **`2`**
    `∧`
    `sizeof(long)` $\geq$ **`4`**
    `∧`
    `sizeof(long long)` $\geq$ **`8`**
  - Файл **`inttypes.h`** задает знаковые и беззнаковые целые типы фиксированных размеров (8, 16, 32, 64 бита)

Типы данных

- Тип **_Bool** (C99, значения 0/1, целый беззнаковый)
  - Необходимо включить **stdbool.h** для объявлений **bool**, **true**, **false**
- Тип **_Complex** (C99, **float/double/long double**)
  - Необходимо включить **complex.h** для объявлений **complex**, **I** и т.п.
  - Тип **_Imaginary** (C99) является необязательным

Переменные

- Переменная = тип + имя + ***значение***

  Каждая переменная является *объектом* программы
- **`Ключевые слова`** (C89 – 32, C99 – C89 + 5) не могут быть именами переменных
- Объявление переменной:

  тип список_переменных

  Можно задать класс памяти и начальное значение переменной

Область действия переменной

- Переменная может быть объявлена:
  - (1) внутри функции или блока (локальная);
  - (2) в объявлении функции (параметр функции);
  - (3) вне всех функций (глобальная).
- Область действия (видимости)
  - локальной переменной – блок, в котором она объявлена (C99 – начиная со строки объявления)
  - глобальной переменной – программный файл, начиная со строки объявления
- В одной области действия нельзя объявлять более одной переменной с одним и тем же именем

Область действия переменной и классы памяти

```
#include <stdio.h>
int count;                      /* глобальная переменная */
void func (void)
{
  int count;                    /* автоматическая переменная */
  count = count - 2;
}
static int mult = 0;            /* статическая переменная */
int sum (int x, int y)
{
  count++;
  return (x + y) * (++mult);
}
int main (void)
{
  register int s = 0;           /* регистровая переменная */
  count = 0;
  s += sum (5, 7);
  s += sum (9, 4);
  func ();
  printf ("Сумма равна %d, вызвали функцию %d раз\n", s, count);
  return 0;
}
```

Инициализация переменной

- При объявлении переменной:

```
int x = 42;
```

  - автоматические переменные инициализируются каждый раз при входе в соответствующий блок;

    если нет инициализации, значение соответствующей переменной не определено!
  - глобальные и статические инициализируются только

    один 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 изменять свое значение

Литералы

- Литералы задают константу (фиксированное значение)
  - символьные константы **`'c'`**, **`L'%'`**, **`'\0x4f'`**, **`'\040'`**
    тип символьной константы – **`int`**!
  - целые константы **`100`**, **`-34l`**, **`1000U`**, **`999llu`**
  - константы с плавающей точкой **`11.123F`**, **`4.56e-4f`**, **`1.0`**, **`-11.123`**, **`3.1415926l`**, **`-6.626068e-34L`**
    тип вещественной константы без суффикса – **`double`**!
  - шестнадцатеричные константы **`0x80`** (*128*)
  - восьмеричные константы **`012`** (*10*)
  - строковые константы **`"a"`**, **`"Hello, World!"`**, **`L"Unicode string"`**
  - специальные символьные константы **`\n`**, **`\t`**, **`\b`**

Операции над целочисленными данными

- ***Арифметические***
  - *Одноместные*:
    изменение знака, или «одноместный минус» (**–**),
    одноместный плюс (**+**).
  - *Двухместные*:
    сложение (**+**), вычитание (**–**), умножение (*****),
    деление нацело (**/**), остаток от деления нацело (**%**).

```
(a/b) * b + (a%b) == a
```

- ***Отношения*** (результат 0/1 типа **`int`**)
  - больше (**>**), больше или равно (**>=**),
    меньше (**<**), меньше или равно (**<=**)
- ***Сравнения*** (результат 0/1 типа **`int`**)
  - равно **(==),** не равно **(!=)**
- ***Логические***
  - отрицание (**!**), конъюнкция **(&&),** дизъюнкция **(||)**
  - ложное значение – 0, истинное – любое ненулевое
  - **“ленивое” вычисление && и ||**

Операции присваивания

- ***Побочные эффекты:*** изменение объекта, вызов функции
- **`lvalue = rvalue`**
  - **`lvalue`** – выражение, указывающее на объект памяти
  - **`rvalue`** – выражение
  - Пример **`a = b = c = d = 0;`**
- ***Укороченное присваивание:*** **`lvalue op= rvalue`**
  где **`op -`** двухместная операция

  Пример **`a += 15;`**
- ***Инкремент и декремент*** (**`++`** и **`--`**)
  - префиксные и постфиксные
- ***Последовательное вычисление:*** операция запятая (**`,`**)

  Пример **`a = (b = 5, b + 2);`**

Точки следования

- ***Побочные эффекты:*** изменение объекта, вызов функции
- ***Точка следования* (*sequence point*)*:*** момент во время выполнения программы, в котором все побочные эффекты предыдущих вычислений закончены, а новых – не начаты
  - первый операнд **&&**, **||**, **,**
  - окончание ***полного*** выражения
  - между двумя точками следования изменение значения переменной возможно **не более одного раза**

    ```
    (a=2) + (a=3)
    ```

    ```
    i++ + ++i
    ```
  - старое значение переменной читается **только** для определения нового

    ```
    a = b++ + b
    ```

Форматный ввод-вывод

```
#include <stdio.h>

int main (void)
{
  int s = 0;
  int a, b;

  scanf ("%d%d", &a, &b);
  s += a + b;
  printf ("Сумма равна %d\n", s);
  return 0;
}
```

Форматный ввод-вывод

Спецификаторы ввода-вывода

%d, **%ld**, **%lld** – напечатать/считать число типа **int**, **long**, **long long**

%u, **%lu**, **%llu** – напечатать/считать число типа **unsigned**, **unsigned long**, **unsigned long long**

%f, **%Lf** – напечатать число типа **double**, **long double**

%f, **%lf** , **%Lf** – считать число типа **float** , **double**, **long double**

%c - напечатать/считать символ

%4d – вывести число типа **int** минимум в четыре символа

%.5f – вывести число типа **double** с пятью знаками

%% - напечатать знак процента

Функция **scanf** возвращает количество удачно считанных элементов

Пример Си-программы

```
/* Решение квадратного уравнения */
#include <stdio.h>
#include <math.h>
int main (void) {
  int a, b, c, d;
  /* Введем коэффициенты */
  if (scanf ("%d%d%d", &a, &b, &c) != 3) {
    printf ("Нужно ввести три коэффициента!\n");
    return 1;
  }
  if (!a) {
    printf ("Уравнение не квадратное!\n");
    return 1;
  }
  d = b*b – 4*a*c;
  if (d < 0)
    printf ("Решений нет\n");
  else if (d == 0) {
    double db = -b;
    printf ("Решение: %.4f\n", db/(2*a));
  } else {
    double db = -b;
    double dd = sqrt (d);
    printf ("Решение 1: %.4f, решение 2: %.4f\n", (db+dd)/(2*a), (db-dd)/(2*a));
  }
  return 0;
}
```

Пример Си-программы

```
/* Решение квадратного уравнения */
#include <stdio.h>
#include <math.h>
int main (void) {
  int a, b, c, d;
  /* Введем коэффициенты */
  if (scanf ("%d%d%d", &a, &b, &c) != 3) {
    printf ("Нужно ввести три коэффициента!\n");
    return 1;
  }
  if (!a) {
    printf ("Уравнение не квадратное!\n");
    return 1;
  }
```

Пример Си-программы

```
  d = b*b – 4*a*c;
  if (d < 0)
    printf ("Решений нет\n");
  else if (d == 0) {
    double db = -b;
    printf ("Решение: %.4f\n", db/(2*a));
  } else {
    double db = -b;
    double dd = sqrt (d);
    printf ("Решение 1: %.4f, решение 2: %.4f\n",
            (db+dd)/(2*a), (db-dd)/(2*a));
  }
  return 0;
}
```

Преобразование типов

- ***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 приведение типов:*** **`(type) expression`**
  - Пример `d = ((double) a+b)/2;`

Приведение типов

- ***Неявное приведение типов:*** происходит, когда операнды двухместной операции имеют разные типы (**6.3.1.8)**
  - 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 – integer promotion)

    `unsigned short(2) + char(1) -> int(4) + int(4)`

    `unsigned short(4) + char(1) -> unsigned int(4) + int(4)`
  - 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`

Приведение типов

- ***Неявное приведение типов:*** происходит, когда операнды двухместной операции имеют разные типы
  - Если операнд беззнакового типа более “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(4) /* Неверные значения */
  - Если тип 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**

Старшинство операций

Операции	Ассоциативность
`! ++ -- + - sizeof (type)`	Справа налево
`* / %`	Слева направо
`+ -`	Слева направо
`== !=`	Слева направо
`&&`	Слева направо
`||`	Слева направо
`= += -= *= /= %=`	Справа налево
`,`	Слева направо

Операторы

- ***Выражение-оператор***: **`expression;`**
- ***Составной оператор***: **`{}`**
- ***Условный оператор***: **`if (expr) stmt; else stmt;`**
  - **`else`** всегда относится к ближайшему **`if`**:

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

- ***Оператор выбора***:

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

  - Оператор **`break`** – немедленный выход из **`switch`**.

Операторы

- ***Цикл while*: `while (expression) stmt;`**
- ***Цикл for*:**

  `for (decl1; expr2; expr3)`
  `  stmt;`

  `decl1 -` возможно определение переменной с инициализатором

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

  - **`for ( ; ; ) stmt;`** – бесконечный цикл.
- ***Цикл do-while*: `do { stmt; } while (expression);`**
  - Проверка условия выхода из цикла после выполнения тела.
- ***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

Пример программы. Количество дней между двумя датами

```
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;
    t1 = check_date (d1, m1, y1);
    if (t1 == 1 || (t2 = check_date (d2, m2, y2)) == 1)
      break;
    else if (t1 == 2 || t2 == 2)
      continue;

    days1 = days_from_jan1 (d1, m1, y1);
    days2 = days_from_jan1 (d2, m2, y2);
    total = days_between_years (y1, y2) + (days2 - days1);
    printf ("Days between dates: %d, weeks between days: %d\n",
             total, total / 7);
  }
  return 0;
}
```

Пример программы. Количество дней между двумя датами

```
#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;
}

while (1) {
<...>
 t1 = check_date (d1, m1, y1);
    if (t1 == 1 || (t2 = check_date (d2, m2, y2)) == 1)
      break;
    else if (t1 == 2 || t2 == 2)
      continue;
<...>
}
```

Пример программы. Количество дней между двумя датами

```
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;
}
```

Пример программы. Количество дней между двумя датами

```
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 (char)

Программа подсчета числа строк во входном потоке

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

Каков должен быть возвращаемый тип функции **getchar**?

Символьный тип данных (`char`)

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
- К символьным данным применимы операции целочисленных типов (но обычно – ***операции отношения и сравнения***)
- Каждый символ, представляющий самого себя, 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

Символьный тип данных (char)

Таблица ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	\0									\t	\n						
1												ESC					
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL	

Символьный тип данных (char)

- В коде *ASCII* буквы верхнего и нижнего регистра составляют непрерывные последовательности:
  между **a** и **z** (соответственно, между **A** и **Z**) нет ничего, кроме букв, расположенных в алфавитном порядке.
- Это же верно и для цифр **0**, **1**, ..., **9**
- Преобразование строки символов цифр **s** в целое число
  (верно для любой кодировки символов)

```
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;
}
```

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 7

Массивы

- Массивы позволяют организовывать непрерывные последовательности нескольких однотипных элементов и обращаться к ним 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]; /* элементы b[0][0], b[0][1], b[0][2],
                          b[1][0], b[1][1], b[1][2],
                          b[2][0], b[2][1], b[2][2]

                */
```

- Контроль правильности индекса массива **не производится!**
- ***Пример***. Программа, подсчитывающая количество вхождений в строку (текст) каждой из десяти цифр (**`ndigit[10]`**), пробельных символов (**`nwhite`**) и остальных символов (**`nother`**).

Массивы

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

Инициализация массивов

***тип имя_массива*[*размер1*]…[*размерN*] = {*список_значений*};**

Пример (справа для наглядности использованы дополнительные фигурные скобки – ***группировка подагрегатов***)

```
int sqrs[10][2] = {
        1, 1,
        2, 4,
        3, 9,
        4, 16,
        5, 25,
        6, 36,
        7, 49,
        8, 64,
        9, 81,
        10, 100
}
```

```
int sqrs[10][2] = {
        {1, 1},
        {2, 4},
        {3, 9},
        {4, 16},
        {5, 25},
        {6, 36},
        {7, 49},
        {8, 64},
        {9, 81},
        {10, 100}
}
```

Инициализация массивов

***тип имя_массива*[*размер1*]…[*размерN*] = {*список_значений*};**

- Можно не указывать размер массива – он будет вычислен по количеству элементов инициализатора

```
int sqrs[] = {1, 4, 9, 16, 25}; /* 5 элементов */
```

- C99: инициализация лишь некоторых элементов (остальные инициализируются нулями)

```
int days[12] = {31, 28, [4] = 31,30,31, [1] = 29};
```

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

- Можно использовать модификаторы **`const`**, **`static`** и т.п.
- Можно использовать любое *константное целочисленное выражение* для определения размера массива
  - **`const`-**переменная не является константным выражением!

Строки

- ***Строка*** – это одномерный массив типа **`char`**

  Объявляя массив, предназначенный для хранения строки, необходимо предусмотреть место для символа **`'\0'`** (конец строки)

- ***Строковая константа*** (например, **`"string"`**).
  В конец 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`**)

Строки

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

int main (void)
{
  char string1[80], string2[80], smp[3];
  fgets (string1, 80, stdin); string1[strlen (string1)-1] = '\0';
  fgets (string2, 80, stdin); string2[strlen (string2)-1] = '\0';

  printf ("Строки имеют длину: первая %d, вторая %d\n,
          strlen (string1), strlen (string2));

  if (!strcmp (string1, string2))
    printf ("строки равны\n");

  strncat (string1, string2, 80 – strlen (string1) - 1);
  printf ("%s\n", string1);

  strcpy (string1, "Привет, ");
  printf ("%s\n", string1);
  return 0;
}
```

Строки

```
#include <stdio.h>
#include <string.h>
int main (void)
{
  char string1[80], string2[80], smp[3];
  fgets (string1, 80, stdin); string1[strlen (string1)-1] = '\0';
  fgets (string2, 80, stdin); string2[strlen( string2)-1] = '\0';

  printf("Строки имеют длину: первая %d, вторая %d\n,
          strlen (string1), strlen (string2));
  if(!strcmp (string1, string2))
    printf ("строки равны\n");
  strncat (string1, string2, 80 – strlen (string1) - 1);
  printf ("%s\n", string1);
  strcpy (string1, "Привет, ");
  printf ("%s\n", string1);
  return 0;
}
```

Если **string1** – **"Здравствуй, "** , а **string2** –**"мир!"**, результат:

```
Строки имеют длину 12 4
Здравствуй, мир!
Привет,
```

Операция `sizeof`

- Одноместная операция **`sizeof`** позволяет определить длину операнда в байтах.

  Операнды – типы либо переменные.

  Результат имеет тип **`size_t`**

- Операция **`sizeof`** выполняется во время компиляции, ее результат представляет собой константу.
- Операция **`sizeof`** помогает улучшить переносимость программ.
- Для определения объема памяти в байтах, нужного для двумерного массива:

  `number_of_bytes = d1 * d2 * sizeof (element_type)`

  где **`d1`** – количество элементов по первому измерению,

  `d2` – количество элементов по второму измерению,
  `element_type` – тип элемента массива.

- Можно поступить и проще:

  `number_of_bytes = sizeof (имя_массива)`

Операция `sizeof`

- **`sizeof`** можно применять только к «полностью» определенным типам.

  Для массивов это означает:

  - 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`**.

Операция sizeof

- Пример:

```
#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 массива
     buffer равно его размеру в байтах */
  strncpy (buffer, argv[1],
           sizeof (buffer) – sizeof (char));
  buffer[sizeof (buffer) - 1] = '\0';
  return 0;
}
```

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 8

Указатели

- **`&`** - операция адресации

  `*` - операция разыменования

```
int a = 1;
int *p;
p = &a;
*p = 2;
printf ("Значение переменной a = %d\n", *p);
printf ("Адрес переменной a = %p\n", p);
```

В результате выполнения фрагмента будет напечатано:

```
Значение переменной a = 2
Адрес переменной a = 0xbffff7a4
```

- **`&foo`** является константой, указатель – переменной

  `foo` должен быть *l*-значением (**`lvalue`**)

- Печать адреса – модификатор **`%p`**
- Нулевой указатель (никуда не указывающий) – **`NULL`**
  (константа в **`stdlib.h`**, может не иметь нулевого значения)

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 **`2012`**.

  После операции **`p1++`** значение **`p1`** будет **`2016`** (а не **`2013`**), после операции **`p1 - 3`** – значение **`2000`**.
  - при увеличении (уменьшении) на целое число **`i`** указатель будет перемещаться на **`i`** ячеек соответствующего типа в сторону увеличения (уменьшения) их адресов.

Преобразование типа указателя

- 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 ссылается на double,
                       а p имеет тип *int */
    y = *p;       /* будет ли y присвоено
                     значение 200.35? */
    printf ("значение x равно %f\n", x);
    printf ("значение y равно %f\n", y);
    return 0;
  }
  ```
  ```

Преобразование типа указателя

- Типичный вывод (GCC, Linux):

```
значение x равно 200.350000
значение y равно 858993459.000000
```

  - В присваивании **`y = *p;`** загрузка **`*p`** считывает только первые **четыре** 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 указателями выполняются **в соответствии с базовым типом указателя**.

Преобразование типа указателя

- Разрешено также преобразование целого в указатель и наоборот (поведение определяется реализацией). Однако пользоваться этим нужно очень осторожно.

  ```
  aux = (void *) -1;
  ```

- Допускается присваивание указателя типа **`void *`** указателю любого другого типа (и наоборот) без явного преобразования типа указателя. Это позволяет использовать указатель типа **`void *`**, когда тип объекта неизвестен.
- Использование типа **`void *`** в качестве параметра функции позволяет передавать в функцию указатель на объект любого типа.

Указатели и массивы

- 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]**
- Значение **array** изменить нельзя, а значение **p** – можно.

array не является *l*-значением, а **p** – является

array = p; array++ – писать нельзя (это ошибки)

p = array; p++ – писать можно (и нужно)

Указатели и массивы

- Индексирование указателей

```
int *p, a[10]; /* два способа присвоить 100 */
                /* 6-ому элементу массива a[10] */
p = a;
*(p + 5) = 100; /* адресная арифметика */
p[5] = 100;  /* индексирование указателя */
```

- Сравнение указателей

  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.

  Можно написать:

```
if (p < q)
  printf ("p ссылается на меньший адрес, чем q");
```

Массивы указателей

- Указатели могут быть собраны в массив:

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

- Пример

```
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**

Функции

- ***Объявление функции***:

  тип_возвр_значения имя_функции(*тип параметр*, *тип параметр*, …, *тип параметр*);

  `int atoi (char s[]);`

  `void QuickSort (char *items, int count);`

- Тип возвращаемого значения **`void`** означает, что функция не возвращает значения.

- ***Определение функции***:

  объявление_функции **{***тело_функции***}**

- ***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

Вызов функции

- Если функция **`f()`** возвращает значение типа *тип*, то вызов этой функции может иметь вид:

  `v = f();`,

  где **v** – переменная типа *тип*.

- Если функция **`f(`***параметр***`)`** не возвращает значений, вызов этой функции имеет вид:

  `f(`*аргумент***`);`**

- **В языке Си все аргументы передаются по значению** (т.е. передаются только значения аргументов, и эти значения копируются в память функции).

- Если аргументом является указатель, его значением может быть адрес объекта вызывающей функции, что обеспечивает вызываемой функции доступ к объекту.

Указатели и аргументы функций

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

Неправильный вариант:

```
void swap (int x, int y)
{
  int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Правильный вариант:

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

Вызов функции

- 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;
}
```

Можно объявить массив **a** в списке параметров как **const a[]**.

- Функции с переменным числом параметров:

```
int scanf (const char *, ...);
```

Всегда должен быть явно задан хотя бы один параметр.
После многоточия не должно быть других явных параметров.

Обработка переменных параметров – файл **stdarg.h**.

Функции

- 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; isdigit (*s); s++)
    n = 10 * (*s - '0');
  return sign * n;
}
```

- Стандартная библиотека **`ctype`** содержит такие функции, как **`isspace()`** , **`isdigit()`** и др.

Возврат из функции

- Возврат из функции в точку вызвавшей ее функции, следующей за точкой вызова функции, осуществляется:
  - либо при 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 функции не **`void`**, то в ее теле должен быть хотя бы один оператор **`return`** с возвращаемым значением
- Если у функции несколько операторов **`return`**, возврат осуществляется **немедленно** по тому из них, который будет выполнен первым.

Результат выполнения функции

- Все функции, кроме тех, которые относятся к типу **`void`**, возвращают значение, которое определяется выражением в операторе **`return`**.
- Помимо вычисления возвращаемого значения, функция может изменять 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 следующие виды функций:
  (1) Функции, которые выполняют операции над своими аргументами с единственной целью – вычислить возвращаемое значение.
  (2) Функции, которые обрабатывают данные и возвращают значение, которое показывает, успешно ли была выполнена эта обработка.
  (3) Функции, возвращающие несколько значений (через указатели-аргументы и через возвращаемое значение).
  (4) Функции, не возвращающие значений. Все такие функции имеют тип **`void`**.

Результат выполнения функции

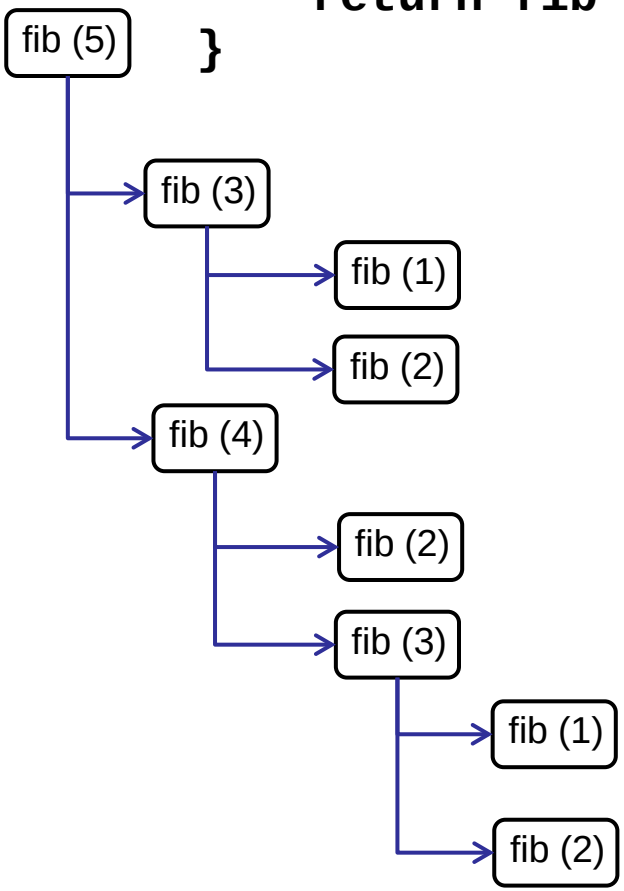
- Возвращаемым значением может быть указатель. Требуется, чтобы в объявлении такой функции тип возвращаемого указателя был объявлен точно: нельзя объявлять возвращаемый тип как **`int *`**, если функция возвращает указатель типа **`char *`**.
- Пример функции, возвращающей указатель (поиск первого вхождения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

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

Рекурсия

- В языке Си функция может быть *рекурсивной*, т.е. вызывать саму себя:

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



Рекурсия

- Рекурсивные функции часто неэффективны по сравнению с их нерекурсивными вариантами:

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

*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);
}
```

```
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;
}
```

Ключевое слово `inline`: встраиваемые функции (C99)

```
#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 ("Наибольшим из чисел %d и %d является %d\n",
           x, y, max (x, y));
  return 0;
}
```

◈ При обычной реализации **`inline`** приведенная программа эквивалентна:

```
#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 ("Наибольшим из чисел %d и %d является %d\n",
           x, y, (x > y ? x : y));
  return 0;
}
```

Указатели на функцию

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

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

Указатели на функцию

- ***Пример***. Сравнение двух строк символов, введенных пользователем (функция **check()**).

```
#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");
}

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); s2[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 два 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)`.**
- У функции **`check`** три параметра: два указателя на тип **`char`** и указатель на функцию **`pf`**. 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 *a, const char *b) {
  return atoi (a) != atoi (b);
}
```

- Массивы указателей на функцию: гибкая обработка событий

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 9

Вычисления с плавающей точкой

- Предпосылки: дробные двоичные числа
- Стандарт арифметики с плавающей точкой IEEE 754: Определение
- Пример и свойства
- Округление, сложение, умножение
- Плавающие типы языка Си
- Выводы

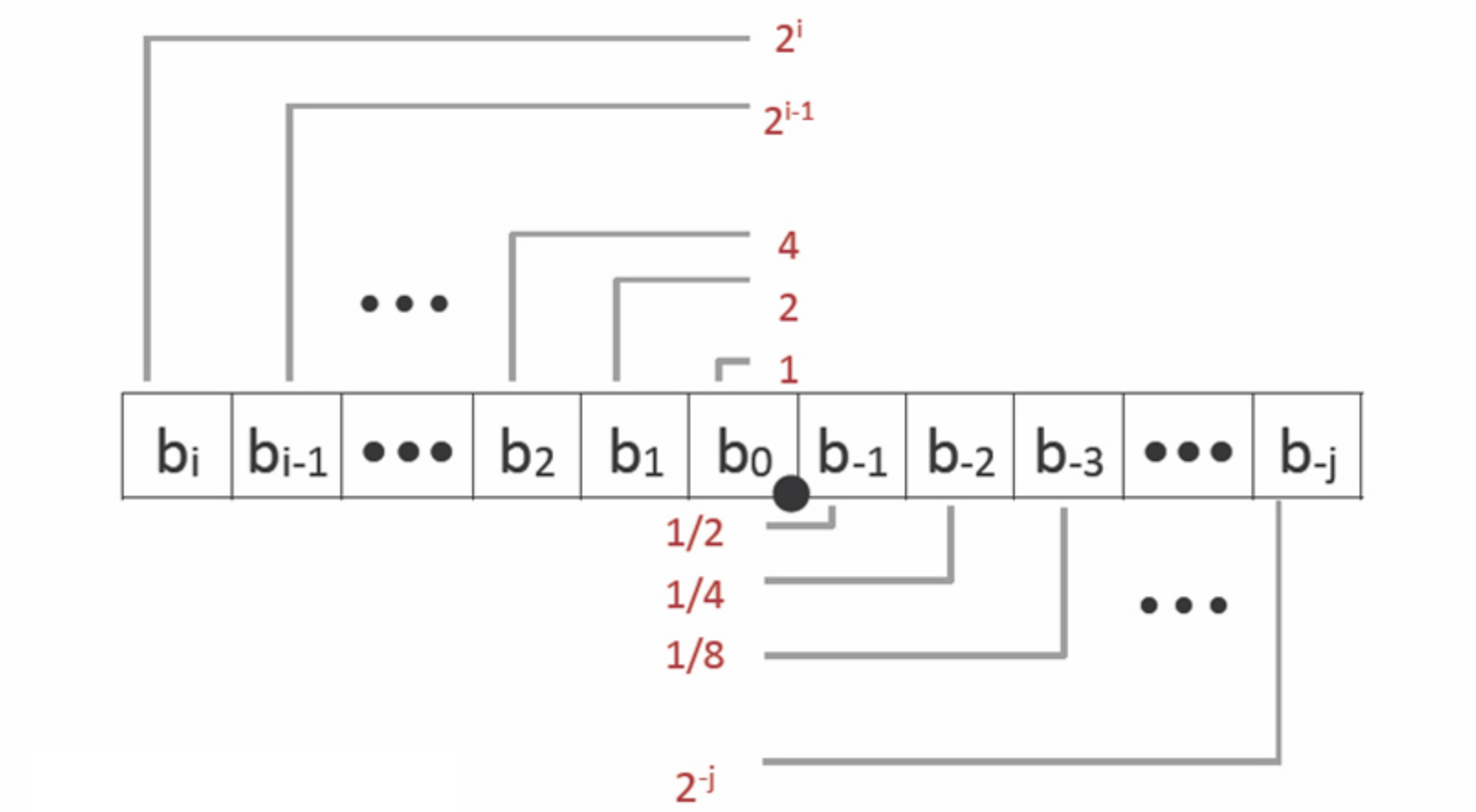
Дробные двоичные числа

- Что такое 1011.101_2 ?

$$1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-3}=$$

$$=11\frac{5}{8}=11.625$$

Дробные двоичные числа



- Черное пятнышко – двоичная точка
- Биты слева от точки умножаются на положительные степени 2
- Биты справа от точки умножаются на отрицательные степени 2

Дробные двоичные числа

- $0.111111\ldots_2 = 1.0 - \varepsilon$ ($\varepsilon \to 0$), так как

$$1/2 + 1/4 + 1/8 + \ldots + 1/2^n + \ldots \to 1 \quad \text{при } n \to \infty$$

- Точно можно представить только числа вида $x/2^k$

- Остальные рациональные числа представляются периодическими двоичными дробями:

$$1/5 = 0.(0011)_2$$

- Иррациональные числа представляются апериодическими двоичными дробями и могут быть представлены только приближенно

Представление чисел с плавающей точкой (IEEE 754)

- Числа с плавающей точкой представляются в нормализованной форме: $(-1^s)\ M\ 2^e$
  - s – код знака числа (он же знак мантиссы)
  - M – мантисса ($1 \le M < 2$)
  - e – (двоичный) порядок

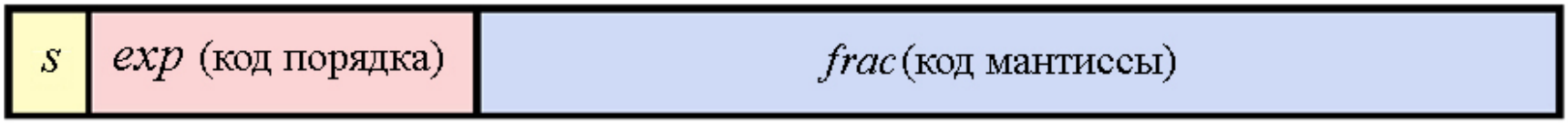
- Первая цифра мантиссы в нормализованном представлении всегда 1. В стандарте принято решение не записывать в представление числа эту единицу (тем самым мантисса как бы увеличивается на разряд).

  Экономия связана с тем, что в представление числа записывается не M, а $frac = M - 1$

Представление чисел с плавающей точкой

- Чтобы не записывать отрицательных чисел в поле порядка, вводится ***смещение*** $bias = 2^{k-1} - 1$, где k – количество бит в поле для записи порядка, и вместо порядка e записывается код порядка exp, связанный с e соотношением $e = exp - bias$.

- Нормализованное число $(-1^s)\ M\ 2^e$ упаковывается в машинное слово (структуру) с полями s, $frac$ и exp



Ширина поля s всегда равна 1.

Ширина полей exp и $frac$ зависит от точности числа

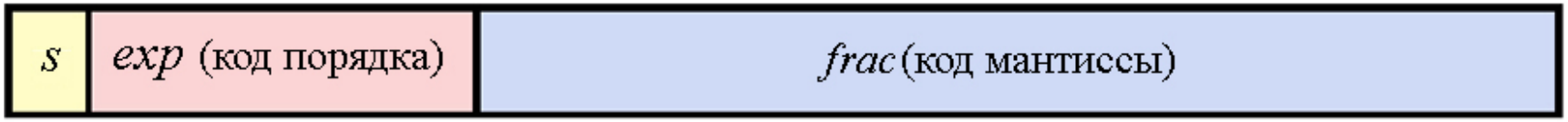
Представление чисел с плавающей точкой

- Одинарная точность (32 бита):

8 бит 23 бита

$bias = 127;$ $-126 \le e \le 127;$ $1 \le exp \le 254$

- Двойная точность (64 бита):



11 бит 52 бита

$bias = 1023;$ $-1022 \le e \le 1023;$ $1 \le exp \le 2046$

- Повышенная точность (80 бит):

15 бит 64 бита

Представление чисел с плавающей точкой

- Пример
  - **Значение** `float f = 15213.0`

    15213_{10} = 11101101101101_2 =

    $1.1101101101101_2 \times 2^{13}$
  - **Значащая часть**

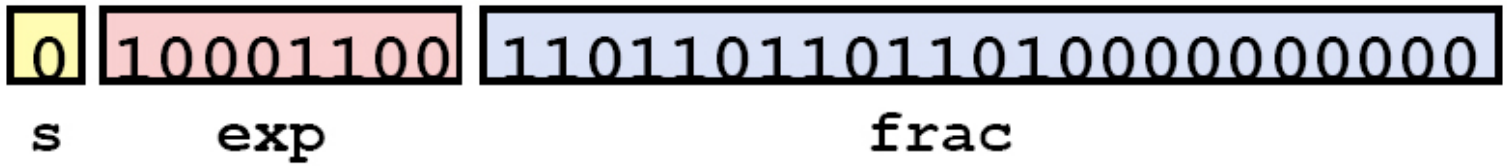
    $M = 1.\underline{1101101101101}_2$,

    $frac = \underline{1101101101101}0000000000_2$
  - **Порядок**

    $e = 13$

    $bias = 127$

    $exp = 140 = 10001100_2$
  - **Результат**



Представление нуля

- Для типа **`float`** код порядка **`exp`** изменяется от **`00000001`** до **`11111110`**
  (значению **`00000001`** соответствует порядок **`e = - 126`**,
  значению **`11111110`** – порядок **`e = 127`**)

- Код **`exp = 00000000, frac = 000…0`**
  представляет нулевое значение; в зависимости от значения знакового разряда **`s`** это либо **`+0`** либо **`-0`**

- А какое значение представляют коды
  `exp = 00000000, frac ≠ 000…0`?
  `exp = 11111111?`

Большие числа

Пусть **exp = 111…1**

- если при этом **frac = 000…0**, то коду будет соответствовать значение ∞ (со знаком **s**)
- если же **frac ≠ 000…0**, то код не будет представлять никакое число («значение», представляемое таким кодом, так и называется: «не число» – **NaN** – Not a number)

Денормализованные числа

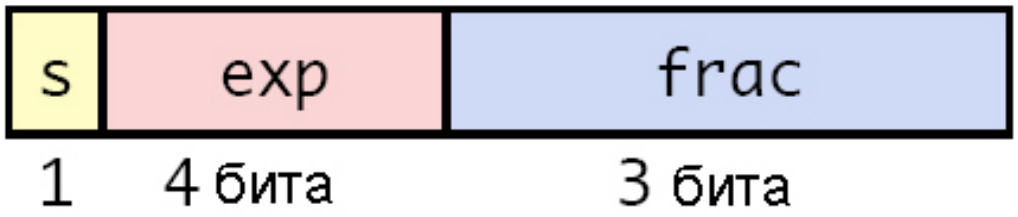
- Это числа, представляемые кодами

  exp = 00000000, frac ≠ 000…0

- **exp** вносит в значение такого числа постоянный вклад 2^{-k-2},

  frac меняется от **000…01** до **111…1 и** рассматривается уже не как мантисса, а как значение, умножаемое на **exp**

- Рассмотрим это на модельном примере:

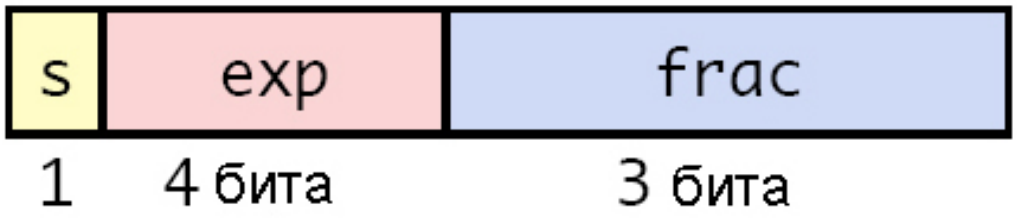


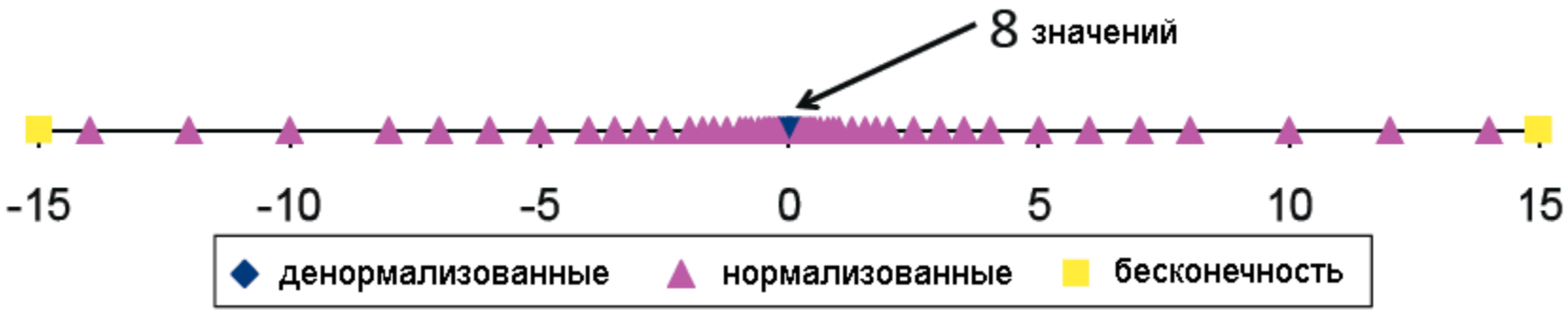
8-разрядные числа с плавающей точкой (положительные)

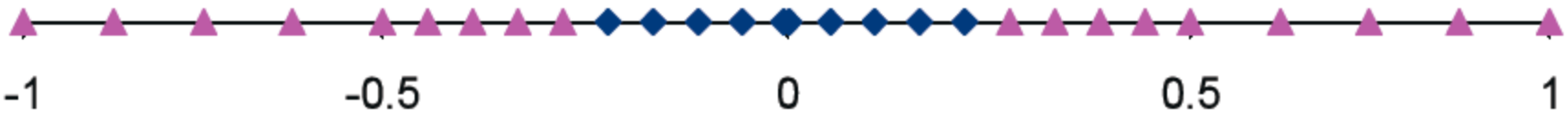
1 4 бита 3 бита

	s	exp	frac	E	Value	
Ненормализованные числа	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	Близкие к 0
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	Наибольшее ненормализованное
Нормализованные числа	0	0001	000	-6	8/8*1/64 = 8/512	Наименьшее нормализованное
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	Ближайшее к 1 снизу
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	Ближайшее к 1 сверху
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	Наибольшее нормализованное
	0	1111	000	n/a	inf	

8-разрядные числа с плавающей точкой



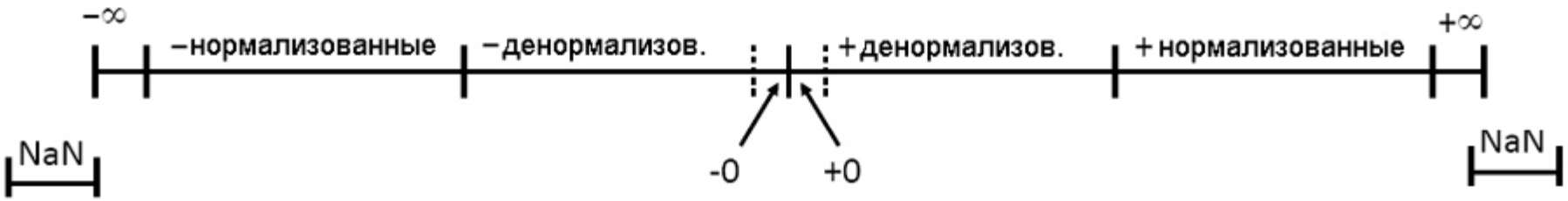




Центральная область более крупно

Важные частные случаи

	exp	frac	Численное значение
◈ Нуль	00…00	00…00	0.0
◈ Наим. положит. денорм.	00…00	00…01	
♦ float ≈ 1.4×10^{-45}			$2^{-23}\times2^{-126}$
♦ double ≈ 4.9×10^{-324}			$2^{-52}\times2^{-1022}$
◈ Наиб. положит. денорм.	00…00	11…11	
♦ float ≈ 1.18×10^{-38}			$(1.0 - \varepsilon)\times2^{-126}$
♦ double ≈ 2.2×10^{-308}			$(1.0 - \varepsilon)\times2^{-1022}$
◈ Наим. положит. норм.	00…01	00…00	
♦ float			1.0×2^{-126}
♦ double			1.0×2^{-1022}
◈ Единица	01…11	00…00	1.0
◈ Наиб. положит. норм.			
♦ float ≈ 3.4×10^{38}			$(2.0 - \varepsilon)\times2^{127}$
♦ double ≈ 1.8×10^{308}			$(2.0 - \varepsilon)\times2^{1023}$



Операции над числами с плавающей точкой

- $x +_{FP} y = Round(x + y)$

  $x \times_{FP} y = Round(x \times y)$

  где $Round()$ означает округление

- Выполнение операции
  - Сначала вычисляется точный результат (получается более длинная мантисса, чем запоминаемая, иногда в два раза)
  - Потом фиксируется исключение (например, переполнение)
  - Потом результат округляется, чтобы поместиться в поле $frac$

Умножение чисел с плавающей точкой

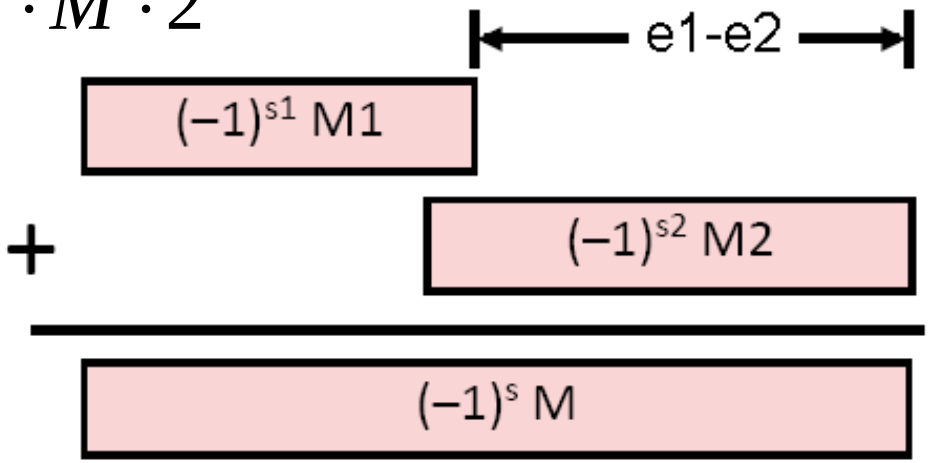
- $(-1)^{s1} \cdot M1 \cdot 2^{e1} \times (-1)^{s2} \cdot M2 \cdot 2^{e2}$
- Точный результат $(-1)^{s} \cdot M \cdot 2^{e}$
  - Знак s $s1 \wedge s2$
  - Значащие цифры M $M1 \times M2$
  - Порядок e $e1 + e2$
- Преобразование
  - Если $M \geq 2$, сдвиг M вправо с одновременным увеличением e
  - Если e не помещается в поле exp, переполнение
  - Округление M, чтобы оно поместилось в поле $frac$
- Основные затраты на перемножение мантисс

Сложение чисел с плавающей точкой

- $(-1)^{s1} \cdot M1 \cdot 2^{e1} + (-1)^{s2} \cdot M2 \cdot 2^{e2}$

  Пусть $e1 > e2$

- Точный результат $(-1)^{s} \cdot M \cdot 2^{e}$
  - Знак s и значащие цифры M вычисляются как показано на рисунке
  - Порядок суммы – $e1$



- Преобразование
  - Если $M \geq 2$, 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, переполнение
  - Округление M, чтобы оно поместилось в поле $frac$

Плавающие типы языка Си

`float`, `double`, `long double`

- ***Операции над данными с плавающей точкой***.
  - *Одноместные*: изменение знака («одноместный минус»: **–**), одноместный плюс (**+**).
  - *Двухместные*: сложение (**+**), вычитание (**–**), умножение (*****), деление (**/**).
- ***Порядок выполнения арифметических операций в выражениях (приоритет).***
  - самый низкий приоритет у двуместных **+** и **–**,
  - более высокий приоритет у двуместных ***** и **/**,
  - еще более высокий приоритет у одноместных **+** и **–**.
  - В выражениях без скобок операции с более высоким приоритетом выполняются раньше.
  - Скобки позволяют изменить порядок выполнения операций.

Пример 1. Вычисление суммы 5 чисел типа **`float`**
(мантисса – 6 десятичных цифр, порядок – 2 десятичных цифры):

$0.231876*10^{02} + 0.645391*10^{-03} + 0.231834*10^{-01} + 0.245383*10^{-02} + 0.945722*10^{-03} =$

a) $\mathbf{0.231876*10^{02} + 0.645391*10^{-03} + 0.231834*10^{-01} + 0.245383*10^{-02} + 0.945722*10^{-03} = 0.2321\underline{4}7*10^{02}}$;

23.1876 + 0.000645391 = 23.188245391 = 23.1882 = 0. $231882*10^{02}$;

23.1882 + 0.0231834 = 23.2113834 = 23.2114 = $0.232114*10^{02}$;

23.2114 + 0.00245383 = 23.21385383 = $23.2138*10^{02}$;

23. 2138 + 0.000945722 = 23.214745722 = 23.2147 = $0.232147*10^{02}$;

b) $\mathbf{0.645391*10^{-03} + 0.9457*10^{-03} + 0.245383*10^{-02} + 0.231834*10^{-01} + 0.231876*10^{02} = 0.2321\underline{5}7*10^{02}}$;

0.000645391 + 0.000945722 = 0.001591113 = 0.00159111 = 0. $159111*10^{-02}$;

0.00159111 + 0.00245383 = 0.00494493 = 0. $494493*10^{-02}$;

0.00494493 + 0.0231834 = 0.02812833 = 0.0281283 = 0. $281283*10^{-01}$;

0.0281283 + 23.1876 = 23.2157283 = 23.2157 = $0.232157*10^{02}$;

Пример 2. Вычисление разности плавающих чисел

(мантисса – 6 десятичных цифр, порядок – 2 десятичных цифры):

$0.238617*10^{02} – 0.238616*10^{02} + 0.645391*10^{04} – 0.645392*10^{04} + 0.845791*10^{00} – 0.835790*10^{00} =$

a) $0.238617*10^{02} – 0.238616*10^{02} + 0.645391*10^{04} – 0.645392*10^{04} + 0.845791*10^{00} – 0.835790*10^{00} =$ **$0.100000*10^{-05}$**

$0.238617*10^{02} – 0.238616*10^{02} = 23.8617 – 23.8616 = 0.0001 =$ **$0.100000*10^{03}$**

$0.100000*10^{-03} + 0.645391*10^{04} = 0.0001 + 6453.91 = 6453.9101 =$ **$0.645391*10^{04}$**

$0.645391*10^{04} – 0.645392*10^{04} = – 0.000001*10^{04} =$ **$– 0.100000*10^{-01}$**

$– 0.100000*10^{-01} + 0.845791*10^{00} = – 0.01 + 0.845791 = 0.835791 *10^{00}$

$0.835791 *10^{00} – 0.835790*10^{00} =$ **$0.000001*10^{00} = 0.100000*\underline{10^{-05}}$**

b) $0.238617*10^{02} + 0.645391*10^{04} + 0.845791*10^{00} – (0.238616*10^{02} + 0.645392*10^{04} + 0.835790*10^{00}) =$ **$0.100000*10^{00}$**

$0.238617*10^{02} + 0.645391*10^{04} = 23.8617 – 6453.91 = 6478.6 =$ **$0.647777*10^{04}$**

$0.647777*10^{04} + 0.845791*10^{00} = 6477.77 + 0.845791 = 6478.615791 =$ ***$0.647862*10^{04}$***

$0.238616*10^{02} + 0.645392*10^{04} = 23.8616 + 6453.92 = 6477.7816 =$ **$6477.78*10^{04}$**

$6477.78*10^{04} + 0.835790*10^{00} = 6477.78 + 0.835790 = 6478.61579 =$ ***$0.647852*10^{04}$***

$0.647862*10^{04} – 0.647852*10^{04}=$ **$0.000010*10^{04} = 0.100000*\underline{10^{-00}}$**

Выводы

(1) ***При вычислении суммы чисел с одинаковыми знаками*** необходимо упорядочить слагаемые по возрастанию и складывать, начиная с наименьших слагаемых.

(2) ***При вычислении суммы чисел с разными знаками*** необходимо сначала сложить все положительные числа, потом – все отрицательные числа и в конце выполнить одно вычитание.

(3) ***Вычитание*** (сложение чисел с противоположными знаками) ***часто приводит к потере точности***, которая у чисел с плавающей точкой определяется 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*.**

Значащими цифрами числа с плавающей точкой называются все цифры его мантиссы за исключением нулей, стоящих в ее конце. Например, у числа 0.**6700089**0000 * 10^3 все цифры, выделенные жирным шрифтом, значащие. При вычитании двух близких чисел почти все значащие цифры пропадают. Например, 0.67000890 * 10^3 - 0.67000880 * 10^3 = 0.00000010 * 10^3 = 0.10 * 10^{-4}. Таким образом, у результата всего одна значащая цифра, хотя у операндов было по 7 значащих цифр.

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 10

Поразрядные операции

- **&** (поразрядное И)
- **|** (поразрядное включающее ИЛИ)
- **^** (поразрядное исключающее ИЛИ)
- **<<** (сдвиг влево)
- **>>** (сдвиг вправо)
  - Беззнаковое число – заполнение нулями
  - Знаковое число – заполнение значением знакового разряда («арифметический сдвиг») или нулями («логический сдвиг»)
- **~** (дополнение до 1, или *инверсия*)

Тип «степень множества» (булеан)

- Пусть U – множество. Множество всех подмножеств множества U называется *степенью множества* U.
- Пусть B – степень конечного множества U. Тогда $|B| = 2^{|U|}$.
- *Характеристической функцией* χ_C подмножества C множества U называется функция, принимающая значение 1 на элементах U, входящих в состав C, и значение 0 на остальных элементах U.
- Множества удобно задавать через их характеристические функции. При этом в зависимости от количества элементов базового множества U его характеристическая функция, кодированная битами целого типа, может иметь тип

  `unsigned char` (если $|U| \le 8$)

  `unsigned int` (если $|U| \le 32$)

  `unsigned long long` (если $|U| \le 64$)

Тип «степень множества»

- **Пример**. Пусть $U = \{r, o, y, g, c, b, v, w\}$. Тогда его подмножества задаются переменными типа **`unsigned char`**: $|B| = 2^{|U|}$

  $\{\}$ задается значением **`00000000`**,

  $\{r, y, g\}$ – значением **`10110000`**

  $\{r, o, y, g, c, b, v, w\}$– значением **`11111111`**

буквы r, o, y, g, c, b, v, w являются первыми буквами семи цветов спектра и белого цвета:

$r - red$	$c - cyan$
$o - orange$	$b - blue$
$y - yellow$	$v - violet$
$g - green$	$w - white$

Реализация операций над множествами с помощью поразрядных операций

- **&** (поразрядное И) соответствует пересечению множеств:

  $B = \{r, y, g, b, w\}$, $C = \{r, o, y, g, v\}$

  χ_B **= 10110101,** χ_C **= 11110010**

  χ_B **&** χ_C **= 10110101 & 11110010 = 10110000**

  $B \cap C = \{r, y, g\}$

Реализация операций над множествами с помощью поразрядных операций

- **|** (поразрядное включающее ИЛИ) соответствует объединению множеств:

$B = \{r, y, g, b, w\}$, $C = \{r, o, y, g, v\}$

χ_B **= 10110101,** χ_C **= 11110010**

χ_B **|** χ_C **= 10110101 | 11110010 = 11110111**

$B \cup C = \{r, o, y, g, b, v, w\}$

Реализация операций над множествами с помощью поразрядных операций

- ~ (инверсия) соответствует дополнению до множества U:

  $B = \{r, y, g, b, w\}$

  χ_B **= 10110101**

  $\chi_{\sim B}$ **= 01001010**

  $\sim B = \{o, c, v\}$

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 *метка структуры* **{** *поля структуры* **};**

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

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**)

Структуры

- Доступ к полям структуры: операция точка "**.**"
  - **f.x, g.y, r.pt1.x**
- Присваивание структур целиком: **f = g;**
- Массивы структур

```
#define NREСT 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->pt1 (= (*pr).pt1), pr->pt2.x
- Адресная арифметика:

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

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 11

Составные инициализаторы структур (C99)

- ```
  struct rect r;
  r = (struct rect) { {4, 4},
                      {7, 6}};
  ```
- Составной инициализатор генерирует **lvalue**!

  Т.е. 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}});
```
```

Приоритеты операций

Операции	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
- Можно выполнять те же операции, что и со структурами

Анонимные объединения и структуры (C11)

- 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.i);
    break;
  case CF:
    printf("%f",sc.f);
    break;
  case CS: puts(sc.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, набор 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;
}
```

  - Адрес битового поля брать запрещено
  - Можно объявить анонимные поля (для выравнивания)
  - Можно объявить битовое поле ширины 0 (для перехода на следующий байт)

Перечисления

- Перечисления – целочисленные типы данных, определяемые программистом.
- Определение перечисления:

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

- Значения перечисления нумеруются с 0, но можно присваивать свои значения

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

- Доступны операции над целочисленными типами и объявление указателей на переменные перечислимых типов
- Проверка корректности присваиваемых значений не производится

Программа: вычисление площадей фигур

- Фигура определяется общей структурой, в которой объединены различные конкретные структуры для определенных фигур и поле тега фигуры
- Используется перечисление для задания возможных типов фигур
- Используется адресная арифметика для обхода массива считанных фигур
- Для обработки фигуры используется оператор выбора по тегу фигуры

Программа: типы данных

```
enum figure_tag {
  CIRCLE,
  RECTANGLE,
  TRIANGLE,
  LAST_FIGURE
};
struct point {
  double x, y;
};
struct fcircle {
  struct point center;
  double radius;
};
struct frectangle {
  struct point ll, ur;
};
```

Программа: типы данных

```
struct ftriangle {
  struct point a, b, c;
};
struct figure {
  enum figure_tag tag;
  union {
    struct fcircle fc;
    struct frectangle fr;
    struct ftriangle ft;
  } u;
};
struct farea {
  double min, max;
  int initialized : 1;
};
```

Программа: основная функция

```
int main (void) {
  enum { MAXFIG = 128 };
  struct figure figs[MAXFIG + 1];
  struct farea areas[LAST_FIGURE];
  int i = 0;

  while (i < MAXFIG)
    if (!read_figure (&figs[i]))
      break;
    else
      i++;

  if (i == 0)
    return 1;

  <... Обработка данных и вывод ...>
}
```

Программа: основная функция

```
int main (void) {
  enum { MAXFIG = 128 };
  struct figure figs[MAXFIG + 1];
  struct farea areas[LAST_FIGURE];
  int i = 0;

  <... Считывание данных ...>

  figs[i].tag = LAST_FIGURE;
  for (i = CIRCLE; i < LAST_FIGURE; i++) {
    areas[i].initialized = 0;
    calculate_minmax_areas (i, figs, &areas[i]);
    output_minmax_areas (i, &areas[i]);
  }
  return 0;
}
```

Программа: подсчет площади

```
static void calculate_minmax_areas (enum figure_tag tag,
  struct figure *f, struct farea *fa) {
  for (; f->tag != LAST_FIGURE; f++)
    if (f->tag == tag) {
      double a = calculate_area (f);
      if (!fa->initialized) {
        fa->min = fa->max = a;
        fa->initialized = 1;
      } else {
        if (a < fa->min)
          fa->min = a;
        if (a > fa->max)
          fa->max = a;
      }
    }
}
```

Программа: подсчет площади

```
static double calculate_area (struct figure *f) {
  double a = 0.0;

  switch (f->tag) {
    case CIRCLE: a = f->u.fc.radius * f->u.fc.radius *
      M_PI; break;
    case RECTANGLE: a = fabs ((f->u.fr.ll.x –
      f->u.fr.ur.x) * (f->u.fr.ll.y - f->u.fr.ur.y));
      break;
    case TRIANGLE: <...>
    default: assert (0);
  }
  return a;
}
```

Программа: подсчет площади

```
static double calculate_area (struct figure *f) {
  double a = 0.0;
  switch (f->tag) {
    <...>
    case TRIANGLE: {
      double x1, y1, x2, y2;
      x1 = f->u.ft.b.x - f->u.ft.a.x;
      y1 = f->u.ft.b.y - f->u.ft.a.y;
      x2 = f->u.ft.c.x - f->u.ft.a.x;
      y2 = f->u.ft.c.y - f->u.ft.a.y;
      a = fabs (x1 * y2 - x2 * y1) / 2;
      break;
    }
  <...>
  }
  return a;
}
```

Программа: ввод

```
static int read_figure (struct figure *f) {
  char name;

  if (scanf (" %c", &name) != 1)
    return 0;

  switch (toupper (name)) {
    case 'C': f->tag = CIRCLE; break;
    case 'R': f->tag = RECTANGLE; break;
    case 'T': f->tag = TRIANGLE; break;
    default: return 0;
  }
  <...>
  return 1;
}
```

Программа: ввод

```
static int read_figure (struct figure *f) {
  <...>
  switch (f->tag) {
    case CIRCLE: if (scanf ("%lf%lf%lf", &f->u.fc.center.x,
      &f->u.fc.center.y, &f->u.fc.radius) != 3)
                    return 0;
                  break;
    case RECTANGLE: if (scanf ("%lf%lf%lf%lf",
      &f->u.fr.ll.x, &f->u.fr.ll.y, &f->u.fr.ur.x,
      &f->u.fr.ur.y) != 4)
                      return 0;
                    break;
    case TRIANGLE:  if (scanf ("%lf%lf%lf%lf%lf%lf",
      &f->u.ft.a.x, &f->u.ft.a.y, &f->u.ft.b.x,
      &f->u.ft.b.y, &f->u.ft.c.x, &f->u.ft.c.y) != 6)
                      return 0;
                    break;
  }
```

Программа: вывод и заголовочные файлы

```
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <assert.h>
static void output_minmax_areas (enum figure_tag ft,
  struct farea *fa) {
  static char *fnames[LAST_FIGURE] = {"CIRCLE",
  "RECTANGLE", "TRIANGLE"};
  if (!fa->initialized)
    printf ("No figures of type %s were met\n",
            fnames[ft]);
  else {
    printf ("Minimal area of %s(s): %.5f\n", fnames[ft],
            fa->min);
    printf ("Maximal area of %s(s): %.5f\n", fnames[ft],
            fa->max);
  }
}
```

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 12

Динамическое распределение памяти

- Функция

  `void *malloc (size_t size)`

  выделяет область памяти 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`**.

- Тривиальные примеры:

  (1) Выделение непрерывного участка памяти объемом 1000 байтов:

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

  (2) Выделение памяти для 50 целых:

```
int *p;
/* явное приведение типа необязательно */
p = malloc (50 * sizeof (int));
```

Динамическое распределение памяти

- Функция

  `void free (void *p)`

  возвращает системе выделенный ранее участок памяти с указателем **`p`**.

- ***Важное замечание***: Аргументом функции **`free()`** обязательно должен быть указатель **`p`** на участок памяти, выделенной ранее функцией **`malloc()`**.
  Вызов функции **`free()`** с неправильным указателем не определен и может привести к разрушению системы распределения памяти.
  Вызов функции **`free()`** с указателем **`NULL`** не приводит ни к каким действиям (C99).

- Функции **`malloc()`** и **`free()`** входят в состав библиотеки **`stdlib.h`**.

Динамическое распределение памяти

- ***Пример.*** Динамическое выделение памяти для строки:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main (void) {
  char *s;
  int t;

  s = (char *) 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]);
  free (s);
  return 0;
}
```

Динамическое распределение памяти

- ***Пример.*** Динамическое выделение памяти для двухмерного целочисленного массива (матрицы):

```
#include <stdio.h>
#include <stdlib.h>
long pwr (int a, int b) {
  long t = 1;
  for (; b; b--) t *= a;
  return t;
}

int main (void) {
  long *p[6];
  int i, j;
  for (i = 0; i < 6; i++)
    if (!(p[i] = malloc (4 * sizeof (long))))  {
      printf ("требуемая память не выделена. \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");
  }
  /* Цикл с освобождением памяти */
  return 0;
}
```

Динамическое распределение памяти

- ***Пример.*** Динамическое выделение памяти для двухмерного целочисленного массива (матрицы):

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

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

int main (void) {
  long *p[6];
  int i, j;
  for (i = 0; i < 6; i++)
    if (!(p[i] = malloc (4 * sizeof (long))))  {
      printf ("требуемая память не выделена. \n");
      exit(1);
    }
```

Динамическое распределение памяти

- ***Пример.*** Динамическое выделение памяти для двухмерного целочисленного массива (матрицы):

```
  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");
  }
  for (i = 0; i < 6; i++)
    free (p[i]);
  return 0;
}
```

VLA-массивы

- В Си-89 размер массива обязан являться константой. Это неудобно при передаче массивов (многомерных) в функции:

```
/* 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;
}
/* 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];
  return s;
}
```

VLA-массивы

- В Си-99 размер массива автоматического класса памяти может задаваться во время выполнения программы:

```
int foo (int n) {
  int a[n];
  <... Можно обрабатывать a[i]...>
}
/* можно передать 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;
}
```

- Объявление функции asum2d:

```
int asum2d (int m, int n, int a[m][n]);
int asum2d (int, int, int [*][*]);
```

VLA-массивы и динамическое выделение памяти

- 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);
}
```

- Можно выделить VLA-массив в динамической памяти:

```
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);
}
```

Динамическое распределение памяти

- Состав функций динамического распределения памяти библиотеки **stdlib** (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
  void *realloc (void *p, size_t size)
  согласно стандарту Си99 сначала выполняет **free (p)**, а потом **p = malloc (size)**, возвращая новое значение указателя **p**. При этом значения первых **size** байтов новой и старой областей совпадают.
- Функция
  void *calloc (size_t num, size_t size)
  работает аналогично функции **malloc (size1)**, где **size1** = **num * size** (т.е. выделяет память для размещения массива из **num** объектов размера **size**).

  Выделенная память инициализируется нулевыми значениями

Массив переменного размера в структуре (C99)

- Flexible array member – последнее поле структуры:

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

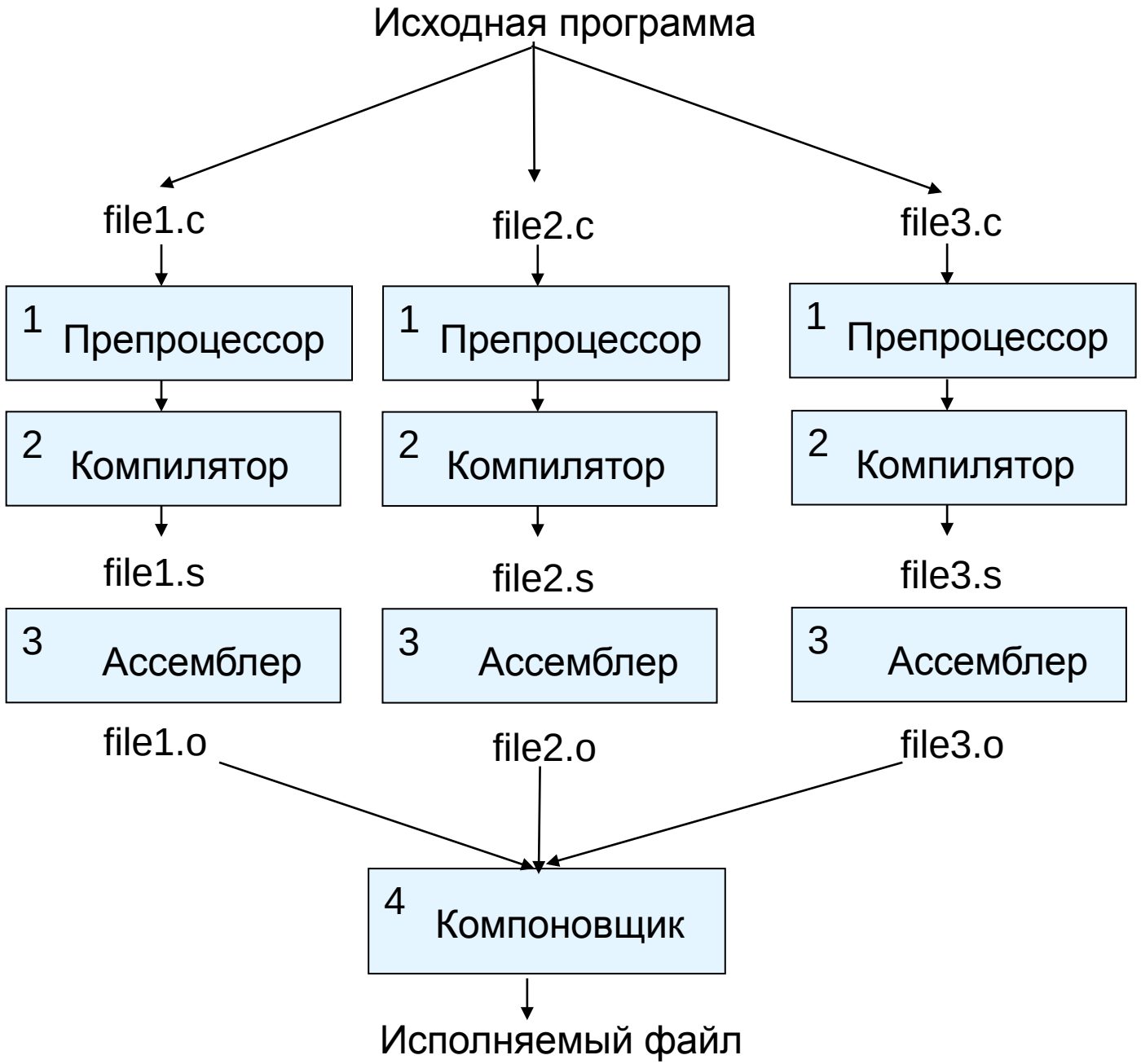
- Варьирование размера переменного массива:

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

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 13

Схема компиляции



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(parameters) text
#undef name

#define MAX 128
#define ABS(x) ((x) >= 0 ? (x) : -(x))

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

x -> a-- ?
```

Препроцессор и условная компиляция

- Препроцессор позволяет организовать условное включение фрагментов кода в программу

#ifdef name / #endif – проверка определения имени

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

Препроцессор и условная компиляция

- Препроцессор позволяет организовать условное включение фрагментов кода в программу

#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_LONG >= 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
```

Препроцессор: операции # и

- 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;
}

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

Препроцессор: операции # и

- Операция ## позволяет объединить фактические аргументы макроса в одну строку

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)
JAVAOP (impdep2,        255, IMPL,    ANY,    2)
```

Препроцессор: операции # и

- Операция ## позволяет объединить фактические аргументы макроса в одну строку

```
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
};
```

Компоновка и классы памяти

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
- Классы памяти функций:
  - статическая (объявлена с квалифатором **`static`**)
  - внешняя (**`extern`**), по умолчанию
  - встраиваемая (**`inline`**, C99)
- Объявление внешних функций в заголовочных файлах:

```
extern void *realloc (void *ptr, size_t size);
```

Компоновщик

- Организовывает слияние нескольких объектных файлов в одну программу
- Разрешает неизвестные символы (внешние переменные и функции)
  - Глобальные переменные с одним именем получают одну область памяти
  - Ошибки, если необходимых имен нет или есть несколько объектов с одним именем
  - Опции для указания места поиска
- Хорошим стилем программирования является экспорт лишь тех объектов, которые используются в других файлах (интерфейс модуля)
  - Используйте квалификатор **`static`**
- Сборка исполняемого файла или библиотеки (*статической* или *динамической*)

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 отсутствия ошибок
- Для отладки используют инструменты, позволяющие получить информацию о поведении программы на некоторых входных данных, не изменяя ее поведения
- Простейший метод: *отладочная печать*

```
int *a; int n;
n = read_array (a);
debug_array (a, n);
```

```
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");
}
```

- Отладочная печать может контролироваться макросом (**NDEBUG**)

Отладка программ: отладчики

- Отладчик – основной инструмент отладки программы
- Отладчик позволяет
  - запустить программу для заданных входных данных
  - останавливать выполнение по достижении заданных точек программы безусловно или при выполнении некоторого условия на значения переменных
  - останавливать выполнение, когда некоторая переменная изменяет свое значение
  - выполнить текущую строку исходного кода программы и снова остановить выполнение
  - посмотреть/изменить значения переменных, памяти
  - посмотреть текущий стек вызовов
- Необходимое условие для отладки на уровне исходного кода: наличие в исполняемом файле программы *отладочной информации* (связи между командами процессора и строками исходного кода программы, связь между адресами и переменными и т.д.)

Отладка программ: отладчик gdb

- Компиляция с отладочной информацией: **`gcc -g`**
- Некоторые 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
- Среда Code::Blocks поддерживает **`gdb`** в своем интерфейсе

Отладка программ: примеры команд gdb

- Установка точек останова
  - можно использовать `'.'` вместо `'->'`

```
b fancy_abort
b 7199
b sel-sched.c:7199
cond 2 insn.u.fld.rt_int == 112
cond 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 sched_verbose=5
call debug_vinsn (0x4744540)
```

- Установка точек наблюдения

```
wa can_issue_more
wa ((basic_block) 0x7ffff58b5680)->preds.base.prefix.num
```

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 14

Динамические структуры данных. Стек

- **Стек** (*stack*) – это динамическая последовательность *элементов*, количество которых изменяется, причем как добавление, так и удаление элементов возможно только с одной стороны последовательности (вершина стека).
- Работа со стеком осуществляется с помощью функций:

  `push(x)` – *затолкать* элемент **`x`** в стек;

  `x = pop()` – *вытолкнуть* элемент из стека.
- Стек можно организовать на базе:
  - 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 (например, список).

Организация стека на динамическом массиве

```
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;
}
```

Организация стека на динамическом массиве

```
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--];
}

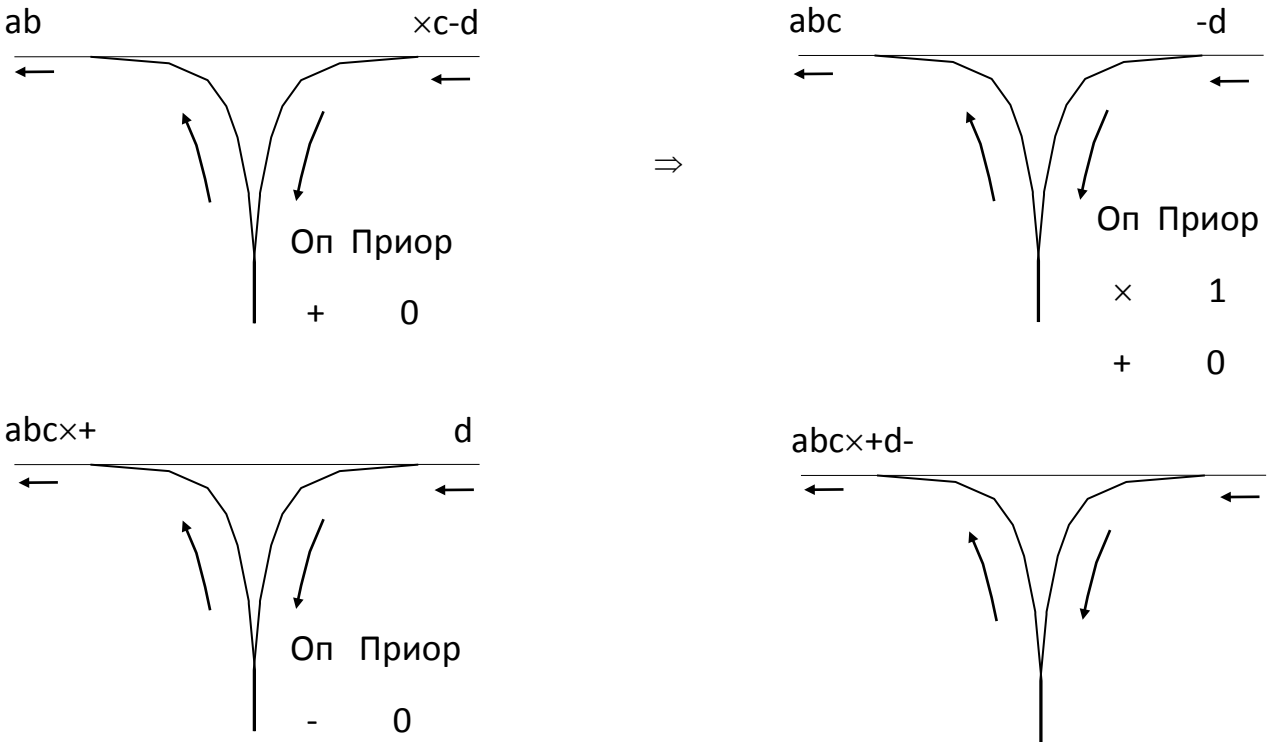
static int isempty (void) {
  return stack.sp == -1;
}
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись (постфиксную).

a + b × c - d → abc × + d -

c × (a + b) - (d + e)/f → cab + × de + f/-



Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

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

#include "stack.c"

/* Считывание символа-операции или переменной */
static char getop (void) {
  int c;
  while ((c = getchar ()) != EOF && isblank (c))
    ;
  return c == EOF || c == '\n' ? 0 : c;
}
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
/* Является ли символ операцией */
static int 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;
}
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
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);
    else <...>
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
/* Операция заносится в стек в зависимости от приоритета */
else if (isop (c)) {
      while (! isempty ()) {
        op = pop ();
        /* Заносим, если больший приоритет */
        if (prio (c) > prio (op)) {
          push (op); break;
        } else
        /* Иначе выталкиваем операцию из стека */
          putchar (op);
      }
      push (c);
    } else <...>
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
/* 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;
}
```

Организация стека как библиотеки

◊ 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
      = { <...> };

void push (char c ) { <...> }
char pop (void) { <...> }
int isempty (void) { <...> }
```

◊ main.c:

```
#include "stack.h"

int main (void) {
  <...push (c), pop (), ...>
}
```

```
$gcc main.c stack.c –o main
```

Организация стека как библиотеки - II

◊ 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) <...>
}
<...>
```

Организация стека как библиотеки - III

◊ 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 *s) {
  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*.

  Для списка можно использовать статический массив: количество элементов массива (*MAX*) = наибольшей допустимой длине очереди.

- Работа с очередью осуществляется с помощью **двух функций**:

  `qstore()` – *поместить* элемент в конец очереди;

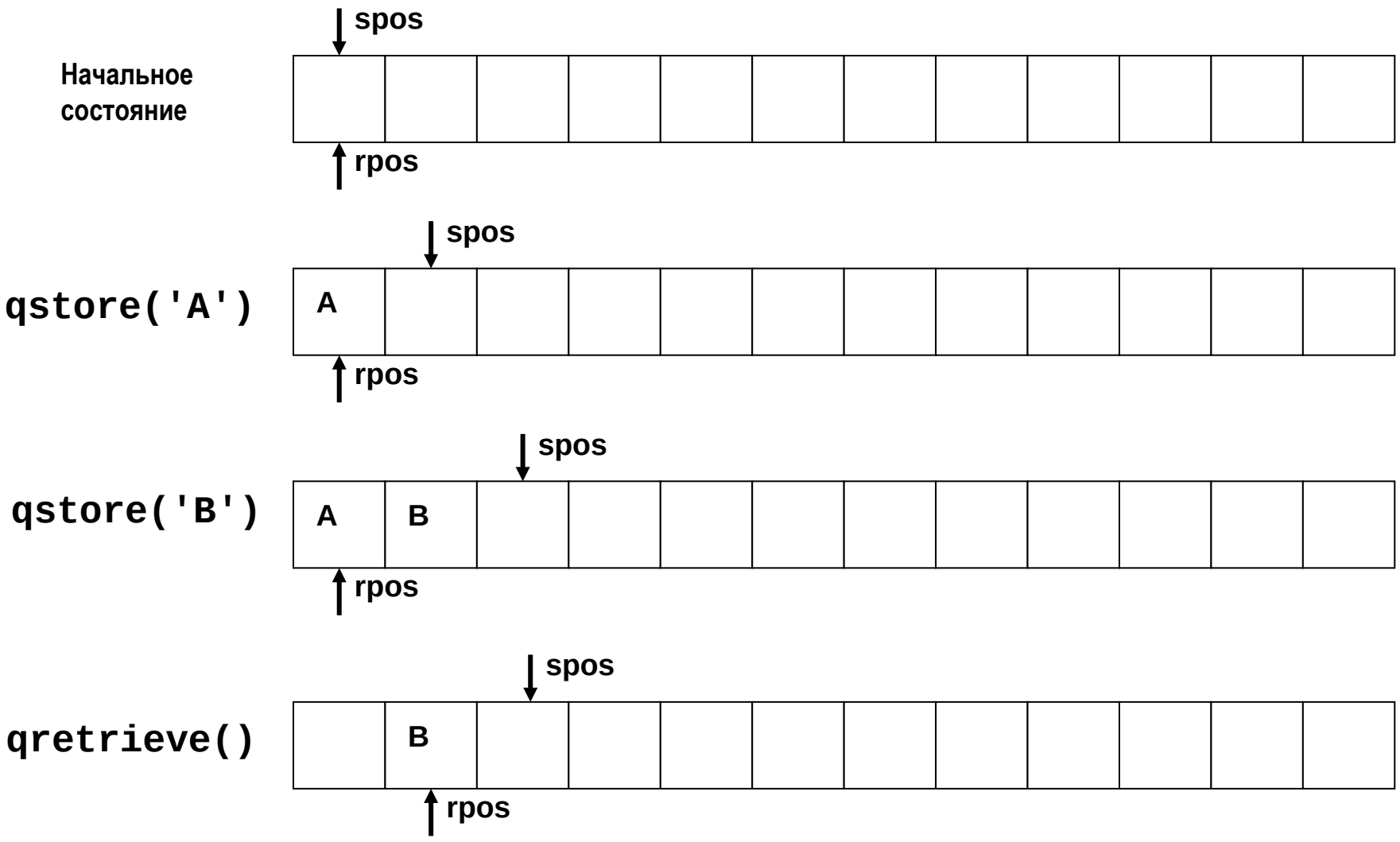
  `qretrieve()` – *удалить* элемент из начала очереди;

  и **двух глобальных переменных**:

  `spos` (индекс первого свободного элемента очереди: его значение < **`MAX`**)

  `rpos` (индекс очередного элемента, подлежащего удалению: «кто первый?»)

Очередь


spos
Начальное состояние
rpos
spos
qstore('A')
A
rpos
spos
qstore('B')
A
B
rpos
spos
qretrieve()
B
rpos

Очередь

- Тексты функций **qstore()** и **qretrieve()**

```
#define MAX    67
int queue[MAX];
int spos = 0, rpos = 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) {
    printf ("Очередь пуста \n");
    return -1;
  }
  return queue[rpos++];
}
```

Улучшение – «зацикленная» очередь

```
#define MAX    67
int queue[MAX];
int spos = 0, rpos = 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;
}
```

Улучшение – «зацикленная» очередь

```
int qretrieve (void) {
  if (rpos == spos) {
    printf ("Очередь пуста \n");
    return -1;
  }
  if (rpos == MAX - 1) {
    rpos = 0;
    return queue[MAX – 1];
  }
  return queue[rpos++];
}
```

- Зацикленная очередь переполняется, когда **spos** находится непосредственно перед **rpos**, так как в этом случае запись приведет к **rpos == spos**, т.е. к пустой очереди.

Списки

- ***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.

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

- Выделение элемента

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

Списки

- Добавление элемента в начало

```
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;
}
```

Списки

- Добавление элемента в конец

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

Списки

- Поиск элемента

```
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;
}
```

Списки

- Удаление элемента

```
struct list *remove (struct list *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;
}
```

Списки

- Удаление элемента (двойной указатель)

```
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);
}
```

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 15

Топологическая сортировка узлов ациклического ориентированного графа

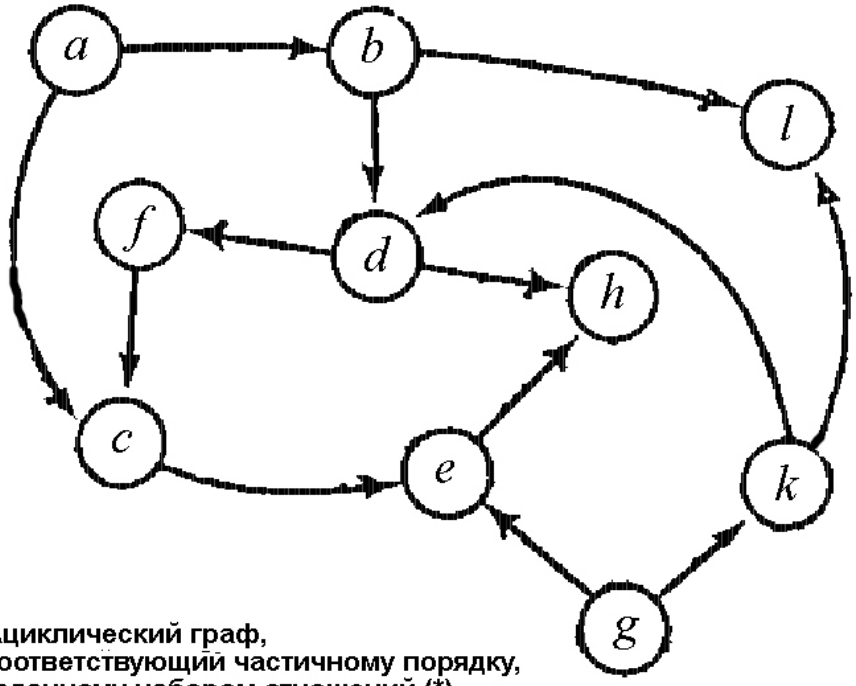
- Ациклический граф можно использовать для графического изображения *частично упорядоченного множества.*
  Цель топологической сортировки:
  преобразовать частичный порядок в линейный.
  Графически это означает, что
  все узлы графа нужно расположить на одной прямой таким образом, чтобы все дуги графа были направлены в одну сторону.

Топологическая сортировка узлов ациклического ориентированного графа

- ***Пример***. Частичный порядок (<) задается следующим набором отношений :

$$a < b,\ b < d,\ d < f,\ b < l,\ d < h,\ f < c,\ a < c, \qquad (*)$$
$$c < e,\ e < h,\ g < e,\ g < k,\ k < d,\ k < l$$

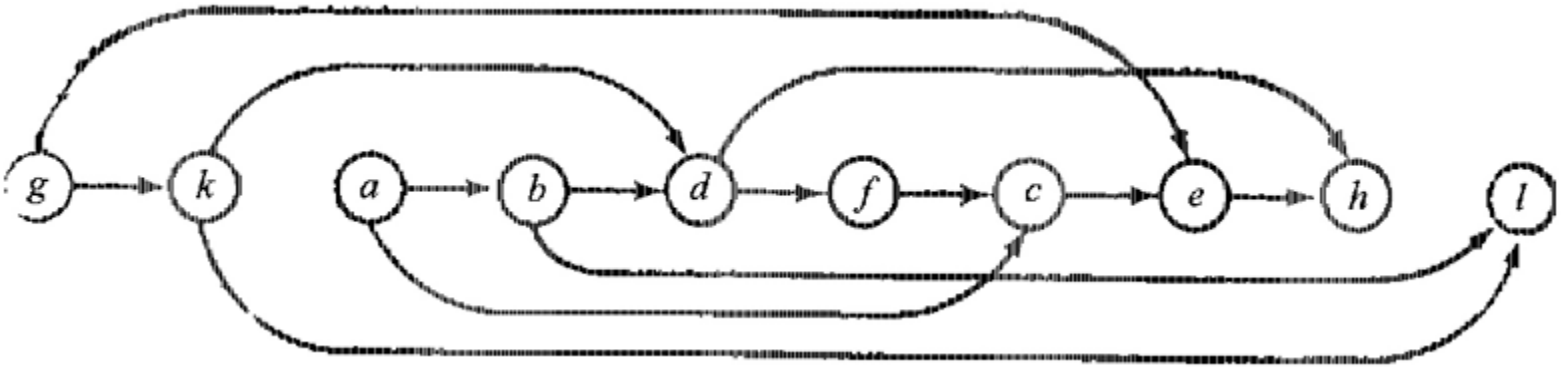
Набор отношений (*) можно представить в виде следующего ациклического графа (см. рисунок):



Ациклический граф, соответствующий частичному порядку, заданному набором отношений (*).

Топологическая сортировка узлов ациклического ориентированного графа

- Требуется привести рассматриваемый граф к линейному графу:



- На этом графе ключи расположены в следующем порядке:

g k a b d f c e h l

(поскольку топологическая сортировка неоднозначна, это один из возможных топологических порядков).

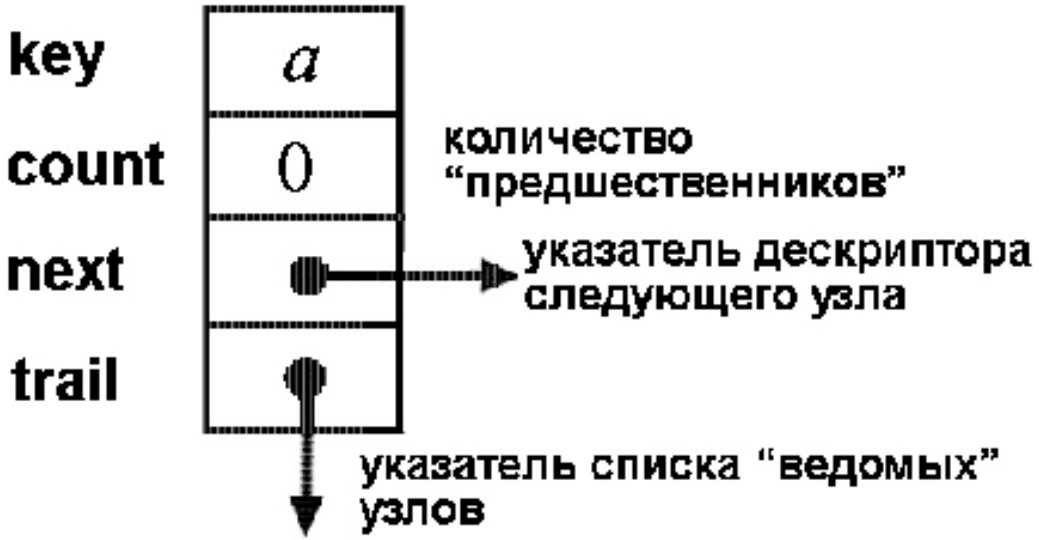
- Последовательная обработка полученного линейного списка узлов графа эквивалентна их обработке в порядке обхода графа.

Топологическая сортировка узлов ациклического ориентированного графа

- Поскольку рассматриваемый граф – ациклический, **существует** хотя бы один узел графа, у которого нет предшествующих узлов. Каждый такой узел будем называть *ведущим* узлом.
  Шаг алгоритма: Выберем один из ведущих узлов и поместим его в начало линейного списка отсортированных узлов, удалив его из исходного графа.
- Поскольку граф, у которого удалили один из ведущих узлов, останется ациклическим, в нем будет хотя бы один ведущий узел. Следовательно, можно повторить шаг алгоритма.
- Легко видеть, что каждый узел графа рано или поздно станет ведущим и попадет в формируемый линейный список, и алгоритм завершится через несколько шагов.

Топологическая сортировка узлов ациклического ориентированного графа

- Структуры данных для представления узлов:
  - Каждый узел исходного графа представляется с помощью дескриптора узла, который имеет вид:

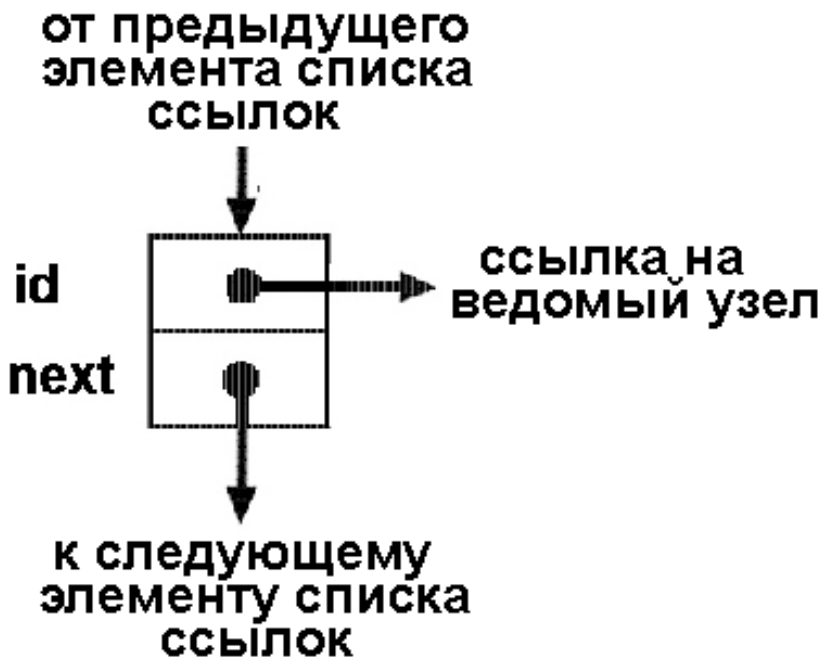


Дескриптор узла.

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

Топологическая сортировка узлов ациклического ориентированного графа

- Структуры данных для представления узлов:
  - Дескриптор каждого узла содержит ссылки на ведомые узлы. Так как заранее неясно, сколько у узла будет ведомых узлов, эти ссылки помещаются в список. На рисунке представлен элемент списка ссылок.



Топологическая сортировка узлов ациклического ориентированного графа

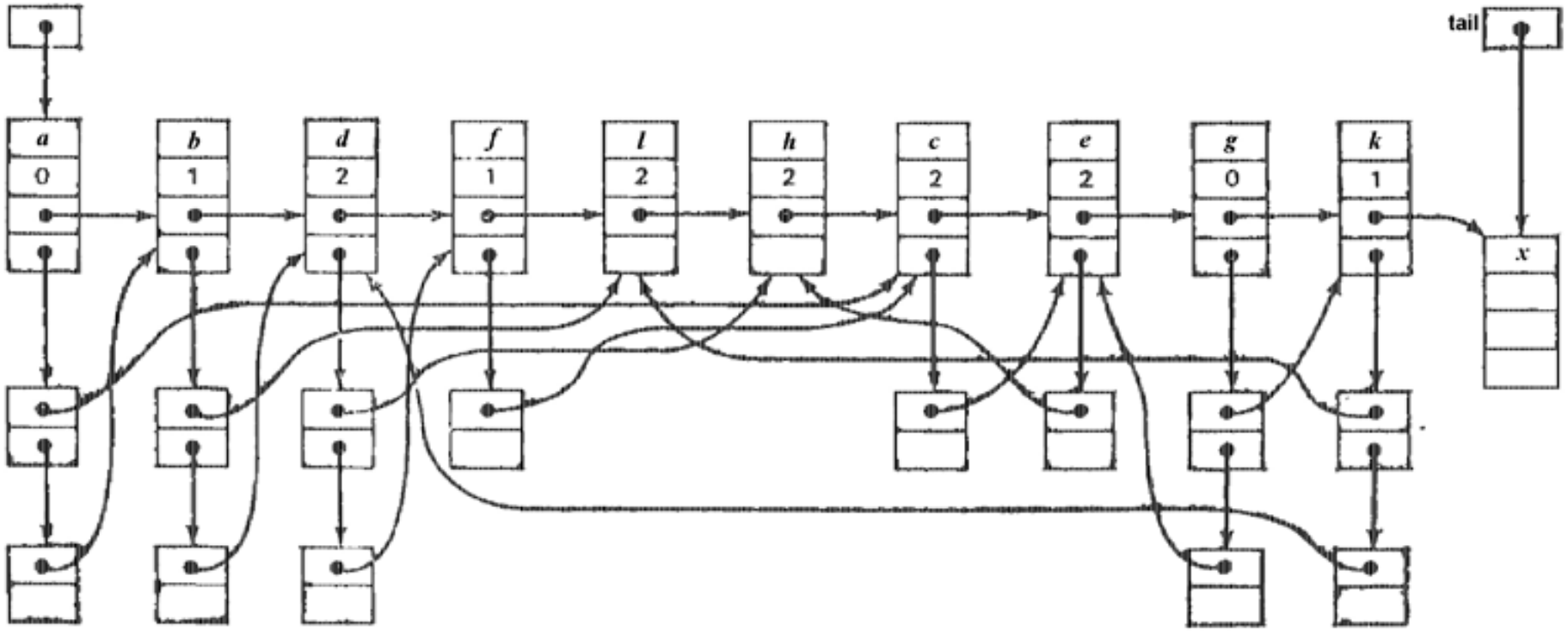
- ***1 фаза алгоритма: ввод исходного графа***

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

  - 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).

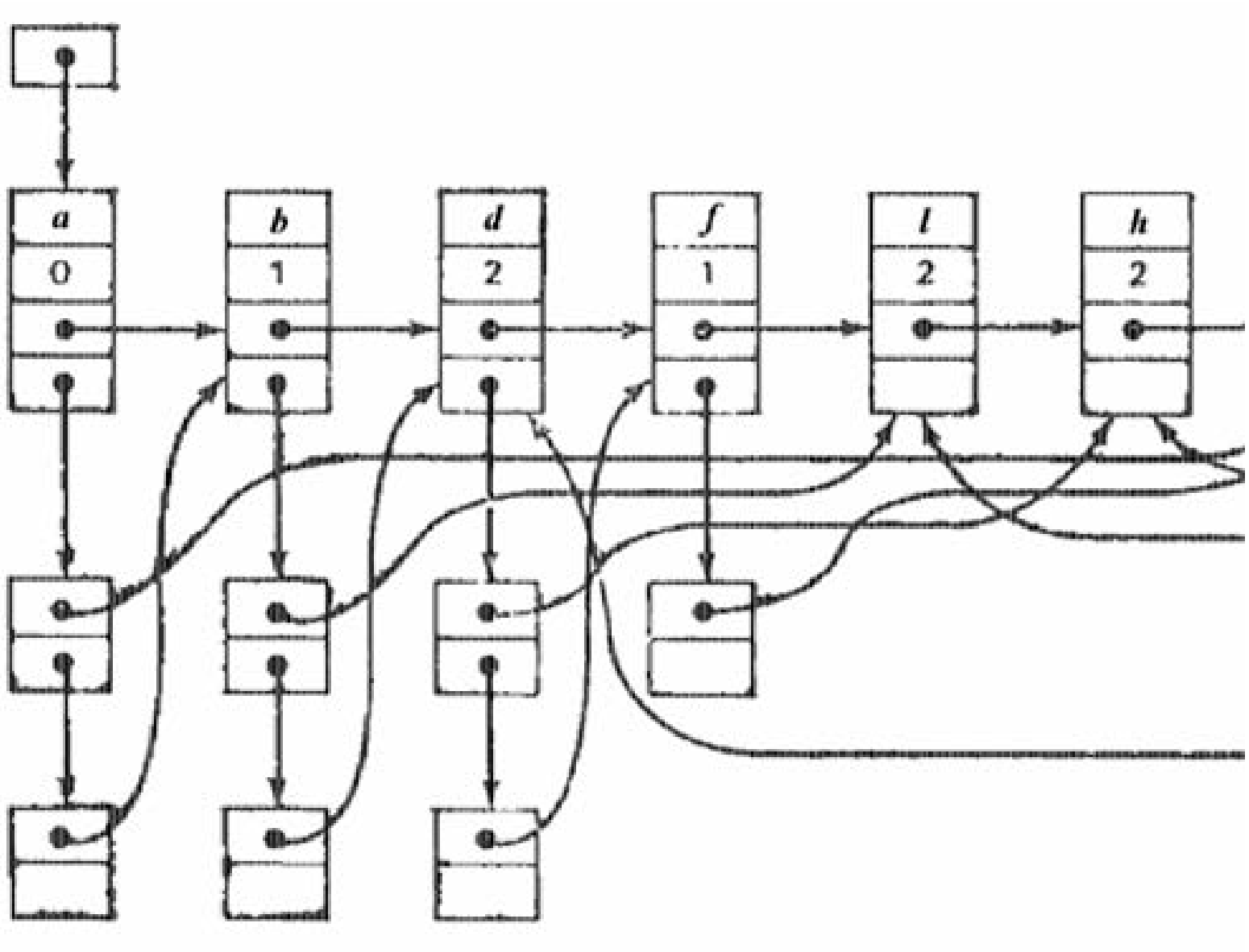
По окончании фазы ввода будет сформирована структура, показанная на следующем слайде (для множества пар ключей (*)).

Топологическая сортировка узлов ациклического ориентированного графа

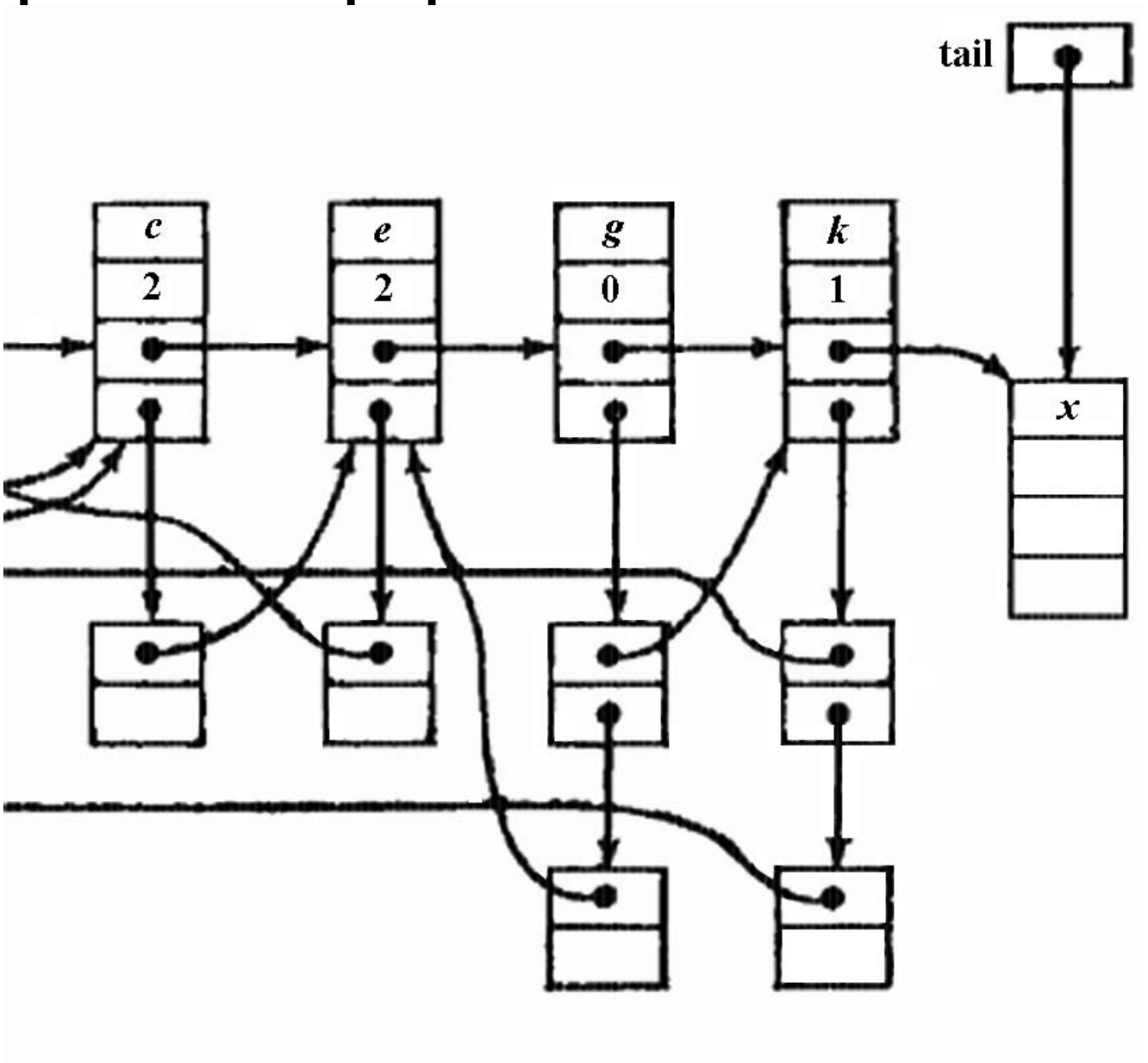


Структура данных, сформированная фазой ввода

Топологическая сортировка узлов ациклического ориентированного графа



Топологическая сортировка узлов ациклического ориентированного графа



Топологическая сортировка узлов ациклического ориентированного графа

- ***2 фаза алгоритма: сортировка***

  (1) В списке «ведущих» находим дескриптор узла ***z***, у которого значение поля **`count`** равно 0.

  (2) Включаем узел ***z*** в результирующую цепочку.

  (3) Если у узла ***z*** есть «ведомые» узлы (значение поля **`trail`** не **`NULL`**)

  - (a) просматриваем очередной элемент списка «ведомых» узлов
  - (b) корректируем поле **`count`** дескриптора соответствующего «ведомого» узла.

  (4) Переходим к шагу (1)

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

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 16

Описание алгоритма топологической сортировки на языке Си

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

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

Описание алгоритма топологической сортировки на языке Си

```
/* поиск по ключу w */

leader *find (char w) {
  leader *h = head;
  /* "барьер" на случай отсутствия w */
  tail->key = w;
  while (h->key != w)
    h = h->next;
  if (h == tail) {
    /* генерация нового ведущего узла */
    tail = (leader *) malloc (sizeof (leader));
    /* старый tail становится новым элементом списка */
    lnum++;
    h->count = 0;
    h->trail = NULL;
    h->next = tail;
  }
  return h;
}
```

Описание алгоритма топологической сортировки на языке Си

```
void init_list() {
  /* инициализация списка «ведущих» */
  leader *p, *q;
  trailer *t;
  char x, y;

  head = (leader *) malloc (sizeof (leader));
  tail = head;
  lnum = 0;          /* начальная установка */
  while (1) {
    if (scanf ("%c %c", &x, &y) != 2)
      break;
    /* включение пары в список */
    p = find (x);
    q = find (y);
    /* коррекция списка */
    t = (trailer *) malloc (sizeof (trailer));
    t->id = q;
    t->next = p->trail;
    p->trail = t;
    q->count += 1;
  }
}
```

Описание алгоритма топологической сортировки на языке Си

```
/* Исходный список построен. Организация нового списка */
void sort_list() {
  leader *p, *q;
  trailer *t;
  /* В выходной список включаются все узлы старого
     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;
    }
  }
<...>
```

Описание алгоритма топологической сортировки на языке Си

```
/* Фаза сортировки и вывода результатов из нового списка */


<...>
  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;
        q = p;
      }
      t = t->next;
      /* здесь можно удалить ведомый элемент */
    }
  }
  /* lnum == 0 */
}
```

Описание алгоритма топологической сортировки на языке Си

```
int main() {
  init_list ();
  sort_list ();
  return 0;
}
```

Сортировка

- ***Постановка задачи***

  Сортировка – это упорядочение наборов однотипных данных, для которых определено отношение линейного порядка (например, <) по возрастанию или по убыванию.
  Здесь будут рассматриваться целочисленные данные и отношение порядка "<".

- В стандартную библиотеку **`stdlib`** входит функция **`qsort`**:

  ```
  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.`**

Сортировка

- Функция, указатель на которую передается в **`qsort`** в качестве аргумента, соответствующего параметру **`int(*compare)(const void *, const void *)`**, должна иметь описание:

  `int` ***`имя функции`*** **`(const void *`*`arg1`**`, const void *`*`arg2`**`)`**

  и возвращать:

  - целое < 0, если ***`arg1`*** < ***`arg2`***,
  - целое = 0, если ***`arg1`*** = ***`arg2`***
  - целое > 0, если ***`arg1`*** > ***`arg2`***

Сложность алгоритмов

- ***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 только самые “дорогие” операции
- ***Сложность в среднем*:** функция размера входа, отражающая средние затраты на выполнение алгоритма для входа данного размера (учет вероятностей входа)
- ***Асимптотические оценки сложности*:** O-нотация (оценка сверху), точная O-оценка, Θ-оценка.
- Подробности: С.А. Абрамов. Лекции о сложности алгоритмов. М.: МЦНМО, 2009

Сортировка

- ***Простейший алгоритм сортировки***: сведение сортировки к задаче нахождения максимального (минимального) из *n* чисел.

  Нахождение максимума *n* чисел (*n* сравнений):

  Числа содержатся в массиве **`int a[n];`**

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

- ***Алгоритм сортировки***: находим максимальное из *n* чисел, получаем последний элемент отсортированного массива (*n* сравнений);
  находим максимальное из *n* – 1 оставшихся чисел, получаем предпоследний элемент отсортированного массива (еще *n* – 1 сравнений); и так далее.
- ***Общее количество сравнений***: $1 + 2 + \ldots + n\text{-}1 + n = n(n - 1)/2$.

  Сложность алгоритма $O(n^2)$.

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 17

Сортировка

- ***Простейший алгоритм сортировки***: сведение сортировки к задаче нахождения максимального (минимального) из *n* чисел.

  Нахождение максимума *n* чисел (*n* сравнений):

  Числа содержатся в массиве **`int a[n];`**

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

- ***Алгоритм сортировки***: находим максимальное из *n* чисел, получаем последний элемент отсортированного массива (*n* сравнений);
  находим максимальное из *n* – 1 оставшихся чисел, получаем предпоследний элемент отсортированного массива (еще *n* – 1 сравнений); и так далее.
- ***Общее количество сравнений***: $1 + 2 + \ldots + n\text{-}1 + n = n(n - 1)/2$.

  Сложность алгоритма $O(n^2)$.

Сортировка

- ***Классификация алгоритмов сортировки***

  Различают *внешнюю* и *внутреннюю* сортировку.

  Рассматривается только *внутренняя сортировка*: сортируемый массив находится в основной памяти компьютера. *Внешняя сортировка* применяется к записям на внешних файлах.

  3 общих метода внутренней сортировки:

  (1) *сортировка обменами*: рассматриваются соседние элементы сортируемого массива и при необходимости меняются местами;

  (2) *сортировка выборкой*: идея описана на предыдущем слайде

  (3) *сортировка вставками*: сначала сортируются два элемента массива, потом выбирается третий элемент и вставляется в нужную позицию относительно первых двух и т.д.

Сортировка

- ***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;
      }
    }
}
```

Сортировка

- ***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.

```
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;
  }
}
```

Сортировка

- ***Оценка сложности алгоритмов сортировки***
- 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 на уже 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 дублировать стек, в котором расположены некоторые промежуточные данные.

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \mathrm{O}(n \cdot \log_2(n))$$

- Доказательство.

(1) *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \log_2(n!).$$

(a) Алгоритм S можно представить в виде двоичного дерева сравнений.
Так как любая перестановка индексов рассматриваемого массива может быть ответом в алгоритме, она должна быть приписана хотя бы одному листу дерева сравнений.
Таким образом, дерево сравнений будет иметь не менее n! листьев.

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \mathrm{O}(n \cdot \log_2(n))$$

- Доказательство.

(1) *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \log_2(n!). \qquad (*)$$

(б) 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 оценку (*)

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq O(n \cdot \log_2(n))$$

- Доказательство.

(2) К $\log_2(n!)$ применим формулу Стирлинга

$$n! = \sqrt{2\pi n} \cdot n^n e^{-n} e^{\vartheta(n)} \qquad (**)$$

$$|\vartheta(n)| \leq \frac{1}{12n}$$

Логарифмируя (**), получаем

$$\log(n!) = \frac{1}{2}\log(2\pi n) + n \cdot \log(n) - n + \vartheta(n)$$

$$\log(n!) \geq O(n \cdot \log(n))$$

Быстрая сортировка

- ***QuickSort*** **– рекурсивная Си-функция следующего вида:**

```
/* Быстрая сортировка. Предполагается, что left<right */
static void QuickSort (int *a, int left, int right) {
  /* comp – компаранд, i, j – значения индексов */

  int comp, tmp, i, j;
  i = left; j = right;
  comp = a[(left + right)/2]; //можно a[left] или a[right]

  /* построение Partition – цикл do-while */
  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);
  ...
}
```

Быстрая сортировка

- ***QuickSort* – рекурсивная Си-функция следующего вида**:

```
static void QuickSort (int *a, int left, int right) {
  ...

  /* продолжение сортировки, если не все отсортировано */
  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

Быстрая сортировка

- Покажем, что цикл **do-while** действительно строит нужное нам разбиение массива **a[]**.

  (1) В процессе работы цикла индексы **i** и **j** не выходят за пределы отрезка [**left, right**], так как в циклах **while** выполняются соответствующие проверки.

  (2) В момент окончания работы цикла
  do-while j ≤ 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.

  (3) Аналогично, в момент окончания работы цикла
  do-while i ≥ left.

  (4) В момент окончания работы цикла **do-while** любой элемент подмассива **a[left..j]** не больше любого элемента подмассива **a[i..right]**, что очевидно.

Быстрая сортировка

- Работа цикла **do-while** на примере: **5 3 2 6 4 1 3 7**.
  - 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 **i**,
    в фигурных – элемент с индексом **j**).
  - Поскольку **i < j**, элементы, выделенные скобками, нужно поменять местами (оператор **if**):

    3 (3) 2 6 4 {1} 5 7;
  - 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**.
    и нужно рекурсивно применить метод к этим подмассивам.

Быстрая сортировка

- 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; **[6] 5 7.**

- Если $f(n)$ и $g(n)$ – некоторые функции, то запись $g(n) = \Theta(f(n))$ означает, что найдутся такие константы $c1, c2 > 0$ и такое n_0, что для всех $n \geq n_0$ выполняются соотношения

  $$0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n).$$

  т.е. при больших n
  $f(n)$ хорошо описывает поведение $g(n)$.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

(1) Время выполнения цикла **`do-while`**
$\Theta(n)$, где $n = right - left + 1$.

(2) для алгоритма **`QuickSort`** максимальное (наихудшее) время выполнения $T_{max}(n) = \Theta(n^2)$.

Наихудшее время: при каждом *Partition* массив длины n разбивается на подмассивы длины 1 и $n - 1$.

(2Д) Для $T_{max}(n)$ имеет место соотношение
$T_{max}(n) = T_{max}(n - 1) + \Theta(n)$.
Очевидно, что $T_{max}(1) = \Theta(1)$.
Следовательно,

$$T_{max}(n) = T_{max}(n - 1) + \Theta(n) =$$

$$\sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \quad n \cdot (n - 1)/2 = \Theta(n^2).$$

(3) Если исходный массив **`a`** отсортирован в порядке убывания, время его сортировки в порядке возрастания с помощью алгоритма **`QuickSort`** будет $\Theta(n^2)$.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

(4) Минимальное и среднее время выполнения алгоритма *QuickSort*

$$T_{mean}(n) = \Theta(n \cdot \log n)$$

с разными константами: чем ближе разбиение на подмассивы к сбалансированному, тем константы меньше.

(4Д) Доказательство использует теорему о рекуррентных оценках [1]

(5) Рекуррентное соотношение для минимального (наилучшего) времени сортировки $T_{min}(n)$ имеет вид

$$T_{min}(n) = 2 \cdot T_{min}(n/2) + \Theta(n),$$

так как минимальное время получается тогда, когда на каждом шаге удается выбрать компаранд, который делит массив на два подмассива одинаковой длины $\lceil n/2 \rceil$.

Применяя ту же теорему, получаем $T_{min}(n) = \Theta(n \cdot \log n)$.

[1] Т. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 1999. ISBN 5-900916-37-5, с. 66 – 73.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

  (6) 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 $T(n) = \Theta(n \cdot \log n)$ (та же книга, с.160 – 164).

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 18

Формальная постановка задачи поиска по образцу

- Даны *текст* – массив $T[N]$ длины N и *образец* – массив $P[m]$ длины $m \leq N$, где значениями элементов массивов T и P являются символы некоторого *алфавита* A.
- Говорят, что образец P входит в текст T со сдвигом s, если $0 \leq s \leq N - m$ и для всех $i = 1, 2, \ldots, m$ $T[s + i] = P[i]$.
- Сдвиг $s(T, P)$ называется *допустимым*, если P входит в T со сдвигом $s = s(T, P)$ и *недопустимым* в противном случае.
- ***Задача поиска подстрок*** состоит в нахождении множества допустимых сдвигов $s(T, P)$ для заданного текста T и образца P.

Формальная постановка задачи поиска по образцу

- ***Терминология***. Пусть строки *x, y, w* $\in A^*$, $\varepsilon \in A^*$ - пустая строка; |*x*| - длина строки *x*;

  xy – конкатенация строк *x* и *y*; |*xy*| = |*x*| + |*y*|;

  x = *wy* -> *w* – *префикс* (начало) *x* (обозначение $w \prec x$);

  x = *yw* -> *w* – *суффикс* (конец) *x* (обозначение $w \succ x$);

  если *w* – префикс или суффикс *x*, то |*w*| $\leq$ |*x*|;

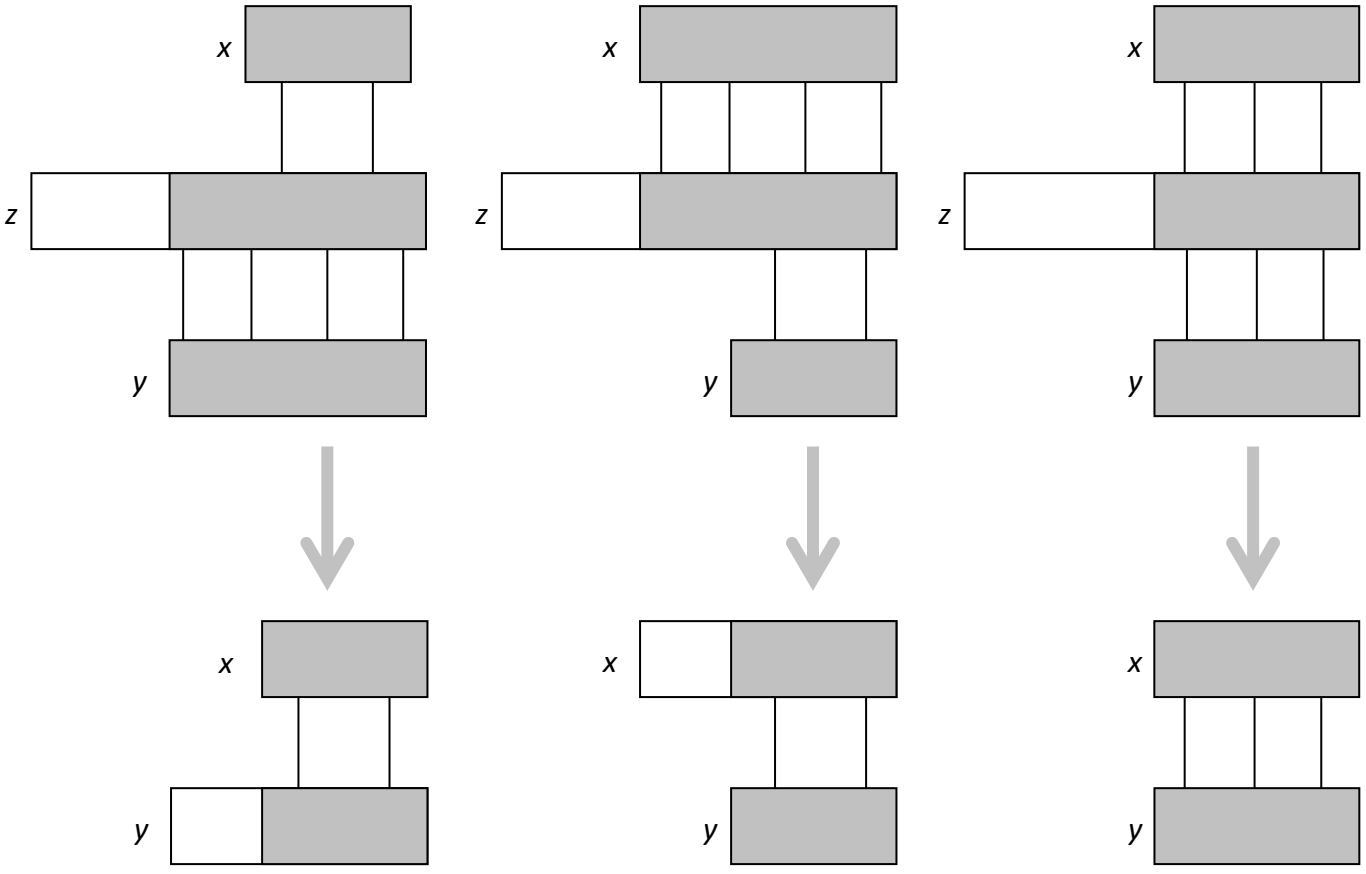
  отношения префикса и суффикса *транзитивны*.

  Для любых *x, y* $\in A^*$ и любого *a* $\in A$ соотношения $x \succ y$ и $xa \succ ya$ *равносильны*.

- Если *S* = S[*r*] – строка длины *r*, то ее префикс длины *k*, *k* $\leq$ *r* будет обозначаться S_k = *S*[*k*]; ясно, что $S_0 = \varepsilon$, S_r = S.

Лемма (*о двух суффиксах*)

- Пусть *x*, *y* и *z* – строки, для которых $x \succ z$ и $y \succ z$. Тогда если |*x*| ≤ |*y*|, то $x \succ y$, если |*x*| ≥ |*y*|, то $y \succ x$, если |*x*| = |*y*|, то $x = y$.



Простой алгоритм

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

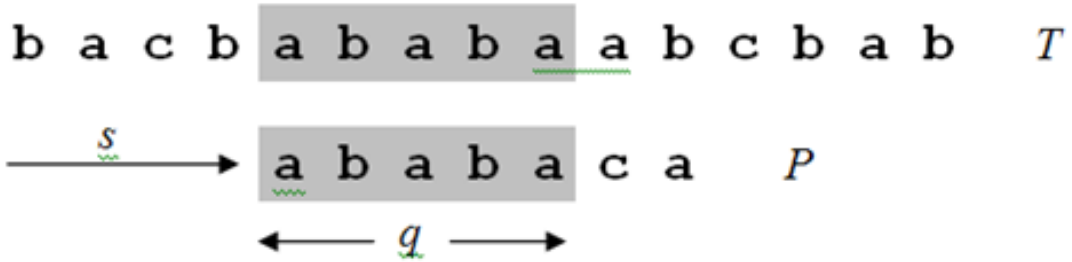
```
for (s = 0; s <= n – m; s++) {
  for (i = 0; i < m && P[i] == T[s + i]; i++)
   ;
  if (i == m)
    printf ("%d\n", s);
}
```

- Время работы в худшем случае $\Theta((n – m + 1)\cdot m) \sim \Theta(nm)$.

  Причина: информация о тексте T, полученная при проверке данного 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**.

Алгоритм Кнута – Морриса – Пратта. Идея

- *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* = **a b a b a c a** 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.



Алгоритм Кнута – Морриса – Пратта. Идея

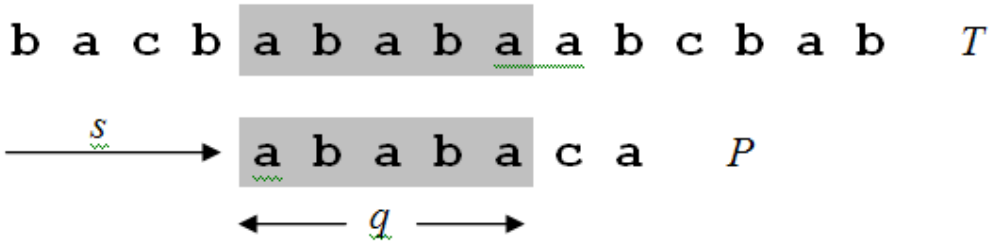
- Пусть $P[1..q] = T[s+1..s+q]$; каково минимальное значение сдвига $s' > s$, для которого $P[1..k] = T[s'+1..s'+k]$, где $s'+k = s+q$?
  - Число s' - минимальное значение сдвига, большего s, которое совместимо с тем, что $T[s+1..s+q] = P[1..q]$. Следовательно, значения сдвигов, меньшие s', проверять не нужно.
    Лучше всего, когда $s' = s+q$, так как в этом случае не нужно рассматривать сдвиги $s+q-1, s+q-2, \ldots, s+1$.
    Кроме того, при проверке нового сдвига s' можно не рассматривать первые его k символов образца: они заведомо совпадут.
- Чтобы найти s', достаточно знать образец P и число q:
  $T[s'+1..s'+k]$ – 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 s' по формуле $s' = s + (q - k)$.

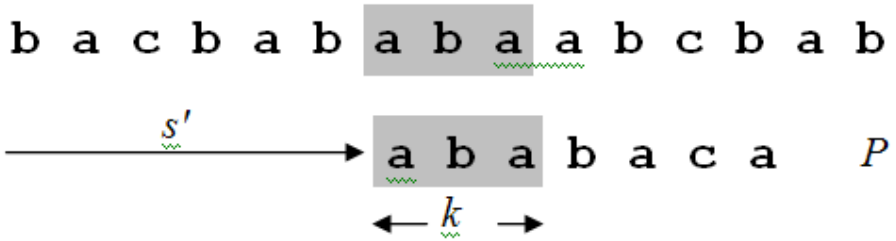
Алгоритм Кнута – Морриса – Пратта. Префикс-функция

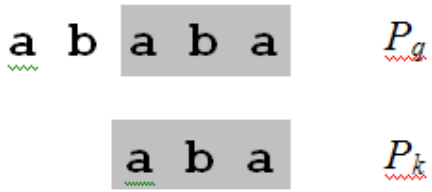
- ***Определение***. Префикс-функцией, ассоциированной со строкой $P[1..m]$, называется функция π: $\{1,2, \ldots, m\} \to \{0,1, \ldots, m-1\}$, определенная следующим образом:

$$\pi[q] = \max\{k: k<q \ \& \ P_k \succ P_q\}$$

Иными словами, $\pi[q]$ – длина наибольшего префикса P, являющегося суффиксом P_q.







Алгоритм Кнута – Морриса – Пратта. Префикс-функция

```
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;
  }
}
```

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***Лемма 1***. Обозначим $\pi^*[q] = \{q, \pi[q], \pi^2[q], ..., \pi^t[q]\}$,
  где $\pi^i[q]$ есть i-я итерация префикс-функции, $\pi^t[q] = 0$.
  Пусть *P* – строка длины *m* с префикс-функцией π.
  Тогда для всех *q* = 1, 2, ..., *m* имеем $\pi^*[q] = \{k : P_k \succ P_q\}$.
- 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* принадлежит $\pi^*[q]$, 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.

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

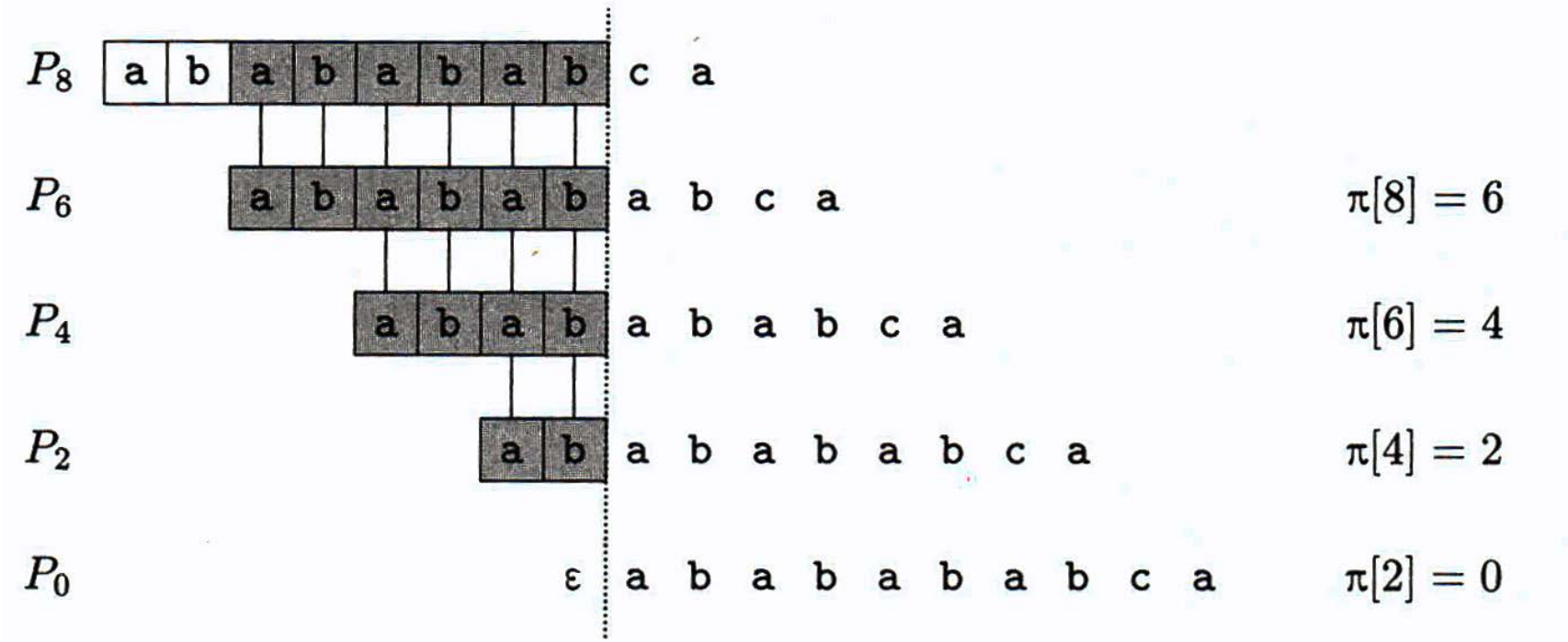
- Доказательство.

  (2) Покажем, что наоборот, если P_i является суффиксом P_q, то i принадлежит $\pi^*[q]$.

  - 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 $j = i_{k+1}$.
  - (2) можно доказать и от противного: для наибольшего числа j такого, что $P_j \succ P_q$, но j не входит в $\pi^*[q]$, определение префикс-функции нарушается.

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1



$$\pi^*[8] = \{8,6,4,2,0\}$$

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***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 $k-1 \in \pi^*[q-1]$, т.е. $\pi[q]-1 \in \pi^*[q-1]$.
- Определим множества E_{q-1} как

  $E_{q-1} = \{k : k \in \pi^*[q-1] \text{ и } P[k+1] = P[q]\}$.

  Множество E_{q-1} состоит из таких *k*, что P_k является суффиксом P_{q-1}, и за ними идут одинаковые буквы *P*[*k*+1] и *P*[*q*].

  Из определения вытекает, что P_{k+1} есть суффикс P_q.

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***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 $r - 1 = \pi[q] - 1 \in \pi^*[q-1]$.
  - Т.к. $P[r] = P[q]$, то $P[(r\text{-}1)+1] = P[q]$.
  - Поэтому $r - 1 \in E_{q-1}$ по определению E_{q-1} и из $\pi[q] \geq 1$ следует непустота E_{q-1}.
  - Следовательно, если E_{q-1} пусто, то $\pi[q] = 0$ (от противного).
  - Если $k \in E_{q-1}$, то P_{k+1} есть суффикс P_q (из определения), следовательно, $\pi[q] \geq k + 1$ 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 $\pi[q] - 1 \in E_{q-1}$, $\pi[q] - 1$ не больше максимума из E_{q-1}, т.е. $\pi[q] \leq 1 + \max\{k \in E_{q-1}\}$.

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

```
1  void prefix_func (char *pat, int *pi, int m) {
2    int k, q;
3
4    /* Считаем, что pat и pi нумеруются от 1 */
5    pi[1] = 0; k = 0;
6    for (q = 2; q <= m; q++) {
7      while (k > 0 && pat[k + 1] != pat[q])
8        k = pi[k];
9      if (pat[k + 1] == pat[q])
10       k++;
11     pi[q] = k;
12   }
13 }
```

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***Теорема 1***. Функция **`prefix_func`** правильно вычисляет префикс-функцию π.

- Доказательство.
  - Покажем, что при входе в цикл функции $k = \pi[q\text{-}1]$.
  - <u>База индукции</u>.

    При **`q = 2`** **`k = 0`**, **`pi[q-1] = pi[1] = 0`**.
  - <u>Шаг индукции</u>.

    Пусть при входе в цикл функции $k = \pi[q\text{-}1]$.

    Код на строках 7-8

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

    находит наибольший элемент E_{q-1} (т.к. цикл перебирает в порядке убывания элементы из $\pi^*[q\text{-}1]$ и для каждого проверяет равенство **`pat[k + 1] != pat[q])`**).

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***Теорема 1***. Функция **`prefix_func`** правильно вычисляет префикс-функцию π.

- Доказательство.
  - После выхода из цикла на строках 7-8

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

1) если **`pat[k + 1] == pat[q]`**, то выполняется код на строке 10:

```
k++;
```

что из Следствия 1 дает нам $\pi[q]$.

2) если **`pat[k + 1] != pat[q]`**, то **`k == 0`**, множество E_{q-1} пусто и $\pi[q] = 0$.

Алгоритм Кнута – Морриса – Пратта. Функция kmp

```
void kmp (char *text, char *pat, int m, int n) {
  int q;
  int pi[m + 1];  /* VLA-массив */
  /* Через alloca: int *pi = alloca ((m + 1) * sizeof (int)); */

  /* Считаем, что pat и text нумеруются от 1 */
  prefix_func (pat, pi, m);
  q = 0;
  for (i = 1; i <= 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 Кнута – Морриса – Пратта. Функция `kmp`

- Алгоритм КМП для подстроки *P* и текста *T* эквивалентен вычислению префикс-функции для строки *Q* = *P*#*T*, где # – символ, заведомо не встречающийся в обеих строках
  - Длина максимального префикса *Q*, являющегося её 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

Алгоритм Кнута – Морриса – Пратта. Время работы

- Функция **prefix_func** выполняет $\leq (m - 1)$ итераций цикла **`for`**. Стоимость каждой итерации можно считать равной $O(1)$, а стоимость всей процедуры $O(m)$.
  - Каждая итерация цикла **`while`** (строки 7-8) уменьшает **`k`**
  - Увеличивается **`k`** только в строке 10 не более одного раза на итерацию цикла **`for`** (строки 6-11)
  - Следовательно, операций уменьшения не больше, чем итераций цикла **`for`**, то есть $\leq (m - 1)$ на весь цикл и $O(1)$ на итерацию в среднем
- Аналогично, функция **kmp** выполняет $\leq (n - 1)$ итераций, и ее стоимость (без учета вызова **prefix_func**) есть $O(n)$. Следовательно, время выполнения всей процедуры $O(m + n)$.

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 19

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*, и
    двоичное дерево, называемое *правым поддеревом*.
- Двоичное дерево не является частным случаем обычного дерева, хотя у этих структур много общего. Основные отличия:
  (1) Пустое дерево является двоичным деревом, но не является обычным деревом.
  (2) Двоичные деревья **(*A*(*B*,*NULL*))** и **(*A*(*NULL*,*B*))** различны, а обычные деревья – одинаковы.
- Термины: узлы, ветви, корень, листья, высота

Двоичное дерево

- Представление двоичного дерева в памяти компьютера

Описание узла двоичного дерева на Си:

```
typedef struct bin_tree {
  char info;
  struct bin_tree *left;
  struct bin_tree *right;
} node;
```

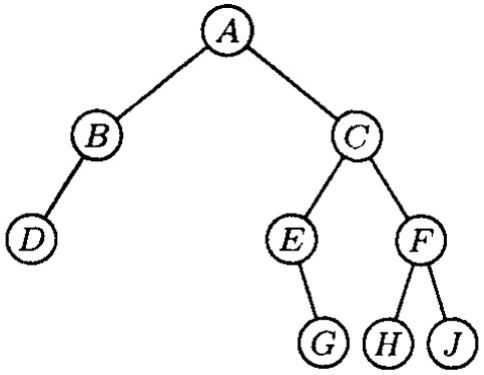


Рис. 1. Двоичное дерево

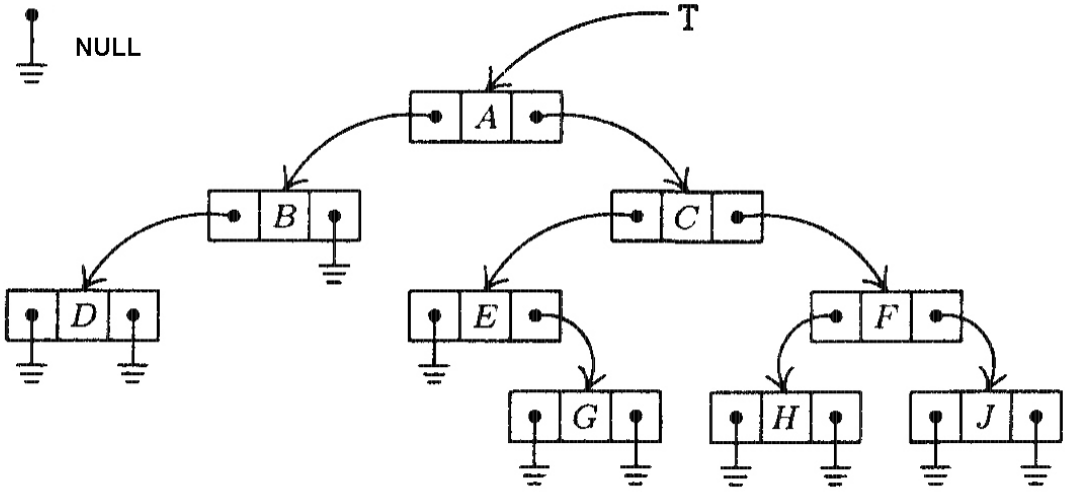


Рис. 2 Представление дерева с рис.1 в компьютере.

Двоичное дерево

- ***Обход двоичного дерева.***

  Обход дерева позволяет выполнить топологическую сортировку узлов дерева и расположить их в линейном одномерном массиве, порядок узлов дерева в котором таков, что их можно обрабатывать в цикле

  ```
  for (i = 0; i < N; i++)
  ```

  (*топологический порядок*)

Двоичное дерево

Различные способы обхода двоичного дерева

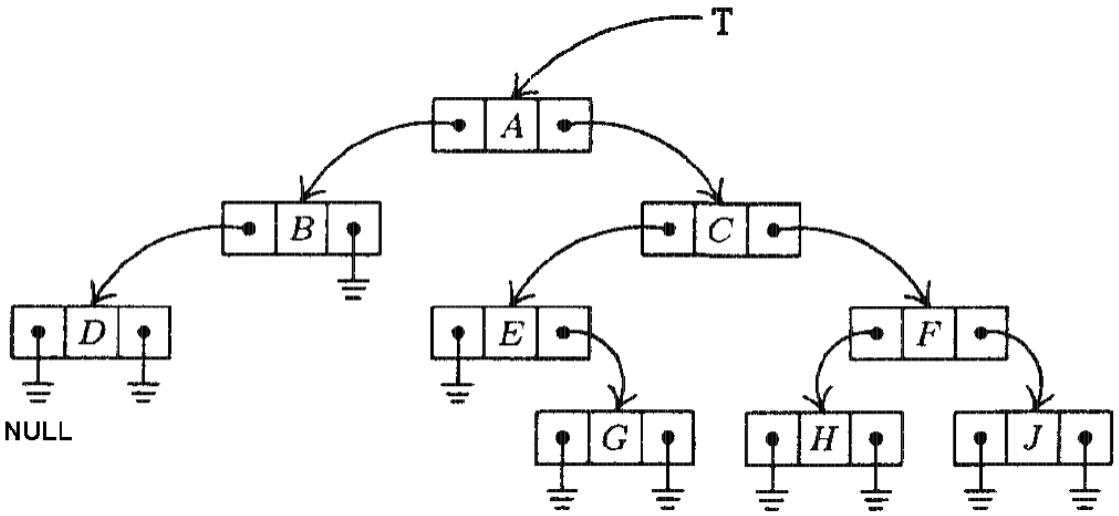
(1) Обход в глубину в *прямом порядке*:

- обработать корень,
- обойти левое поддерево,
- обойти правое поддерево.

Порядок обработки узлов дерева на рисунке

A B D C E G F H J

Линейная последовательность узлов, полученная при прямом обходе, отражает «спуск» информации от корня дерева к листьям.



Двоичное дерево

Различные способы обхода двоичного дерева

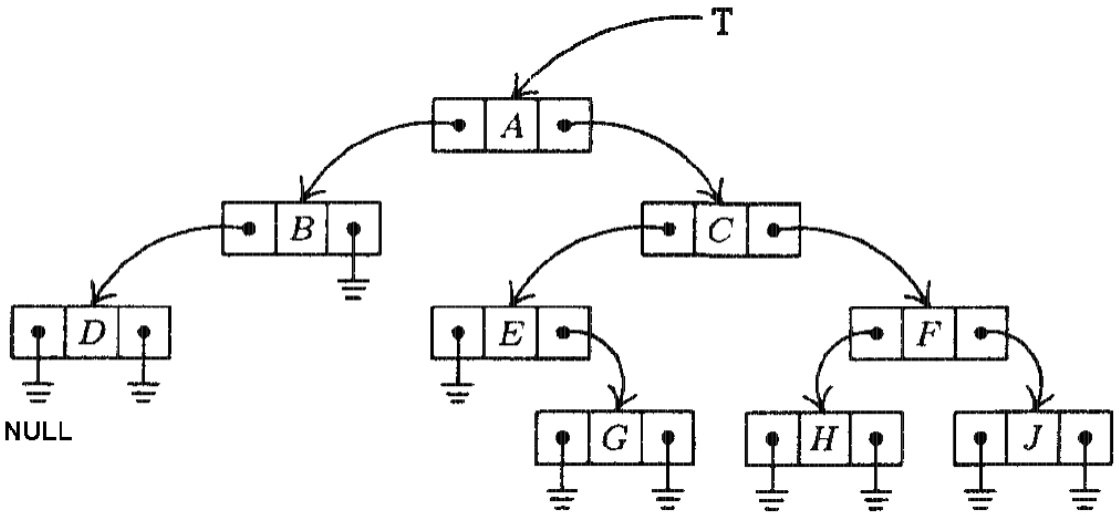
(2) Обход в глубину в *обратном порядке*:

- обойти левое поддерево,
- обойти правое поддерево,
- обработать корень.

Порядок обработки узлов дерева на рисунке:

D B G E H J F C A

Линейная последовательность узлов, полученная при обратном обходе, отражает «подъем» информации от листьев к корню дерева.



Двоичное дерево

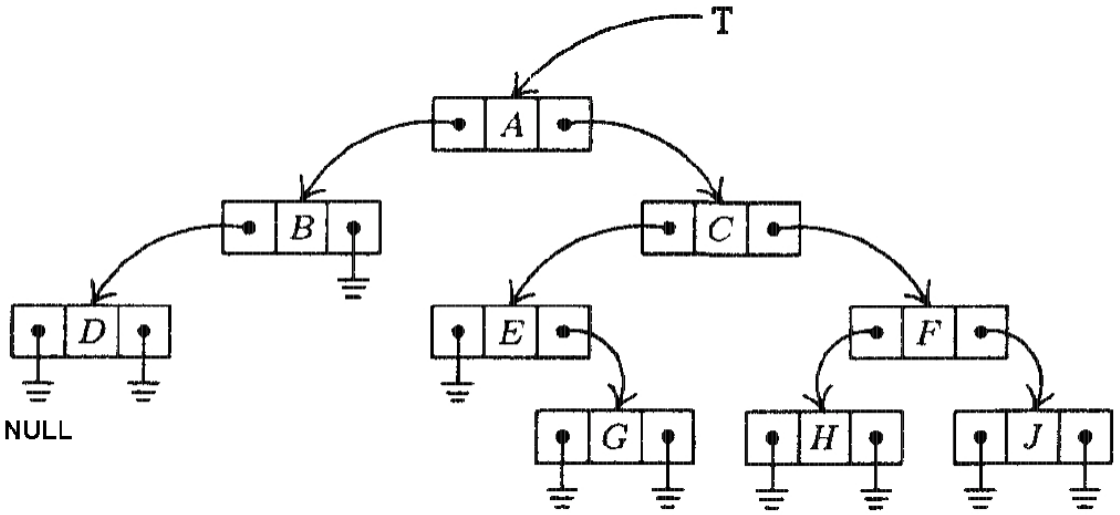
- ***Различные способы обхода двоичного дерева***

(3) Симметричный обход в глубину (обход в *симметричном порядке*):

- обойти левое поддерево,
- обработать корень.
- обойти правое поддерево,

Порядок обработки узлов дерева на рисунке:

D B A E G C H F J



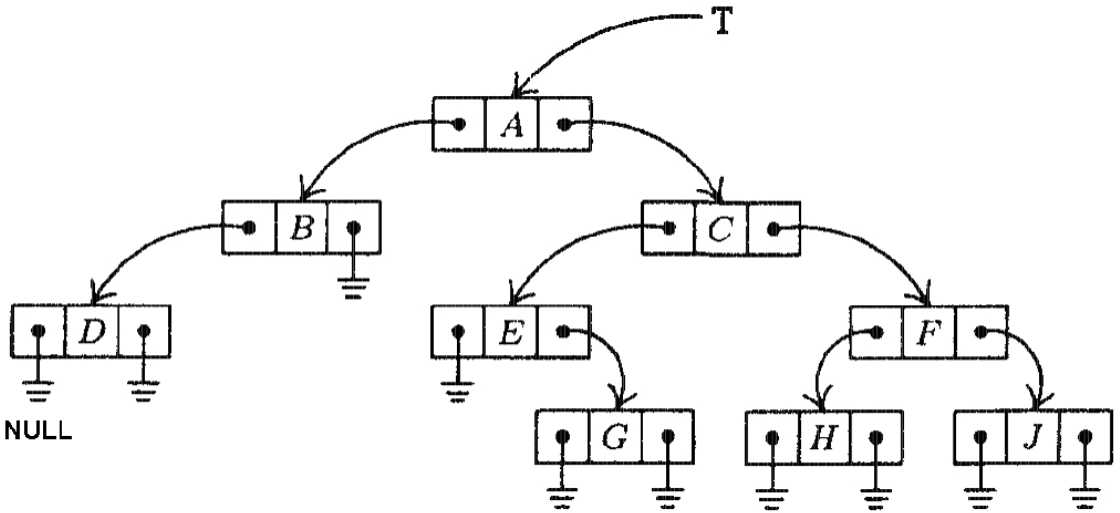
Двоичное дерево

- ***Различные способы обхода двоичного дерева***

(4) Обход двоичного дерева в ширину:
узлы дерева обрабатываются «по уровням»
(уровень составляют все узлы, находящиеся на одинаковом расстоянии от корня)

Порядок обработки узлов дерева на рисунке:

A B C D E F G H J



Двоичное дерево

- Функции, реализующие обходы двоичного дерева, позволяют по указателю каждого узла дерева **P** вычислить указатели узлов

  P_next_pre, P_next_post и **P_next_in,**
  P_pred_pre, P_pred_post и **P_pred_in.**

- Рекурсивные Си-функции обхода двоичного дерева в глубину

(1)
```
void preorder (node * r) {
  if (r == NULL)
    return;
  if (r->info)
    printf ("%c", r->info);
  preorder (r->left);
  preorder (r->right);
}
```

Двоичное дерево

- Рекурсивные Си-функции обхода двоичного дерева в глубину

(2)
```
void postorder (node *r) {
  if (r == NULL)
    return;
  postorder (r->left);
  postorder (r->right);
  if (r->info)
    printf ("%c", r->info);
}
```

(3)
```
void inorder (node *r) {
  if (r == NULL)
    return;
  inorder (r->left);
  if (r->info)
    printf ("%c", r->info);
  inorder (r->right);
}
```

Двоичное дерево

- Нерекурсивная функция обхода двоичного дерева (управление стеком ведется не автоматически, а в самой функции).

  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 (если 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

Двоичное дерево

- Нерекурсивная функция обхода двоичного дерева

Алгоритм:

(1) [Инициализация]. Сделать стек пустым, т.е. затолкнуть **NULL** на дно стека: **`stack[0] = NULL;`** установить указатель стека на дно стека: **`top = 0;`** установить указатель **`t`** на корень дерева: **`t = r.`**

(2) [Конец ветви].Если **`t == NULL`**, перейти к (4).

(3) [Продолжение ветви]. Затолкнуть **`t`** в стек:

`stack[++top] = t;`

установить **`t = t->left`** и вернуться к шагу (2).

(4) [К обработке правой 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).

Двоичное дерево

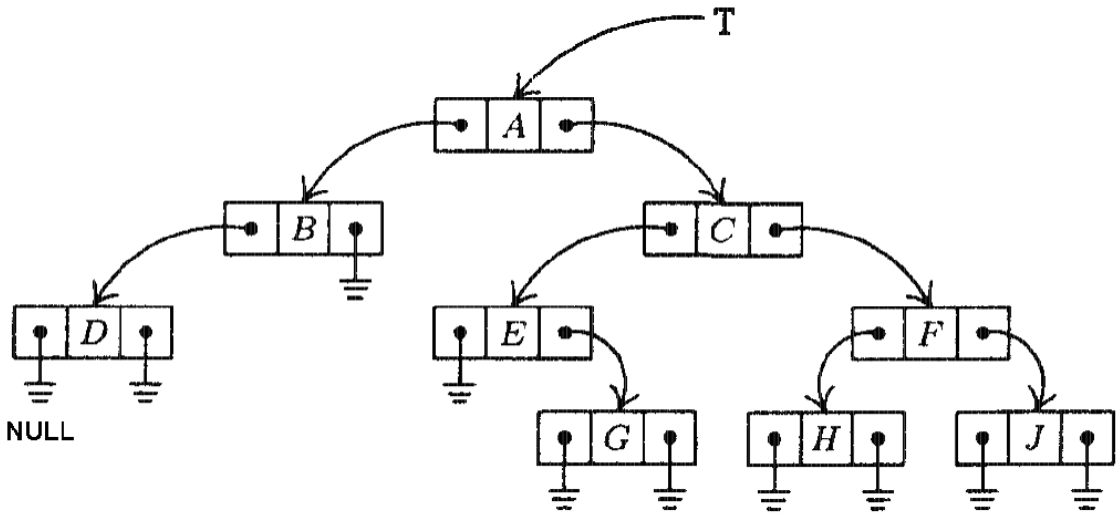
- Нерекурсивная функция обхода двоичного дерева

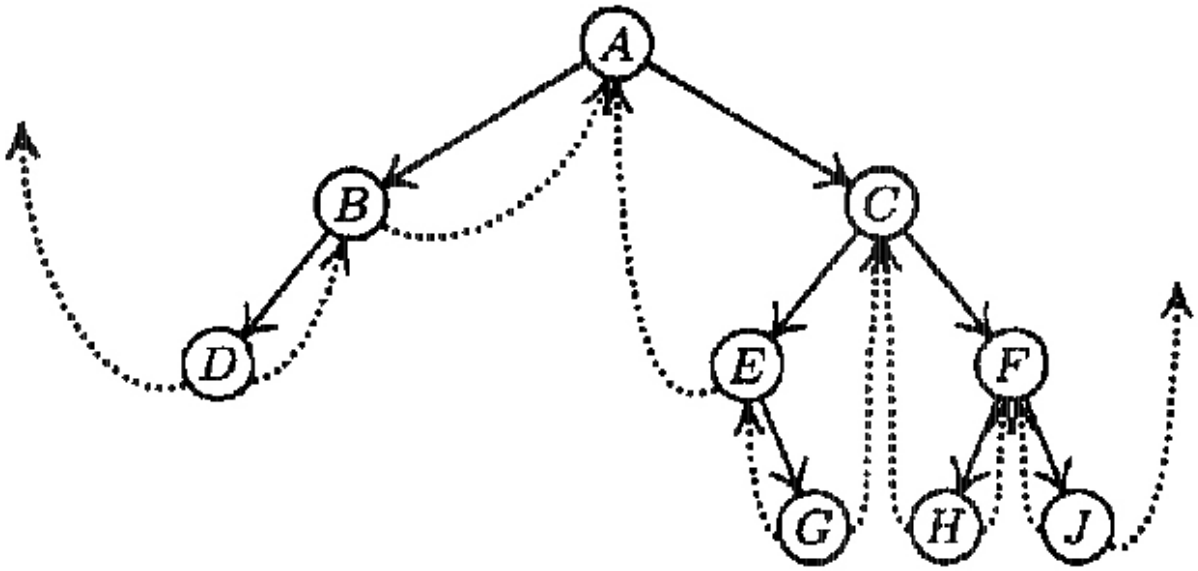
```
int inorder (node *r, char *order) {
  node *t = r;
  node *stack[depth];                    // depth = ?
  int top = 0, i = 0;

  if (!t)
    return 0;
  stack[0] = NULL;                       //Шаг 1
  while (1) {
    while (t) {                          //Шаг 2
      stack[++top] = t;                  //Шаг 3
      t = t->left;
    }
    t = stack[top--];                    //Шаг 4
    if (t) {
      order[i++] = t->info;              //обработка
      t = t->right;                      //Шаг 5
    } else                               //t == NULL
      break;                             //Шаг 4
  }
  return i;
}
```

Двоичное дерево

- ***Прошитое двоичное дерево***





Рассмотрим двоичное дерево на верхнем рисунке.
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 не использовать стек.

Двоичное дерево

- ***Прошитое двоичное дерево***
- Описание узла прошитого двоичного дерева

```
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. Например, в случае симметричного обхода:

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`

Двоичное дерево

- ***Прошитое двоичное дерево***
- Нити существенно упрощают алгоритмы обхода двоичных деревьев. Например, для вычисления для каждого узла **p** указатель узла **P_next_in** можно использовать следующий простой алгоритм:

```
threaded_node * next_in (threaded_node *p) {
  threaded_node *q = p->right;
  if (p->right_tag == 1)
    return q;
  while (q->left_tag == 0) //q != NULL
    q = q->left;           //q->left != NULL
  return q;
}
```

- Функция **next_in** фактически реализует симметричный обход дерева, так как позволяет для произвольного узла дерева **P** найти **P_next_in**. Многократно применяя эту функцию, можно вычислить топологический порядок узлов двоичного дерева, соответствующий симметричному обходу.

Двоичное дерево

- ***Прошитое двоичное дерево***
- Аналогичным образом можно вычислить **P_next_pred** и **P_next_post**.

  Применяя функции **next_pred** (либо **next_post**), можно вычислить топологический порядок узлов, соответствующий прямому (либо обратному) обходу.
- ***Замечания***

  (1) С помощью обычного представления невозможно для произвольного узла **P** вычислить **P_next_in**, не вычисляя всей последовательности узлов.

  (2) Функции **next_in** не требуется стек ни в явной, ни в неявной (рекурсия) форме.

Двоичное дерево

- ***Прошитое двоичное дерево***
- Сравнение функций **`inorder()`**и **`next_in()`** позволяет сделать следующие выводы:

  - Если **`p`** – произвольно выбранный узел дерева,то следующий фрагмент функции **`next_in()`**:

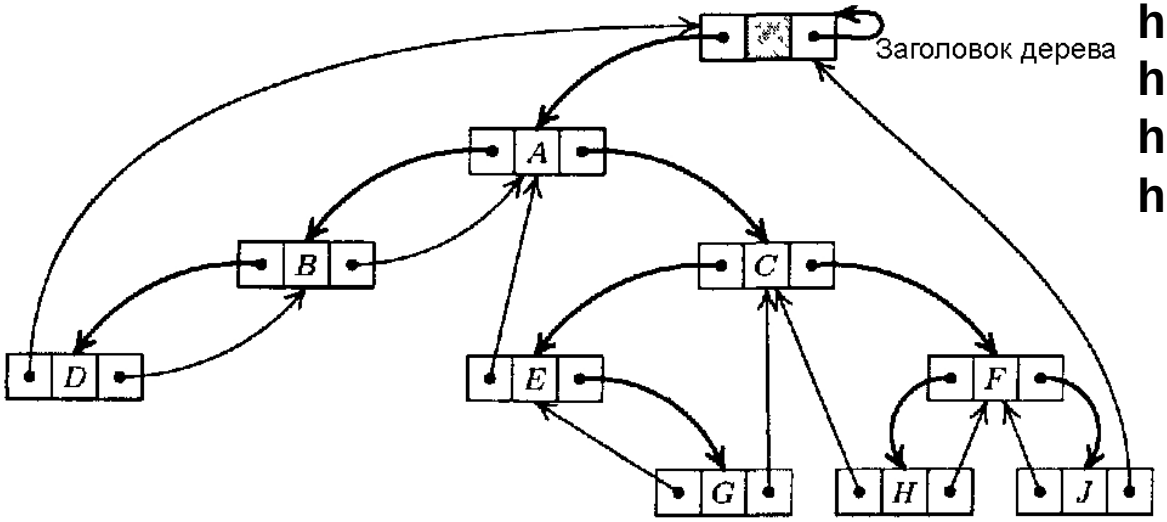
```
q = p->right;
if (p->right_tag == 1)
  return q;
```

    выполняется только один 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]`**. **`depth`** стараются взять не очень большим, но **`depth`** не может быть меньше высоты двоичного дерева. **Нельзя допускать переполнение стека деревьев**.

Двоичное дерево

◈ *Прошитое двоичное дерево*

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. Для этого к дереву добавляется еще один узел – заголовок дерева (**header**).



поля структуры

```
typedef struct bin_tree
{
  char info;
  struct bin_tree *left;
  struct bin_tree *right;
  char left_tag;
  char right_tag;
} threaded_node;
threaded_node *header;
```

заполняются в заголовке следующим образом

```
header->left_tag = 0;
header->right_tag = 0;
header->left = r;
header->right = header;
```

На рисунке дуги дерева показаны более жирными линиями, чем нити.

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 20

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* (найти, иногда называется *fetch*), *insert* (вставить) и *delete* (удалить).

  Также предоставляется один или несколько вариантов *обхода* хранилища (посещения всех данных).

- Варианты решения – деревья поиска, хеширование

Двоичные деревья поиска

- Структура для представления узла двоичного дерева поиска:

```
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 *y* находится в правом поддереве узла *x*, то

  $$key[y] > key[x].$$

- Возможно хранение дублирующихся ключей (нестрогие неравенства), не рассматривающееся в данном курсе

Двоичные деревья поиска: *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* (если такой узел есть), либо пустой указатель NULL.

```
struct BT_node *Btsearch (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);
}
```

Двоичные деревья поиска: *поиск узла*

- Итеративная версия поиска.

```
struct BT_node *Btsearch (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;
}
```

- Среднее время поиска $O(h)$, где h – высота дерева.

Двоичные деревья поиска: *минимум и максимум*

- **На входе**: указатель *root* на корень поддерева.
  На выходе: указатель на узел с минимальным ключом *k*.

```
struct BT_node *Btmin (struct BT_node *root)
{
  struct BT_node *p = root;
  while (p->left)
    p = p->left;
  return p;
}
```

- Среднее время выполнения $O(h)$, где h – высота дерева.

Двоичные деревья поиска: *следующий элемент*

- **На входе**: указатель *node* на узел дерева.
  На выходе: указатель на следующий за *node* узел дерева.

```
struct BT_node *Btsucc (struct BT_node *node) {
  struct BT_node *p = node, *q;
  /* I случай: правое поддерево узла не пусто */
  if (p->right)
    return Btmin (p->right);
  /* II 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;
}
```

- Среднее время выполнения $O(h)$, где h – высота дерева.
- Связь с симметричным порядком обхода и прошитыми деревьями.

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 *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;
  return root;
}
```

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); (2) у узла *n* только один ребенок; (3) у узла *n* два ребенка.
  - (1) просто удаляем узел *n*;
  - (2) вырезаем узел *n*, соединив единственного ребенка узла *n* с родителем узла *n*.
  - (3) находим *succ*(*n*) и удаляем его, поместив ключ *succ*(*n*) в узел *n*.

Двоичные деревья поиска: удаление

- Шаг 1: если у *n* меньше двух детей, удаляем *n*, иначе удаляем *succ*(*n*); устанавливаем указатель *y* на удаляемый узел.
- Шаг 2: находим ребенка удаляемого узла (ребенка либо нет, либо он единственный) и устанавливаем на него указатель *x*.
- Шаг 3: подвешиваем ребенка *y* (указатель *x*) к родителю *y*; если у *y* нет родителя, новым корнем дерева становится *x*; устанавливаем в соответствующем поле родителя указатель на *x*, полностью исключая *y* из дерева.
- Шаг 4: если удаляемый узел *succ*(*n*), заменяем данные узла *n* на данные узла *succ*(*n*).

Двоичные деревья поиска: удаление

```
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 – ребенок y, вырезаем y из родителей */
  if (x)
    x->parent = y->parent;
  /* Шаг 3: если у y нет родителя, новым корнем дерева становится x */
  if (! y->parent)
    *root = x;
  else {
    /* Шаг 3: x присоединяется к y->parent с требуемой стороны */
    if (y == y->parent->left)
      y->parent->left = x;
    else
      y->parent->right = x;
  }
  <...>
```

Двоичные деревья поиска: удаление

```
struct BT_node * BTdelete (struct BT_node **root,
                           struct BT_node *n) {
  struct BT_node *x, *y;
   <...>
  /* Шаг 4: если удалялся не узел n, а succ(n), необходимо
     заменить 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;
}
```

- Среднее время выполнения $O(h)$, где h – высота дерева.

Построение двоичного дерева поиска.

- **Постановка задачи.** Пусть имеется множество K из m ключей:

$$K = \{k_0, k_1, \ldots, k_{m-1}\}$$

Разбиение K на три подмножества K_1, K_2, K_3:

- $|K_2| = 1, |K_1| \geq 0, |K_3| \geq 0.$
- $K_2 = \{k\} \Rightarrow \forall l \in K_1: l < k$ и $\forall r \in K_3: r \geq k$

Далее по рекурсии: разбиваем K_1 на K_{11}, K_{12}, K_{13}
K_3 на K_{31}, K_{32}, K_{33}

и т.д. пока ключи не кончатся

- **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.

Построение двоичного дерева поиска.

- Пусть h – высота полностью сбалансированного двоичного дерева. Тогда число вершин m должно быть равно:

$$m = 1 + 2 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$$

откуда $h = \log_2(m + 1)$.

- Если все m ключей известны заранее, их можно отсортировать за $O(m \cdot \log_2 m)$, 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: 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 $n \geq 3$

  $$f_n = f_{n\text{-}1} + f_{n\text{-}2}.$$

Деревья Фибоначчи

- *n*-е число Фибоначчи вычисляет следующая функция:

```
int Fbn (int n) {
      if (n == 1 || n == 2)
            return 1;
      else {
            int g, h, k, Fb;
            g = h = 1;
            for (k = 2; k < n; k++) {
              Fb = g + h;
              h = g;
              g = Fb;
            }
            return Fb;
      }
}
```

Деревья Фибоначчи

- ***Определение*** дерева Фибоначчи
  (это тоже искусственное дерево).

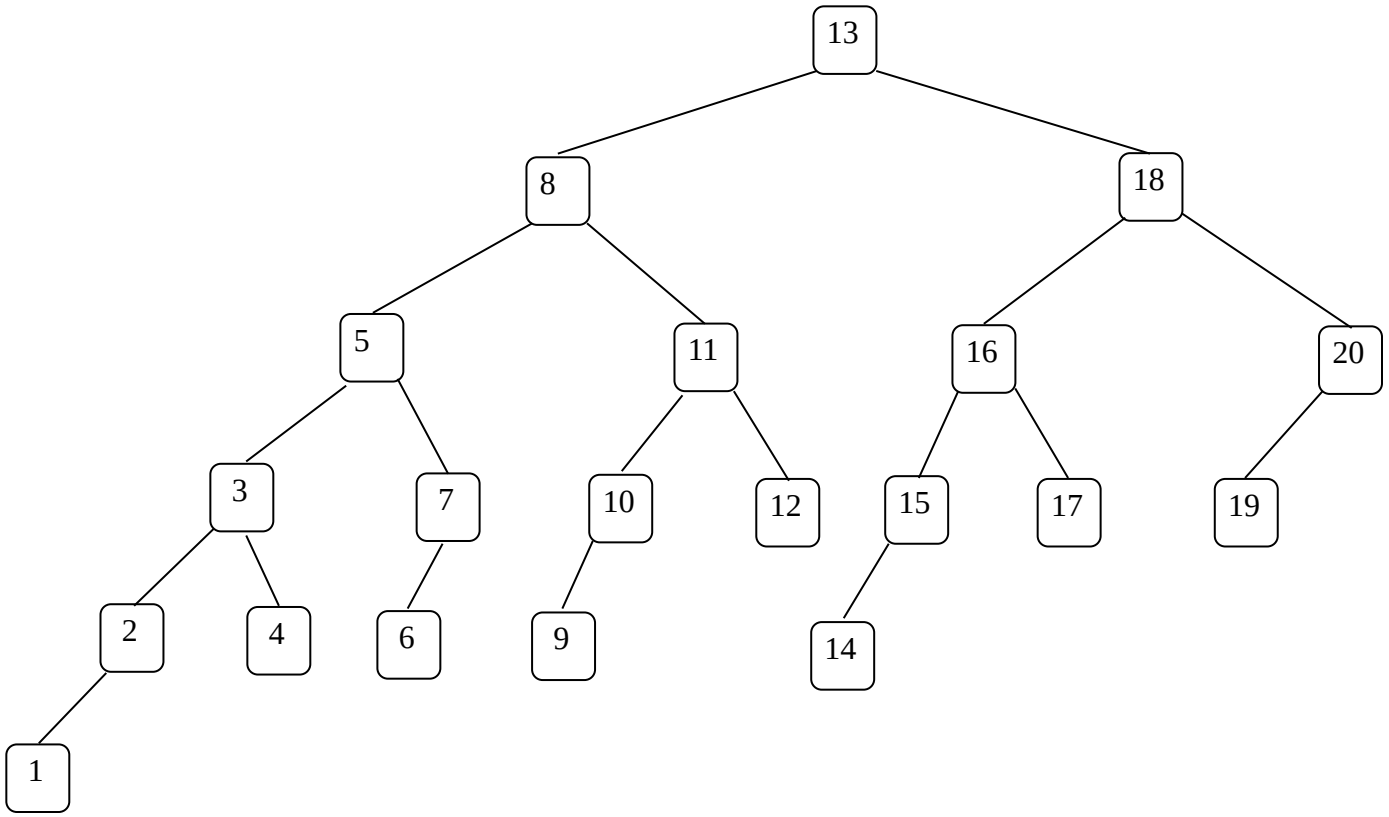
  (1) Пустое дерево – это дерево Фибоначчи с высотой $h = 0$.

  (2) 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.

Деревья Фибоначчи

- ***Пример***. Дерево Фибоначчи с $h = 6$.



Деревья Фибоначчи

- ***Теорема 1***. Число вершин в дереве Фибоначчи F_h высоты h равно $m(h) = f_{h+2} - 1$.

Доказательство (по индукции).

$h = 0$: $m(0) = f_2 - 1 = 0$

$m(1) = f_3 - 1 = 1$.

Шаг: по определению $m(h) = m(h-1) + m(h-2) + 1$.

Имеем $m(h) = (f_{h+1} - 1) + (f_h - 1) + 1 = f_{h+2} - 1$,

так как $f_h + f_{h+1} = f_{h+2}$

Деревья Фибоначчи

◈ ***Теорема 2***. Пусть C_1 и C_2 таковы, что уравнение

$$r^2 - C_1 r - C_2 = 0 \qquad (*)$$

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. r_1 и r_2 – корни уравнения (*),

то $r_1^2 = C_1 r_1 + C_2$

$r_2^2 = C_1 r_2 + C_2$.

Имеем:

$$C_1 a_{n-1} + C_2 a_{n-2} = C_1(\alpha_1 r_1^{n-1} + \alpha_2 r_2^{n-1}) + C_2(\alpha_1 r_1^{n-2} + \alpha_2 r_2^{n-2}) =$$
$$= \alpha_1 r_1^{n-2}(C_1 r_1 + C_2) + \alpha_2 r_2^{n-2}(C_1 r_2 + C_2) =$$
$$= \alpha_1 r_1^{n-2} r_1^2 + \alpha_2 r_2^{n-2} r_2^2 = \alpha_1 r_1^n + \alpha_2 r_2^n = a_n \qquad (**)$$

Теорема доказана.

Деревья Фибоначчи

◈ ***Теорема 3***. Пусть C_1 и C_2 таковы, что уравнение

$$r^2 - C_1 r - C_2 = 0 \qquad (*)$$

имеет два 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 и a_1

следует $a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ для n = 1, 2, ...

Доказательство. Нужно не только повторить в обратном порядке вывод (**), но и подобрать такие α_1 и α_2, чтобы

$$a_0 = \alpha_1 + \alpha_2,\ a_1 = \alpha_1 r_1 + \alpha_2 r_2 \qquad (***)$$

Рассматривая (***) как систему линейных уравнений относительно α_1 и α_2, получим:

$$\alpha_1 = \frac{a_1 - a_0 \cdot r_2}{r_1 - r_2}, \qquad \alpha_2 = \frac{-a_1 + a_0 \cdot r_1}{r_1 - r_2}$$

Теорема доказана.

Деревья Фибоначчи

- Применим доказанные теоремы к числам Фибоначчи:

$f_n = f_{n-1} + f_{n-2}$.

Уравнение $r^2 - r - 1 = 0$ имеет корни

$$r_1 = \frac{1+\sqrt{5}}{2} \qquad r_2 = \frac{1-\sqrt{5}}{2}$$

Следовательно, согласно теореме 3

$$f_n = \alpha_1 \cdot r_1^n + \alpha_2 \cdot r_2^n = \alpha_1 \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + \alpha_2 \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n,$$

$$f_0 = \alpha_1 + \alpha_2 = 0,$$

$$f_1 = \alpha_1 \cdot \left(\frac{1+\sqrt{5}}{2}\right) + \alpha_2 \cdot \left(\frac{1-\sqrt{5}}{2}\right) = 1$$

$$\alpha_1 = \frac{1}{\sqrt{5}}, \alpha_2 = -\frac{1}{\sqrt{5}}$$

Деревья Фибоначчи

Откуда

$$f_n = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2} \right)^n$$

Согласно теореме 1

$$m(h) = f_{h+2} - 1 = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2} - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^{h+2} - 1$$

$$\left| \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^{h+2} \right| < 1$$

$$m(h) + 1 > \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2}$$

Деревья Фибоначчи

Обозначение $\gamma = \frac{1+\sqrt{5}}{2}$

$$m(h)+1 > \frac{1}{\sqrt{5}}\gamma^{h+2} \qquad (****)$$

Логарифмируя обе части (****) , получаем

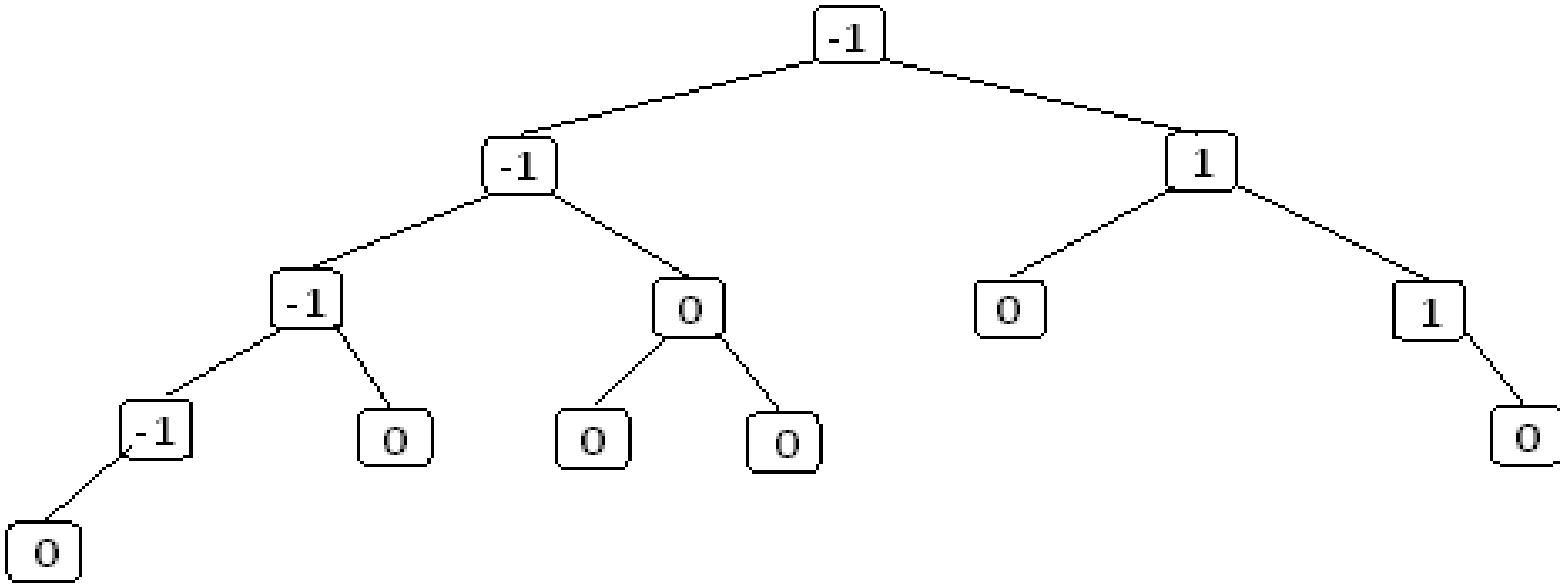
$$h+2 < \frac{\log_2(m+1)}{\log_2 \gamma} + \frac{\log_2 \sqrt{5}}{\log_2 \gamma}$$

откуда $h < 1{,}44 \cdot \log_2(m+1) - 0{,}32$

Таким образом, мы доказали, что для деревьев Фибоначчи с числом вершин m количество сравнений в худшем случае не превышает $1{,}44 \cdot \log_2(m+1) - 0{,}32$

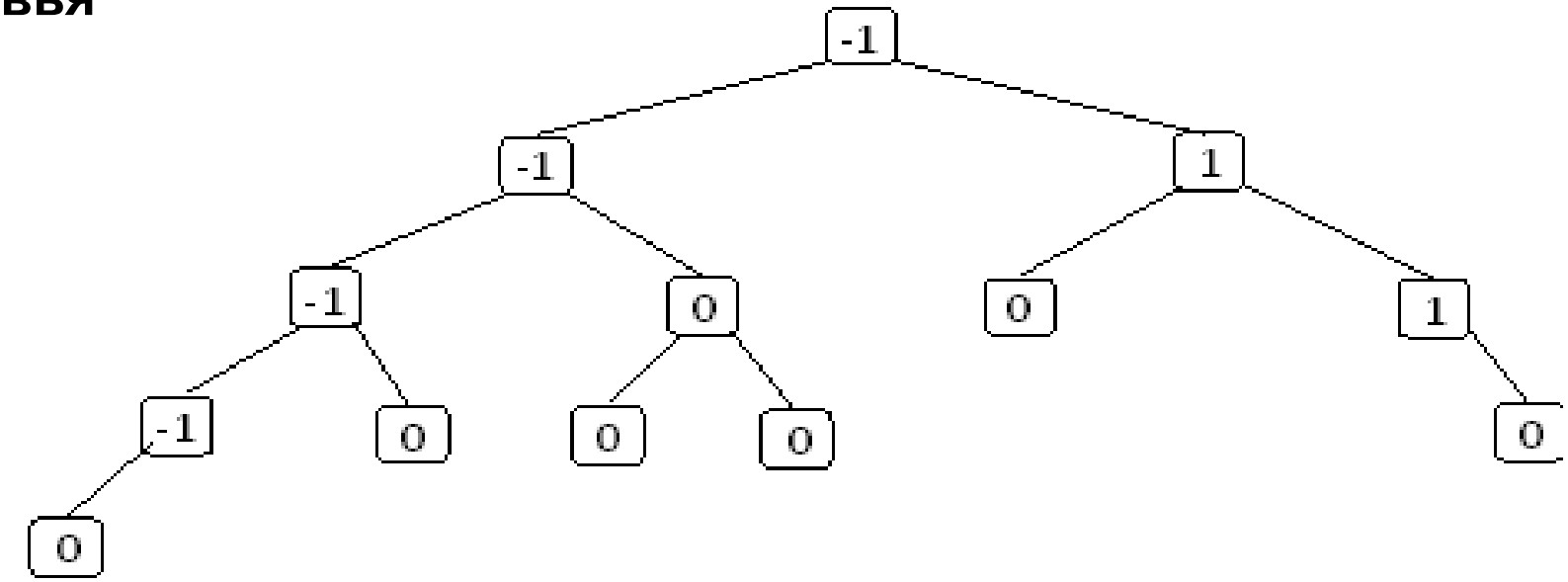
АВЛ-деревья

- 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.
- *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.



Пример АВЛ-дерева.

АВЛ-деревья



- В узлах дерева записаны значения *показателя сбалансированности* (*balance factor*), определяемого по формуле:

  $$balance\ factor = height(right\ subtree) - height(left\ subtree)$$

  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 баланса 0 (это самое «хорошее» АВЛ-дерево) а у дерева Фибоначчи *все* узлы имеют показатель баланса +1 (либо –1) (это самое «плохое» АВЛ-дерево).

АВЛ-деревья

- Типичная структура узла АВЛ-дерева:

```
typedef int key_t;
struct avlnode;
typedef struct avlnode *avltree;
struct avlnode {
  key_t     key;        //ключ
  avltree   left;       //левое поддерево
  avltree   right;      //правое поддерево
  //  int   balance;    показатель баланса
  int       height;     //высота поддерева
};
```

АВЛ-деревья.

- Базовые операции над АВЛ-деревьями.

```
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
```

Реализация простейших базовых операций

- Удалить дерево:

```
avltree makeempty (avltree t) {
  if (t != NULL) {
    makeempty (t->left);
    makeempty (t->right);
    free (t);
  }
  return NULL;
}
```

- Поиск по ключу:

```
avltree find (key_t x, avltree t) {
 if (t == NULL || x == t->key)
   return t;
 if (x < t->key)
   return find (x, t->left);
 if (x > t->key)
   return find (x, t->right);
}
```

Реализация простейших базовых операций

- Минимальный и максимальный ключи:

```
avltree findmin (avltree t) {
  if (t == NULL)
    return NULL;
  else if (t->left == NULL)
    return t;
  else
    return findmin (t->left);
}

avltree findmax (avltree t) {
  if (t != NULL)
    while (t->right != NULL)
      t = t->right;
  return t;
}
```

Включение узла в АВЛ-дерево

- ***Поддержка балансировки АВЛ-дерева при выполнении операции включения ключей***

  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.

- h_L **не изменяется** $\Rightarrow$ не изменяются соотношения между h_L и h_R $\Rightarrow$ свойства АВЛ-дерева сохраняются.

- h_L **увеличивается на 1** $\Rightarrow$ возможны три случая:

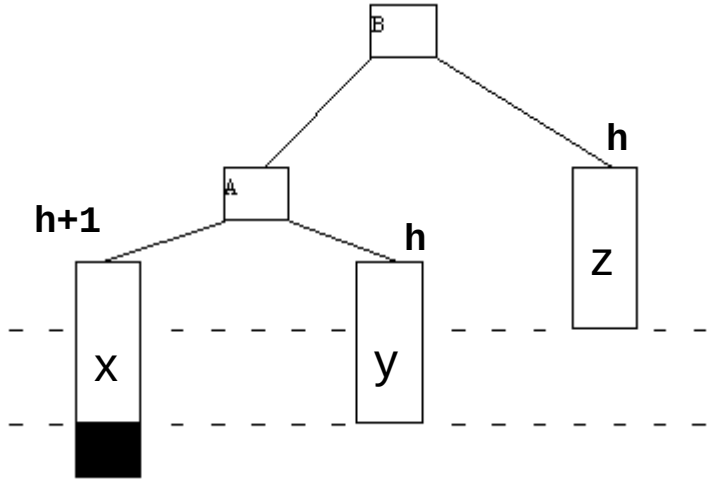
  (1) $h_L = h_R \Rightarrow$ после добавления вершины L и R станут разной высоты, но свойство сбалансированности сохранится

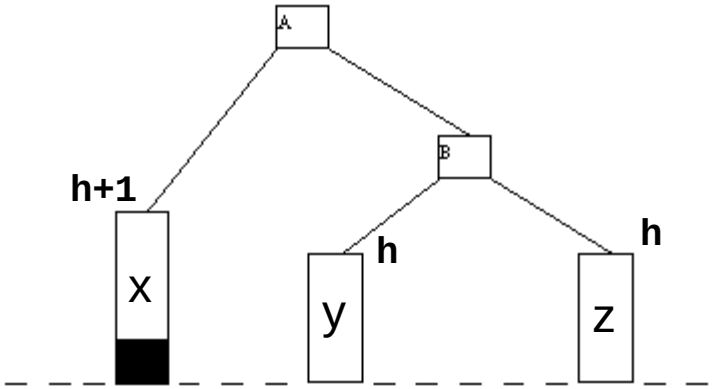
  (2) $h_L < h_R \Rightarrow$ после добавления новой вершины L и R станут равной высоты, т.е. сбалансированность общего дерева даже улучшится

  (3) $h_L > h_R \Rightarrow$ после включения ключа сбалансированность нарушится, и *потребуется перестройка дерева*.

Включение узла в АВЛ-дерево

- (3а) **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.



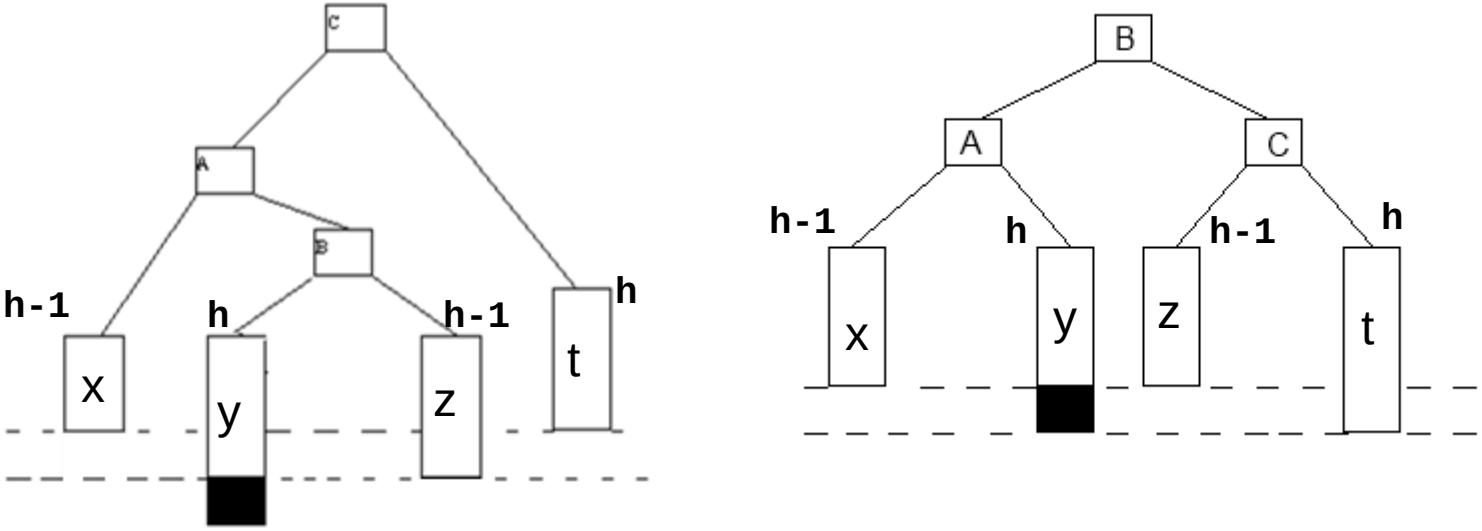


Преобразование, разрешающее ситуацию (3а) (однократный поворот $\boldsymbol{RR}$):

Делаем узел A корневым узлом поддерева, в результате правое поддерева с корнам в узле B «опускается» и разность высот становится равной 0

Включение узла в АВЛ-дерево

- (3b) **Новая вершина добавляется к правому поддереву поддерева** L. В результате поддерево с корнем в C разбалансировалось: разность высот его левого и правого поддеревьев стала равной -2.



Преобразование, разрешающее ситуацию (3b) (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 корневыми узлами B и C.

Построение АВЛ-дерева

- Высота поддерева с корнем в узле P.

```
static inline int height (avltree p) {
  return p ? p->height : 0;
}
```

- Выбор более длинного поддерева

```
static inline int max (int lhs, int rhs) {
  return lhs > rhs ? lhs : rhs;
}
```

Построение АВЛ-дерева

- Однократные повороты
  - Между узлом и его левым сыном

Функция SingleRotateWithLeft вызывается только в том случае, когда у узла К2 есть левый сын. Функция выполняет поворот между узлом (К2) 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 переставленных узлов */
  k2->height = max (height (k2->left),
                    height (k2->right)) + 1;
  k1->height = max (height (k1->left), k2->height) + 1;
  return k1; /* новый корень */
}
```

Построение АВЛ-дерева

- Однократные повороты
  - Между узлом и его правым сыном

Эта функция 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;
  k2 = k1->right;
  k1->right = k2->left;
  k2->left = k1;
  k1->height = max (height (k1->left),
                    height (k1->right)) + 1;
  k2->height = max (height (k2->right), k1->height) + 1;
  return k2; /* новый корень */
}
```

Построение АВЛ-дерева

- Двойные повороты
  - *LR*- поворот

```
Эта функция вызывается только тогда, когда
у узла K3 есть левый сын, а у левого сына
K3 есть правый сын. Функция выполняет двойной
поворот LR, корректирует высоты поддеревьев, 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);
}
```

Построение АВЛ-дерева

- Двойные повороты
  - *RL*- поворот

Эта функция вызывается только в том случае, когда у узла K1 есть правый сын, а у правого сына узла K1 есть левый сын. Функция выполняет двойной поворот RL, корректирует высоты поддеревьев, после чего возвращает новый корень

```
static avltree DoubleRotateWithRight (avltree k1){
  /* Поворот между K3 и K2 */
  k1->right = SingleRotateWithLeft (k1->right);
  /* Поворот между K1 и K2 */
  return SingleRotateWithRight(k1);
}
```

Построение АВЛ-дерева

- Вставить новый узел (все)

```
avltree insert (key_t x,
                avltree t) {
  if (t == NULL) {
    /* создание дерева с одним узлом */
    t = malloc (sizeof
               (struct avlnode));
    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;
    //t->balance = height (t->right)
    //             – height (t->left);
    return t;
}
```

Построение АВЛ-дерева

- Вставить новый узел

```
avltree insert (key_t x, avltree t) {
  if (t == NULL) {
    /* создание дерева с одним узлом */
    t = malloc (sizeof (struct avlnode));
    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) {
```

- Вставить новый узел

```
    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;
  //t->balance = height (t->right) – height (t->left);
  return t;
}
```

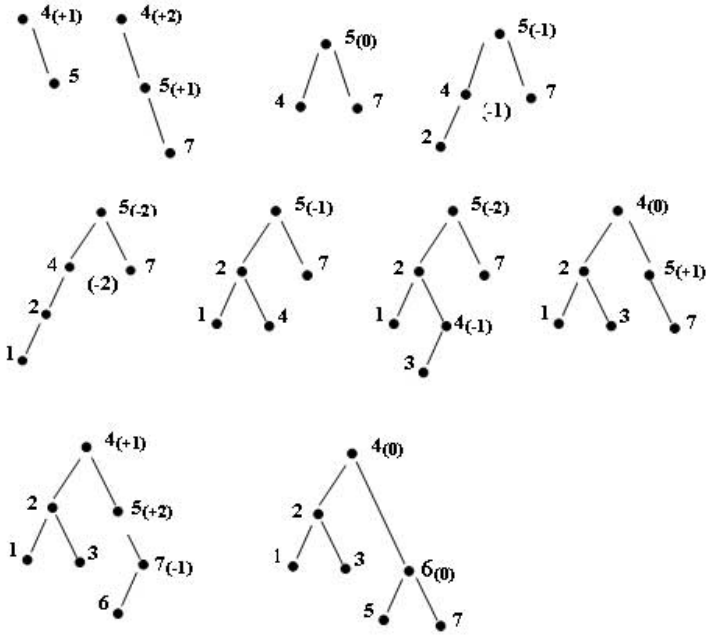
Построение АВЛ-дерева

- ***Пример***

Пусть на «вход» функции **`insert()`** последовательно поступают целые числа 4,5,7,2,1,3,6.

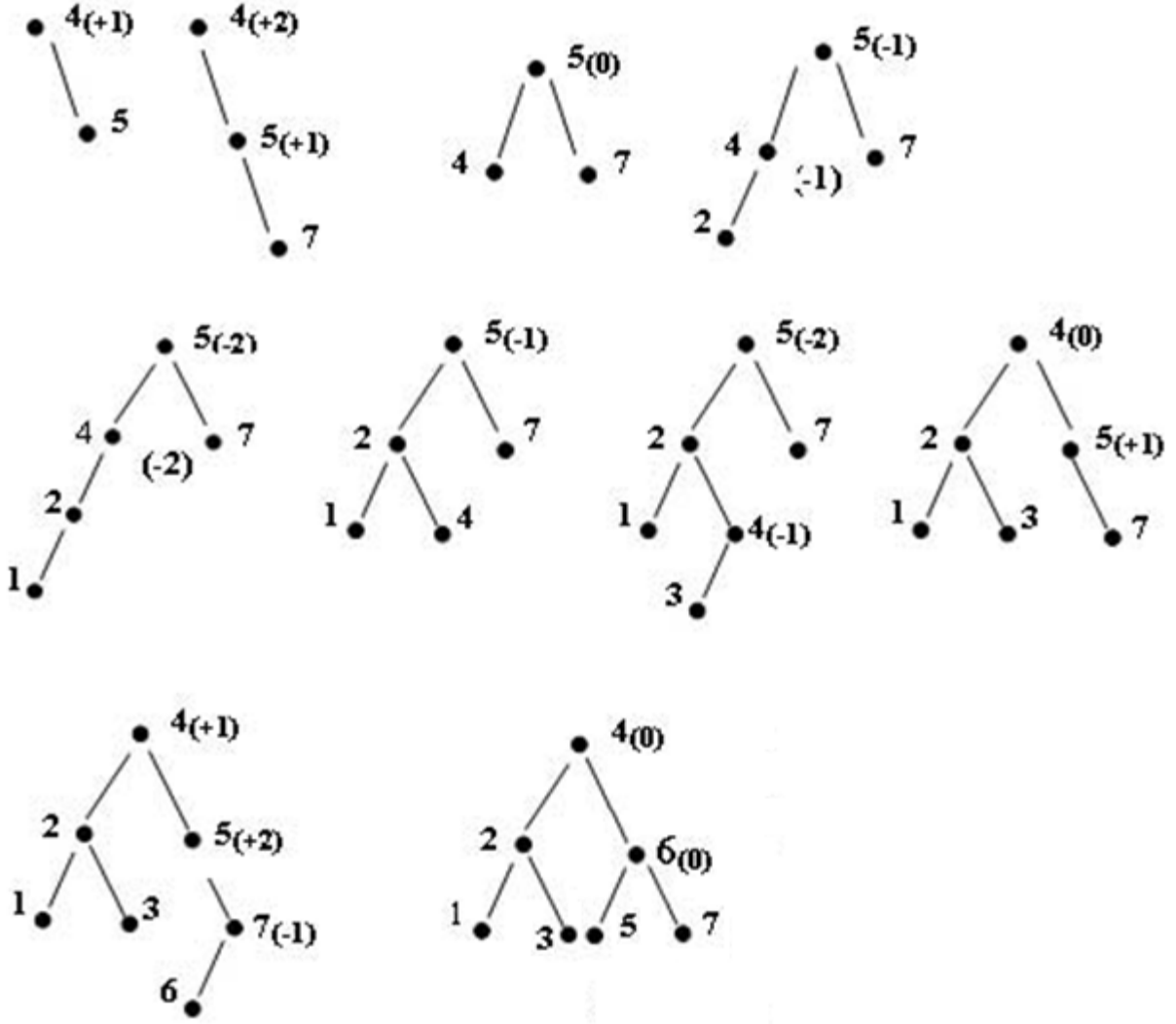
Изобразим процесс «роста» АВЛ-дерева (в скобках для части вершин указан показатель сбалансированности):

Числа в примере подобраны так, чтобы обеспечить как можно больше поворотов при минимальном числе включений.



Построение АВЛ-дерева

- ***Пример построения АВЛ-дерева***



Исключение узла из АВЛ-дерева

- Объявление функции:

```
avltree delete (key_t x, avltree t);
```

- Исключение узла из АВЛ-дерева требует балансировки дерева.

  Иными словами, в конец функции, выполняющей исключение узла, необходимо добавить вызовы функций:
  `SingleRotateWithRight(T)`, **`SingleRotateWithLeft(T)`**, **`DoubleRotateWithRight(T)`** и **`DoubleRotateWithLeft(T)`**

- Возможны случаи вращения, не встречавшиеся при вставке.
- Может оказаться необходимым выполнить несколько вращений.

Оценка сложности

- Ранее были получены оценки высоты

(1) самого «хорошего» АВЛ-дерева, содержащего m узлов (*полностью сбалансированное дерево*)

$$h = O(\log_2(m+1))$$

(2) самого «плохого» АВЛ-дерева, содержащего m узлов (*дерево Фибоначчи*)

$$h \le 1.44 \cdot \log_2(m+1) - 0.32$$

Следовательно, для «среднего» АВЛ-дерева, содержащего m узлов оценка высоты будет где-то посередине:

$$\log_2(m+1) \le h \le 1.44 \cdot \log_2(m+1) - 0.32$$

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 21

Красно-черные деревья

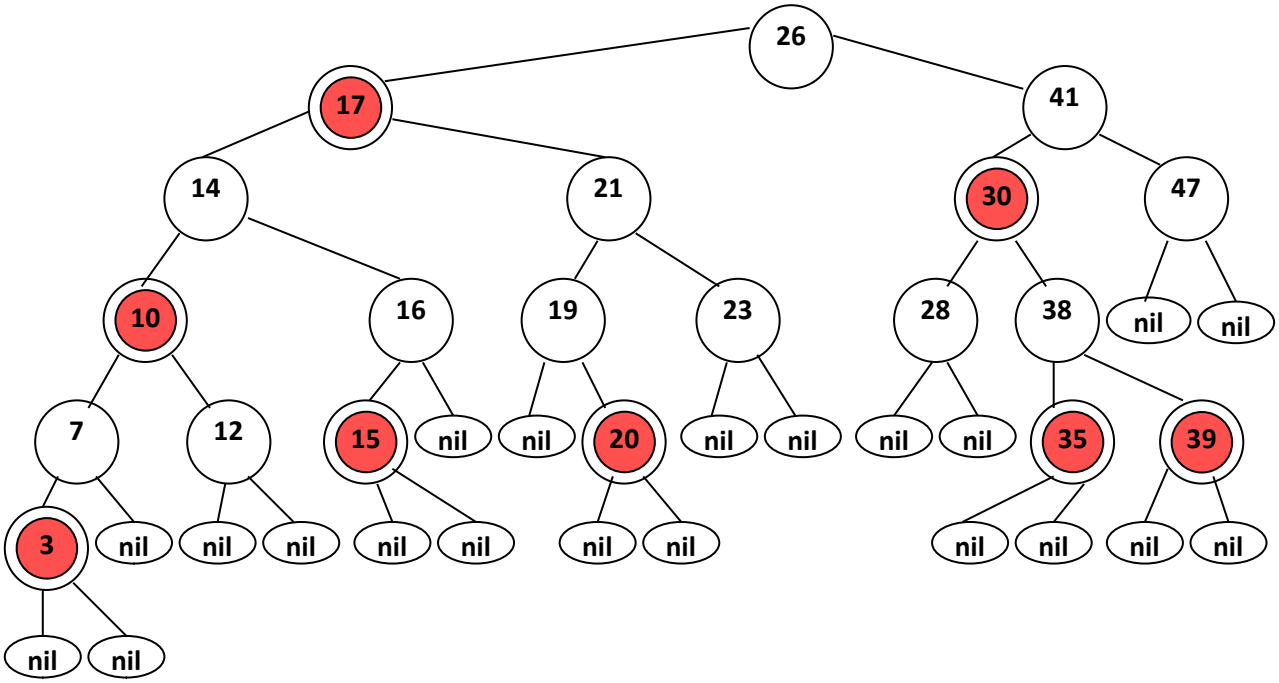
- Красно-черное дерево – двоичное дерево поиска, каждая вершина которого окрашена либо в красный, либо в черный цвет
- Поля – цвет, дети, родители

```
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, т.е. все вершины внутренние

Красно-черные деревья

- Свойства красно-черных деревьев:
  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



Красно-черные деревья

- Обозначим $bh(x)$ – "черную" высоту поддерева с корнем x (саму вершину в число не включаем), т.е. количество черных вершин от x до листа
- Черная высота дерева – черная высота его корня
- *Лемма*: Красно-черное дерево с n внутренними вершинами (без фиктивных листьев) имеет высоту не более $2log_2(n+1)$.

  (1) Покажем вначале, что поддерево x содержит не меньше $2^{bh(x)} - 1$ внутренних вершин

  (1а) Индукция. Для листьев $bh = 0$, т.е. $2^{bh(x)} - 1 = 2^0 - 1 = 0$.

  (1б) 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 – $k - 1$).

  (1в) По предположению индукции каждый сын имеет не меньше $2^{k-1} - 1$ вершин. Поэтому поддерево x имеет не меньше $2^{k-1} - 1 + 2^{k-1} - 1 + 1 = 2^k - 1$.

Красно-черные деревья

- *Лемма*: Красно-черное дерево с n внутренними вершинами (без фиктивных листьев) имеет высоту не более $2log_2(n+1)$.

  (2) Теперь пусть высота дерева равна h.

  (2а) По свойству 3 черные вершины составляют не меньше половины всех вершин на пути от корня к листу. Поэтому черная высота дерева bh не меньше $h/2$.

  (2б) Тогда $n \geq 2^{h/2} - 1$ и $h \leq 2log_2(n + 1)$. Лемма доказана.

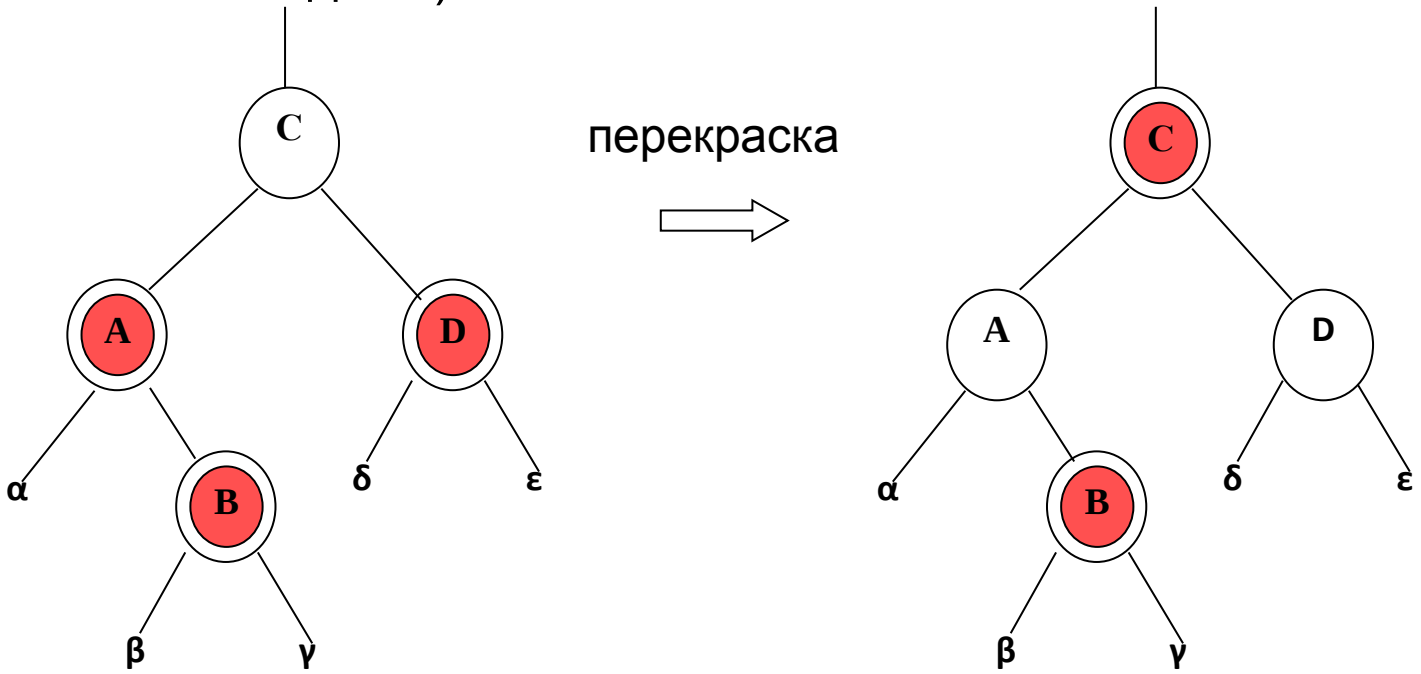
- Следовательно, поиск по красно-черному дереву имеет сложность $O\ (log_2 n)$.

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, то красим новый корень в черный цвет
- Свойство 4 при вставке изначально не нарушено, т.к. новая вершина красная
- Если родитель новой вершины черный (новая – красная), то свойство 3 также не нарушено
- Иначе (родитель красный) свойство 3 нарушено

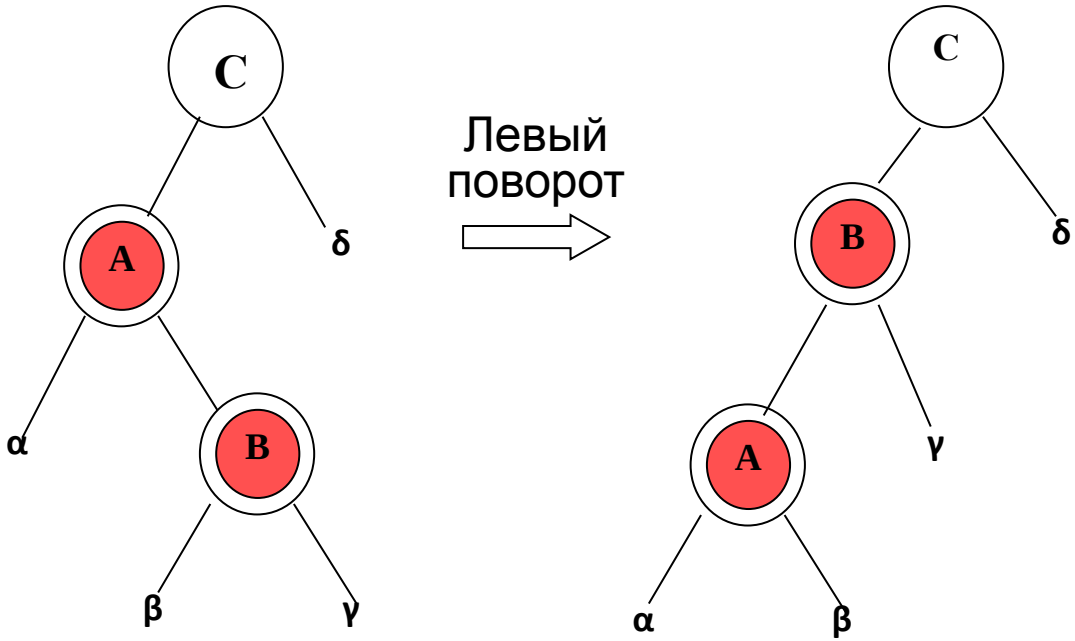
Красно-черные деревья: вставка вершины

- Случай 1: “дядя” (второй сын родителя родителя текущей вершины) тоже красный (как текущая вершина и родитель)
  - Возможно выполнить перекраску:
    родителя и дядю (вершины A и D) – в черный цвет,
    деда – (вершина C) – в красный цвет
  - Свойство 4 не нарушено (черные высоты поддеревьев совпадают)



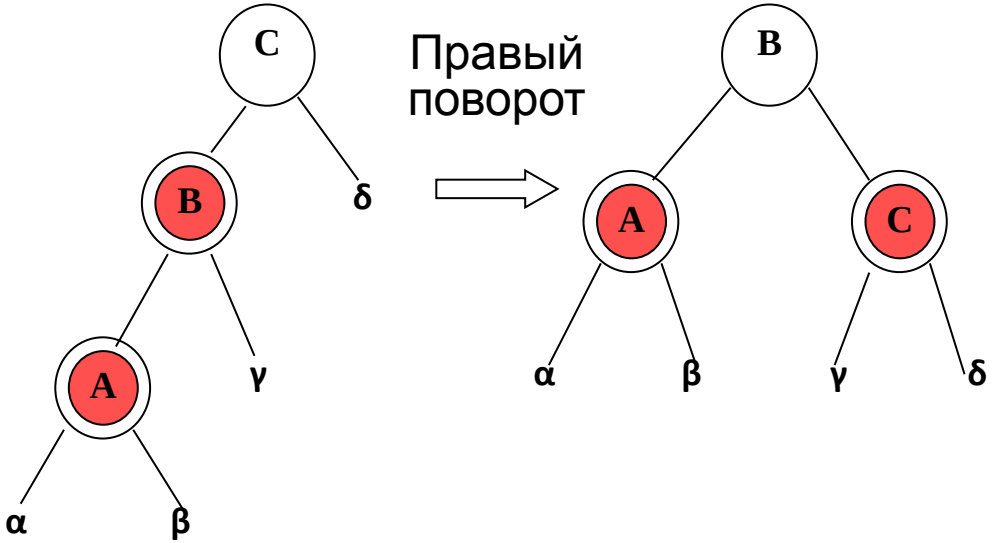
Красно-черные деревья: вставка вершины

- Случай 2: “дядя” (второй сын родителя родителя текущей вершины) черный
  - Шаг 1: Необходимо выполнить левый поворот родителя текущей вершины (вершины A)



Красно-черные деревья: вставка вершины

- Случай 2: “дядя” (второй сын родителя родителя текущей вершины) черный
  - Шаг 2: Необходимо выполнить правый поворот вершины C, после чего …
  - Шаг 3: … перекрасить вершины B и C
  - Все поддеревья имеют черные корни и одинаковую черную высоту, поэтому свойства 3 и 4 верны



Пирамидальная сортировка (heapsort)

- Можно использовать дерево поиска для сортировки
- Например, последовательный поиск минимального элемента, удаление его и вставка в отсортированный массив
  - Сложность такого алгоритма есть $O(nh)$, где h – высота дерева
- Недостатки:
  - Требуется дополнительная память для дерева
  - Требуется построить само дерево (с минимальной высотой)
- Можно ли построить похожий алгоритм без требований к дополнительной памяти?

Пирамидальная сортировка: пирамида (двоичная куча)

- Рассматриваем массив **a** как двоичное дерево:
  - Элемент **a[i]** является узлом дерева
  - Элемент **a[i/2]** является родителем узла **a[i]**
  - Элементы **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]**

Пирамидальная сортировка: пирамида (двоичная куча)

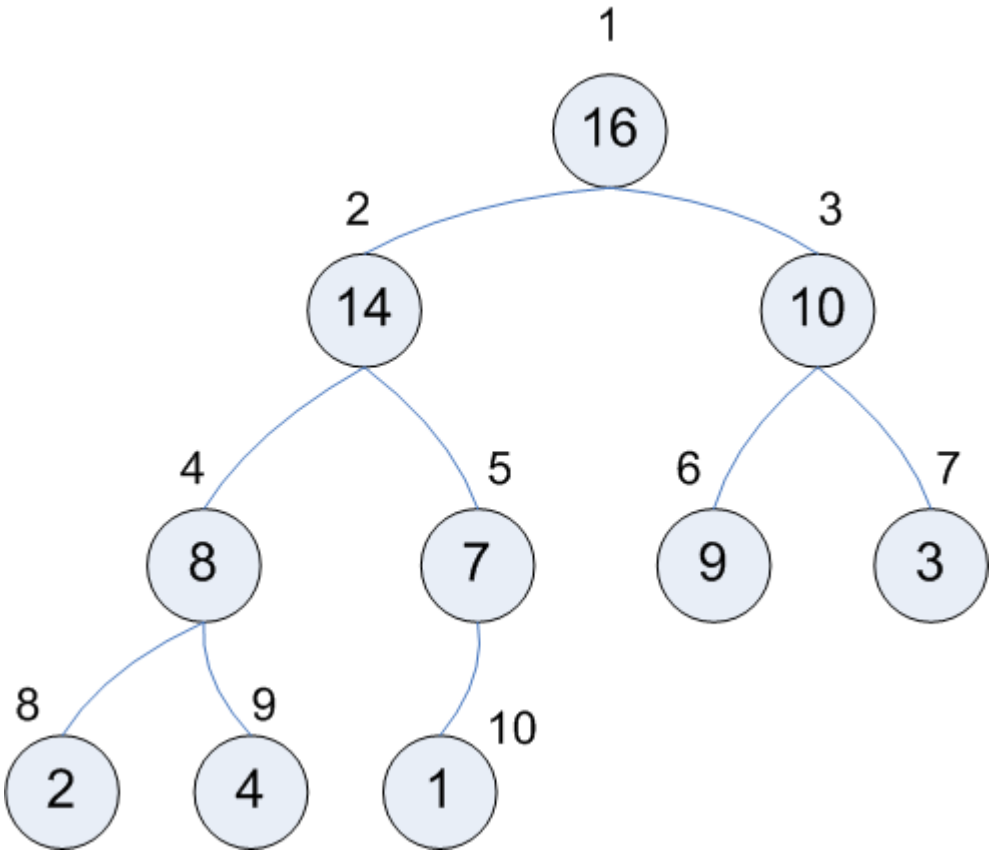
- Для всех элементов пирамиды выполняется соотношение:

  `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



Пирамидальная сортировка: просеивание элемента

- Как добавить элемент в уже существующую пирамиду?
- Алгоритм:
  - Поместим новый элемент в корень пирамиды
  - Если этот элемент меньше одного из сыновей:
    - Элемент меньше наибольшего сына
    - Обменяем элемент с наибольшим сыном (это позволит сохранить свойство пирамиды для другого сына)
    - Повторим процедуру для обмененного сына

Пирамидальная сортировка: просеивание элемента

```
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++;
  }
}
```

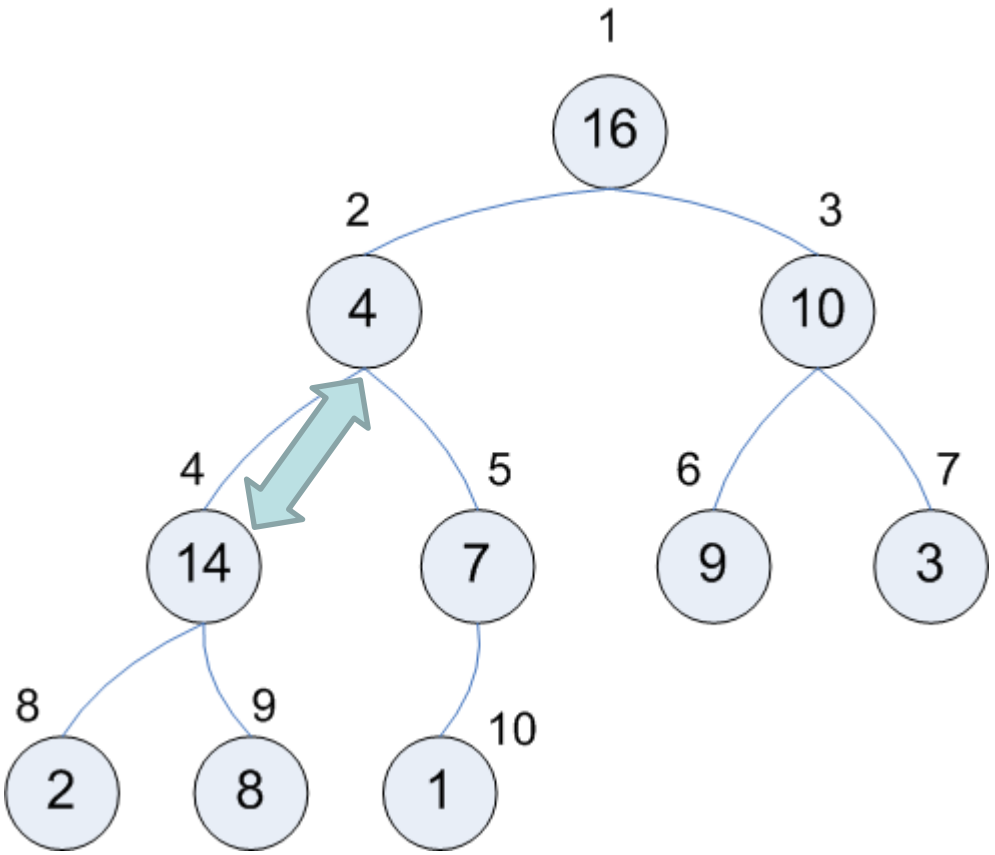
Пирамидальная сортировка: просеивание элемента

```
/* l, r - от 0 до n-1 */
static void sift (int *a, int l, int r) {
  int i, j, x;

  /* Теперь l, r, i, j от 1 до n, а индексы массива
     уменьшаются на 1 при доступе */
  l++, r++;
  i = l; j = 2*l; x = a[l-1];
  /* j указывает на наибольшего сына */
  if (j < r && a[j-1] < a[j])
    j++;
  /* i указывает на отца */
  while (j <= r && x < a[j-1]) {
    /* обмен с наибольшим сыном: a[i-1] == x */
    a[i-1] = a[j-1]; a[j-1] = x;
    /* продвижение индексов к следующему сыну */
    i = j; j = 2*j;
    /* выбор наибольшего сына */
    if (j < r && a[j-1] < a[j])
      j++;
  }
}
```

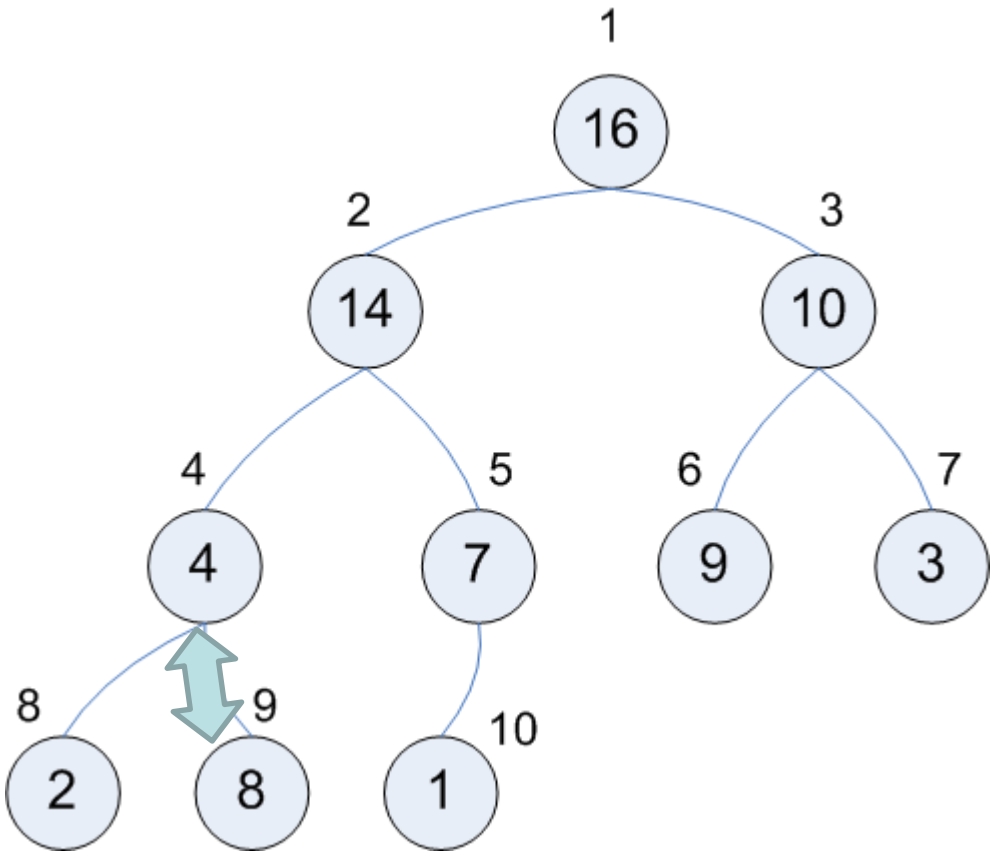
Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



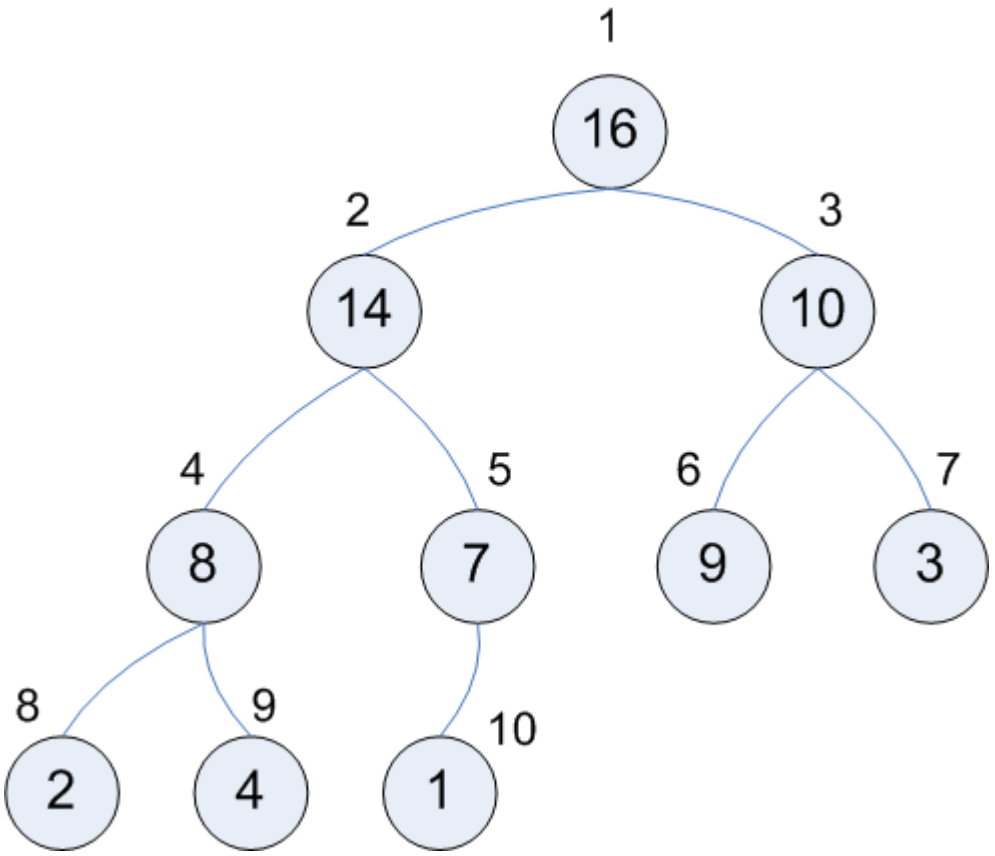
Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



Пирамидальная сортировка: алгоритм

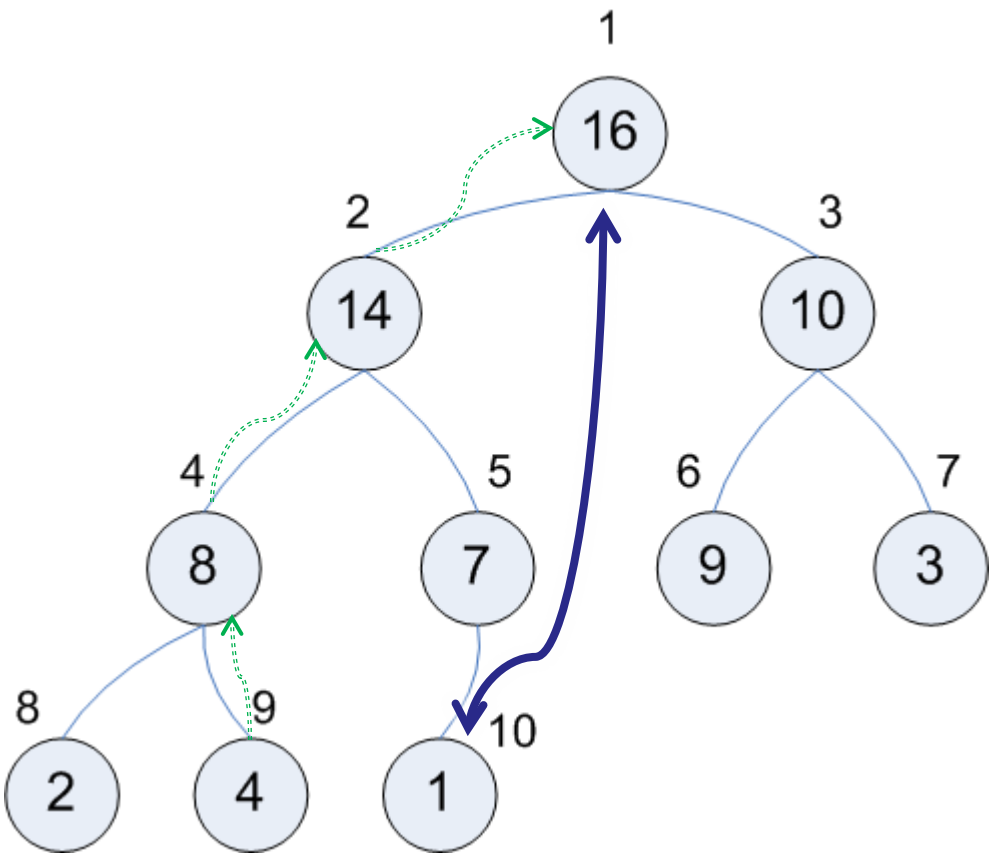
- (1) Построим пирамиду по сортируемому массиву
  - Элементы массива от n/2 до n являются листьями дерева, а следовательно, правильными пирамидами из одного элемента
  - Для остальных элементов в порядке уменьшения индекса просеиваем их через правую часть массива
- (2) Отсортируем массив по пирамиде
  - Первый элемент массива максимален (корень пирамиды)
  - Поменяем первый элемент с последним (таким образом, последний 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.

Пирамидальная сортировка: программа

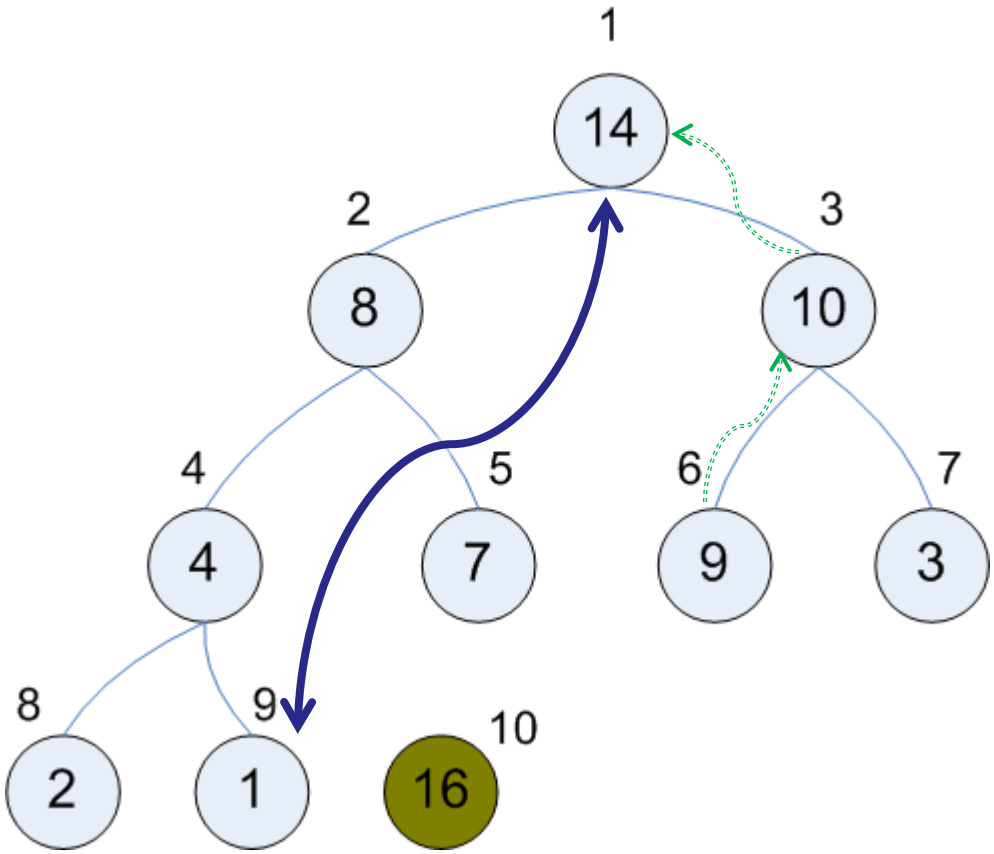
```
void heapsort (int *a, int n) {
  int i, x;

  /* Построим пирамиду по сортируемому массиву */
  /* Элементы нумеруются с 0 -> идем от 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);
  }
}
```

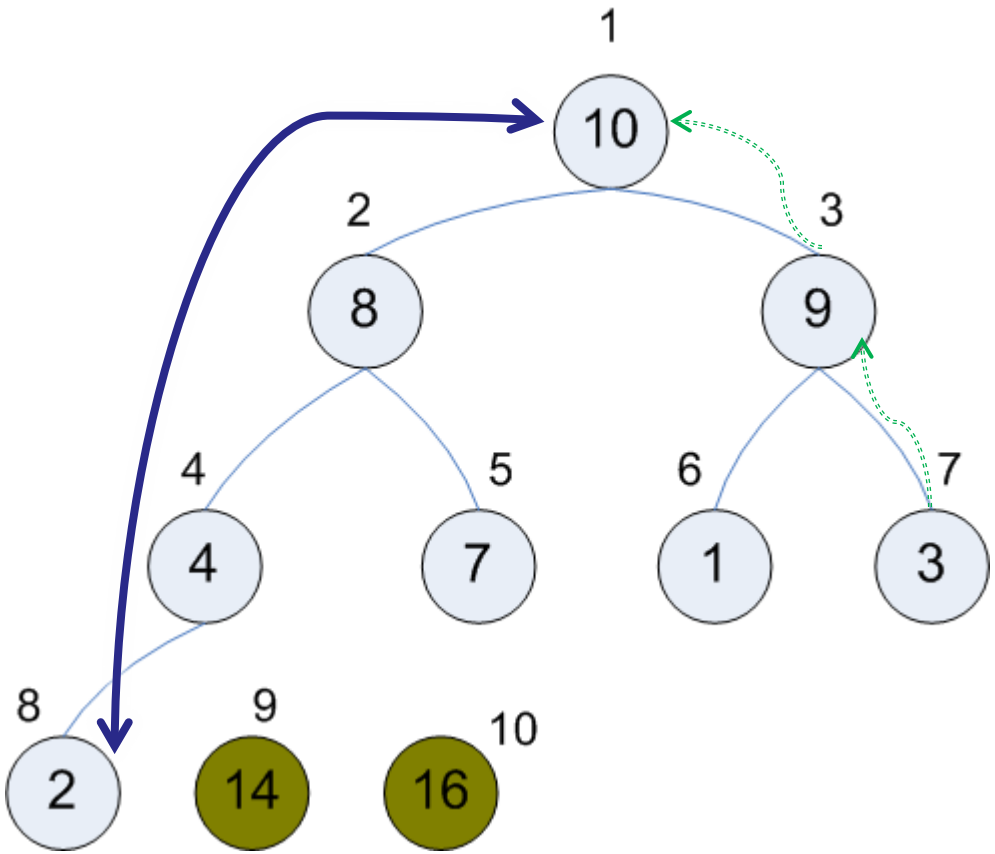
Пирамидальная сортировка: пример



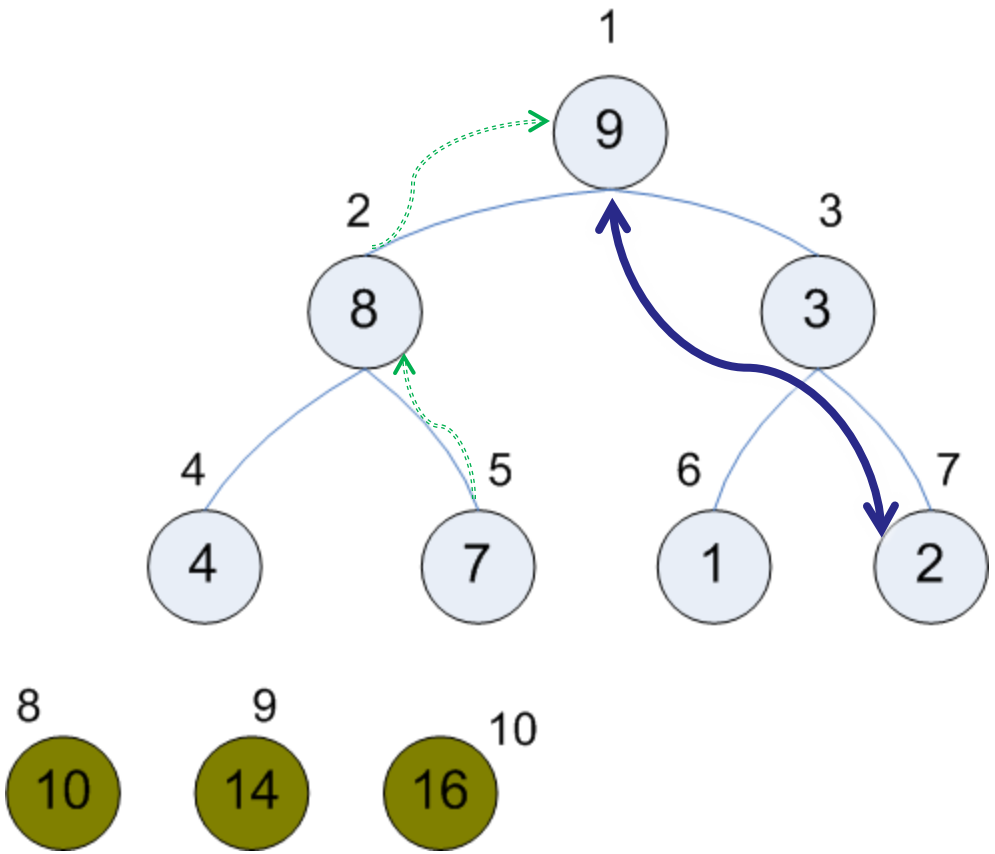
Пирамидальная сортировка: пример



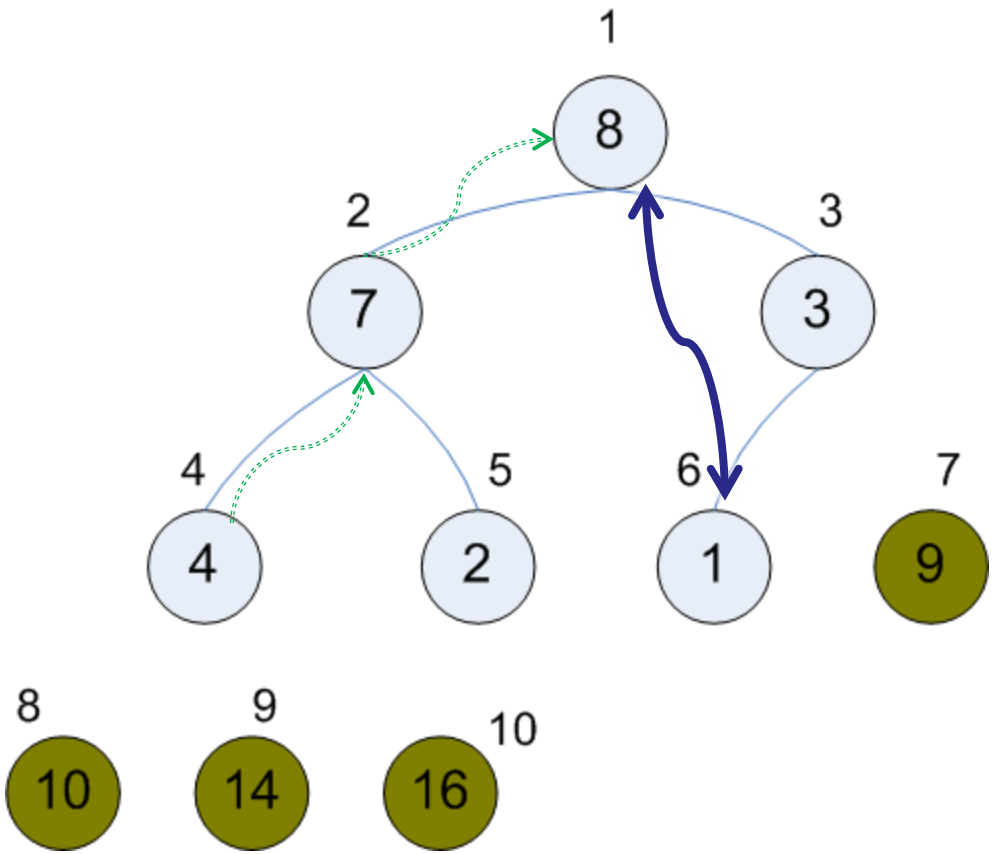
Пирамидальная сортировка: пример



Пирамидальная сортировка: пример



Пирамидальная сортировка: пример



Пирамидальная сортировка: сложность алгоритма

- (1) Построим пирамиду по сортируемому массиву
  - Элементы массива от n/2 до n являются листьями дерева, а следовательно, правильными пирамидами из 1 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, последний 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.

- Сложность этапа построения пирамиды есть $O(n)$
- Сложность этапа сортировки есть $O(n \log n)$
- Сложность в худшем случае также $O(n \log n)$
- Среднее количество обменов – $n/2 * \log n$

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 22

Хеш-таблицы

- Словарные операции: *добавление*, *поиск* и *удаление* элементов по их ключам.
- Организуется таблица ключей: массив **Index[m]** длины **m**, элементы которого содержат значение ключа и указатель на данные (информацию), соответствующие этому ключу.
  - ***Прямая адресация***. Применяется, когда количество возможных ключей невелико: например, ключи перенумерованы целыми числами из множества $U = \{0, 1, 2, \ldots, m - 1\}$, 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, т.к. совпадает с индексом.
  - Все три словарные операции 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).

Хеш-таблицы

- ***Хеширование*** тоже позволяет обеспечить среднее время операций с данными $T_{ср}(n) = O(1)$ и тоже за счет использования таблицы *Index*.
- Хеш-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).
- Пример использования хеширования – таблица идентификаторов программы, составляемая компилятором.
- В случае хеш-адресации элементу с ключом *key* отводится строка таблицы с номером *hash*(*key*), где
  hash: $U \to \{0, 1, 2, \ldots, m - 1\}$ – *хеш-функция*.
  Число *hash*(*key*) называется *хеш-значением* ключа *key*.
- Если хеш-значения ключей key_1 и key_2 совпадают ($hash(key_1) == hash(key_2)$), 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 $|U| > m$.

Хеш-таблицы

- Простейший способ обработки коллизий – сцепление элементов с одинаковыми значениями хеш-функции: все такие элементы сцепляются в список, а в хеш-таблицу помещается указатель на первый элемент этого списка. В пределах каждого такого списка осуществляется последовательный поиск.
- В случае использования двусвязного списка среднее время выполнения каждой из трех словарных 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.

Хеш-таблицы

- Устройство простой хеш-таблицы (реализация хеширования с цепочками).
  - Задается некоторое фиксированное число *m* (типичные значения *m* от 100 до 1,000,000).
  - Создается массив **Index[m]** указателей начал двусвязных списков (цепочек), который называется *индексом* хеш-таблицы. В начале работы все указатели имеют значения *NULL*.
  - Задается хеш-функция *hash*(), которая получает на вход ключи и выдает значение от 0 до *m* – 1.
  - При добавлении пары (*key*, *value*) вычисляется *h* = *hash*(*key*) и пара добавляется в список **Index[h].**
  - При удалении либо поиске пары (*key*, *value*) вычисляется *h* = *hash*(*key*) и происходит удаление либо поиск пары (*key*, *value*) в списке **Index[h]**.

Хеш-таблицы

- Анализ хеширования с цепочками.
  - Пусть **Index[m]** – хеш-таблица с *m* позициями, 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 $D = \frac{1}{m}\sum_{i=0}^{m-1} \left(M - |\, Index[i]\,|\right)^2 \le \alpha$.
  - Это условие исключает наихудший случай, когда хеш-значения всех ключей одинаковы, заполнен только один список и поиск в этом списке из *n* элементов имеет среднее время $\Theta(n)$.

Хеш-таблицы

- Анализ хеширования с цепочками.
  - *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.
  - Условие из предыдущего слайда выполняется и *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 словарных операций с учетом вычисления хеш-функции равно $\Theta(1 + \alpha)$.

Хеш-таблицы

- ***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 на вычисление 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 ее элементы равновероятны.
- Из теорем следует, что в случае равномерного хеширования среднее время выполнения любой словарной операции есть $O(1)$.

Методы построения хеш-функций

- 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 к равномерному.
  - Правильный выбор *m*: в качестве *m* выбирается достаточно большое простое число, далеко отстоящее от степеней двойки.
  - Например, если устраивает средняя длина списков 3, а число записей, доступ к которым нужно обеспечить с помощью хеш-таблицы ≈ 2000, то можно взять *m* = 2000/3 ≈ 701. Тогда *hash*(*key*) = *key %* 701.
  - Недостаток: в качестве *m* нельзя брать степень двойки, так как если $m = 2^p$, то *hash*(*key*) – это просто *p* младших битов числа *key*.

Методы построения хеш-функций

- Построение хеш-функции ***методом умножения***.
  - Пусть количество хеш-значений равно *m*.
    Выберем и зафиксируем вещественную константу
    v, 0 < *v* < 1; положим ***hash(key) =*** $\lfloor$**m(frac(key·*v*)**$\rfloor$
    frac(key·*v*) – дробная часть числа *key·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 случае 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 *key* на *w*-битовое целое число $v \cdot 2^w$; получится 2*w*-битовое число r_0.

    В качестве значения *hash*(*key*) возьмем старшие *p* битов “дробной” части числа $r_0 / 2^w$ ($r_0 \,\%2^w$ или обнуление *w* старших разрядов, потом умножение на $m = 2^p$).
  - Согласно Д. Кнуту выбор $v = \left(\sqrt{5}-1\right)/2 = 0.6180339887...$ является удачным.

Хеш-функции: программы

```
#define MAX 701   /* размер хеш-таблицы */
struct htype {
  int key;        /* ключ */
  int val;        /* значение элемента данных */
  struct htype *next;   /* указатель на следующий элемент
                           цепочки */
  struct htype *prvs;   /* указатель на предыдущий элемент
                           цепочки */
};
struct htype *index[MAX];
```

Хеш-функции: программы

```
#define MAX 701   /* размер хеш-таблицы */
static inline int hash (int key) {
  return key % MAX;
}
/* инициализация хеш-таблицы */
void init (void) {
  int i;
  for (i = 0; i < MAX; i++)
    index[i] = NULL; /* массив начал цепочек */
}
```

Хеш-функции: программы

```
/* Вычисление хеш-адреса и поиск по ключу k:
   если элемент с ключом k найден, возвращаем указатель
   на него, если нет, возвращаем NULL */
struct htype *search (int k) {
  /* вычисление хеш-адреса */
  int h = hash (k);
  /* поиск ключа k */
  if (index[h]) {
    struct htype *p = index[h];
    do {
      if (p->key == k)
        return p;
      else
        p = p->next;
    } while (p);
  }
  return NULL;
}
```

Хеш-функции: программы

```
/* Порождение нового элемента цепочки и возврат указателя
   на него */
struct htype *new (void) {
  struct htype *p;
  p = (struct htype *) malloc (sizeof (struct htype));
  if (!p)
    abort ();
  p->key = -1;
  p->val = 0;
  p->next = NULL;
  p->prvs = NULL;
  return p;
}
```

Хеш-функции: программы

```
/* Вычисление хеш-адреса и поиск по ключу k: если элемент с ключом k
   найден, 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 search_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;
}
```

Хеш-функции: программы

```
/* 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 цепочки */
    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;
  }
}
```

Хеш-функции: программы

```
/* 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 */
}
```

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»:

$$hash: U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

- Функцию *hash* нужно выбрать такой, чтобы в последовательности проб $\langle hash(k, 0), hash(k, 1), \ldots, hash(k, m-1)\rangle$ каждый номер ячейки 0, 1, …, m – 1 встретился только один 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).

Хеширование с открытой адресацией: программы

```
#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 (key, i);
    if (index[j] && index[j]->key == k)
      return index[j];
  } while (index[j] && ++i < m);
  return NULL;
}
```

Хеширование с открытой адресацией: программы

```
/* Добавление элемента */
int insert (int k, int v) {
  int i = 0, j;

  do {
    j = hash (key, 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; /* Или расширим index */
  index[j] = new ();
  index[j]->key = k, index[j]->val = v;
  return j;
}
```

Хеширование с открытой адресацией: программы

```
/* Внутренний поиск: вернем индекс массива */
static int search_internal (int k) {
 int i = 0, j;

  do {
    j = hash (key, i);
    if (index[j] && index[j]->key == k)
      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;
}
```

Хеширование с открытой адресацией: программы

```
/* 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 удаляемым ключом (с тем же хешем). */
  ???
}
```

Хеширование с открытой адресацией: программы

```
#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;
}
```

Хеширование с открытой адресацией: программы

```
#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 (key, i);
    if (!ISEMPTY (index[j]) && index[j]->key == k)
      return j;
   } while (index[j] && ++i < m);
  return -1;
}
```

Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)
#define ISEMPTY(el) ((!el) || (el) == SHADOW)

/* Добавление элемента */
int insert (int k, int v) {
  int i = 0, j;

  do {
    j = hash (key, 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; /* Или расширим index */
  index[j] = new ();
  index[j]->key = k, index[j]->val = v;
  return j;
}
```

Хеш-функции для открытой адресации

- *Линейная последовательность* проб.
  Пусть $hash'$: $U \to \{0, 1, \ldots, m - 1\}$ – обычная хеш-функция.
  Функция $hash(k, i) = (hash'(k) + i) \bmod m$
  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, т.к. каждая последовательность однозначно определяется своим первым элементом.

Хеш-функции для открытой адресации

- Серьезный недостаток – тенденция к образованию *кластеров* (длинных последовательностей занятых ячеек, идущих подряд), что удлиняет поиск:
  - 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 не равномерное.
- Возможное улучшение: добавляем не 1, а константу c, взаимно простую с m (для полного обхода таблицы).

Хеш-функции для открытой адресации

- *Квадратичная последовательность* проб:
  $hash(k, i) = (hash'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$,
  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. Метод работает значительно лучше, чем линейный.
- Чтобы при просмотре таблицы *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 $j \leftarrow (j + 1) \bmod m$, $i \leftarrow (i + j) \bmod m$ и повторяем текущий шаг.

Хеш-функции для открытой адресации

- *Двойное хеширование* – один из лучших методов открытой адресации.

  $hash(k, i) = (h_1(k) + i\, h_2(k)) \bmod m,$

  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 (т.е. вероятность совпадения их хешей обратно пропорциональна 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 $\frac{1}{1-\alpha}$.

Хеширование других данных

- Хеширование идентификаторов в компиляторе

```
hashval_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;
}
```

- Хеширование ключа переменной длины: в GCC используется http://burtleburtle.net/bob/hash/evahash.html

Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

Лекция 23

Цифровой поиск

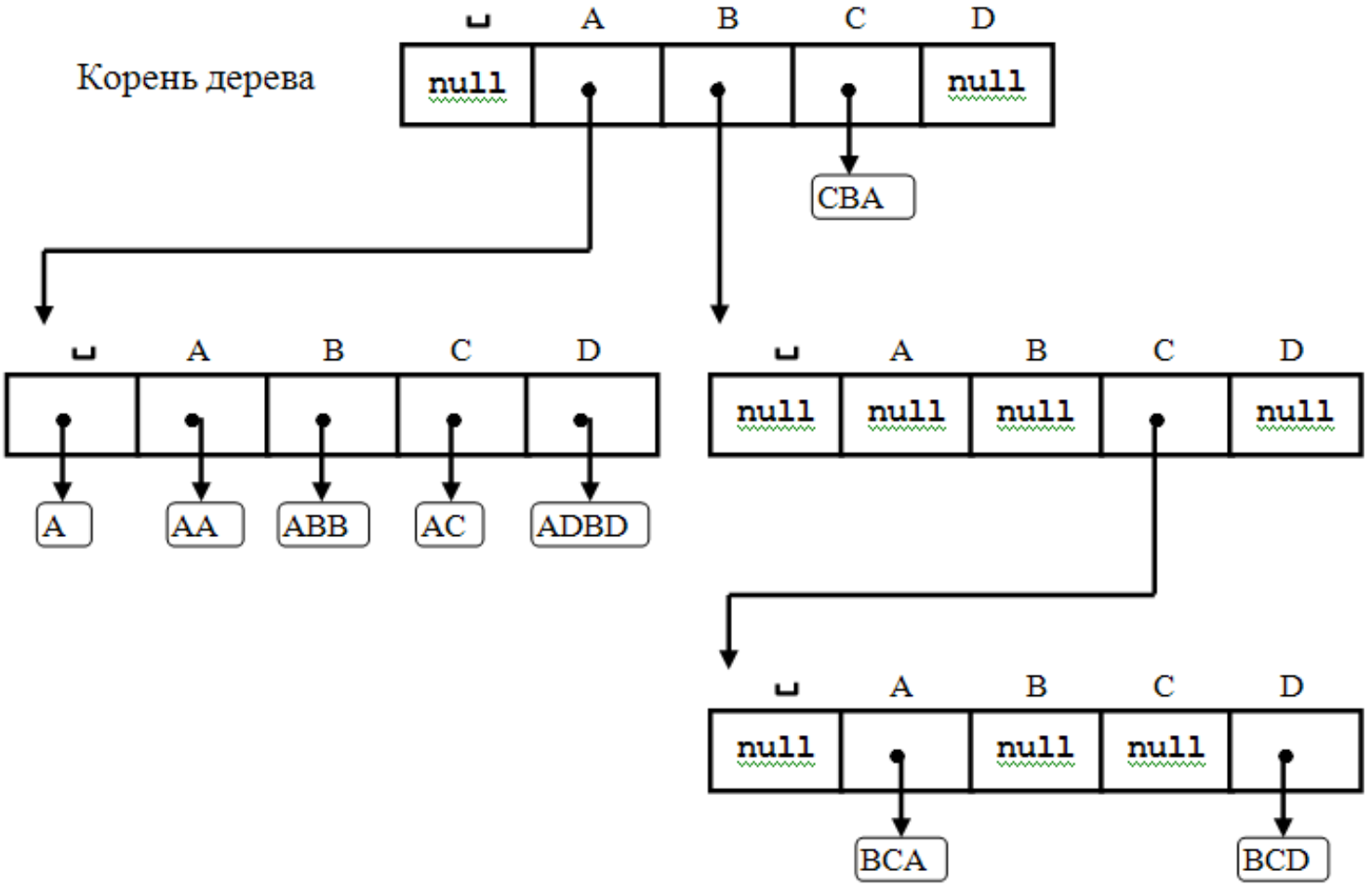
- *Цифровой поиск* – частный случай поиска заданной подстроки (*образца*) в длинной строке (*тексте*).
- *Примеры цифрового поиска*: поиск в словаре, в библиотечном каталоге и т.п., когда делается поиск по образцу в нескольких текстах (названиях книг, фамилиях авторов, текстах на вызванных сайтах и т.п.).
- Хороший пример – *словарь с высечками*, т.е. словарь, в котором обеспечен быстрый доступ к некоторым страницам (например, начальным страницам списков слов, начинающихся на очередную букву алфавита). Иногда используются *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.

Цифровой поиск

- *Пример*. Пусть множество используемых букв (алфавит) {*A*, *B*, *C*, *D*}. Мы добавим к алфавиту еще одну букву␣(пробел). По определению слова *AA*, *AA*␣, *AA*␣␣ совпадают.
  Пусть {*A*, *AA*, *ABB*, *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 $a_1a_2a_3...a_k$ и первые *i* его букв (*i* < *k*) задают уникальное значение: комбинация $a_1...a_i$ 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, т.е. строить 2^m-ичное дерево поиска (на двоичных деревьях для разветвления берется один бит: *m* = 1).

Цифровой поиск

- Прямоугольниками изображены вершины дерева, в овалах – значения слов (ключей) и связанная с ним информация.
  Тем самым любая вершина дерева – массив из *m* элементов.
  Каждый элемент вершины содержит либо ссылку на другую вершину *m*-ичного дерева, либо на овал (ключ).



Цифровой поиск

- Иногда используют комбинации нескольких методов: цифровой поиск вначале, а затем переключение на поиск в последовательных таблицах.
  - Именно так мы и работаем со словарем с высечками: вначале на высечку, а затем либо последовательный поиск, либо дихотомический.
- Обычно предлагается пользоваться цифровым поиском, пока количество различных слов не меньше некоторого k, а затем переключаться на последовательные таблицы.
- Обобщения: поиск по неполным ключам, поиск по образцу.

Цифровой поиск

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

#define M 5

typedef enum {word, node} tag_t;
struct record {
  char *key;
  int value;
};

struct tree {
  tag_t tag;
  union {
    struct record *r;
    struct tree *nodes[M+1];
  };  /* анонимное объединение */
};
```

Цифровой поиск: *поиск элемента*

```
static inline int ord (char c) {
  return c ? c - 'A' + 1 : 0;
}

struct record *find (struct tree *t, char *key) {
  int i = 0;

  while (t) {
    switch (t->tag) {
    case word:
      for (; key[i]; i++)
        if (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;
}
```

Цифровой поиск: *вставка – вспомогательные функции*

```
struct record *make_record (char *key, int value) {
  struct record *r = malloc (sizeof (struct record));
  r->key = strdup (key);
  r->value = value;
  return r;
}

struct tree *make_word (char *key, int value) {
  struct tree *t = malloc (sizeof (struct tree));
  t->tag = word;
  t->r = make_record (key, value);
  return t;
}

struct tree *make_node (void) {
  struct tree *t = calloc (1, sizeof (struct tree));
  t->tag = node;
  return t;
}

struct tree *make_from_record (struct record *r) {
  struct tree *t = malloc (sizeof (struct tree));
  t->tag = word;
  t->r = r;
  return t;
}
```

Цифровой поиск: *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 word skipped -- key exists, update value */
  if (i && !key[i - 1]) {
    t->r->value = value;
    return root;
  }
```

Цифровой поиск: *вставка элемента*

```
/* 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));
```

Цифровой поиск: *вставка элемента*

```
 /* 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;
}
```

Цифровой поиск: *печать элементов*

```
void print (struct tree *t, char c) {
  static int level = 0;
  if (!t) {
    printf ("empty\n");
    return;
  }
  for (int i = 0; i < level; i++)
    putchar (' ');
  if (level)
    printf ("%c: ", chr (c));
  if (t->tag == word) {
    printf ("word: %s %d\n", t->r->key, t->r->value);
  } else {
    printf ("node: ");
    for (int i = 0; i < M + 1; i++)
      if (t->nodes[i])
        printf ("%c ", chr(i));
    putchar ('\n');
    level++;
    for (int i = 0; i < M + 1; i++)
      if (t->nodes[i])
        print (t->nodes[i], i);
    level--;
  }
}
```

Самоперестраивающиеся деревья (splay trees)

- 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 $\Theta(n)$
  - Не делается предположений о распределении вероятностей ключей дерева и словарных операций (т.е. что некоторые операции выполнялись чаще других)
- Хорошее описание в:

  Harry R. Lewis, Larry Denenberg. Data Structures and Their Algorithms. HarperCollins, 1991. Глава 7.3.

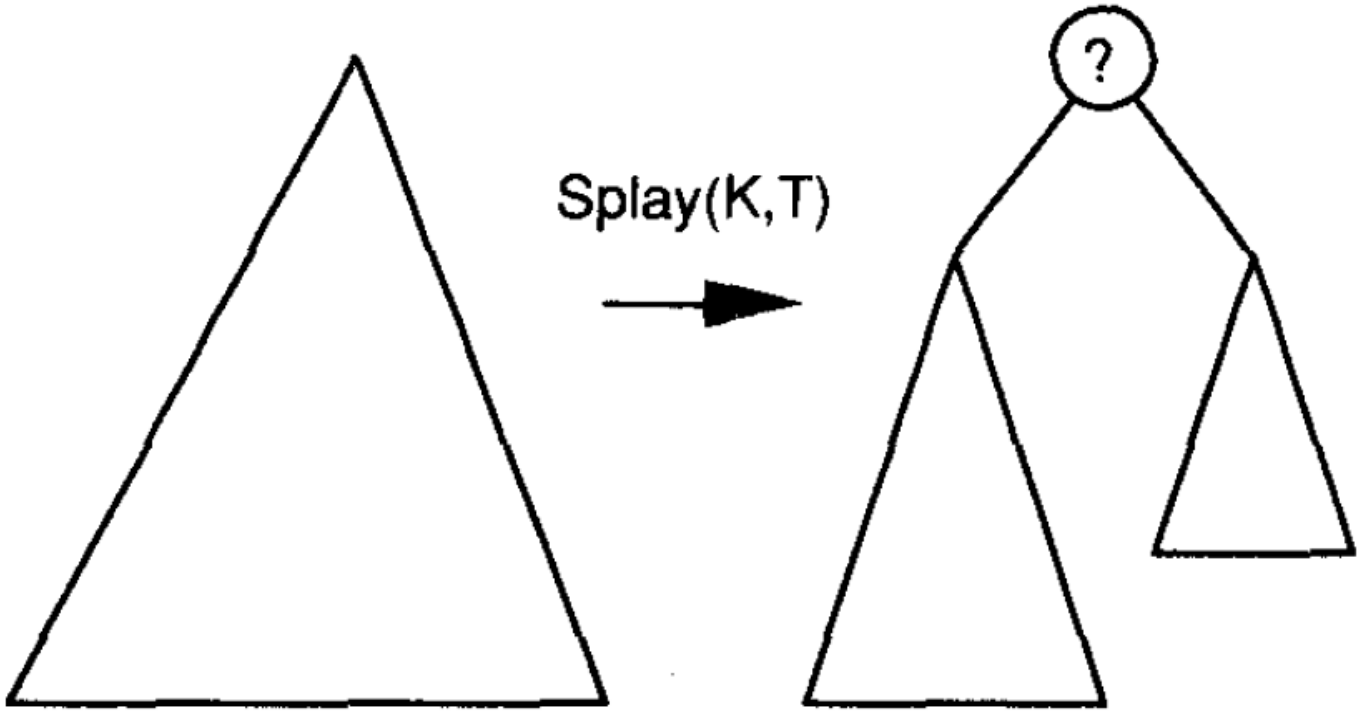
  http://www.amazon.com/Structures-Their-Algorithms-Harry-Lewis/dp/067339736X

Самоперестраивающиеся деревья (splay trees)

- Идея: эвристика Move-to-Front
  - Список: давайте при поиске элемента в списке перемещать найденный элемент в начало списка
  - Если он потребуется снова в обозримом будущем, он найдется быстрее
- Move-to-Front для двоичного дерева поиска: операция $Splay(K, T)$ (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

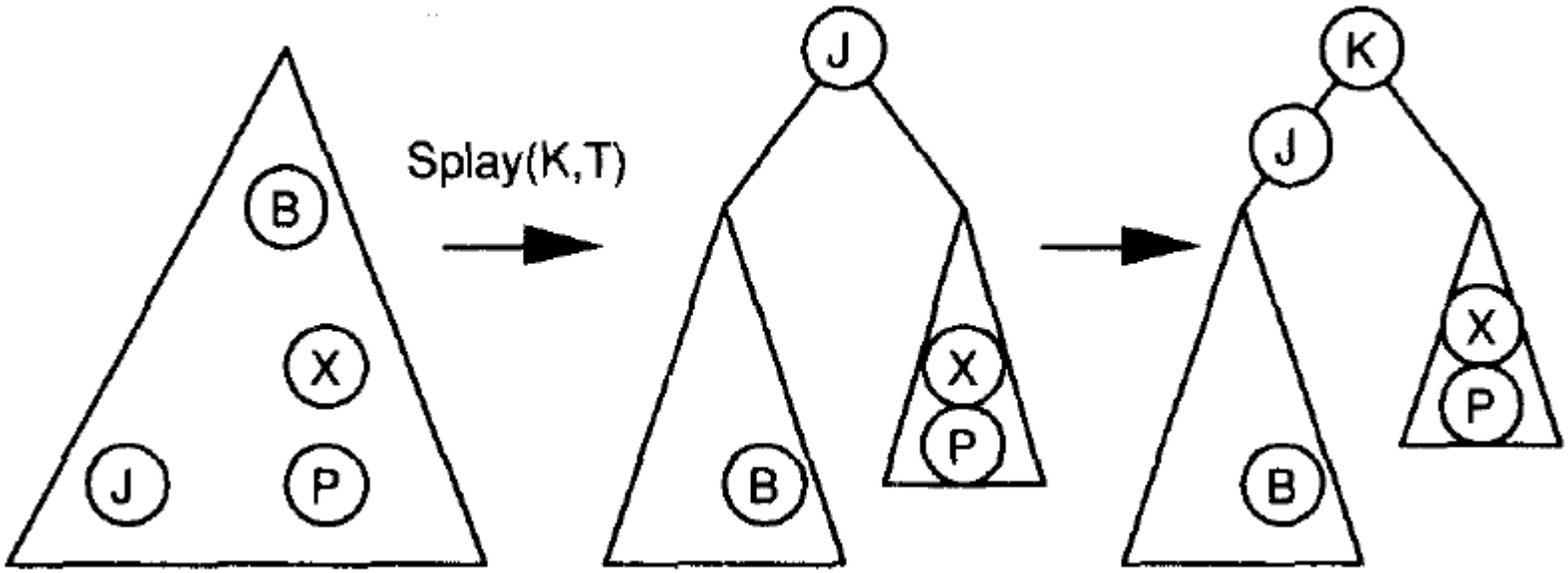
Реализация словарных операций через splay

- Поиск (LookUp): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
  - если значение равно K, то ключ найден



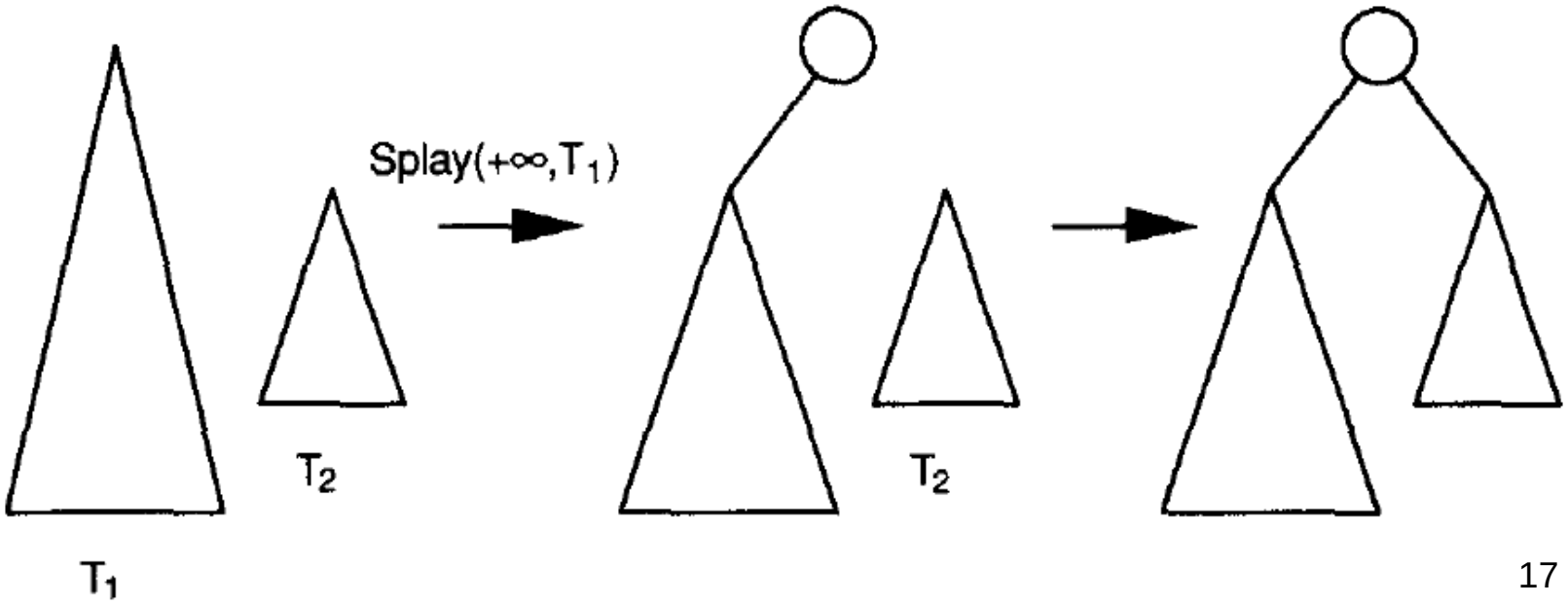
Реализация словарных операций через splay

- Вставка (Insert): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
  - если значение уже равно K, то обновим данные ключа
  - если значение другое, то вставим новый корень K и поместим старый корень J слева или справа (в зависимости от значения J)



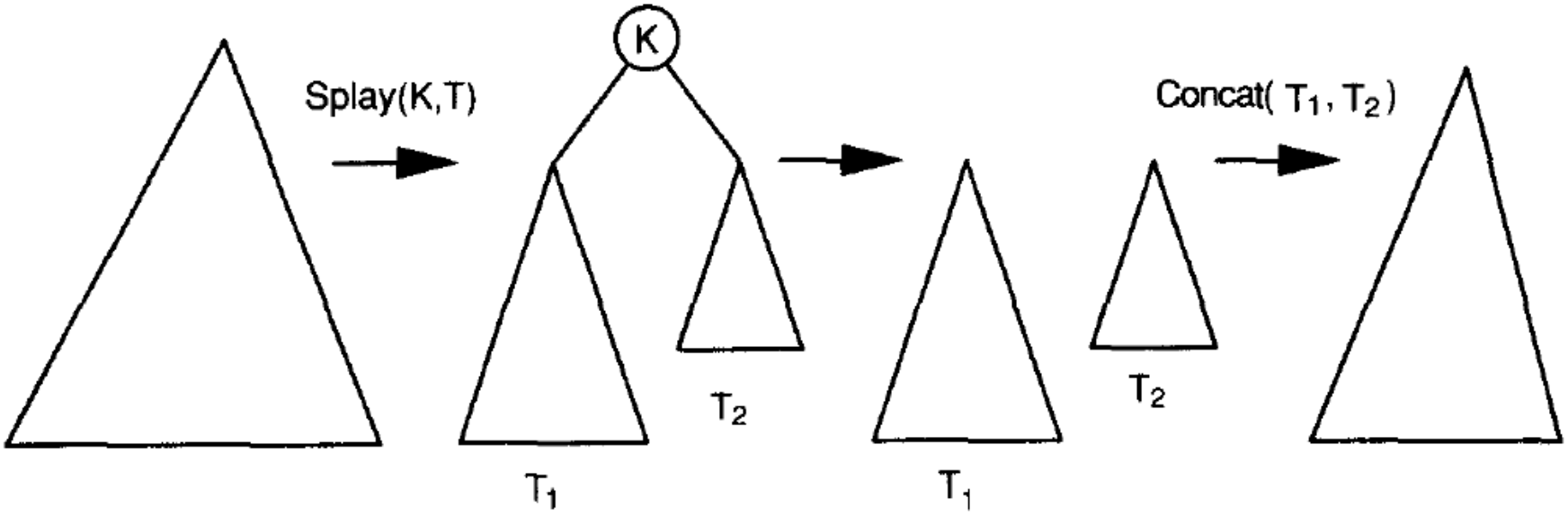
Реализация словарных операций через splay

- 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 дерево поиска
- Слияние (Concat): выполним операцию $Splay(+\infty, T_1)$ со значением ключа, заведомо больше любого другого в T_1
  - После $Splay(+\infty, T_1)$ у корня дерева T_1 нет правого сына
  - Присоединим дерево T_2 как правый сын корня T_1



Реализация словарных операций через splay

- Удаление (Delete): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
  - если значение **не равно** K, то ключа в дереве нет и удалять нам нечего
  - иначе (ключ был найден) выполним операцию *Concat* над левым и правым сыновьями корня, а корень удалим

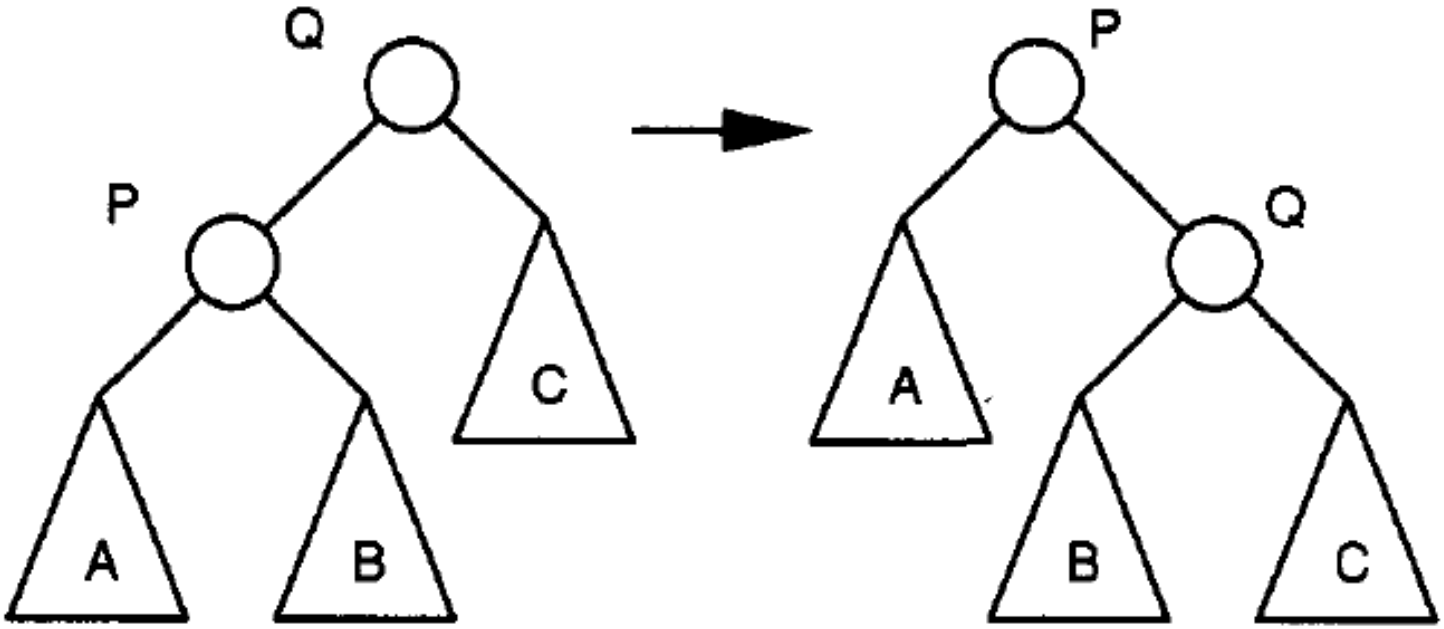


Реализация операции splay

- Шаг 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)
    - перенаправление указателей на сына назад на родителя вдоль пути по дереву плюс 1 бит на обозначение направления
- Шаг 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)
- Шаг 3: возвращаемся назад вдоль запомненного пути, перемещая узел P к корню (узел P будет новым корнем)

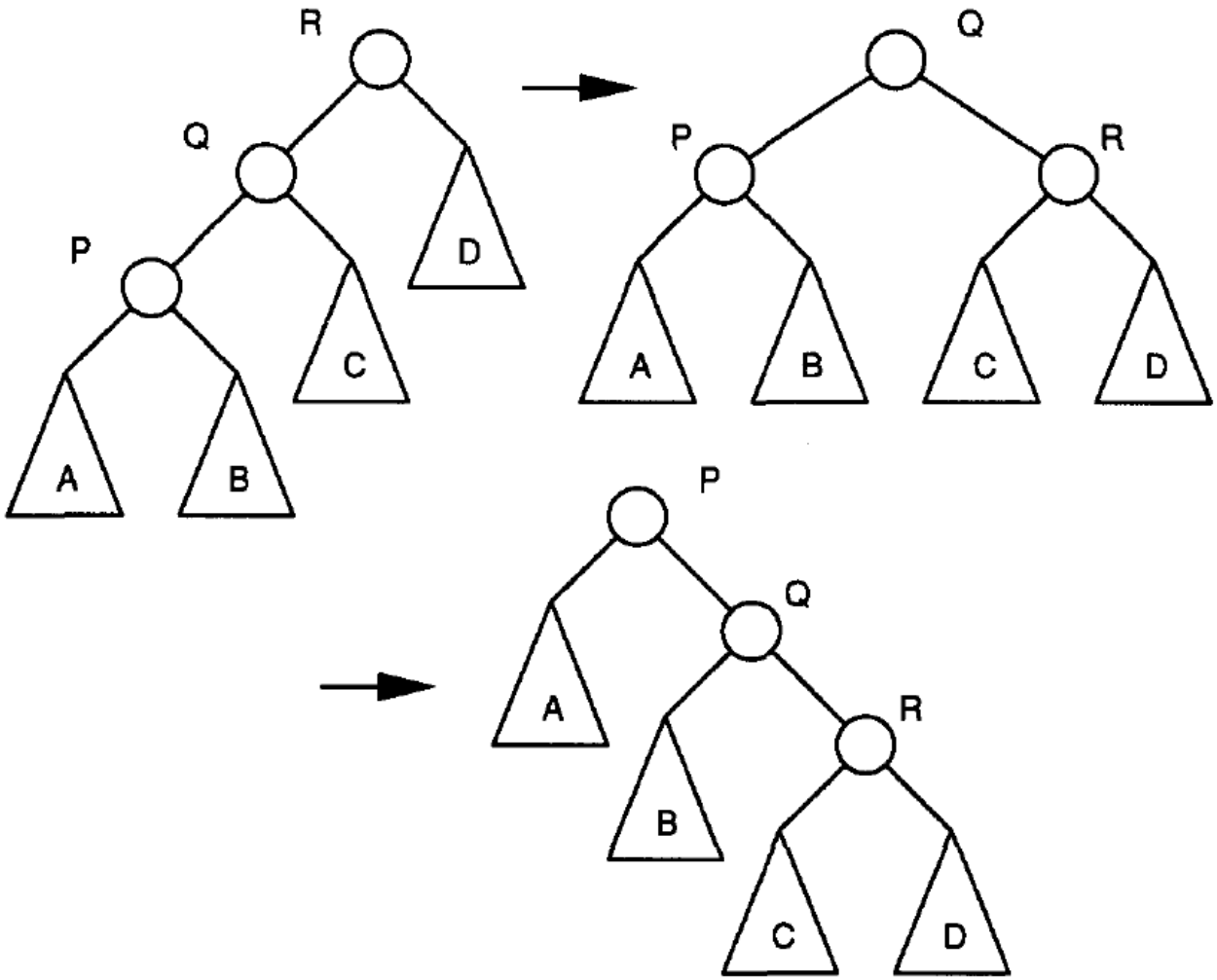
Реализация операции splay

- Шаг 3а): отец узла P – корень дерева (или у P нет деда)
  - выполняем однократный поворот налево или направо



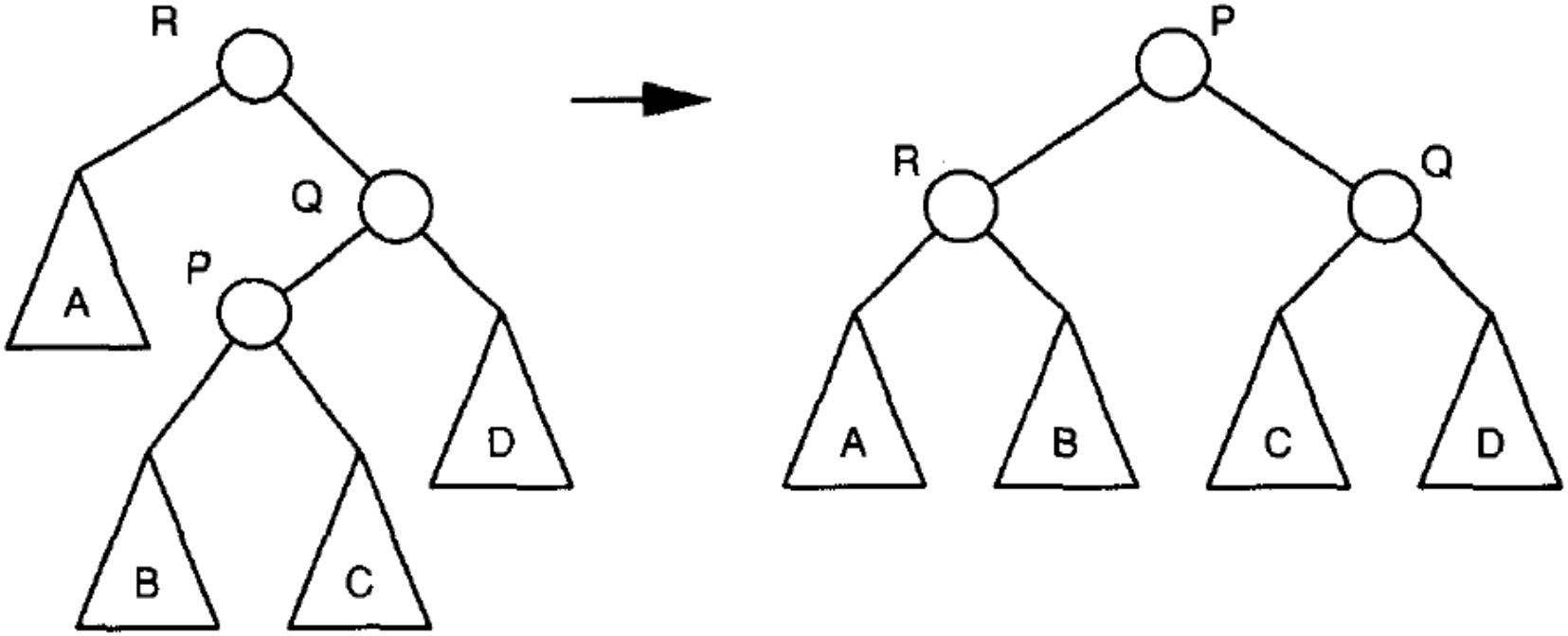
Реализация операции splay

- Шаг 3б): узел P и отец узла P – оба левые или правые дети
  - выполняем два однократных поворота направо (налево), сначала вокруг деда P, потом вокруг отца P



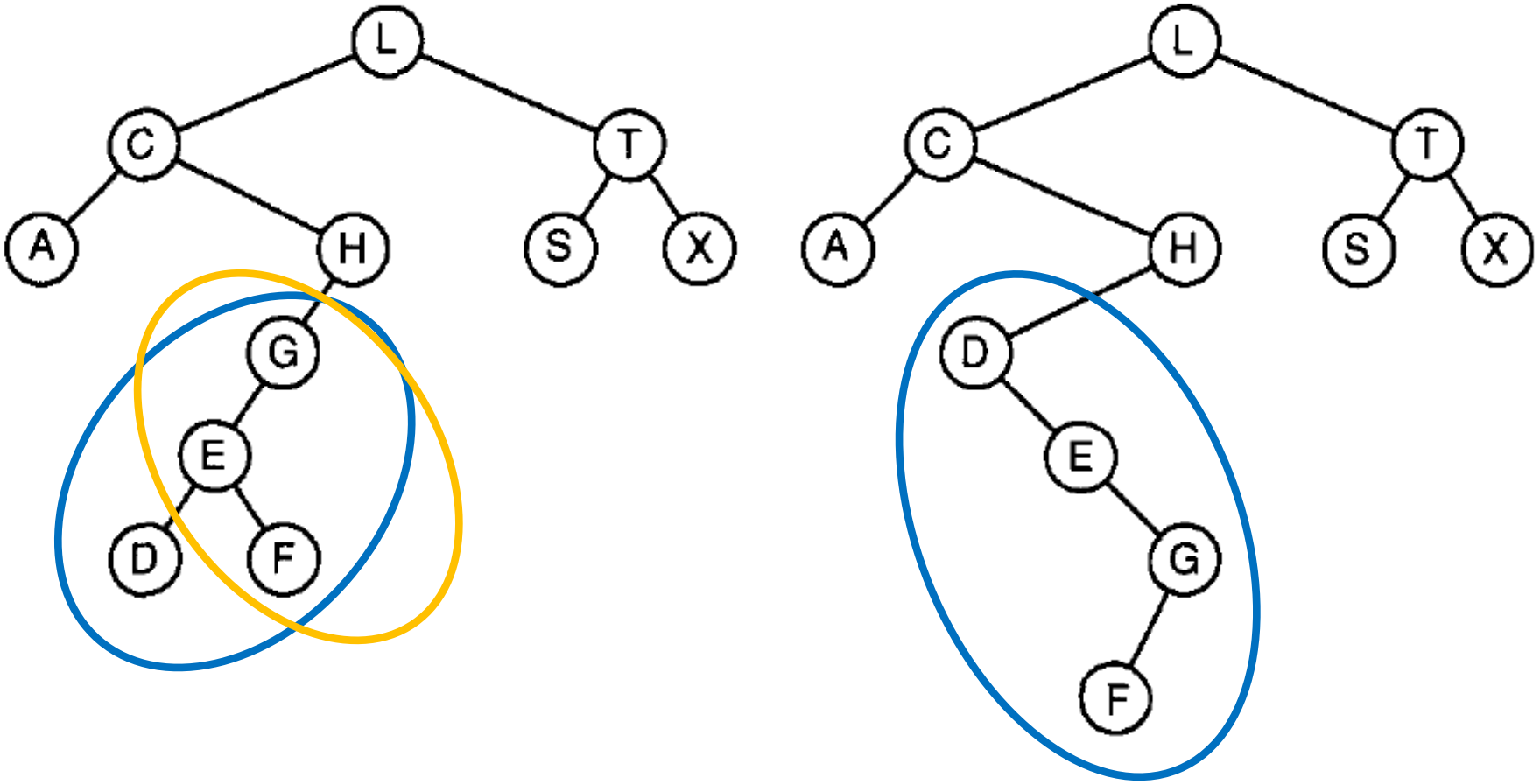
Реализация операции splay

- Шаг 3в): отец узла P – правый сын, а P – левый сын (или наоборот)
  - выполняем два однократных поворота в противоположных направлениях (сначала вокруг отца P направо, потом вокруг деда P налево)



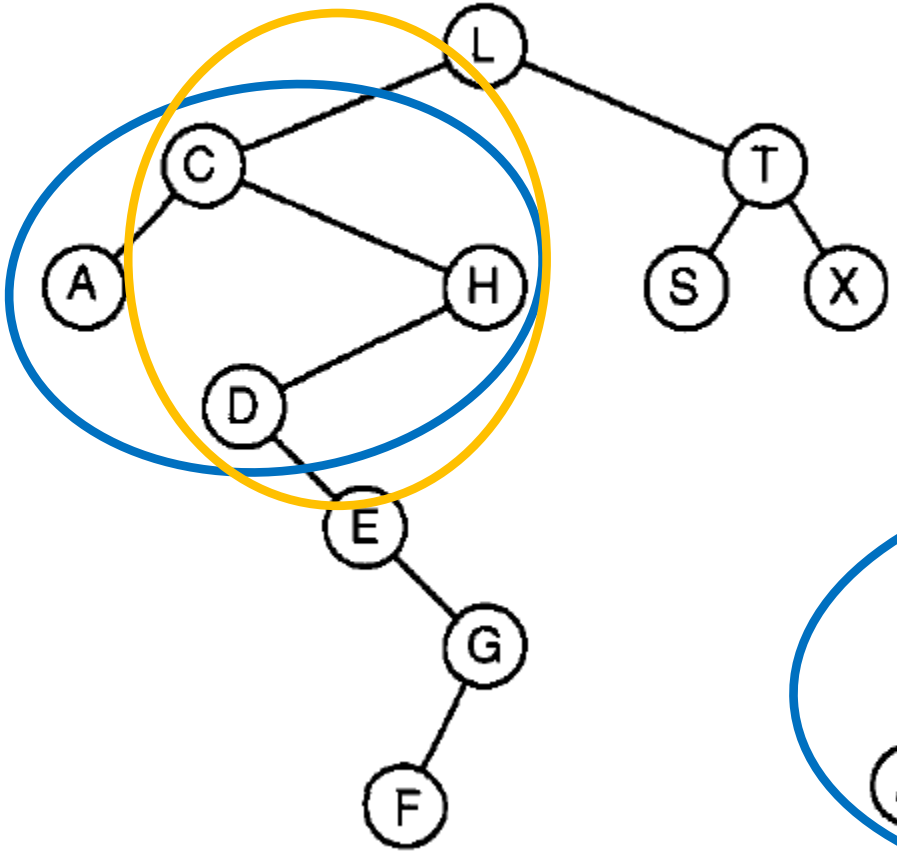
Пример операции splay над узлом D

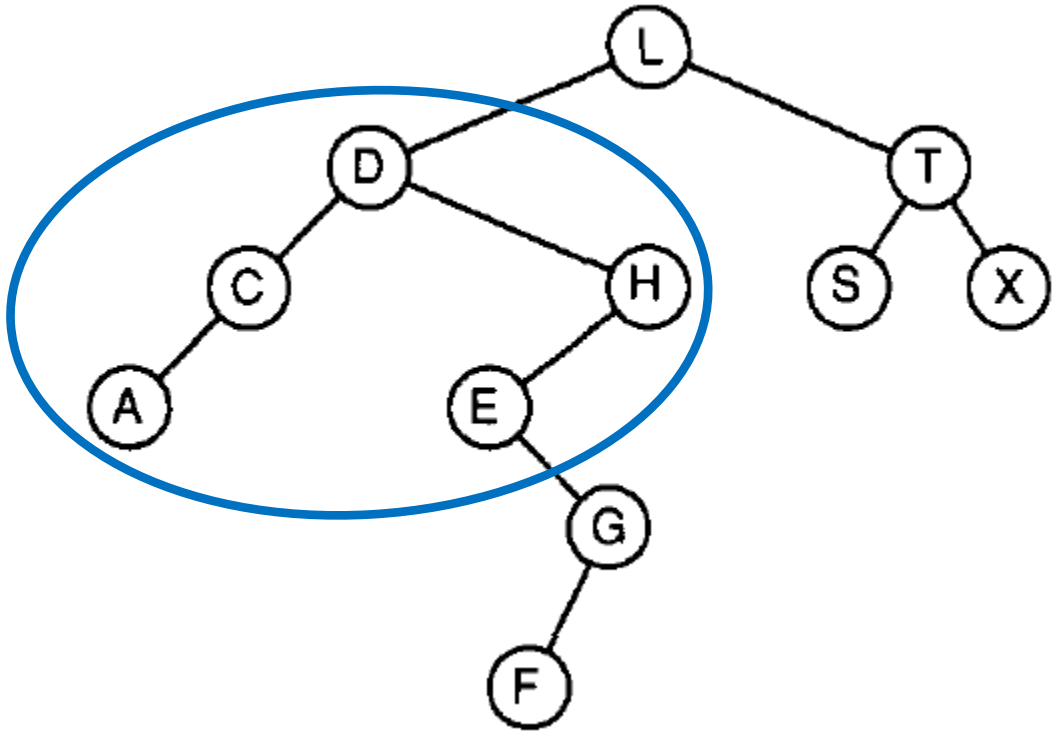
- Случай б): отец узла D (E) и сам узел D – оба левые сыновья



Пример операции splay над узлом D

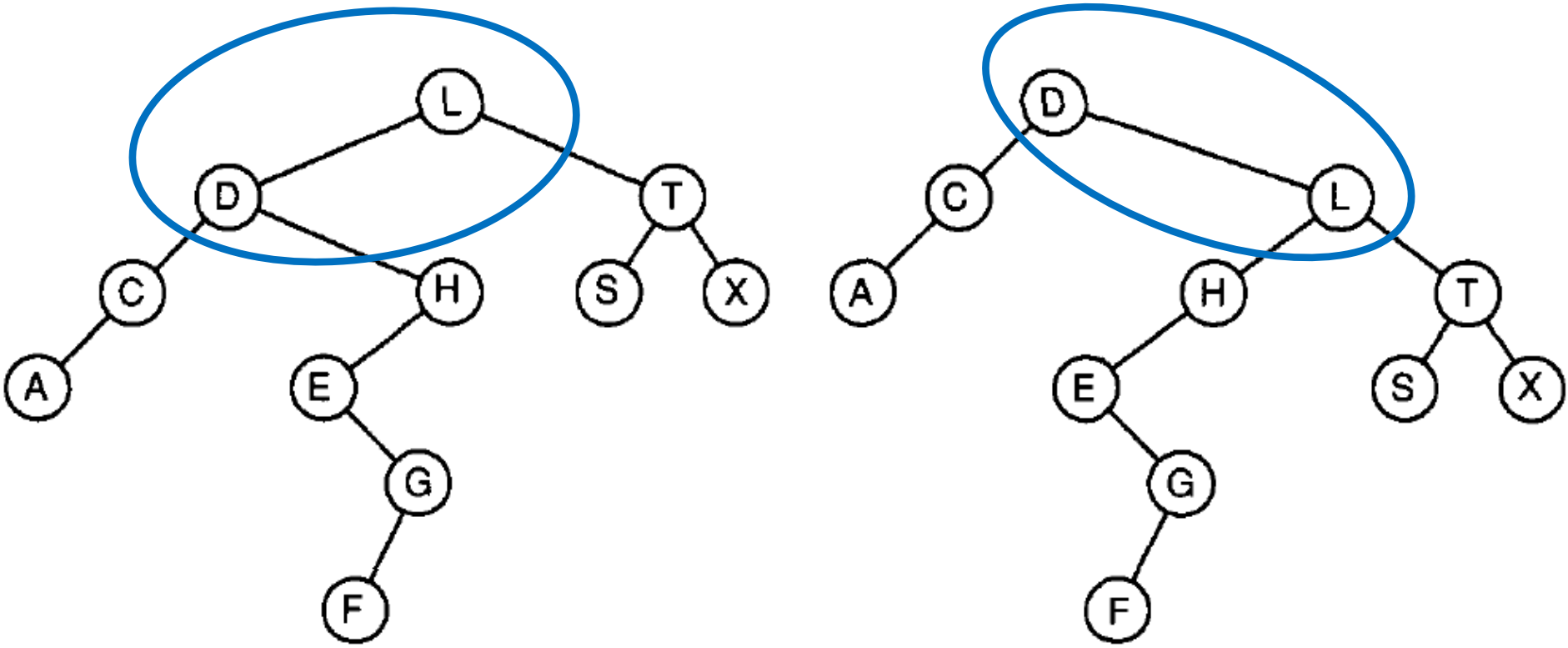
- Случай в): отец узла D (H) – правый сын, а сам узел D – левый сын





Пример операции splay над узлом D

- Случай а): отец узла D (L) – корень дерева



Сложность операции splay

- Пусть каждый узел дерева содержит некоторую сумму денег.
  - *Весом* узла является количество ее потомков, включая сам узел
  - *Рангом* узла $r(N)$ называется логарифм ее 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 в $3\lfloor \lg n \rfloor + 1$ 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.
  - Каждая операция требует не более $O(\log n)$ инвестиций, при этом может использовать деньги узла
  - По лемме инвестируется всего не более $m(3\lfloor \lg n \rfloor + 1)$ рублей, сначала дерево содержит 0 рублей, в конце содержит ≥0 рублей – $O(m \log n)$ хватает на все операции.

Сбалансированные деревья: *обобщение через ранги*

- Haeupler, Sen, Tarjan. Rank-balanced trees. ACM Transactions on Algorithms, 2014 (to appear).

- Обобщение разных видов сбалансированных деревьев через понятие ранга (rank) и ранговой разницы (rank difference)
  - АВЛ, красно-черные деревья, 2-3 деревья, B-деревья

- Новый вид деревьев: слабые АВЛ-деревья (weak AVL)

- Анализ слабых АВЛ-деревьев, анализ потенциалов

Сбалансированные деревья: *понятие ранга*

- Ранг (rank) вершины r(x): неотрицательное целое число
  - Ранг отсутствующей (null) вершины равен -1
- Ранг дерева: ранг корня дерева
- Ранговая разница (rank difference): если у вершины x есть родитель p(x), то это число r(p(x)) – r(x).
  - У корня дерева нет ранговой разницы
- *i*-сын: вершина с ранговой разницей, равной i.
- *i*,*j*-вершина: вершина, у которой левый сын – это *i*-сын, а правый сын – это *j*-сын. Один или оба сына могут отсутствовать. *i*,*j*- и *j*,*i*-вершины не различаются.

Сбалансированные деревья: *ранговый формализм*

- Конкретный вид сбалансированного дерева определяется *рангом* и *ранговым правилом*.

- Ранговое правило должно гарантировать:
  - Высота дерева (h) превосходит его ранг не более чем в константное количество 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 – число ее потомков, включая себя, т.е. размер поддерева с корнем в этой вершине
  - T.e. $h = O(k)$, $k = O(\log n) \rightarrow h = O(\log n)$

- Совершенное дерево:

  ранг дерева – его высота; 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)
  - 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
  - Корень не имеет цвета (т.к. 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
  - Бит на узел для ранговой разницы или ее *четности*
  - Балансировка: не более двух поворотов и O(log n) изменений ранга для вставки/удаления, при этом амортизированно – лишь O(1) изменений.
  - Слабое АВЛ-дерево является красно-черным деревом