

Министерство информационных технологий и связи
Российской Федерации

Сибирский государственный университет
телекоммуникаций и информатики

Е. В. Курапова
Е. П. Мачикина

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Методическое пособие

Новосибирск 2006

УДК 681.3.06

ктн Е. В. Курапова, кф-мн Е. П. Мачикина

Структуры и алгоритмы обработки данных: Методическое пособие. / Сиб. гос. ун-т телекоммуникаций и информатики. – Новосибирск, 2006. – 105 с.

Методическое пособие предназначено для студентов технических специальностей, обучающихся по направлению “550400 Телекоммуникации” и изучающих дисциплину «Структуры и алгоритмы обработки данных». Пособие содержит необходимый теоретический минимум по данному предмету и варианты заданий для самостоятельного выполнения.

Рисунков — 69, таблиц — 13. Список лит. — 5 назв.

Кафедра прикладной математики и кибернетики.

Рецензент: Зайцев М.Г., Венедиктов М.Д.

Утверждено редакционно-издательским советом СибГУТИ
в качестве методического пособия.

© Сибирский государственный университет
телекоммуникаций и информатики, 2006 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	7
1. НЕОБХОДИМЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ	7
1.1 Основные структуры данных.....	7
1.2 Задача сортировки массивов	8
1.3 Трудоемкость методов сортировки массивов	9
1.4 Задача сортировки последовательностей.....	10
1.5 Теорема о сложности сортировки.....	10
1.6 Задача поиска элементов с заданным ключом.....	11
2. МЕТОДЫ СОРТИРОВКИ С КВАДРАТИЧНОЙ ТРУДОЕМКОСТЬЮ	12
2.1 Метод прямого выбора	12
2.2 Пузырьковая сортировка.....	13
2.3 Шейкерная сортировка	15
2.4 Варианты заданий	17
3. МЕТОД ШЕЛЛА	17
3.1 Метод прямого включения.....	17
3.2 Метод Шелла.....	18
3.3 Варианты заданий	20
4. БЫСТРЫЕ МЕТОДЫ СОРТИРОВКИ МАССИВОВ	20
4.1 Пирамидальная сортировка	20
4.2 Метод Хоара.....	23
4.3 Проблема глубины рекурсии.....	25
4.4 Варианты заданий	26
5. РАБОТА С ЛИНЕЙНЫМИ СПИСКАМИ	27
5.1 Указатели. Основные операции с указателями.....	27
5.2 Основные операции с линейными списками	28
6. МЕТОДЫ СОРТИРОВКИ ПОСЛЕДОВАТЕЛЬНОСТЕЙ	31
6.1 Метод прямого слияния	31
6.2 Цифровая сортировка.....	34
6.3 Варианты заданий	36
7. ДВОИЧНЫЙ ПОИСК В УПОРЯДОЧЕННОМ МАССИВЕ	36
7.1 Алгоритм двоичного поиска	36
7.2 Варианты заданий	38
8. СОРТИРОВКА ДАННЫХ С ПРОИЗВОЛЬНОЙ СТРУКТУРОЙ.....	38
8.1 Сравнение данных произвольной структуры	38
8.2 Сортировка по множеству ключей. Индексация.....	39
8.3 Индексация через массив указателей	40
8.4 Варианты заданий	40
9. ДВОИЧНЫЕ ДЕРЕВЬЯ	41
9.1 Основные определения и понятия.....	41
9.2 Различные обходы двоичных деревьев.....	41
9.3 Вычисление основных характеристик дерева	42
9.4 Варианты заданий	43
10. ДЕРЕВЬЯ ПОИСКА	44

10.1	<i>Поиск в дереве</i>	44
10.2	<i>Идеально сбалансированное дерево поиска</i>	45
10.3	<i>Варианты заданий</i>	46
11.	СЛУЧАЙНОЕ ДЕРЕВО ПОИСКА	47
11.1	<i>Определение случайного дерева поиска</i>	47
11.2	<i>Добавление вершины в дерево</i>	48
11.3	<i>Удаление вершины из дерева</i>	49
11.4	<i>Варианты заданий</i>	51
12.	СБАЛАНСИРОВАННЫЕ ПО ВЫСОТЕ ДЕРЕВЬЯ (АВЛ-ДЕРЕВЬЯ)	51
12.1	<i>Определение и свойства АВЛ-дерева</i>	51
12.2	<i>Повороты при балансировке</i>	53
12.3	<i>Добавление вершины в дерево</i>	55
12.4	<i>Удаление вершины из дерева</i>	57
12.5	<i>Варианты заданий</i>	62
13.	Б-ДЕРЕВЬЯ	62
13.1	<i>Определение Б-дерева порядка m</i>	62
13.2	<i>Поиск в Б-дереве</i>	63
13.3	<i>Построение Б-дерева</i>	65
13.4	<i>Определение двоичного Б-дерева</i>	67
13.5	<i>Добавление вершины в дерево</i>	68
13.6	<i>Варианты заданий</i>	72
14.	ДЕРЕВЬЯ ОПТИМАЛЬНОГО ПОИСКА (ДОП)	72
14.1	<i>Определение дерева оптимального поиска</i>	72
14.2	<i>Точный алгоритм построения ДОП</i>	74
14.3	<i>Приближенные алгоритмы построения ДОП</i>	77
14.4	<i>Варианты заданий</i>	80
15.	ХЭШИРОВАНИЕ И ПОИСК	80
15.1	<i>Понятие хэш-функции</i>	80
15.2	<i>Метод прямого связывания</i>	82
15.3	<i>Метод открытой адресации</i>	84
15.4	<i>Варианты заданий</i>	86
16.	ЭЛЕМЕНТЫ ТЕОРИИ КОДИРОВАНИЯ ИНФОРМАЦИИ	87
16.1	<i>Необходимые понятия</i>	87
16.2	<i>Кодирование целых чисел</i>	88
16.3	<i>Алфавитное кодирование</i>	92
16.4	<i>Оптимальное алфавитное кодирование</i>	95
16.5	<i>Почти оптимальное алфавитное кодирование</i>	98
16.6	<i>Варианты заданий</i>	102
	РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА	103
	ПРИЛОЖЕНИЕ А	104

СПИСОК РИСУНКОВ

Рисунок 1	Дерево решений для 6 элементов.....	10
Рисунок 2	Метод прямого выбора.....	12
Рисунок 3	Пузырьковая сортировка.....	14
Рисунок 4	Шейкерная сортировка.....	16
Рисунок 5	Метод прямого включения	18
Рисунок 6	Метод Шелла.....	19
Рисунок 7	Добавление в пирамиду нового элемента	21
Рисунок 8	Пирамидальная сортировка	22
Рисунок 9	Метод Хоара.....	24
Рисунок 10	Схема вызовов при вычислении 4!	25
Рисунок 11	Структура фрейма.....	26
Рисунок 12	Равенство указателей.....	27
Рисунок 13	Указатель на элемент списка	28
Рисунок 14	Добавление в стек	29
Рисунок 15	Удаление из стека	29
Рисунок 16	Добавление в очередь.....	29
Рисунок 17	Структура очереди.....	30
Рисунок 18	Добавление в очередь.....	30
Рисунок 19	Перемещение элемента	30
Рисунок 20	Слияние серий.....	31
Рисунок 21	Метод прямого слияния	32
Рисунок 22	Начальное расщепление	33
Рисунок 23	Цифровая сортировка	34
Рисунок 24	Первая версия поиска	37
Рисунок 25	Вторая версия поиска	37
Рисунок 26	Список абонентов	39
Рисунок 27	Пример двоичного дерева	41
Рисунок 28	42
Рисунок 29	Дерево поиска	44
Рисунок 30	Примеры ИСД и неИСД.....	45
Рисунок 31	Построение ИСДП	46
Рисунок 32	Случайное дерево поиска.....	47
Рисунок 33	Плохие СДП	48
Рисунок 34	Добавление вершины В.....	49
Рисунок 35	Добавление вершины 9	49
Рисунок 36	Варианты удаления вершин.....	49
Рисунок 37	Удаляемая вершина с двумя поддеревьями	50
Рисунок 38	Порядок изменения указателей при удалении вершины	50
Рисунок 39	Пример АВЛ-дерева и не АВЛ-дерева	52
Рисунок 40	Деревья Фибоначчи	52
Рисунок 41	LL - поворот	53
Рисунок 42	LR – поворот.....	54
Рисунок 43	RR – поворот	54
Рисунок 44	RL – поворот.....	55
Рисунок 45	Построение АВЛ-дерева	57
Рисунок 46	Три случая при удалении вершины из левого (для BL) поддерева	58
Рисунок 47	Три случая при удалении вершины правого (для BR) поддерева.....	59
Рисунок 48	Удаление из АВЛ-дерева	61
Рисунок 49	Страница Б-дерева	63
Рисунок 50	Пример Б-дерева	63
Рисунок 51	Структура страницы Б-дерева	64

Рисунок 52 Построение Б-дерева	65
Рисунок 53 Виды вершин ДБД	67
Рисунок 54 Вершины двоичного Б-дерева	68
Рисунок 55 Четыре ситуации, возникающих при росте левых или правых поддеревьев	69
Рисунок 56 Построение двоичного Б-дерева	71
Рисунок 57 Различные деревья поиска с вершинами $V_1=1, V_2=2, V_3=3$	73
Рисунок 58 ДОП для $w_1=60, w_2=30, w_3=10$	76
Рисунок 59 Дерево, построенное приближенным алгоритмом A1	78
Рисунок 60 Дерево, построенное приближенным алгоритмом A2	79
Рисунок 61 Отображение $H: K \rightarrow A$	81
Рисунок 62 Хеш-таблица, построенная методом прямого связывания	83
Рисунок 63 Использование квадратичных проб	86
Рисунок 64 Модель системы передачи сигналов	87
Рисунок 65 Полное двоичное дерево с помеченными вершинами	93
Рисунок 66 Построение префиксного кода с заданными длинами	94
Рисунок 67 Процесс построения кода Хаффмена	97
Рисунок 68 Кодовое дерево для кода Хаффмена	97
Рисунок 69 Кодовое дерево для кода Фано	100

СПИСОК ТАБЛИЦ

Таблица 1 Различные типы данных	7
Таблица 2 Основные операции с указателями	28
Таблица 3 Частоты вхождения символов в строку	78
Таблица 4 Упорядоченный набор вершин	79
Таблица 5 Номера символов строки	85
Таблица 6 Код класса <i>Fixed + Variable</i>	89
Таблица 7 Код класса <i>Variable + Variable</i>	90
Таблица 8 γ -код Элиаса	90
Таблица 9 ω -код Элиаса	91
Таблица 10 Код Хаффмена	97
Таблица 11 Код Шеннона	99
Таблица 12 Код Фано	100
Таблица 13 Код Гилберта-Мура	102

ВВЕДЕНИЕ

Изучение дисциплины «Структуры и алгоритмы обработки данных» является одним из основных моментов в процессе подготовки специалистов по разработке программного обеспечения для компьютерных систем. Это связано с тем, что первичная задача программиста заключается в применении решения о форме представления данных и выборе алгоритмов, применяемых к этим данным. И лишь затем выбранная структура программы и данных реализуется на конкретном языке программирования. В связи с этим знание классических методов и приемов обработки данных позволяет избежать ошибок, которые могут возникать при чисто интуитивной разработке программ.

Данное пособие содержит необходимый теоретический материал по четырем основным разделам курса «Структуры и алгоритмы обработки данных»: методы сортировки данных, древовидные структуры данных, хэширование и кодирование информации. Все рассмотренные методы проиллюстрированы наглядными примерами. Также для каждого метода приведен конкретный алгоритм, позволяющий легко программировать данный метод. Кроме того, в каждой главе даны варианты заданий, выполнив которые, можно окончательно уяснить все особенности изучаемых методов.

1. НЕОБХОДИМЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

1.1 Основные структуры данных

Любая программа в процессе работы обрабатывает некоторые данные. По способу представления в памяти компьютера данные можно разделить на две группы: статические и динамические. Статические данные имеют фиксированную структуру, поэтому размер выделенной для них памяти постоянен. Динамические данные изменяют свою структуру в процессе работы программы, при этом объём памяти изменяется. Основные структуры данных представлены в следующей таблице.

Таблица 1 Различные типы данных

Статические	Простые	Перечислимые
		Целые
		Вещественные
		Логические
		Символьные
Динамические	Структурированные	Массивы
		Записи
		Объединения
	Списки	Стек
		Очередь
	Деревья	АВЛ-деревья
		Б-деревья

Основной характеристикой данных в программировании является тип данных. Любая константа, переменная, выражение, функция всегда относятся к определенному типу. Тип характеризует множество значений, которые может принимать переменная. К простым типам языка программирования относятся целые, вещественные, логические, символьные. Целые типы в языках программирования различаются количеством байт, отведённых в памяти (диапазоном значений) и наличием знака. Вещественные типы характеризуются точностью представления числа. Перечислимые типы образуются в процессе перечисления всех возможных значений. Логический тип является частным случаем перечислимого типа с двумя возможными значениями ИСТИНА и ЛОЖЬ.

Структурированные (составные) типы отличаются от простых тем, что состоят из набора компонент одинакового или разного типа. К структурированным типам относят массивы, записи (структуры), объединения (записи с вариантами).

Массивы – это наиболее известная и распространённая структура данных. Массив представляет собой фиксированное количество элементов одного типа. Всем элементам присваивается одно имя, а к отдельному элементу обращаются по его номеру (индексу). Тип элементов массива может быть любым, но тип индексов должен быть скалярным. Массив – структура данных со случайным (прямым) доступом, т.е. в любой момент времени доступен любой элемент массива, при этом индекс элемента можно вычислять. Обычный способ работы с массивом – выбор определённых элементов и их изменение. Также часто используется перебор элементов в цикле. В памяти элементы массива располагаются последовательно, без разрывов, по возрастанию адресов памяти.

Другой вид структурированных типов – запись состоит из фиксированного числа компонент называемых **полями**, которые в отличие от элементов массива могут быть разных типов. **Запись** также является структурой данных со случайным доступом.

Объединения или (записи с вариантами) используются для размещения в одной и той же области памяти данных различного типа. В один и тот же момент времени в памяти находятся данные только одного типа.

Динамические структуры данных позволяют работать с большими объемами данных. Наиболее используемые динамические структуры данных – списки и деревья.

1.2 Задача сортировки массивов

Пусть дан массив $A=(a_1, a_2, \dots, a_n)$ и для всех его элементов определены операции отношения: меньше, больше, равно ($<$, $>$, $=$). Необходимо отсортировать массив, т.е. переставить элементы массива A таким образом, что выполняется одно из следующих неравенств:

$$\begin{aligned} a_1 &\leq a_2 \leq a_3 \leq \dots \leq a_n \\ a_1 &\geq a_2 \geq a_3 \geq \dots \geq a_n \end{aligned}$$

Если выполняется первое неравенство, то массив сортируется по возрастанию и такой порядок элементов будем называть *прямым*. Если выполняется второе неравенство, то массив отсортирован по убыванию и такой порядок элементов будем называть *обратным*. В дальнейшем, если не оговорено особо, исполь-

зуется прямой порядок сортировки.

При сортировке массивов элементов со сложной структурой возникает задача определить отношение порядка между элементами. Та часть информации, которая учитывается при определении отношения порядка называется *ключом сортировки*.

Сортировка называется *устойчивой*, если после её проведения в массиве не меняется относительный порядок элементов с одинаковыми ключами.

Для проверки правильности сортировки массива могут использоваться следующие приемы. Вычисление контрольной суммы элементов массива до и после сортировки дает возможность проверить потерю элементов массива во время процесса сортировки. Определение количества серий элементов массива, т.е. убывающих последовательностей из элементов массива, позволяет проверить правильность упорядочивания массива, поскольку массив, отсортированный по возрастанию, состоит из одной серии, а в массиве, отсортированном по убыванию, количество серий равно количеству элементов в массиве.

1.3 Трудоемкость методов сортировки массивов

Существует много способов или методов сортировки массивов. Для того, чтобы оценить насколько один метод сортировки лучше другого необходимо каким-то образом оценивать эффективность метода сортировки. Естественно отличать методы сортировки по времени, затрачиваемому на реализацию сортировки. Для сортировок основными считаются две операции: операция сравнения элементов и операция пересылки элемента. Будем считать, что в единицу времени выполняется одна операция сравнения или пересылки. Таким образом, время или трудоемкость метода имеет две составляющие M и C , где

M – количество операций пересылки.

C – количество операций сравнения.

Нетрудно видеть, что M и C – зависят от количества элементов в массиве, т.е. являются функциями от длины массива. Часто бывает трудно определить точное выражение для трудоемкости алгоритма. В этом случае пользуются асимптотической оценкой времени работы.

Будем говорить, что функция $g(x)$ *асимптотически доминирует* на $f(x)$ или $g(x) = O(f(x))$, если $|g(x)| \leq \text{const} |f(x)|$ при $x \rightarrow \infty$. В дальнейшем будем рассматривать асимптотическое поведение величин M и C в зависимости от числа элементов в массиве n , при $n \rightarrow \infty$.

Для функций f, f_1, f_2 и константы k справедливы свойства:

1. $f = O(f)$
2. $O(k \cdot f) = O(f)$
3. $O(f_1 + f_2) = O(f_1) + O(f_2)$

Пример. $T = 10n + 20$

$T = O(10n + 20) = O(10n) + O(20) = O(n) + O(1) = O(n)$, при $n \rightarrow \infty$.

Приведем ряд доминирования основных функций

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^a) < O(a^n) < O(n!) < O(n^n)$, при $n \rightarrow \infty, a > 1$

Поскольку для различных массивов один и тот же метод сортировки может иметь различную трудоемкость, то необходимо знать в каких пределах могут изменяться величины характеризующие трудоемкость, т.е. определить минимальное и максимальное значения трудоемкости и массивы, на которых достигаются эти значения, а также средние значения величин M и C .

1.4 Задача сортировки последовательностей

Пусть дана последовательность $S=S_1S_2 \dots S_n$, т.е. совокупность данных с последовательным доступом к элементам. Примерами такой последовательности могут служить файл на магнитной ленте, линейный список:

Необходимо переставить элементы последовательности так, чтобы выполнялись неравенства: $S_1 \leq S_2 \leq \dots \leq S_n$. Последовательный доступ означает, что любой элемент списка может быть получен только путём просмотра предыдущих элементов, причём просмотр возможен только в одном направлении. Это является существенным ограничением по сравнению с массивом, где можно было обратиться к любому элементу массиву, используя индекс. Поэтому методы сортировки, разработанные специально для массивов, не годятся для последовательностей, в то время как методы сортировки последовательностей используются и для сортировки массивов. Трудоемкость методов сортировки последовательностей измеряется количеством операций, затрачиваемых на сортировку. Характерными операциями при сортировке последовательностей являются операция сравнения элементов и операция пересылки элемента одной последовательности в другую. Как и прежде будем обозначать количества операций сравнения и пересылки C и M соответственно.

1.5 Теорема о сложности сортировки

При изучении различных методов сортировок возникает закономерный вопрос о построении метода сортировки с минимально возможной трудоемкостью. Следующая теорема устанавливает нижнюю границу трудоемкости для сортировки массива из n элементов.

Теорема. Если все перестановки из n элементов равновероятны, то любое дерево решений, сортирующее последовательность из n элементов имеет среднюю высоту не менее $\log(n!)$.

Приведем нестрогое доказательство. Рассмотрим дерево решений для трех элементов a, b, c .

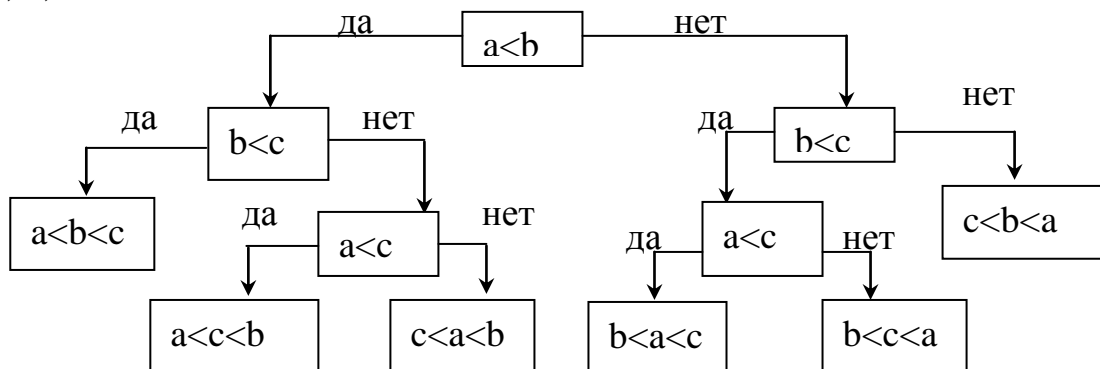


Рисунок 1 Дерево решений для 3 элементов

Все возможные перестановки – это листья дерева (6 вариантов). Чтобы получить конкретную перестановку нужно сделать два или три сравнения. Оценим среднее количество сравнений, необходимых для упорядочивания массива или среднюю длину пути от корня дерева до листьев. Для этого посчитаем сумму длин всевозможных путей от корня до листьев (длина внешнего пути двоичного дерева) и поделим ее на количество листьев $C_{cp} = (2+3+3+3+3+2)/6 = 2,6$.

Из теории графов известно, что длина внешнего пути двоичного дерева с m листьями $D(m) \geq m \log m$. Поскольку в общем случае на дереве имеется $n!$ листьев. Тогда $C_{cp} \geq n! \log(n!)/n! = \log n! > n \log n - n \log e$. Последнее неравенство является известной нижней оценкой для значения факториала. Таким образом, не существует алгоритма сортировки n элементов, использующего в среднем меньше чем $(n \log n - \log e)$ операций сравнения. Класс сложности $n \log n$ является предельно достижимым для алгоритмов сортировки с использованием операций сравнения. Что касается количества пересылок, то если мы определим требуемую перестановку, и имеем память для второй копии массива, то достаточно сделать n пересылок. На сегодняшний день алгоритм, требующий $n \log n$ сравнений и n пересылок, неизвестен.

1.6 Задача поиска элементов с заданным ключом

Пусть имеется массив $A = (a_1, a_2, \dots, a_n)$ и задан ключ поиска X . Необходимо найти элемент (элементы) массива с ключом X , или определить, что элемента с заданным ключом в массиве нет. Если массив не упорядочен, то единственный путь поиска – просмотр всех элементов массива (линейный поиск) до тех пор, пока не будет найден этот элемент, или просмотрен весь массив. Трудоемкость поиска в этом случае равна $O(n)$, $n \rightarrow \infty$. Более эффективные алгоритмы поиска можно построить, если предполагать, что массив отсортирован, т.е. $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n$. Трудоемкость метода поиска будем оценивать по количеству сравнений, требуемых для поиска нужного элемента, при этом нас интересует только асимптотическая оценка трудоемкости.

2. МЕТОДЫ СОРТИРОВКИ С КВАДРАТИЧНОЙ ТРУДОЕМКОСТЬЮ

2.1 Метод прямого выбора

Один из самых простых методов сортировки, метод прямого выбора, заключается в следующем. Находим наименьший элемент массива и обмениваем его с первым элементом массива. Таким образом, первый элемент можно больше не рассматривать. Среди оставшихся элементов находим наименьший элемент и обмениваем его со вторым элементом массива. Среди оставшихся элементов находим наименьший и переставляем его на третье место и т. д.

Пример. Упорядочим слово методом прямого выбора.

Условные обозначения

\underline{X} сравнение элемента X с текущим максимальным элементом

\dot{X} текущий максимальный элемент

\frown Перестановка элементов

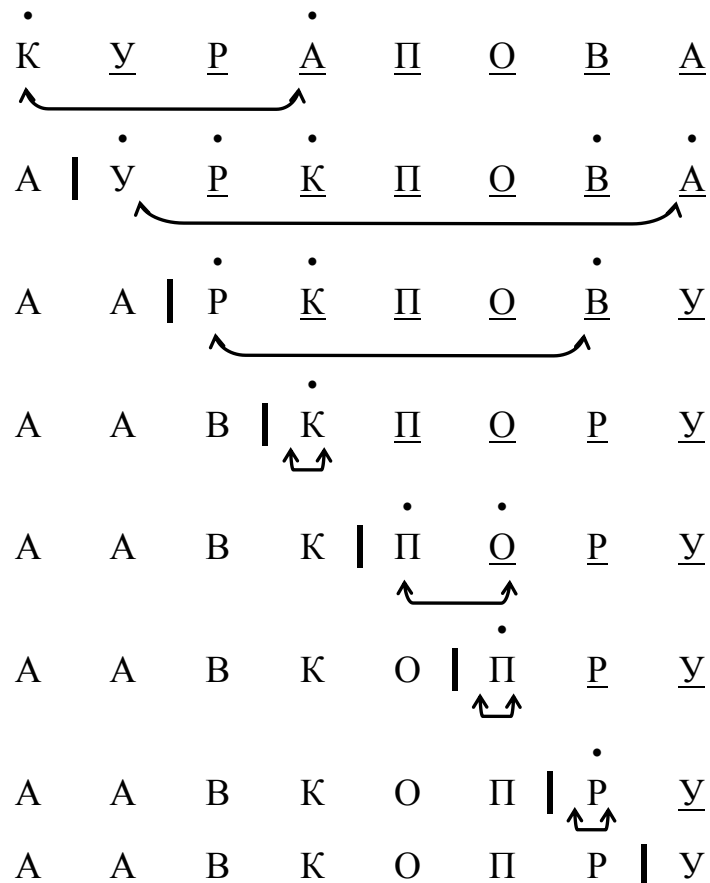


Рисунок 2 Метод прямого выбора

Алгоритм на псевдокоде
Метод прямого выбора

```
DO (i = 1, 2, ... n-1)
    k:=<номер наименьшего элемента из ai, ... an>
    ai ↔ ak
OD
```

Дадим оценку трудоёмкости метода прямого выбора. В данном случае можно найти точные выражения для M и C . Поскольку на каждой итерации происходит точно один обмен, то $M = 3(n-1)$. Определим теперь количество сравнений. На первом этапе имеем $(n-1)$ сравнений, на втором – $(n-2)$ сравнений, на i -том этапе происходит $(n-i)$ сравнений и т.д. Суммируя, получим $C = \frac{n^2 - n}{2}$

Отметим важные особенности метода прямого выбора. Метод не зависит от исходной отсортированности массива, т.е. значения M и C не меняются, даже если сортируется уже отсортированный массив. Сортировка методом прямого выбора неустойчива.

2.2 Пузырьковая сортировка

Популярный метод пузырьковой сортировки заключается в следующем. Двигаясь от конца массива к его началу, будем сравнивать между собой соседние элементы. При этом если правый элемент a_j будет меньше чем левый a_{j-1} , $j=n, n-1, \dots, 2$, то поменяем их местами. Таким образом, при первом проходе наименьший элемент переместится на первое место и можно не учитывать его при сортировке оставшейся части массива. При втором проходе наименьший элемент из оставшихся “всплывёт” на второе место. Через $(n-1)$ итераций массив будет отсортирован.

Пример. Отсортировать слово методом пузырьковой сортировки. Подчёркнутые пары, в которых произошёл обмен. Вертикальной чертой ограничена отсортированная часть массива.

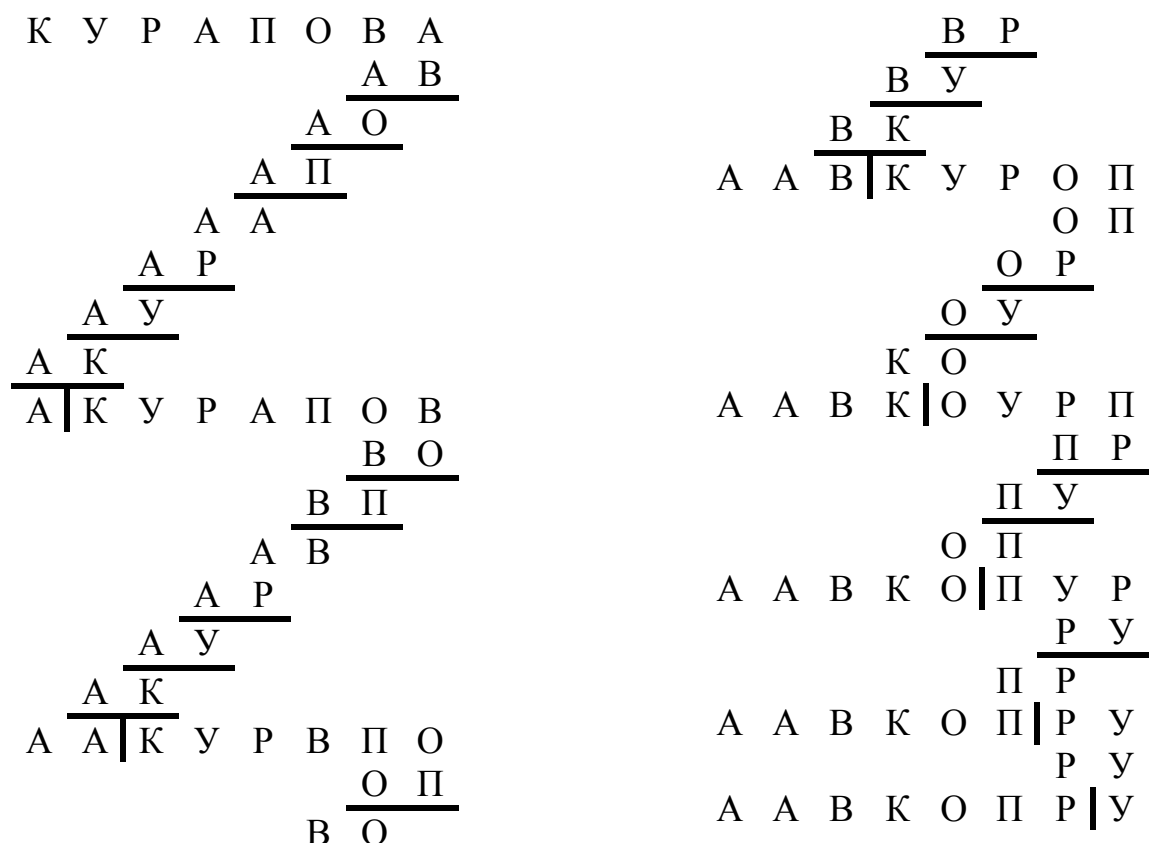


Рисунок 3 Пузырьковая сортировка

Алгоритм на псевдокоде *Пузырьковая сортировка*

Обозначим i – номер итерации,

j – индекс правого элемента в текущей паре.

DO ($i = 1, 2, \dots, n-1$)

DO ($j = n, n-1, \dots, i+1$)

IF ($a_j < a_{j-1}$) $a_j \leftrightarrow a_{j-1}$ FI

OD

OD

Проанализируем сложность пузырьковой сортировки. Количество сравнений в методе прямого выбора и в методе пузырьковой сортировки одинаково и не зависит от исходной отсортированности массива. Однако количество пересылок M зависит от того, как часто выполняется условие $a_j < a_{j-1}$. Можно определить максимальное и минимальное значения $M_{\min} \leq M_{\text{сред}} \leq M_{\max}$, где

$M_{\min} = 0$, при сортировке упорядоченного по возрастанию массива.

$M_{\max} = 3C = \frac{3(n^2 - n)}{2}$, при сортировке упорядоченного по убыванию массива.

Отсюда следует, что $M_{\text{сред}} = O(n^2)$, при $n \rightarrow \infty$

Таким образом, пузырьковая сортировка сильно зависит от исходной упорядоченности массива по количеству сравнений. Метод обеспечивает устойчивую сортировку.

2.3 Шейкерная сортировка

Анализируя метод пузырьковой сортировки можно отметить два обстоятельства. Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, ее можно исключить из рассмотрения. Во-вторых, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо. Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (т.е. части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.

Пример. Отсортировать слово методом шейкерной сортировки. Подчёркнутые пары, в которых произошёл обмен. Вертикальной чертой ограничена рабочая часть массива.

```

      К  У  Р  А  П  О  В  А
                        А  В
                        А  О
                        А  П
                      А  А
                    А  Р
                  А  У
                А  К
              А | К  У  Р  А  П  О  В
                К  У
                Р  У
                А  У
                П  У
                О  У
              В  У
            А | К  Р  А  П  О  В | У
              В  О
              В  П
            А  В
            А  Р
          А  К
        А | А | К  Р  В  П  О | У
          К  Р
          В  Р
          П  Р
        О  Р
      А  А | К  В  П  О | Р  У
  
```

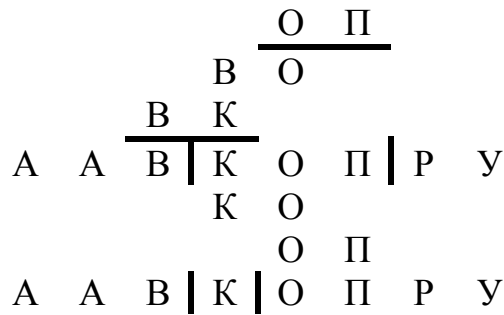


Рисунок 4 Шейкерная сортировка

Алгоритм на псевдокоде
Метод шейкерной сортировки

Обозначим

L – левая граница рабочей части массива.
R – правая граница рабочей части массива.
n – количество элементов в массиве

```

L: = 1, R: = n, k: = n,
DO
    DO (j = R, R-1, ... L+1)
        IF (aj < aj-1) aj ↔ aj-1, k: = j FI
    OD
    L: = k
    DO (j: = L, L+1, ... R-1)
        IF (aj > aj+1) aj ↔ aj+1, k: = j, FI
    OD
    R: = k
OD (L < R)

```

Оценим трудоемкость метода. Количество пересылок такое же, как и в методе пузырьковой сортировки $M_{\text{сред}} = O(n^2)$, при $n \rightarrow \infty$. Улучшения в методе шейкерной сортировки приводят к снижению количества сравнений. Точное выражение для величины C получить не удастся, поэтому определим границы, в которых изменяется C. Если сортируется массив, в котором элементы расположены в порядке возрастания, то в методе шейкерной сортировки достаточно один раз просмотреть массив. Тогда $C_{\min} = n - 1$, где n – количество элементов в массиве. Если массив отсортирован в обратном порядке, то на каждой итерации границы слева и справа сдвигаются на одну позицию и $C_{\max} = \frac{(n^2 - n)}{2}$. Следовательно, $C_{\text{сред}} = O(n^2)$, при $n \rightarrow \infty$. Таким образом, как и пузырьковая сортировка, метод шейкерной сортировки сильно зависит от исходной упорядоченности массива по количеству сравнений. Метод обеспечивает устойчивую сортировку.

2.4 Варианты заданий

1. Разработать процедуру сортировки массива целых чисел методом прямого выбора (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (M и C), сравнить их с теоретическими оценками.
2. В методе прямого выбора придумать способ устранения фиктивных перестановок и оценить его влияние на общую трудоемкость метода сортировки.
3. Разработать процедуру построения графика зависимости величин M и C от размера массива.
4. Разработать процедуру сортировки массива целых чисел методом пузырьковой сортировки (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (M и C), сравнить с теоретическими оценками.
5. Разработать процедуру сортировки массива целых чисел методом шейкерной сортировки (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (M и C), сравнить с теоретическими оценками.
6. Сравнить трудоемкости пузырьковой и шейкерной сортировок на массивах убывающих, случайных и возрастающих чисел.
7. Сравнить трудоемкости метода прямого выбора и шейкерной сортировки на массивах убывающих, случайных и возрастающих чисел.

3. МЕТОД ШЕЛЛА

3.1 Метод прямого включения

Сначала рассмотрим метод сортировки, который является базовым для метода Шелла. Метод прямого включения заключается в следующем. Начиная с $i = 2, i=2, \dots, n$, берём очередной i -й элемент массива и включаем его на нужное место среди первых $(i-1)$ элементов, при этом все элементы, которые больше a_i сдвигаются на одну позицию вправо.

Пример. Отсортировать слово методом прямого включения.

Условные обозначения

\boxed{X} i -тый элемент

\underline{X} сравнение элемента X с i -тым элементом

\rightarrow сдвиг элемента на одну позицию вправо



Рисунок 5 Метод прямого включения

Алгоритм на псевдокоде
Метод прямого включения

```

DO (i: = 2, 3, ... n)
    t: = ai, j: = i-1
    DO (j > 0 и t < aj)
        aj+1: = aj
        j: = j-1
    OD
    aj+1: = t

```

OD

Для метода прямого включения справедливы следующие оценки величин М и С.

$$C_{\min} \leq C_{\text{сред}} \leq C_{\max}, \text{ где } C_{\min} = n-1, C_{\max} = \frac{n^2 - n}{2},$$

$$M_{\min} \leq M_{\text{сред}} \leq M_{\max}, \text{ где } M_{\min} = 2(n-1), M_{\max} = \frac{n^2 - n}{2} + 2n - 2.$$

Минимальные и максимальные значения величин С и М достигаются на прямо отсортированном и обратно отсортированном массивах соответственно. Таким образом, средняя трудоемкость этого метода имеет квадратичный порядок, т.е. $C = O(n^2)$ $M = O(n^2)$, при $n \rightarrow \infty$.

Метод прямого включения устойчивый.

3.2 Метод Шелла

На базе метода прямого включения разработан алгоритм, обеспечивающий значительную производительность сортировки. Мы заметили, что в методе прямого включения, чем больше упорядочен массив, тем меньше операций требуется для его сортировки. При сортировке уже упорядоченного массива трудоемкость имеет линейный порядок. Поэтому имеет смысл попытаться предварительно несколько улучшить порядок элементов в массиве, а затем отсортиро-

вать массив методом прямого включения.

Предварительное упорядочивание будем проводить с помощью k – сортировок. Суть k – сортировки заключается в следующем. Массив разбивается на последовательности с шагом k

$$a_i, a_{k+i}, a_{2k+i}, \dots, a_{[n/k]k+i}, i = 1, 2, \dots, k$$

и сортировка происходит только внутри этих последовательностей.

Обозначим через H последовательность из m возрастающих шагов

$$H=(h_1, h_2, \dots, h_m), \text{ где } h_1=1, h_1 < h_2 < h_3 < \dots < h_m$$

Метод Шелла состоит в последовательном проведении h_i -сортировки, $i=m, m-1, \dots, 1$, причем $h_1=1$ гарантирует, что массив будет полностью отсортирован, поскольку 1-сортировка является методом прямого включения.

Пример. Отсортировать слово методом Шелла. Последовательность шагов выберем следующим образом $h_1=1, h_2=2$.

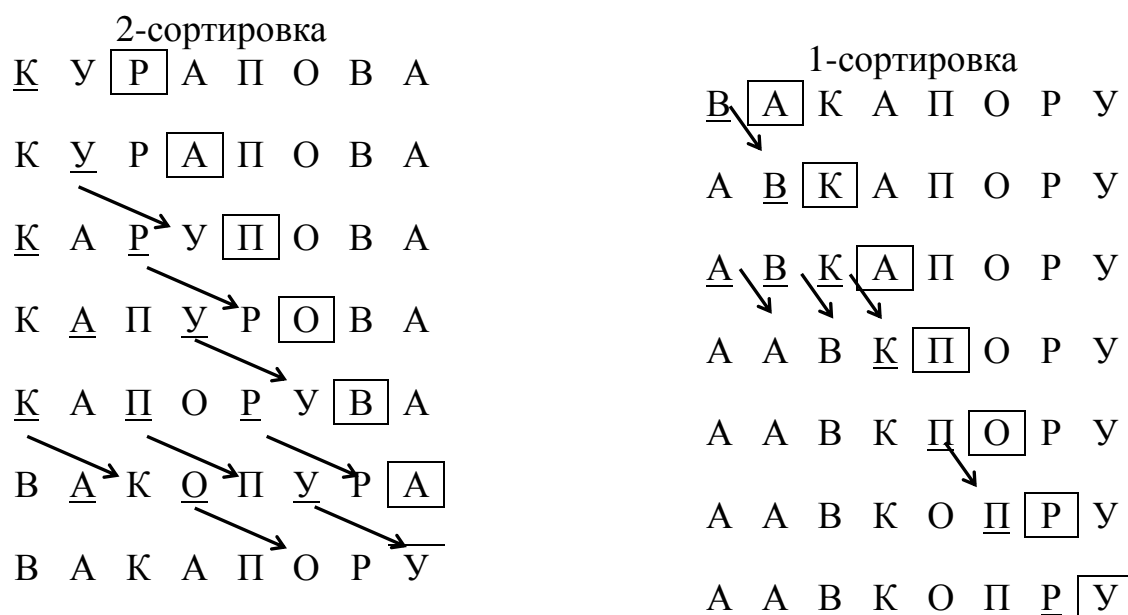


Рисунок 6 Метод Шелла

Алгоритм на псевдокоде

Сортировка методом Шелла

```

DO (k=hm, hm-1, ... 1)
  DO (i=k+1, k+2, ... n)
    t: = ai, j: =i-k
    DO (j>0 и t < aj)
      aj+k: = aj
      j: = j-k
    OD
    aj+k: = t
  OD
OD

```

OD

Эффективность метода зависит от выбора значений шагов. Последовательность значений шагов, которая дает наилучшую трудоемкость, пока неизвестна,

метод продолжает исследоваться, но существует и часто используется следующая последовательность шагов, предложенная Кнутом.

$$h_1=1, h_i=2h_{i-1}+1, i=2, \dots, m, m=\lfloor \log_2 n \rfloor - 1$$

При такой последовательности шагов средний порядок трудоёмкости $O(n^{1.2})$, $n \rightarrow \infty$. Таким образом, метод Шелла существенно быстрее методов с квадратичной трудоёмкостью. Метод Шелла не устойчив.

3.3 Варианты заданий

1. Разработать процедуру сортировки массива целых чисел методом прямого включения (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (М и С), сравнить их с теоретическими оценками.
2. Разработать процедуру сортировки массива целых чисел методом Шелла (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (М и С), сравнить их с теоретическими оценками.
3. Сравнить метод прямого включения и метод Шелла по количеству сравнений и пересылок для различных типов массивов. Разработать процедуру построения графика зависимости величин М и С от размера массива.
4. Сравнить метод прямого включения и метод Шелла с ранее рассмотренными методами сортировки по количеству сравнений и пересылок для различных типов массивов. Разработать процедуру построения графика зависимости величин М и С от размера массива
5. Экспериментально определить подходящую последовательность шагов для метода Шелла.
6. Опытным путем определить константы в выражениях оценки для количества сравнений и пересылок в методе Шелла.

4. БЫСТРЫЕ МЕТОДЫ СОРТИРОВКИ МАССИВОВ

4.1 Пирамидальная сортировка

Пирамидальная сортировка основана на алгоритме построения пирамиды. Последовательность a_i, a_{i+1}, \dots, a_k называется (i, k) -пирамидой, если неравенство

$$a_j \leq \min(a_{2j}, a_{2j+1}) \quad (*)$$

выполняется для каждого $j, j=i, \dots, k$ для которого хотя бы один из элементов a_{2j}, a_{2j+1} существует.

Например, массив А является пирамидой, а массив В — не является.

$$A=(a_2, a_3, a_4, a_5, a_6, a_7, a_8)=(3, 2, 6, 4, 5, 7)$$

$$B=(b_1, b_2, b_3, b_4, b_5, b_6, b_7)=(3, 2, 6, 4, 5, 7)$$

Свойства пирамиды

1. Если последовательность $a_i, a_{i+1}, \dots, a_{k-1}, a_k$ является (i, k) -пирамидой, то последовательность a_{i+1}, \dots, a_{k-1} , полученная усечением элементов с обеих

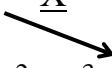
концов последовательности, является $(i+1, k-1)$ -пирамидой.

2. Если последовательность $a_1 \dots a_n$ – $(1, n)$ -пирамида, то a_1 – минимальный элемент последовательности.
3. Если $a_1, a_2, \dots, a_{n/2}, a_{n/2+1}, \dots, a_n$ – произвольная последовательность, то последовательность $a_{n/2+1}, \dots, a_n$ является $(n/2+1, n)$ -пирамидой.

Процесс построения пирамиды выглядит следующим образом. Дана последовательность a_{s+1}, \dots, a_k , которая является $(s+1, k)$ -пирамидой. Добавим новый элемент x и поставим его на s -тую позицию в последовательности, т.е. пирамида всегда будет расширяться влево. Если выполняется (*), то полученная последовательность – (s, k) -пирамида. Иначе найдутся элементы a_{2s+1}, a_{2s} такие, что либо $a_{2s} < a_s$ либо $a_{2s+1} < a_s$.

Пусть имеет место первый случай, второй случай рассматривается аналогично. Поменяем местами элементы a_s и a_{2s} . В результате получим новую последовательность $a_s, a_{s+1}, \dots, a_{2s}, \dots, a_k$. Повторяем все действия для элемента a_{2s} и т.д. пока не получим (s, k) -пирамиду.

Пример. Добавим в $(2, 8)$ -пирамиду новый элемент $x=6$.

Условные обозначения: \boxed{X} – новый элемент
 \underline{X} – сравнение с включаемым элементом
 – обмен элементов

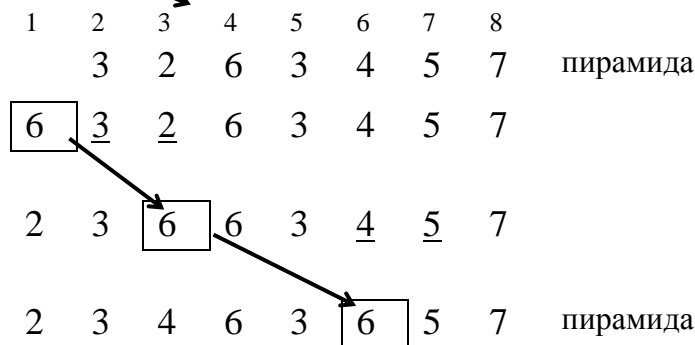


Рисунок 7 Добавление в пирамиду нового элемента

Алгоритм на псевдокоде

Построение (L, R) -пирамиды

a_{L+1}, \dots, a_R – на входе пирамида $(L+1, R)$

a_L – новый элемент

$x := a_L, i := L$

DO

$j := 2i$

 IF $(j > R)$ OD

 IF $((j < R) \text{ и } (a_{j+1} \leq a_j))$ $j := j + 1$ FI

 IF $(x \leq a_j)$ OD

$a_i := a_j$

$i := j$

OD

$a_i := x$

Величины M и C в процессе построения (L, R) -пирамиды имеют следующие оценки $M_{\text{пир}} \leq \log(R/L) + 2$, $C_{\text{пир}} \leq 2 \log(R/L)$

Пирамидальная сортировка производится в два этапа. Сначала строится пирамида из элементов массива. По свойству (3) правая часть массива является $(n/2+1, n)$ -пирамидой. Будем добавлять по одному элементу слева, расширяя пирамиду, пока в неё не войдут все элементы массива. Тогда по свойству (2) первый элемент последовательности – минимальный.

Произведём двустороннее усечение: уберём элементы a_1, a_n . По свойству (1) оставшаяся последовательность является $(2, n-1)$ -пирамидой. Элемент a_1 поставим на последнее место, а элемент a_n добавим к пирамиде a_2, \dots, a_{n-1} слева. Получим новую $(1, n-1)$ -пирамиду. В ней первый элемент является минимальным. Поставим первый элемент пирамиды на позицию $n-1$, а элемент a_{n-1} добавим к пирамиде a_2, \dots, a_{n-1} , и т.д. В результате получим обратно отсортированный массив.

Пример. Отсортировать слово методом пирамидальной сортировки.



Алгоритм на псевдокоде
Пирамидальная сортировка

```

L:=⌊n/2⌋
DO (L>0)
    <Построение (L,n) пирамиды>
    L:=L-1
OD
R:=n
DO (R>1)
    a1↔aR
    R:=R-1
    <Построение (1,R) пирамиды >
OD

```

Общее количество операций сравнений и пересылок для пирамидальной сортировки: $C \leq 2n \log n + n + 2$, $M \leq n \log n + 6.5n - 4$. Таким образом, $C = O(n \log n)$, $M = O(n \log n)$ при $n \rightarrow \infty$.

Отметим некоторые свойства пирамидальной сортировки. Метод пирамидальной сортировки неустойчив и не зависит от исходной отсортированности массива.

4.2 Метод Хоара

Метод Хоара или метод быстрой сортировки заключается в следующем. Возьмём произвольный элемент массива x . Просматривая массив слева, найдём элемент $a_i \geq x$. Просматривая массив справа, найдём $a_j \leq x$. Поменяем местами a_i и a_j . Будем продолжать процесс просмотра и обмена, до тех пор пока i не станет больше j . Тогда массив можно разбить на две части: в левой части все элементы не больше x , в правой части массива не меньше x . Затем к каждой части массива применяется тот же алгоритм.

Пример: Отсортировать слово методом быстрой сортировки.

Условные обозначения: \odot ведущий элемент

\underline{X} сравнение с ведущим элементом при просмотре справа

\overline{X} сравнение с ведущим элементом при просмотре слева

| разделение массива на части \longleftrightarrow обмен элементов

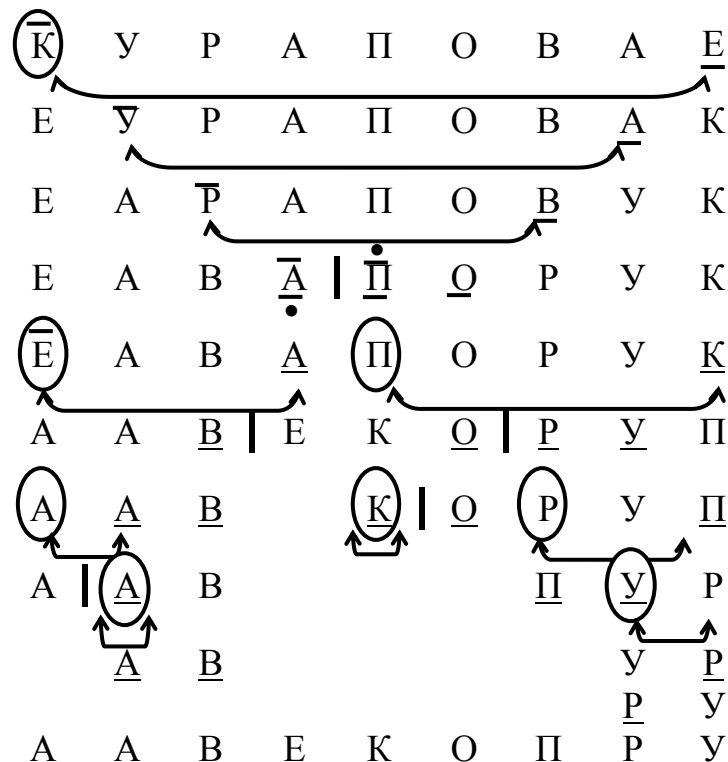


Рисунок 9 Метод Хоара

Алгоритм на псевдокоде

Сортировка части массива с границами (L, R) .

Обозначим: L-левую границу рабочей части массива

R-правую границу рабочей части массива

```

x:=aL, i:=L, j:=R,
DO (i≤j)
    DO (ai<x) i:=i+1 OD
    DO (aj>x) j:=j-1 OD
    IF (i≤j)
        ai↔ aj, i:=i+1, j:=j-1
    FI
OD
IF (L<j)
    <Сортировка части массива с границами (L,j)>
FI
IF (i<R)
    <Сортировка части массива с границами (i,R)>
FI

```

Очевидно, трудоёмкость метода существенно зависит от выбора элемента x , который влияет на разделение массива. Максимальные значения M и C для метода быстрой сортировки достигаются при сортировке упорядоченных массивов (в прямом и обратном порядке). Тогда в этом случае в одной части остаётся только один элемент (минимальный или максимальный), а во второй – все остальные

элементы. Выражения для М и С имеют следующий вид

$$M=3(n-1), C=(n^2+5n+4)/2$$

Таким образом, в случае упорядоченных массивов трудоёмкость сортировки имеет квадратичный порядок.

Элемент a_m называется *медианой* для элементов $a_L...a_R$, если количество элементов меньших a_m равно количеству элементов больших a_m с точностью до одного элемента (если количество элементов нечётно). В примере буква К- медиана для КУРАПОВАЕ.

Минимальная трудоёмкость метода Хоара достигается в случае, когда на каждом шаге алгоритма в качестве ведущего элемента выбирается медиана массива. Количество сравнений в этом случае $C=(n+1)\log(n+1)-(n+1)$. Количество пересылок зависит от положения элементов, но не может быть больше одного обмена на два сравнения. Поэтому количество пересылок – величина того же порядка, что и число сравнений. Асимптотические оценки для средних значений М и С имеют следующий вид

$$C=O(n \log n), M=O(n \log n) \text{ при } n \rightarrow \infty.$$

Метод Хоара неустойчив.

4.3 Проблема глубины рекурсии.

В теле подпрограммы доступны все объекты, описанные в основной программе, в том числе и имя самой подпрограммы. Это позволяет внутри тела подпрограммы осуществлять её вызов. Процедуры и функции, организующие вызовы «самих себя» называются *рекурсивными*. Рекурсия широко используется в программировании, потому что многие математические алгоритмы имеют рекурсивную природу.

В качестве примера приведём известный алгоритм вычисления факториала неотрицательного целого числа:

$$0!=1$$

$$1!=1$$

$$n!=(n-1)!*n$$

```
function fact (n:word):longint;
```

```
begin
```

```
  if (n=0) or (n=1) then fact:=1
```

```
  else fact:=fact(n-1)*n;
```

```
end;
```

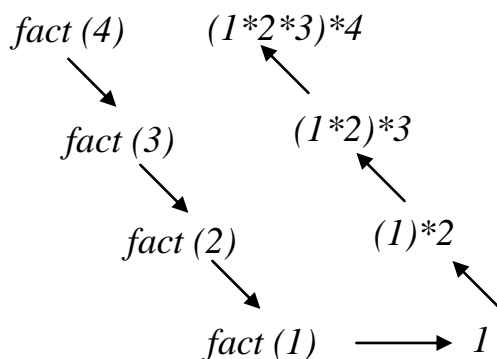


Рисунок 10 Схема вызовов при вычислении 4!

Рекурсивное оформление программы более компактно, наглядно и эффективно. Но существует опасность переполнения стека. Каждый вызов подпрограммы требует специально отведённой области памяти, называемой *фреймом*. В ней хранятся фактические параметры, адреса возврата, локальные переменные и регистры УП.

Фрейм	
Практический параметр	
Адрес возврата	
Регистры из программы	
Локальные переменные	

Рисунок 11 Структура фрейма

При выходе из программы эта память освобождается. Но если подпрограмма вызывает другую подпрограмму или саму себя, то в дополнение к существующему фрейму создаётся новый, т.е. n вложенных вызовов требуют выделения n фреймов в памяти.

Рассмотренный алгоритм Хоара может потребовать n вложенных вызовов (n – размер массива), т.е. глубина рекурсии достигает n . Это большой недостаток предложенного алгоритма. Попробуем уменьшить глубину рекурсии до $\log n$. В рассмотренном алгоритме производится 2 рекурсивных вызова. Но один из них можно заменить простой итерацией, т.е. для одной части массива будем применять рекурсию, а для другого – простую итерацию. Чтобы уменьшить глубину рекурсии нужно делать рекурсивный вызов для меньшей по размеру части массива. Тогда в худшем случае, когда размеры правой и левой частей будут одинаковыми, максимальная глубина рекурсии будет не больше $\log n$. Например, для массива из 1 млн. элементов понадобится одновременно менее 20 фреймов в памяти. Запишем новую версию алгоритма:

Алгоритм на псевдокоде

Сортировка части массива (L,R)

```

DO (есть хотя бы 2 элемента, т.е.  $L < R$ )
    <разделение> (как в 1 версии)
    IF (левая часть длиннее правой, т.е.  $j - L > R - i$ )
        Сортировка части массива (i,R)
        R:=j
    Else
        Сортировка части массива (L,j)
        L:=i;
    FI
OD

```

4.4 Варианты заданий

1. Разработать процедуру сортировки массива целых чисел методом пирамидальной сортировки (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа

- серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (М и С), сравнить их с теоретическими оценками.
2. Разработать процедуру сортировки массива целых чисел методом Хоара (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (М и С), сравнить их с теоретическими оценками.
 3. Сравнить метод пирамидальной сортировки и метод Хоара по количеству сравнений и пересылок для различных типов массивов. Разработать процедуру построения графика зависимости $M+C$ от размера массива n для случайного массива.
 4. Сравнить трудоемкости метода Хоара и метода прямого выбора для различных типов массивов. Разработать процедуру построения графика зависимости величин М и С от размера массива
 5. Экспериментально определить устойчивость и зависимость от начальной отсортированности массива пирамидальной сортировки и метода Хоара.
 6. Опытным путем определить константы в выражениях оценки для количества сравнений и пересылок в методе Хоара.
 7. Реализовать метода Хоара с меньшим количеством рекурсивных вызовов. Определить необходимое количество рекурсивных вызовов.
 8. Запрограммировать метод Хоара без использования рекурсивных вызовов.

5. РАБОТА С ЛИНЕЙНЫМИ СПИСКАМИ

5.1 Указатели. Основные операции с указателями

Каждый элемент данных, хранящихся в памяти компьютера, имеет свой адрес. Адреса могут находиться в специальных переменных, называемых указателями. Мы будем рассматривать типизированные указатели, которые могут хранить адреса только объектов определенного типа.

Пусть указатели p и q содержат адрес объекта x некоторого типа $tData$. Графически будем изображать это следующим образом:

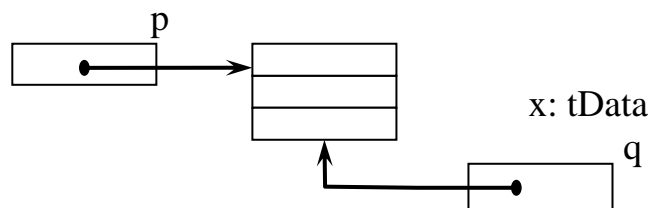


Рисунок 12 Равенство указателей

Стрелка начинается в указателе и указывает на объект в целом, а не на отдельную компоненту, поэтому указатели p и q равны. Заметим, что обычно адрес объекта совпадает с адресом его первой компоненты. Можно обращаться к переменной по имени, а можно по адресу. При обращении по адресу не нужно знать имени переменной.

Основные операции с указателями приведены в следующей таблице.

Таблица 2 Основные операции с указателями

Операция	Псевдокод	Комментарии
1. Присваивание	$p:=q$	Указателю p присвоить значение указателя q
2. Сравнение	$p=q, p \neq q$	Сравнение значений указателей p и q
3. Получение адреса	$p:=@x$	Указателю p присвоить адрес переменной x
4. Доступ по адресу	$*p=y \quad *p:=*q$	Переменной по адресу p присвоить значение переменной по адресу q
5. Доступ к отдельной компоненте	$p \rightarrow \text{comp}:=x$	Компоненте comp переменной по адресу p присвоить значение переменной x
6. Отсутствие адреса	NIL	Значение указателя p не равно никакому адресу

5.2 Основные операции с линейными списками

Списком называется последовательность однотипных элементов, связанных между собой указателями. Будем считать, что элементы списка имеют тип tLE, указатели на элементы списка имеют тип pLE.

X: tLE

p: pLE



Рисунок 13 Указатель на элемент списка

Поле *Next* является указателем на элемент списка и может занимать произвольное место в структуре элемента. Однако если оно является первым элементом структуры, то его адрес совпадает с адресом элемента списка, и это позволяет оптимизировать многие операции со списками. Поле *Data* содержит информацию, которая будет учитываться при сортировке.

Рассмотрим два вида списков: *стек* и *очередь*. *Стек* характеризуется тем, что новый элемент добавляется в начало последовательности, а удаляться может только первый элемент списка. При добавлении в *очередь* новый элемент ставится в конец списка, удаляется первый элемент последовательности.

Рассмотрим основные операции со стеком и очередью. Для работы со стеком необходимо иметь указатель на начало списка. Обозначим его *Head*. При работе с очередью требуется дополнительный указатель на конец очереди. Обозначим его *Tail*. Иногда при работе с очередью удобно объединять указатели *Head* и *Tail* в виде полей некоторой переменной *Queue*.

Добавление элемента по адресу p в стек.

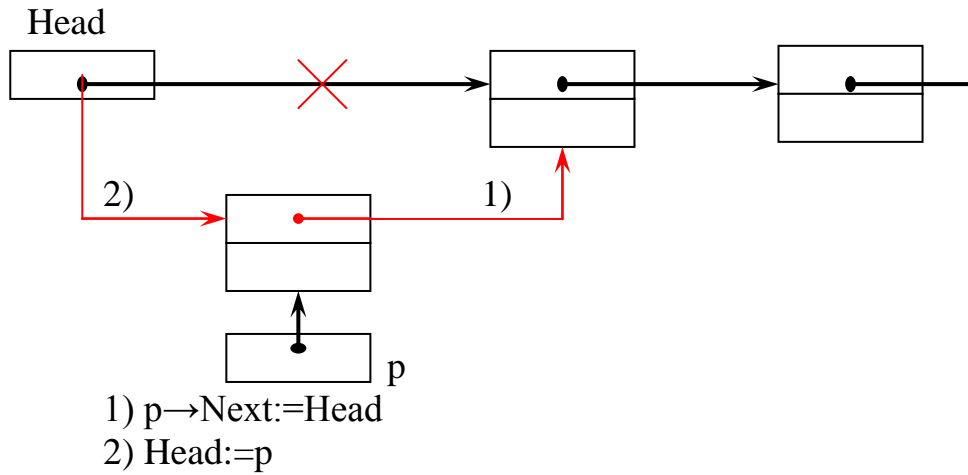


Рисунок 14 Добавление в стек

Удаление из стека или очереди (при условии, что список не пуст, т.е. $\text{Head} \neq \text{NIL}$)

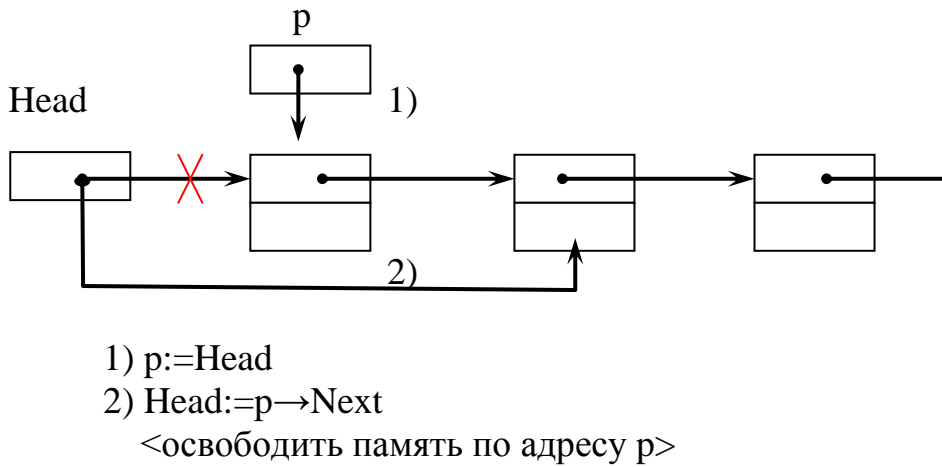


Рисунок 15 Удаление из стека

Добавление элемента по адресу p в очередь.

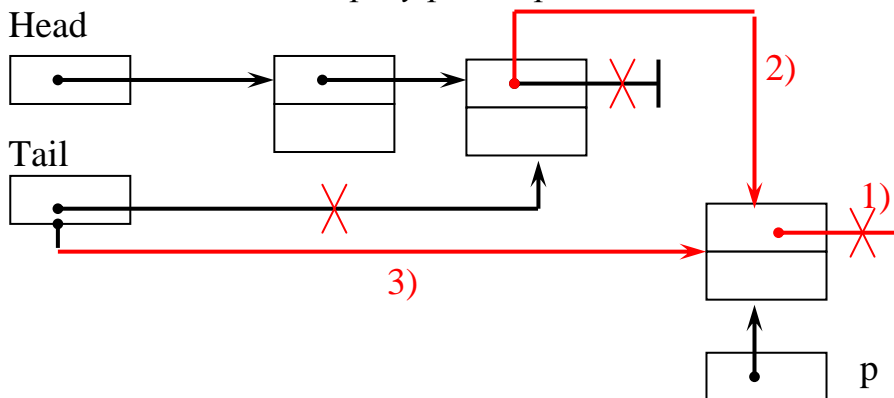


Рисунок 16 Добавление в очередь

1) $p \rightarrow \text{Next} := \text{NIL}$
 2) IF ($\text{Head} \neq \text{NIL}$) $\text{Tail} \rightarrow \text{Next} := p$
 ELSE $\text{Head} := p$
 FI

3) Tail:=p

Операцию добавления элемента в очередь можно оптимизировать в случае, если поле Next является первой компонентой элемента очереди и его адрес совпадает с адресом всего элемента. Зададим пустую очередь следующим образом:

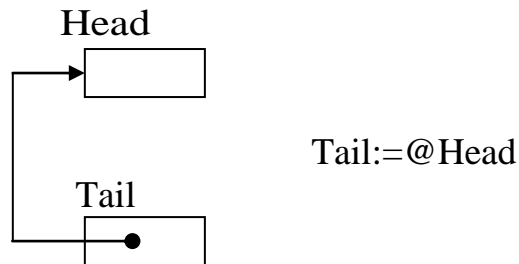


Рисунок 17 Структура очереди

Эту операцию назовем инициализацией очереди. Тогда добавление элемента в очередь будет происходить в два раза быстрее:

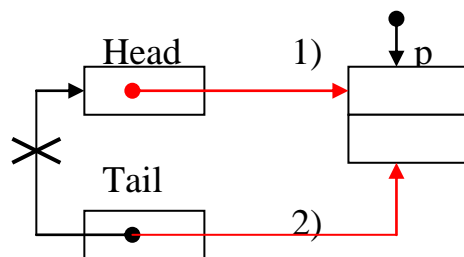


Рисунок 18 Добавление в очередь

- 1) Tail->Next:=p
- 2) Tail:=p

Перемещение элемента из начала списка List в конец очереди Queue.

3)

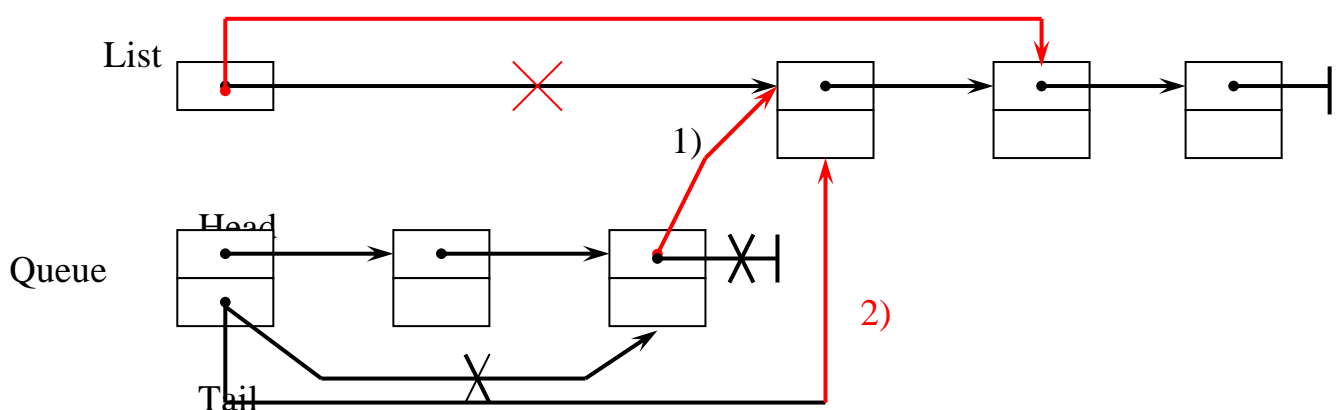


Рисунок 19 Перемещение элемента

- 1) Queue.Tail->Next:=List
- 2) Queue.Tail:=List
- 3) List:=List->Next

6. МЕТОДЫ СОРТИРОВКИ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

6.1 Метод прямого слияния

В основе метода прямого слияния лежит операция слияния серий. *p*-серией называется упорядоченная последовательность из *p* элементов.

Пусть имеются две упорядоченные серии *a* и *b* длины *q* и *r* соответственно. Необходимо получить упорядоченную последовательность *c*, которая состоит из элементов серий *a* и *b*. Сначала сравниваем первые элементы последовательностей *a* и *b*. Минимальный элемент перемещаем в последовательность *c*. Повторяем действия до тех пор, пока одна из последовательностей *a* и *b* не станет пустой, оставшиеся элементы из другой последовательности переносим в последовательность *c*. В результате получим $(q+r)$ -серию.

Пример. Слить две серии $a=(1, 4, 5, 6')$ и $b=(2, 3, 6'', 7, 8)$

Условные обозначения | операция сравнения первых элементов списков.

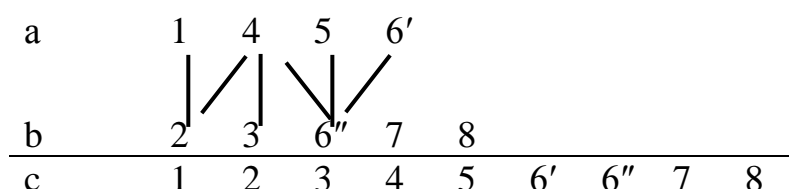


Рисунок 20 Слияние серий

Алгоритм на псевдокоде

Слияние *q* – серии из списка *a* с *r* – серией из списка *b*,
запись результата в очередь *c*
Слияние (*a*, *q*, *b*, *r*, *c*)

```

DO (q ≠ 0 и r ≠ 0)
  IF (a → Data ≤ b → Data)
    <Переместить элемент из списка a в очередь c>
    q:=q-1
  ELSE
    <Переместить элемент из списка b в очередь c>
    r:=r-1
  FI
OD
DO (q > 0)
  <переместить элемент из списка a в очередь c>
  q:=q-1
OD
DO (r > 0)
  <Переместить элемент из списка b в очередь c>
  r:=r-1
OD

```

Для алгоритма слияния серий с длинами *q* и *r* необходимое количество сравнений

и перемещений оценивается следующим образом

$$\min(q, r) \leq C \leq q+r-1, M=q+r$$

Пусть длина списка S равна степени двойки, т.е. 2^k , для некоторого натурального k . Разобьем последовательность S на два списка a и b , записывая поочередно элементы S в списки a и b . Сливаем списки a и b с образованием двойных серий, то есть одиночные элементы сливаются в упорядоченные пары, которые записываются попеременно в очереди c_0 и c_1 . Переписываем очередь c_0 в список a , очередь c_1 – в список b . Вновь сливаем a и b с образованием серий длины 4 и т. д. На каждой итерации размер серий увеличивается вдвое. Сортировка заканчивается, когда длина серии превысит общее количество элементов в обоих списках. Если длина списка S не является степенью двойки, то некоторые серии в процессе сортировки могут быть короче.

Пример. Отсортировать слово методом прямого слияния.

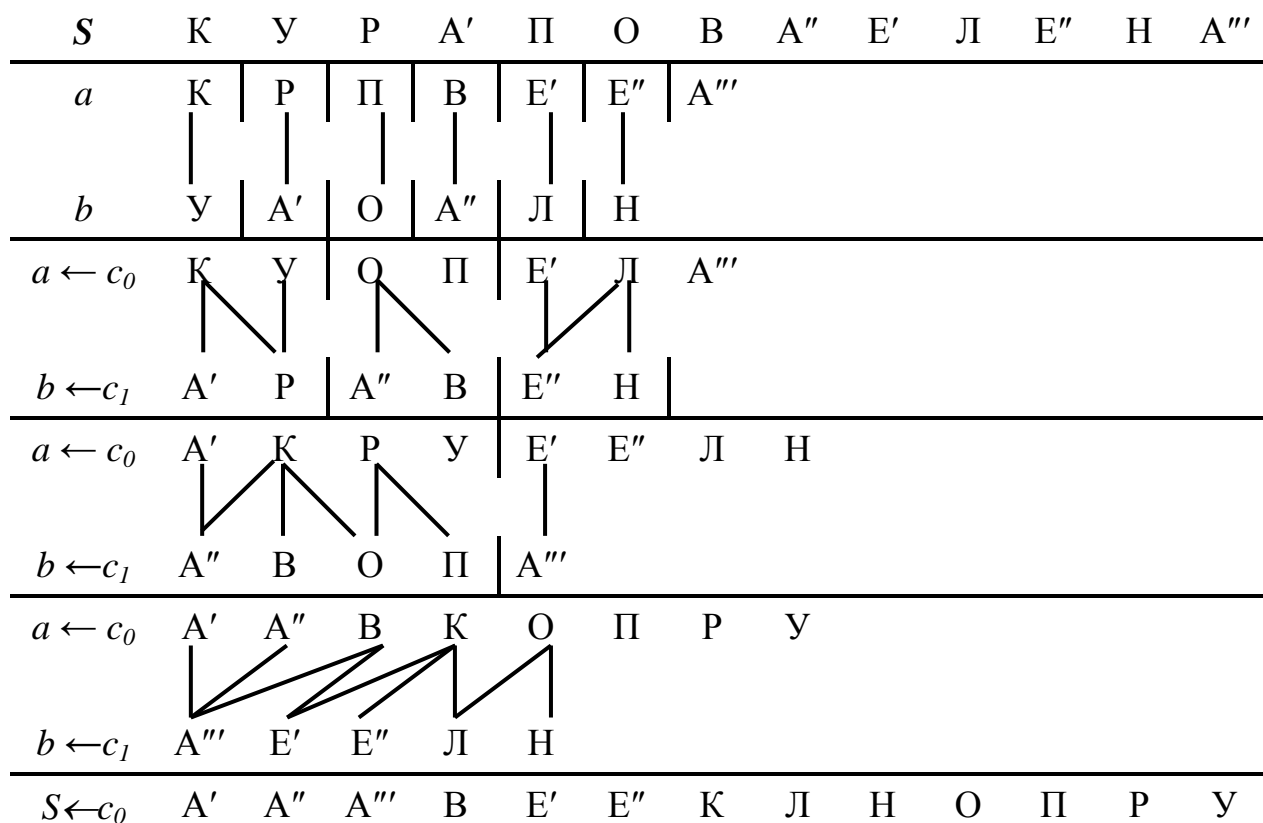


Рисунок 21 Метод прямого слияния

Схематически начальное расщепление последовательности S на списки a и b можно изобразить следующим образом. Ниже приведен алгоритм расщепления на псевдокоде при условии, что список S не пуст.

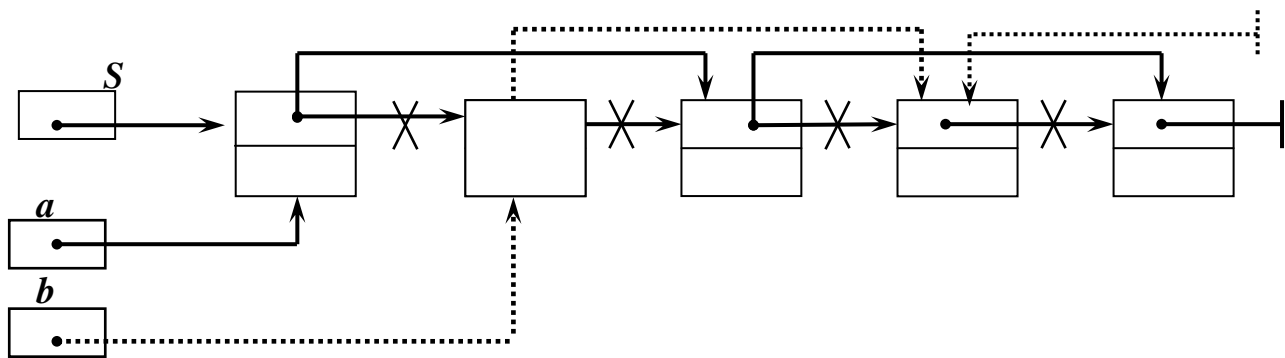


Рисунок 22 Начальное расщепление

Расщепление (S, a, b, n)

Обозначим

n - количество элементов в S

k, p - рабочие указатели

a:=S, b:=S → Next, n:=1

k:=a, p:=b

DO (p ≠ NIL)

n:=n+1

k → next:=p → next

k:=p

p:=p → next

OD

Алгоритм на псевдокоде

Обозначим

n – количество элементов в S

a, b – рабочие списки

c=(c₀, c₁) – массив из двух очередей

p – предполагаемый размер серии

q – фактический размер серии в списке a

r – фактический размер серии в списке b

m – текущее количество элементов в списках a и b

i – номер активной очереди

<Расщепление (S, a, b, n)>

p:= 1

DO (p < n)

<инициализация очередей c₀, c₁>

i:=0, m:=n

DO (m > 0)

IF (m ≥ p) q:=p ELSE q:=m FI

m:= m – q

IF (m ≥ p) r:=p ELSE r:=m FI

m:= m – r

<слияние(a, q, b, r, c_i)>

i:=1–i

```

OD
a:=c0.Head, b:=c1.Head
p:=2p
OD
c0.Tail → next:=NIL
S:=c0.Head

```

При инициализации очереди обнуляются указатели, указывающие на начало и на конец очереди, т.е. очередь становится пустой.

Трудоёмкость метода прямого слияния определяется сложностью операции слияния серий. На каждой итерации происходит ровно n перемещений элементов списка и не более n сравнений. Как нетрудно видеть, количество итераций равно $\lceil \log n \rceil$. Тогда

$$C < n \lceil \log n \rceil, M = n \lceil \log n \rceil + n.$$

Дополнительные n перемещений происходят во время начального расщепления исходного списка. Асимптотические оценки для M и C имеют следующий вид

$$C = O(n \log n), M = O(n \log n) \text{ при } n \rightarrow \infty.$$

Метод обеспечивает устойчивую сортировку. При реализации для массивов, метод требует наличия второго вспомогательного массива, равного по размеру исходному массиву. При реализации со списками дополнительной памяти не требуется.

6.2 Цифровая сортировка

Другим методом сортировки последовательностей является цифровая сортировка. Пусть дана последовательность из S чисел, представленных в m -ичной системе счисления. Каждое число состоит из L цифр $d_1 d_2 \dots d_L$, $0 \leq d_i \leq m - 1$, $i = 1..L$. Сначала числа из списка S распределяются по m очередям, причём номер очереди определяется последней цифрой каждого числа. Затем полученные очереди соединяются в список, для которого все действия повторяются, но распределение по очередям производится в соответствии со следующей цифрой и т.д.

Пример. Отсортировать последовательность 31 03' 20 02 03'' 33 30 21 методом цифровой сортировки. Числа представлены в четверичной системе счисления.

S :	31	03'	20	02	03''	33	30	21
Q ₀ :	20	30						
Q ₁ :	31	21						
Q ₂ :	02							
Q ₃ :	03'	03''	33					
S :	20	30	31	21	02	03'	03''	33
Q ₀ :	02	03'	03''					
Q ₁ :								
Q ₂ :	20	21						
Q ₃ :	30	31	33					
S :	02	03'	03''	20	21	30	31	33

Рисунок 23 Цифровая сортировка

Алгоритм на псевдокоде
Цифровая сортировка

```
DO (j=L, L-1, ... , 1)
    <инициализация очередей Q>
    <расстановка элементов из списка S в очереди Q по j – ой цифре >
    <соединение очередей Q в список S >
```

OD

Цифровой метод может успешно использоваться не только для сортировки чисел, но и для сортировки любой информации, представленной в памяти компьютера. Необходимо лишь рассматривать каждый байт ключа сортировки как цифру, принимающую значения от 0 до 255. Тогда для сортировки потребуется $m=256$ очередей. Для выделения каждого байта ключа сортировки можно использовать массив Digit, наложенный в памяти компьютера на поле элемента последовательности, по которому происходит сортировка. Приведем более детальный алгоритм цифровой сортировки.

Алгоритм на псевдокоде
Цифровая сортировка

```
DO (j=L, L-1, ... 1)
    DO (i=0, 1, ... 255)
        Qi.Tail:=@ Qi.Head
    OD
    p:=S
    DO (p ≠ NIL)
        d:=p → Digit[j]
        Qd.Tail → Next:=p
        Qd.Tail:=p
        p:=p → Next
    OD
    p:=@ S
    DO (i=0, 1, ... 255)
        IF (Qi.Tail ≠ @ Qi.Head)
            p → Next:=Qi.Head
            p:=Qi.Tail
        FI
    OD
    p → Next:=NIL
OD
```

Для цифровой сортировки $M < \text{const } L(m+n)$. При фиксированных m и L $M=O(n)$ при $n \rightarrow \infty$, что значительно быстрее остальных рассмотренных методов. Однако если длина чисел L велика, то метод может проигрывать обычным методам сортировки. Кроме того, Метод применим только, если задача сортировки сводится к задаче упорядочивания чисел, что не всегда возможно.

Метод обеспечивает устойчивую сортировку. Чтобы изменить направление

сортировки на обратное, очереди нужно присоединять в обратном порядке.

6.3 *Варианты заданий*

1. Разработать процедуру сортировки последовательности целых чисел методом прямого слияния (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в последовательности. Предусмотреть подсчет количества пересылок и сравнений (M и C), сравнить их с теоретическими оценками.
2. Разработать процедуру цифровой сортировки последовательности целых чисел (язык программирования Паскаль или Си). Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в последовательности. Предусмотреть подсчет количества пересылок и сравнений (M и C), сравнить их с теоретическими оценками.
3. Сравнить метод прямого слияния и метод цифровой сортировки по количеству сравнений и пересылок для различных последовательностей. Разработать процедуру построения графика зависимости $M+C$ от длины последовательности n .
4. Сравнить трудоемкости методов сортировки последовательностей длины n и методов быстрой сортировки массивов длины n . результаты оформить в виде таблицы.
5. Экспериментально определить устойчивость и зависимость от начальной отсортированности последовательности методов сортировки последовательностей.
6. Опытным путем определить константы в выражениях оценки для количества сравнений и пересылок в методе прямого слияния и методе цифровой сортировки.
7. Определить длину чисел (количество цифр в записи числа) в последовательности, при которой цифровая сортировка работает медленнее, чем метод Хоара для массива той же длины.

7. ДВОИЧНЫЙ ПОИСК В УПОРЯДОЧЕННОМ МАССИВЕ

7.1 *Алгоритм двоичного поиска*

Алгоритм двоичного поиска в упорядоченном массиве сводится к следующему. Берём средний элемент отсортированного массива и сравниваем с ключом X . Возможны три варианта:

1. Выбранный элемент равен X . Поиск завершён.
2. Выбранный элемент меньше X . Продолжаем поиск в правой половине массива.
3. Выбранный элемент больше X . Продолжаем поиск в левой половине массива.

Далее рассмотрим две версии реализации двоичного поиска в упорядоченном массиве.

Версия 1

Пример. Найти в отсортированном массиве элемент с ключом $X = б$.

1	2	3	4	5	6	7	8	9	10	11	12
а	б	б	б	е	ж	и	к	л	м	н	о
а	б	б	б	е							

Рисунок 24 Первая версия поиска

Алгоритм на псевдокоде

Поиск элемента с ключом X

Обозначим

L, R – правая и левая границы рабочей части массива.

Найден – логическая переменная, в которой будем отмечать факт успешного завершения поиска.

$L := 1, R := n, \text{Найден} := \text{нет}$

DO ($L \leq R$)

$m := \lfloor (L + R) / 2 \rfloor$

IF ($a_m = X$) Найден: =да OD FI

IF ($a_m < X$) $L := m + 1$

ELSE $R := m - 1$

FI

OD

Отметим недостаток этой версии алгоритма. На каждой итерации совершается два сравнения, но можно обойтись одним сравнением. Если в массиве несколько элементов с одинаковым ключом, то эта версия находит один из них. Чтобы найти все элементы с одинаковыми ключами, необходимо просмотреть массив влево и вправо от найденного элемента.

Версия 2

Пример. Найти в отсортированном массиве элемент с ключом $X = б$.

1	2	3	4	5	6	7	8	9	10	11	12
а	б	б	б	е	ж	и	к	л	м	н	о
а	б	б	б	е	ж						
а	б	б	б	е	ж						
а	б	б									
а	б										
а											

Рисунок 25 Вторая версия поиска

Алгоритм на псевдокоде
Поиск элемента с ключом X

```
L: = 1, R: =n
DO (L<R)
    m: =  $\lfloor (L + R) / 2 \rfloor$ 
    IF ( $a_m < X$ ) L: = m+1
        ELSE R: = m
    FI
OD
IF ( $a_L = X$ ) Найден: =да
    ELSE Найден: =нет
FI
```

Нетрудно заметить, что после выхода из цикла $L = R$. Если в массиве несколько элементов с одинаковым ключом, то эта версия алгоритма находит самый левый из них. Для поиска остальных элементов с заданным ключом требуется просмотреть массив только в одном направлении – вправо от найденного элемента.

Дадим верхнюю оценку трудоёмкости алгоритма двоичного поиска. На каждой итерации поиска необходимо два сравнения для первой версии, одно сравнение для второй версии. Количество итераций не больше, чем $\lceil \log_2 n \rceil$. Таким образом, трудоёмкость двоичного поиска в обоих случаях

$$C = O(\log n), n \rightarrow \infty.$$

7.2 *Варианты заданий*

1. Написать программу (язык программирования Паскаль или Си) для быстрого поиска в упорядоченном массиве.
2. Сравнить средние трудоёмкости различных версий быстрого поиска.
3. Построить графики зависимости средней трудоёмкости поиска от количества элементов в массиве для различных вариантов поиска.
4. Сравнить трудоёмкость быстрого поиска с трудоёмкостью поиска перебором.
5. Экспериментально оценить долю случаев, когда поиск перебором происходит быстрее, чем быстрый поиск.

8. СОРТИРОВКА ДАННЫХ С ПРОИЗВОЛЬНОЙ СТРУКТУРОЙ

8.1 *Сравнение данных произвольной структуры*

Основная проблема, возникающая при сортировке данных произвольной структуры — неопределенность операции сравнения. Если исходный массив A заполнен числами, то в качестве операции сравнения могут быть использованы стандартные операции сравнения. Если структура сортируемых данных не соответствует простым встроенным типам языка, то необходимо переопределить операции сравнения с помощью логических функций. Например, пусть массив A — телефонный справочник, каждый элемент которого является записью с полями Name (фамилия абонента) и Phone (номер телефона):

А:

<i>Иванов</i>	<i>Петров</i>	<i>Егоров</i>
223455	452185		454455

Рисунок 26 Список абонентов

Если необходимо отсортировать телефонный справочник по фамилиям абонентов, то логическая функция Less (меньше) может выглядеть следующим образом:

```
function less ( x,y: <тип записи>): boolean;  
begin  
    less:=x.Name <y.Name;  
end;
```

Такой подход позволяет путем изменения функции Less учитывать любые сложные условия упорядочивания массива элементов произвольной структуры. Например, если необходимо упорядочить телефонный справочник по номерам телефонов дополнительно по фамилиям абонентов, то функцию Less можно записать так:

```
function less ( x,y: <тип записи>): boolean;  
begin  
  
    IF (x.phone<y.phone)  
        less:=true  
    ELSEIF (x.phone>y.phone )  
        less:=false  
    ELSE less:=(x.name<y.name)  
    FI  
    FI  
end;
```

Кроме того, нужно переопределить операцию сравнения и при организации поиска элементов в отсортированном массиве. Изменение направления упорядочивания массива достигается путем замены операций сравнения на противоположные, т. е. в самой функции меняем «<» на «>». Операция пересылки не требует переопределения и выполняется путем побитового копирования.

8.2 Сортировка по множеству ключей. Индексация

Пусть рассмотренный выше телефонный справочник необходимо использовать для быстрого поочередного поиска абонентов или по номеру телефона, или по фамилии абонента. Пересортировка массива то по одному, то по другому ключу требует значительных затрат времени. Для эффективного решения подобной задачи используется прием, называемый индексацией, или созданием индексного массива.

Вначале построения индексный массив В заполняется целыми числами от 1

до n . Затем производится сортировка, но при условии, что в операциях сравнения элементы массива A индексируются через массив B . Перестановки делаются только в массиве B . Тогда при доступе к элементам массива A через индексный массив B $A[B[i]]$ можно работать с массивом A как с упорядоченным по возрастанию (например, производить быстрый поиск элементов), в то время как сами элементы A физически не переставляются.

Пример. Рассмотрим создание индексного массива, который упорядочивает массив целых чисел $A=(7,1,6,3,2,8,5)$. Чтобы упорядочивать массив A по возрастанию, пронумеруем его элементы. Введем новый массив B и запишем в него номера массива A в последовательности, соответствующей условию упорядочивания (по возрастанию). Получим следующий индексный массив $B=(2,5,4,7,3,1,6)$

Алгоритм на псевдокоде
(на примере пузырьковой сортировки)

```
B:=(1,2,...,n)
DO (i=1,2,...,n-1)
    DO(j=n,n-1,...,i+1)
        IF(a[bj] < a[bj-1]) bi ↔ bj-1 FI
    OD
OD
```

Отметим ряд положительных свойств индексации.

1. Индексация дает возможность построения нескольких различных индексов, которые можно использовать по мере необходимости.
2. Исключается копирование больших массивов данных (физический массив остаётся на месте, а индексы занимают мало места).
3. Имеется возможность фильтрации данных. Фильтрация означает, что при работе с базами данных используются не все элементы, а только те, которые отвечают определённым условиям. В индекс включаются физические номера тех элементов, которые удовлетворяют условию фильтра.

8.3 Индексация через массив указателей

Индексация через массив указателей отличается от обычной индексации тем, что вместо номеров элементов в индексный массив записываются адреса сортируемых элементов. К достоинствам такой индексации можно отнести то, что исходные данные могут располагаться не только в массиве, а произвольным образом в динамической памяти.

8.4 Варианты заданий

Написать программу «Телефонный справочник». Каждый абонент имеет имя, адрес, телефонный номер. С помощью индексов и фильтров

1. упорядочить справочник по имени по возрастанию
2. упорядочить справочник по телефонному номеру по возрастанию
3. упорядочить справочник по адресу по убыванию
4. выбрать тех абонентов, которые имеют номер в заданном диапазоне

5. упорядочить справочник по имени и телефонному номеру по возрастанию
6. выбрать тех абонентов, которые имеют имя в заданном диапазоне
7. выбрать абонентов, которые имеют имя и адрес в заданном диапазоне

9. ДВОИЧНЫЕ ДЕРЕВЬЯ

9.1 Основные определения и понятия

Определим двоичное дерево следующим образом :

1. Отдельная вершина V является двоичным деревом.
2. Двоичное дерево – это вершина V , соединенная с (возможно пустыми) левым (T_L) и правым (T_R) двоичными деревьями.

Пример двоичного дерева приведен на следующем рисунке. Вершины дерева обозначаются кружочками, связи между вершинами стрелками.

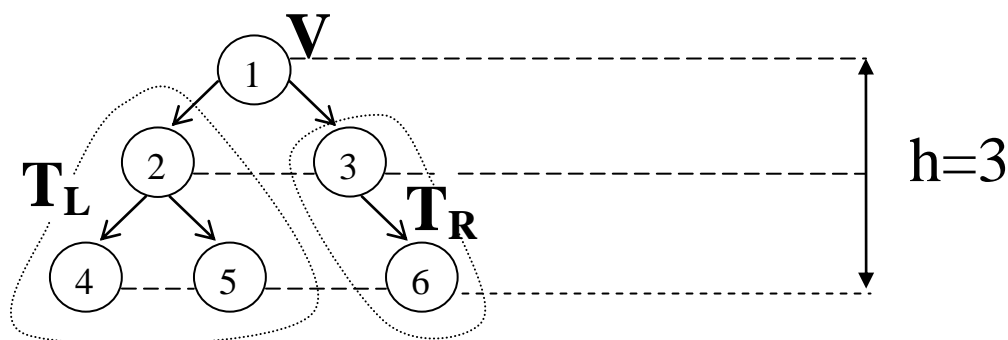


Рисунок 27 Пример двоичного дерева

Каждая вершина дерева может содержать какую-либо информацию. Выделенная вершина дерева называется *корнем*. Концевые вершины дерева, не имеющие поддеревьев, называются *листьями*. Дуги ориентированы по направлению от корня к листьям. Путь от корня к листу называется *ветвью*. Под *длиной ветви* будем понимать число входящих в неё вершин. *Высота дерева h* определяется как число вершин в самой длинной ветви дерева. *Размер дерева* – число входящих в него вершин. Средней высотой дерева называется усредненная по количеству вершин в дереве сумма длин путей от корня к каждой вершине.

Приведем некоторые свойства двоичных деревьев.

Свойство 1. Максимальное число вершин в двоичном дереве высоты h равно

$$n_{\max}(h) = 2^h - 1$$

Свойство 2. Минимально возможная высота двоичного дерева с n вершинами равна $h_{\min}(n) = \lceil \log(n+1) \rceil$

9.2 Различные обходы двоичных деревьев

При разработке алгоритмов для работы с деревьями будем считать, что каждая вершина содержит некоторые данные и указатели на вершины слева и справа. Поэтому вершина дерева – это переменная специального типа. В качестве заголовка для дерева используем переменную *Root*, указывающую на корень.

```
TYPE      pVertex = ^tVertex;
          tVertex = record
```

```

Data: integer;
Left: pVertex;
Right: pVertex;
end;

```

```

VAR      Root: pVertex;

```

При решении многих задач, связанных с деревьями, часто возникает необходимость перебрать все вершины дерева или совершить *обход дерева*, чтобы выполнить некоторые действия с каждой вершиной дерева.

Существуют три основных порядка обхода дерева:

1. Сверху вниз (\downarrow): корень, левое поддерево, правое поддерево.
2. Слева направо (\rightarrow): левое поддерево, корень, правое поддерево.
3. Снизу вверх (\uparrow): левое поддерево, правое поддерево, корень.

Пример. Совершить обход слева направо для двоичного дерева, изображенного на рисунке 28.

Результат обхода: 4 2 5 1 3 6

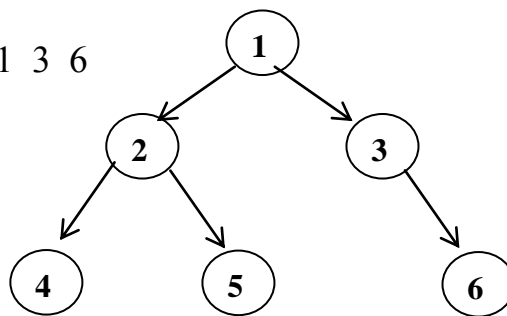


Рисунок 28

Обходы легко программируются с помощью рекурсивных процедур.

Алгоритм на псевдокоде

Обход слева направо(p: pVertex)

IF (p \neq NIL)

Обход слева направо (p \rightarrow Left)

Печать (p \rightarrow Data)

Обход слева направо (p \rightarrow Right)

FI

9.3 Вычисление основных характеристик дерева

Приведем алгоритмы для вычисления различных характеристик двоичных деревьев.

1) Определение размера дерева

Алгоритм на псевдокоде

Размер(p: pVertex)

IF (p = NIL) Размер := 0

ELSE Размер := 1 + Размер (p \rightarrow Left) + Размер (p \rightarrow Right)

FI

2) Определение высоты дерева

Алгоритм на псевдокоде

Высота(p: pVertex)

IF (p = NIL) Высота := 0

ELSE Высота := 1 + max(Высота (p→Left), Высота(p→Right))

FI

3) Определение средней высоты дерева

Для определения средней высоты дерева понадобится функция вычисления суммы длин путей от корня до каждой вершины на L-том уровне.

Алгоритм на псевдокоде

СДП (p: pVertex; L: уровень вершины)

IF (p = NIL) СДП:= 0

ELSE СДП:= L + СДП(p → Left, L+1) + СДП(p → Right, L+1)

FI

Тогда средняя высота вычисляется следующим образом

$$h_{cp} := \text{СДП}(\text{Root}, 1) / \text{Размер}(\text{Root})$$

4) Определение контрольной суммы для дерева

Алгоритм на псевдокоде

Сумма (p: pVertex)

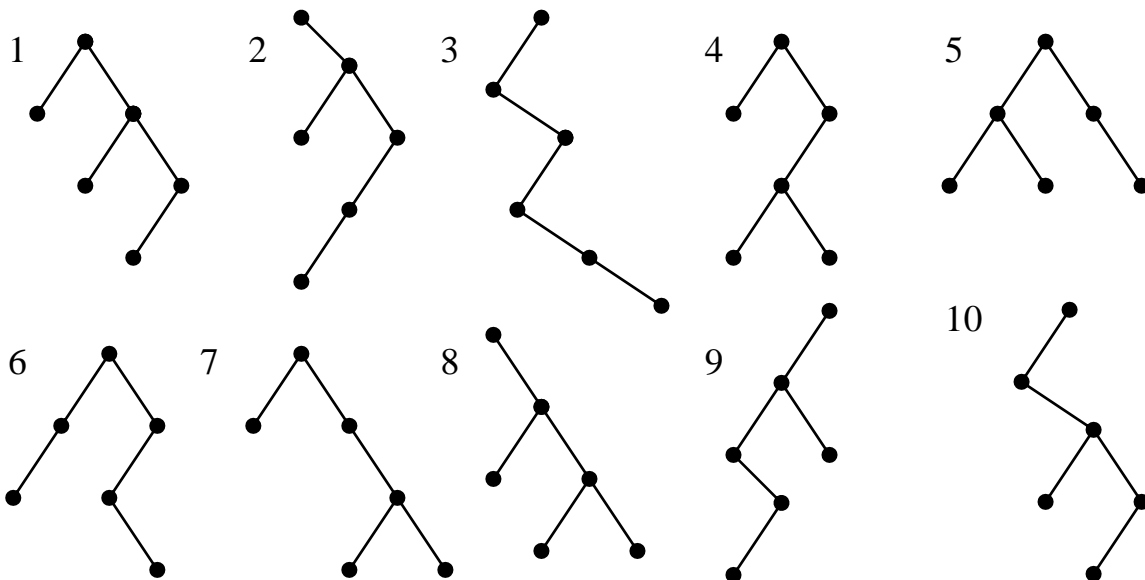
IF (p = NIL) Сумма := 0

ELSE Сумма:= p→Data + Сумма(p→Left) + Сумма(p→Right)

FI

9.4 *Варианты заданий*

Разместить в памяти компьютера данное двоичное дерево (см. ниже), данные в вершинах заполнить случайными числами. Написать процедуры для вычисления размера дерева, высоты дерева, средней высоты дерева, контрольной суммы для дерева и проверить их работу на конкретном примере. Запрограммировать обход двоичного дерева слева направо и вывести на экран получившуюся последовательность данных.



10. ДЕРЕВЬЯ ПОИСКА

10.1 Поиск в дереве

Двоичные деревья часто употребляются для представления множества данных, среди которых идет поиск элементов по уникальному ключу. Будем считать, что

- 1) часть данных, хранящихся в каждой вершине дерева, является ключом для поиска.
- 2) Для всех ключей определены операции сравнения $<$, $>$, $=$.
- 3) В дереве нет элементов с одинаковыми ключами.

Двоичное дерево называется *деревом поиска*, если ключ в каждой его вершине больше ключа любой вершины в левом поддереве и меньше ключа любой вершины в правом поддереве. Пример такого дерева приведен на рисунке.

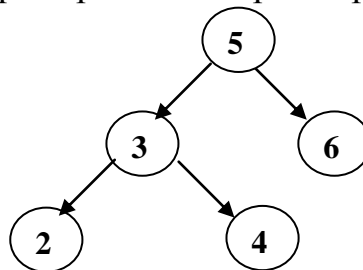


Рисунок 29 Дерево поиска

Чтобы определить является ли двоичное дерево деревом поиска приведем описание на псевдокоде следующей функции. Функция возвращает значение ИСТИНА в случае, если дерево является деревом поиска, и значение ЛОЖЬ в противном случае.

Алгоритм на псевдокоде

Дерево поиска(p: pVertex)

Дерево поиска = ИСТИНА

IF (p \neq NIL и ((p \rightarrow Left \neq NIL и (p \rightarrow Data \leq p \rightarrow Left \rightarrow Data или не Дерево поиска (p \rightarrow Left))) или (p \rightarrow Right \neq NIL и (p \rightarrow Data \geq p \rightarrow Right \rightarrow Data или не Дерево поиска (p \rightarrow Right))))))

Дерево поиска := ЛОЖЬ

FI

В основном деревья поиска используются для организации быстрого и удобного поиска элемента с заданным ключом во множестве данных, которое динамически изменяется. Приведенная ниже процедура поиска элемента в дереве поиска возвращает указатель на вершину с заданным ключом, в противном случае возвращаемое значение равно пустому указателю.

Алгоритм на псевдокоде

Поиск вершины с ключом X

p: = Root

DO (p \neq NIL)

IF (p \rightarrow Data $<$ x) p: = p \rightarrow Right

```

ELSEIF (p→Data > x) p: = p→Left
ELSE OD { p→Data = x }
OD

```

```

IF (p ≠ NIL) <вершина найдена>
ELSE <вершина не найдена>

```

Нетрудно видеть, что максимальное количество сравнений при поиске равно $C_{\max} = 2h$, где h высота дерева.

10.2 Идеально сбалансированное дерево поиска

Двоичное дерево называется *идеально сбалансированным* (ИСД), если для каждой его вершины размеры левого и правого поддеревьев отличаются не более чем на 1.

На рисунке приведены примеры деревьев, одно из которых является идеально сбалансированным, а другое – нет.

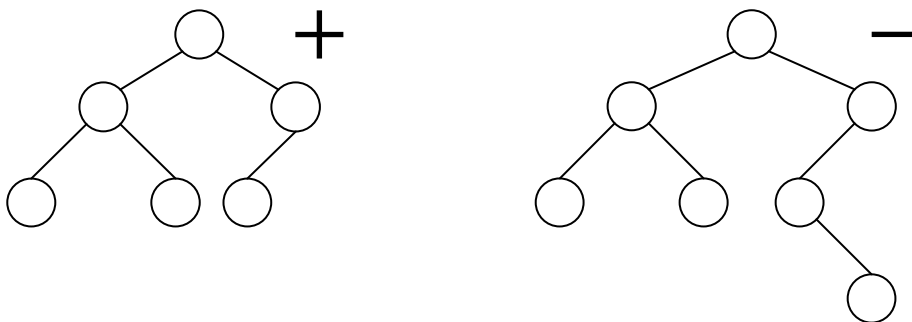


Рисунок 30 Примеры ИСД и неИСД

Отметим некоторые свойства идеально сбалансированного дерева.

Свойство 1. *Высота ИСД с n вершинами минимальна и равна*

$$h_{ИСД(n)} = h_{\min}(n) = \lceil \log(n+1) \rceil.$$

Свойство 2. *Если дерево идеально сбалансировано, то все его поддеревья также идеально сбалансированы.*

Задача построения идеально сбалансированного дерева поиска из элементов массива $A = (a_1, a_2, \dots, a_n)$ решается в два этапа:

1. Сортировка массива A .
2. В качестве корня дерева возьмем средний элемент отсортированного массива, из меньших элементов массива строим левое поддерево, из больших – правое поддерево. Далее процесс построения продолжается до тех пор, пока левое или правое поддерево не станет пустым.

Пример. Построить ИСДП из элементов массива A . Пусть $n = 16$, а элементы массива это числа в 16-ричной системе счисления.

A: B 9 2 4 1 7 E F A D C 3 5 8 6

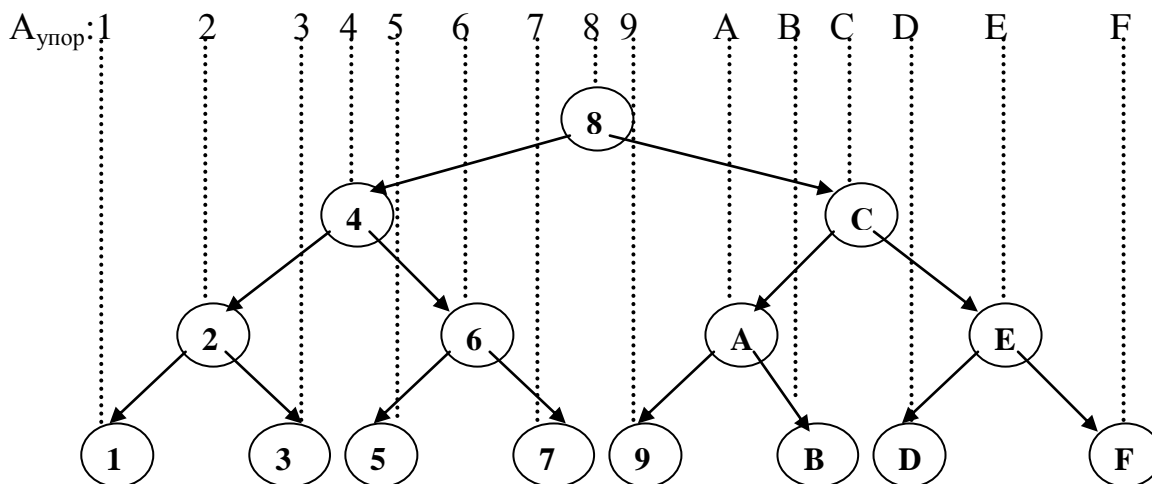


Рисунок 31 Построение ИСДП

Приведем на псевдокоде алгоритм построения ИСДП. Функция ИСДП (L, R) возвращает указатель на построенное дерево, где L, R – левая и правая границы той части массива, из элементов которой строится дерево.

Алгоритм на псевдокоде ИСДП (L, R)

```

IF (L > R) ИСДП:=NIL
ELSE m := [(L+R) / 2]
    <Выделяем память для p>
    p → Data := A[m]
    p → Left := ИСДП ( L, m-1)
    p → Right := ИСДП ( m+1, R)
    ИСДП:= p

```

FI

Для идеально сбалансированного дерева $C_{\max} = 2^{\lceil \log(n+1) \rceil}$. Если считать, что поиск любой вершины происходит одинаково часто, то ИСДП обеспечивает минимальное среднее время поиска. К существенным недостаткам ИСДП можно отнести то, что при добавлении нового элемента к множеству данных необходимо строить заново ИСДП.

10.3 Варианты заданий

1. Написать процедуру, определяющую является ли двоичное дерево деревом поиска. Проверить ее работу на двоичном дереве из предыдущего задания.
2. Запрограммировать процедуру поиска в дереве поиска элемента с заданным ключом и проверить ее работу на конкретном примере.
3. Определить количество операций, необходимых для поиска. Сравнить эту величину с высотой дерева.
4. Определить, что данное дерево является деревом поиска с помощью процедуры обхода дерева слева направо.

5. Определить, что данное дерево является деревом поиска с помощью процедур обхода дерева сверху вниз и снизу вверх.
6. Написать процедуру построения ИСДП из n элементов. Определить высоту построенного ИСДП и сравнить с теоретической оценкой.
7. Сделать поиск элемента в ИСДП. Сравнить количество операций, необходимых для поиска с высотой построенного ИСДП.
8. Графически изобразить ИСДП.

11. СЛУЧАЙНОЕ ДЕРЕВО ПОИСКА

11.1 Определение случайного дерева поиска

При решении многих типов задач объем данных заранее неизвестен, но необходима такая структура данных, для которой достаточно быстро выполняются операции поиска, добавления и удаления вершин. Одно из решений этой проблемы построение *случайного дерева поиска* (СДП). При построении СДП данные поступают последовательно в произвольном порядке и добавление нового элемента происходит в уже имеющееся дерево.

Пример случайного дерева поиска для последовательности D:

В 9 2 4 1 7 E F A D C 3 5 8 6

приведен на рисунке 32.

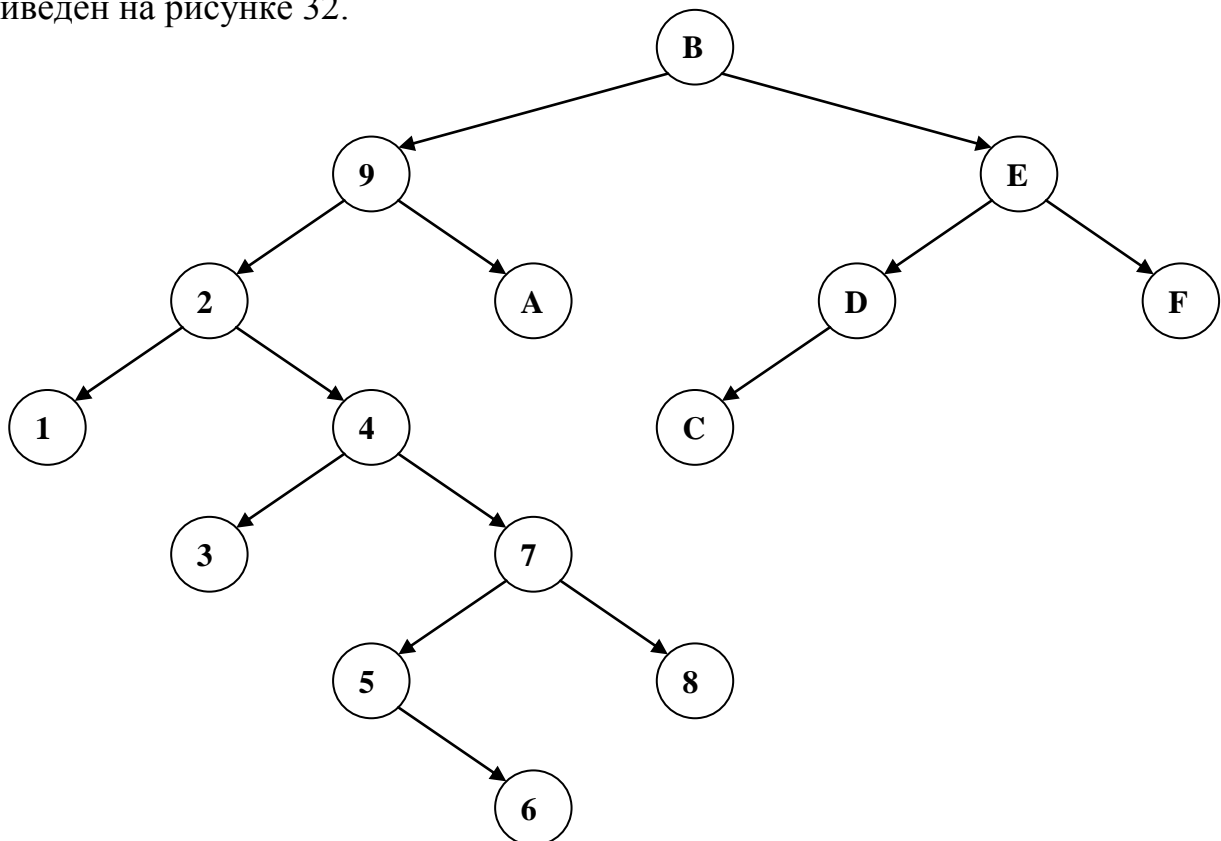
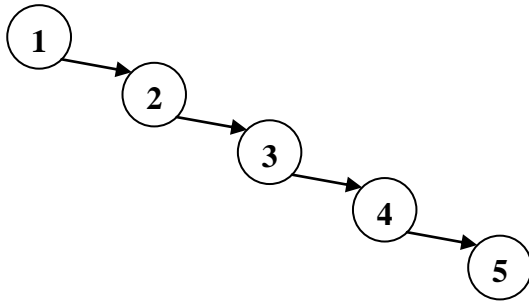


Рисунок 32 Случайное дерево поиска

Это дерево не является ИСДП. В худшем случае СДП может вырождаться в список.

Пример. 1) D: 1 2 3 4 5



2) D: 5 1 2 4 3

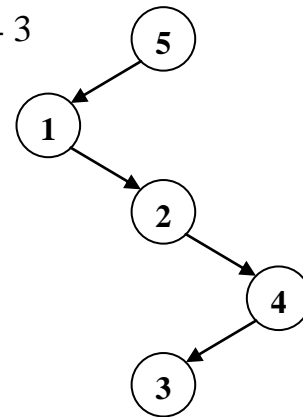


Рисунок 33 Плохие СДП

Таким образом, средняя высота случайного дерева поиска может изменяться в пределах $\log n < h_{\text{ср сд}} < n$ при больших n . В [1] показано, что

$$\lim_{n \rightarrow \infty} (h_{\text{ср сд}} / h_{\text{ср исдп}}) = 2 \ln 2 = 1,386... \text{ и } h_{\text{ср сд}} = O(\log n) \text{ при } n \rightarrow \infty.$$

11.2 Добавление вершины в дерево

Алгоритм добавления вершины в СДП заключается в следующем. Если дерево пустое, то создается корневая вершина, в которую записываются данные. В противном случае вершина добавляется к левому или правому поддереву в зависимости от результата сравнения с данными в текущей вершине.

При создании новой вершины нужно будет изменить значение указателя на нее, поэтому нам понадобится указатель на указатель (двойная косвенность). На языках высокого уровня двойная косвенность обычно может быть реализована с помощью ссылки на указатель.

```
type pVertex = ^tVertex;
var p: ^pVertex;
```

Алгоритм на псевдокоде

Добавление в СДП ($D, *Root$)

Описание переменных: $Root$ – указатель на корень дерева.
 D – данные, которые необходимо добавить.
 P – указатель на указатель на вершину дерева.

```
p := @Root
DO (*p ≠ NIL)
  IF (D < (*p)→Data) p := @((*p)→Left)
  ELSEIF (D > (*p)→Data) p := @((*p)→Right)
  ELSE OD {данные с ключом D уже есть в дереве}
OD
IF (*p = NIL)
  new(*p), (*p)→Data := D,
  (*p)→Right := NIL, (*p)→Left := NIL
FI
```


Пример. Построение дерева для последовательности В 9

1). $D = B$

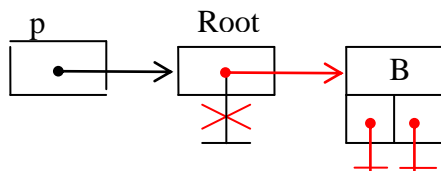


Рисунок 34 Добавление вершины В

2). $D = 9$

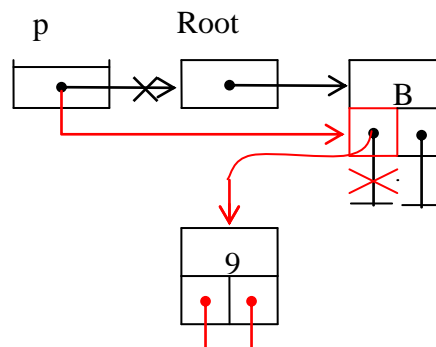


Рисунок 35 Добавление вершины 9

Нетрудно видеть, что трудоемкость добавления вершины в случайное дерево поиска сравнима по порядку величины с трудоемкостью поиска в двоичном дереве, т.е. $C_{cp} = O(\log n)$, $n \rightarrow \infty$.

11.3 Удаление вершины из дерева

Алгоритм удаления вершины с ключом равным X из случайного дерева поиска состоит в следующем. Сначала нужно найти вершину с ключом равным X . Если найденная вершина имеет не более одного поддерева, то её просто удаляем. На рисунке 36 показаны различные варианты расположения вершин. Удаляемая вершина выделена черным цветом.

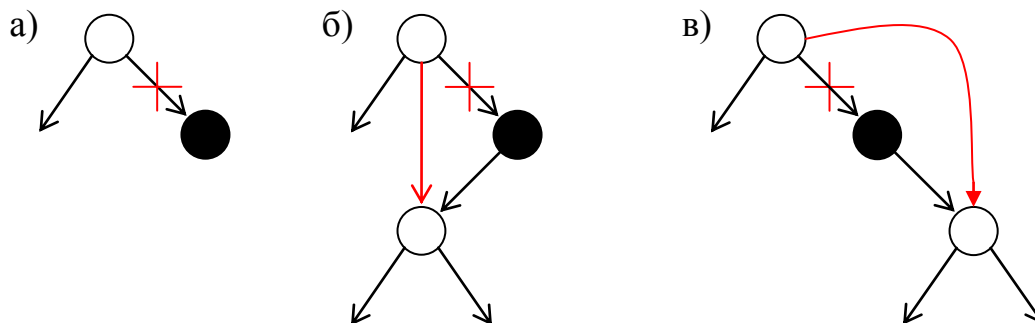


Рисунок 36 Варианты удаления вершин

Если найденная вершина имеет два поддерева (Рисунок 37), то тогда порядок действий следующий. На место удаляемой вершины ставится наибольшая вершина из левого поддерева (наименьшая из правого поддерева), т.е. самая правая вершина левого поддерева (самая левая из правого поддерева), которая не имеет правого поддерева (левого поддерева) (Рисунок 38).

$$r \rightarrow \text{Right} := q \rightarrow \text{Right} \quad 3)$$

$$*p := r \quad 4)$$

FI
dispose(q)
FI

Трудоёмкость удаления вершины складывается из конечного количества операций сравнения и присваивания на каждом шаге поиска в дереве. Таким образом, трудоёмкость удаления такая же, как и в случае добавления в случайное дерево поиска.

11.4 Варианты заданий

1. Написать процедуру включения нового элемента в случайное дерево поиска.
2. Определить необходимое количество операций для включения элемента в дерево. Сравнить эту величину с высотой дерева.
3. Написать процедуру исключения элемента с заданным ключом из случайного дерева поиска.
4. Определить необходимое количество операций для исключения элемента из дерева. Сравнить эту величину с высотой дерева.
5. Построить случайное дерево поиска для данных, которые поступают в случайном порядке. Найти высоту построенного дерева и сравнить с теоретическими оценками.
6. Проверить, является ли построенное случайно дерево деревом поиска с помощью логической функции.
7. Сделать обход слева направо для построенного случайного дерева. Подтвердить, что оно является деревом поиска.
8. Написать процедуру графического изображения СДП.
9. Построить случайное дерево поиска для данных, которые поступают в возрастающем (убывающем) порядке. Найти высоту построенного дерева и сравнить с теоретическими оценками.
10. Построить случайное дерево поиска и ИСДП для одних и тех же данных. Определить высоты построенных деревьев и найти отношение высот. Сравнить полученную величину с теоретической оценкой.

12. СБАЛАНСИРОВАННЫЕ ПО ВЫСОТЕ ДЕРЕВЬЯ (АВЛ-ДЕРЕВЬЯ)

12.1 Определение и свойства АВЛ-дерева

Как было показано выше, ИСДП обеспечивает минимальное среднее время поиска. Однако перестройка дерева после случайного включения вершины – довольно сложная операция. СДП дает среднее время поиска на 40 % больше, но процедура построения достаточно проста. Возможное промежуточное решение – введение менее строгого определения сбалансированности. Одно из таких определений было предложено Г. М. Адельсон – Вельским и Е. М. Ландисом (1962).

Дерево поиска называется *сбалансированным по высоте*, или *АВЛ – деревом*, если для каждой его вершины высоты левого и правого поддеревьев отличаются

не более чем на 1.

На рисунке 39 приведены примеры деревьев, одно из которых является AVL-деревом, а другое – нет. В выделенной вершине нарушается баланс высот левого и правого поддеревьев.

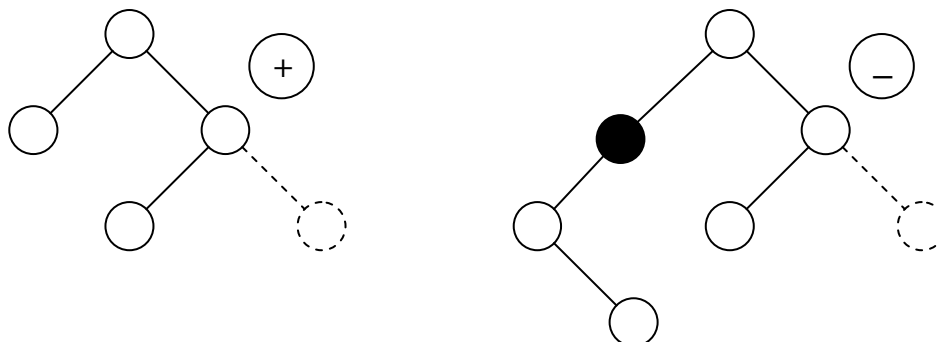


Рисунок 39 Пример AVL-дерева и не AVL-дерева

Заметим, что ИСДП является также и AVL – деревом. Обратное утверждение не верно.

Адельсон – Вельский и Ландис доказали теорему, гарантирующую, что AVL-дерево никогда не будет в среднем по высоте превышать ИСДП более, чем на 45% независимо от количества вершин:

$$\log(n+1) \leq h_{\text{AVL}}(n) < 1,44 \log(n+2) - 0,328 \text{ при } n \rightarrow \infty.$$

Таким образом, лучший случай сбалансированного по высоте дерева – ИСДП, худший случай – плохое AVL – дерево. *Плохое AVL – дерево* это AVL-дерево, которое имеет наименьшее число вершин при фиксированной высоте. Рассмотрим процесс построения плохого AVL-дерева. Возьмём фиксированную высоту h и построим AVL – дерево с минимальным количеством вершин. Обозначим такое дерево через T_h . Ясно, что T_0 – пустое дерево, T_1 – дерево с одной вершиной. Для построения T_h при $h > 1$ будем брать корень и два поддерева с минимальным количеством вершин.

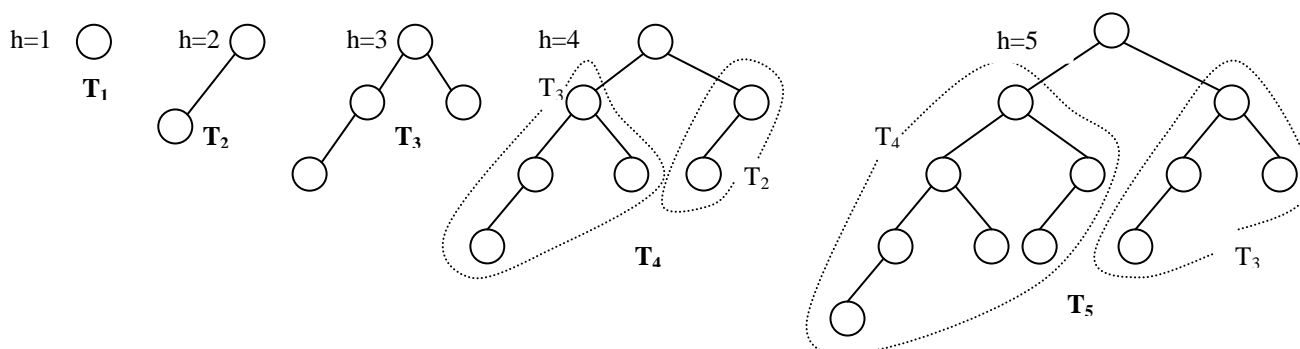


Рисунок 40 Деревья Фибоначчи

Одно поддерево должно быть высотой $h-1$, а другое высотой $h-2$. Поскольку принцип их построения очень напоминает построение чисел Фибоначчи, то такие деревья называют *деревьями Фибоначчи*: $T_h = \langle T_{h-1}, x, T_{h-2} \rangle$. Число вершин в T_h определяется следующим образом:

$$n_0 = 0, n_1 = 1, n_h = n_{h-1} + 1 + n_{h-2}$$

12.2 Повороты при балансировке

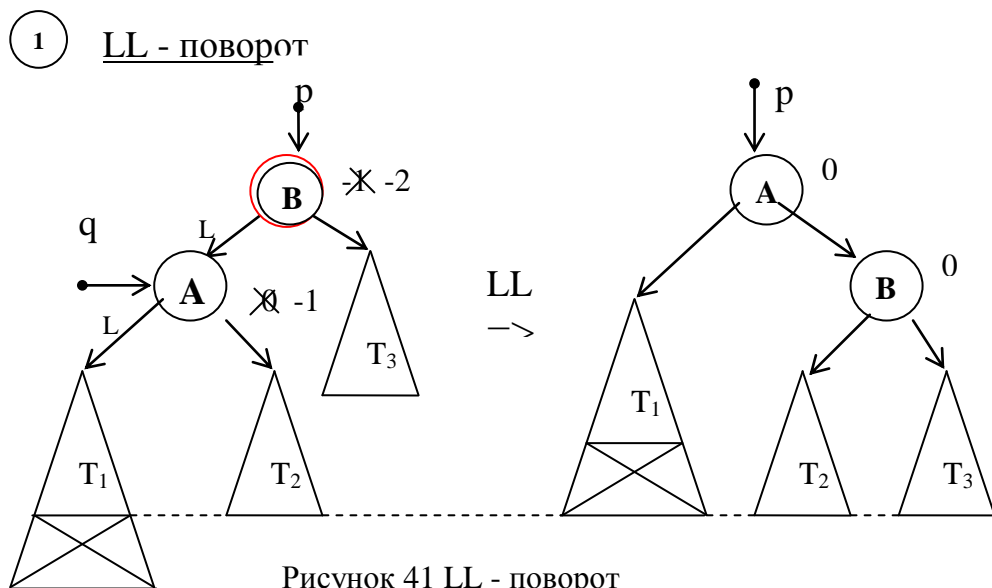
Рассмотрим, что может произойти при включении новой вершины в сбалансированное по высоте дерево. Пусть r – корень АВЛ-дерева, у которого имеется левое поддерево (T_L) и правое поддерево (T_R). Если добавление новой вершины в левое поддерево приведет к увеличению его высоты на 1, то возможны три случая:

- 1) если $h_L = h_R$, то T_L и T_R станут разной высоты, но баланс не будет нарушен;
- 2) если $h_L < h_R$, то T_L и T_R станут равной высоты, т. е. баланс даже улучшится;
- 3) если $h_L > h_R$, то баланс нарушится и дерево необходимо перестраивать.

Введём в каждую вершину дополнительный параметр Balance (показатель баланса), принимающий следующие значения:

- 1, если левое поддерево на единицу выше правого;
- 0, если высоты обоих поддеревьев одинаковы;
- 1, если правое поддерево на единицу выше левого.

Если в какой-либо вершине баланс высот нарушается, то необходимо так перестроить имеющееся дерево, чтобы восстановить баланс в каждой вершине. Для восстановления баланса будем использовать процедуры поворотов АВЛ-дерева.



Алгоритм на псевдокоде

LL - поворот

```

q := p → Left
q → Balance := 0
p → Balance := 0
p → Left := q → Right
q → Right := p
p := q

```

2

LR – поворот

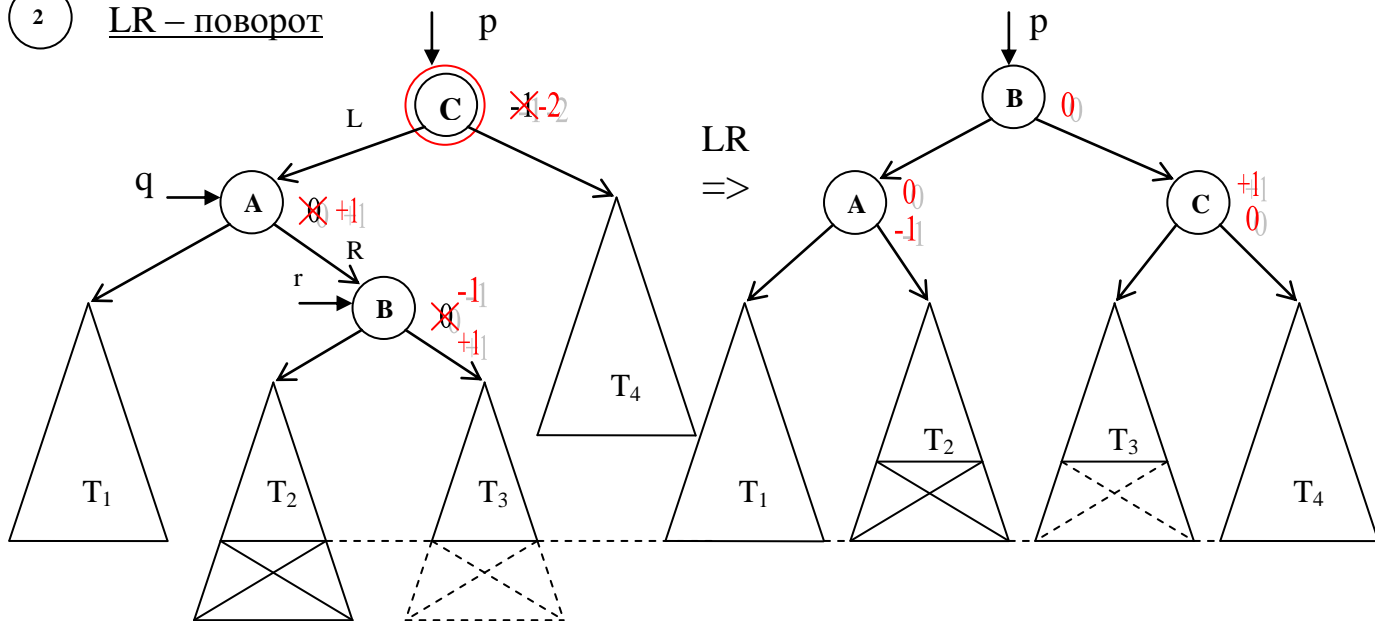


Рисунок 42 LR – поворот

Алгоритм на псевдокоде

LR - поворот

$q := p \rightarrow \text{Left}, r := q \rightarrow \text{Right}$

IF ($r \rightarrow \text{Balance} < 0$) $p \rightarrow \text{Balance} := +1$ ELSE $p \rightarrow \text{Balance} := 0$ FI

IF ($r \rightarrow \text{Balance} > 0$) $q \rightarrow \text{Balance} := -1$ ELSE $q \rightarrow \text{Balance} := 0$ FI

$r \rightarrow \text{Balance} := 0$

$p \rightarrow \text{Left} := r \rightarrow \text{Right}, q \rightarrow \text{Right} := r \rightarrow \text{Left}$

$r \rightarrow \text{Left} := q, r \rightarrow \text{Right} := p, p := r$

3

RR – поворот

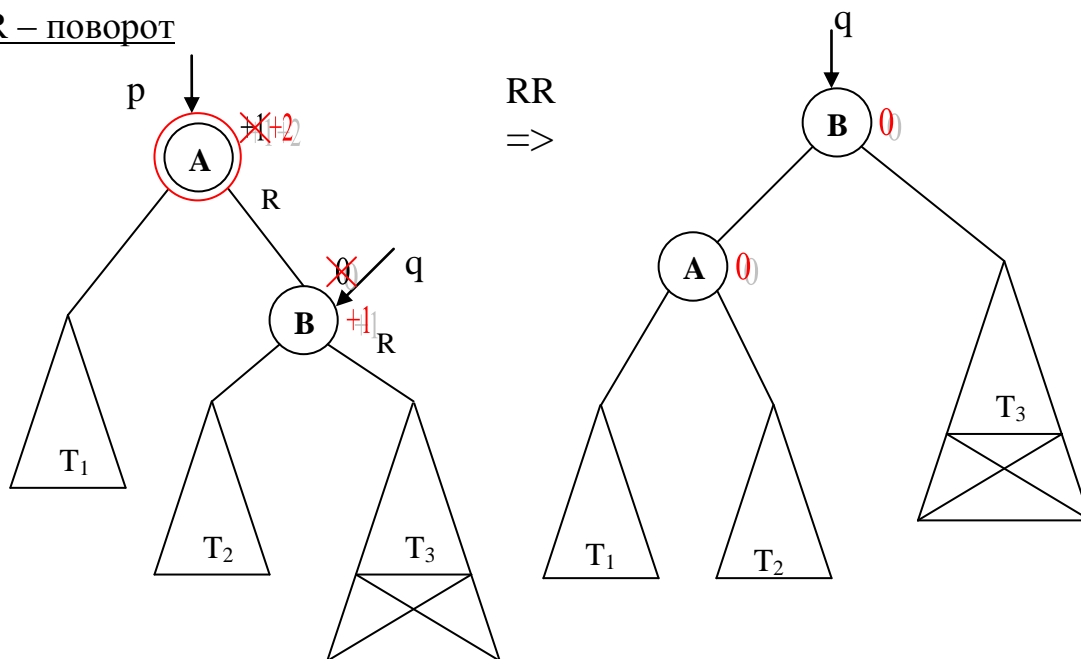


Рисунок 43 RR – поворот

Алгоритм на псевдокоде RR - поворот

```

q := p → Right
q → Balance := 0
p → Balance := 0
p → Right := q → Left
q → Left := p
p := q
4  RL – поворот

```

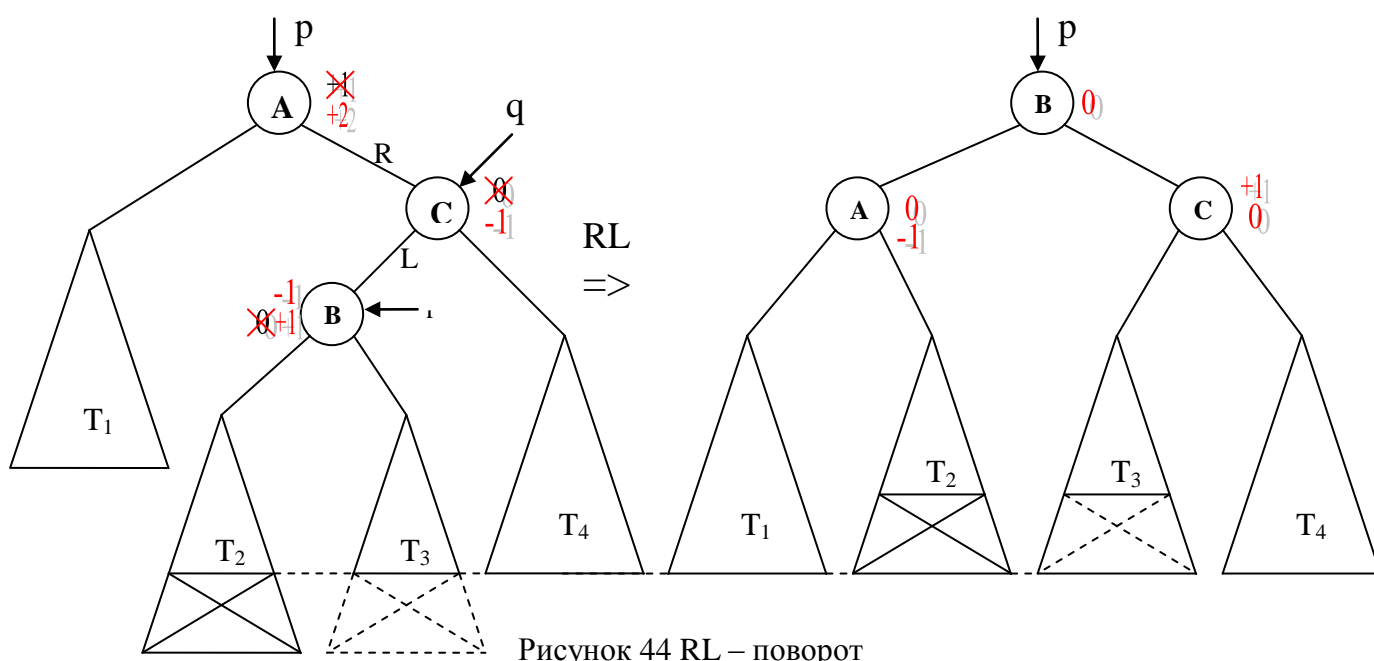


Рисунок 44 RL – поворот

Алгоритм на псевдокоде RL - поворот

```

q := p → Right, r := q → Left
IF (r → Balance > 0) p → Balance := -1 ELSE p → Balance := 0 FI
IF (r → Balance < 0) q → Balance := 1 ELSE q → Balance := 0 FI
r → Balance := 0
p → Right := r → Left, q → Left := r → Right
r → Left := p, r → Right := q, p := r

```

12.3 Добавление вершины в дерево

Добавление новой вершины в АВЛ-дерево происходит следующим образом. Вначале добавим новую вершину в дерево так же как в случайное дерево поиска (проход по пути поиска до нужного места). Затем, двигаясь назад по пути поиска от новой вершины к корню дерева, будем искать вершину, в которой нарушился баланс (т. е. высоты левого и правого поддеревьев стали отличаться более чем на 1). Если такая вершина найдена, то изменим структуру дерева для восстановления баланса с помощью процедур поворотов.

Алгоритм на псевдокоде

Добавление в АВЛ – дерево (D : данные; $Var p: pVertex$);

Обозначим

Рост – логическая переменная, которая показывает выросло дерево или нет.

IF ($p = NIL$)

new(p), $p \rightarrow Data := D$, $p \rightarrow Left := NIL$, $p \rightarrow Right := NIL$

$p \rightarrow Balance := 0$, Рост := ИСТИНА

ELSE

IF ($p \rightarrow Data > D$)

Добавление в АВЛ – дерево (D , $p \rightarrow Left$)

IF (Рост = ИСТИНА) {выросла левая ветвь}

IF ($p \rightarrow Balance > 0$) $p \rightarrow Balance := 0$, Рост := ЛОЖЬ

ELSE IF ($p \rightarrow Balance = 0$) $p \rightarrow Balance := -1$

ELSE

IF ($p \rightarrow Left \rightarrow Balance < 0$) <LL – поворот>

ELSE <LR – поворот> Рост := ЛОЖЬ

FI

FI

ELSE IF ($p \rightarrow Data < D$)

<аналогичные действия для правого поддерева>

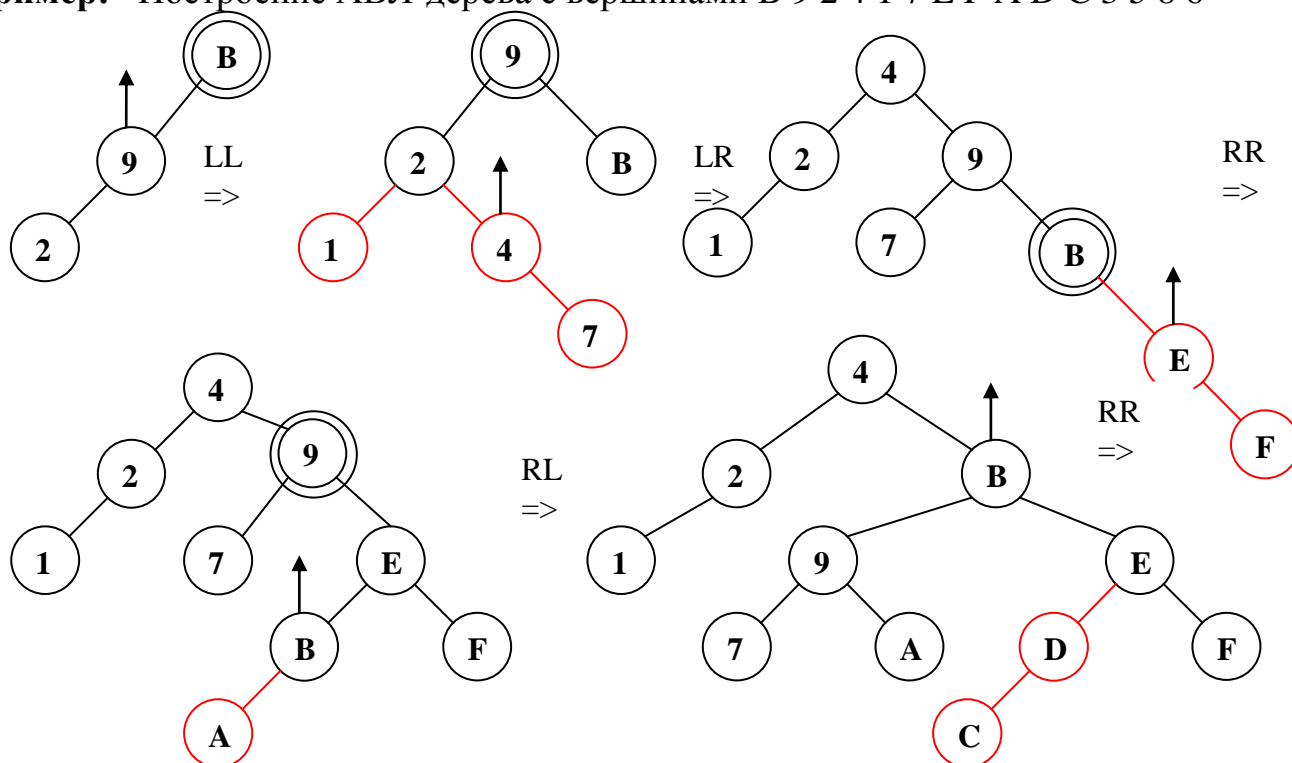
ELSE { $p \rightarrow Data = D$, такая вершина уже есть}

FI

FI

FI

Пример: Построение АВЛ-дерева с вершинами В 9 2 4 1 7 Е F А D С 3 5 8 6



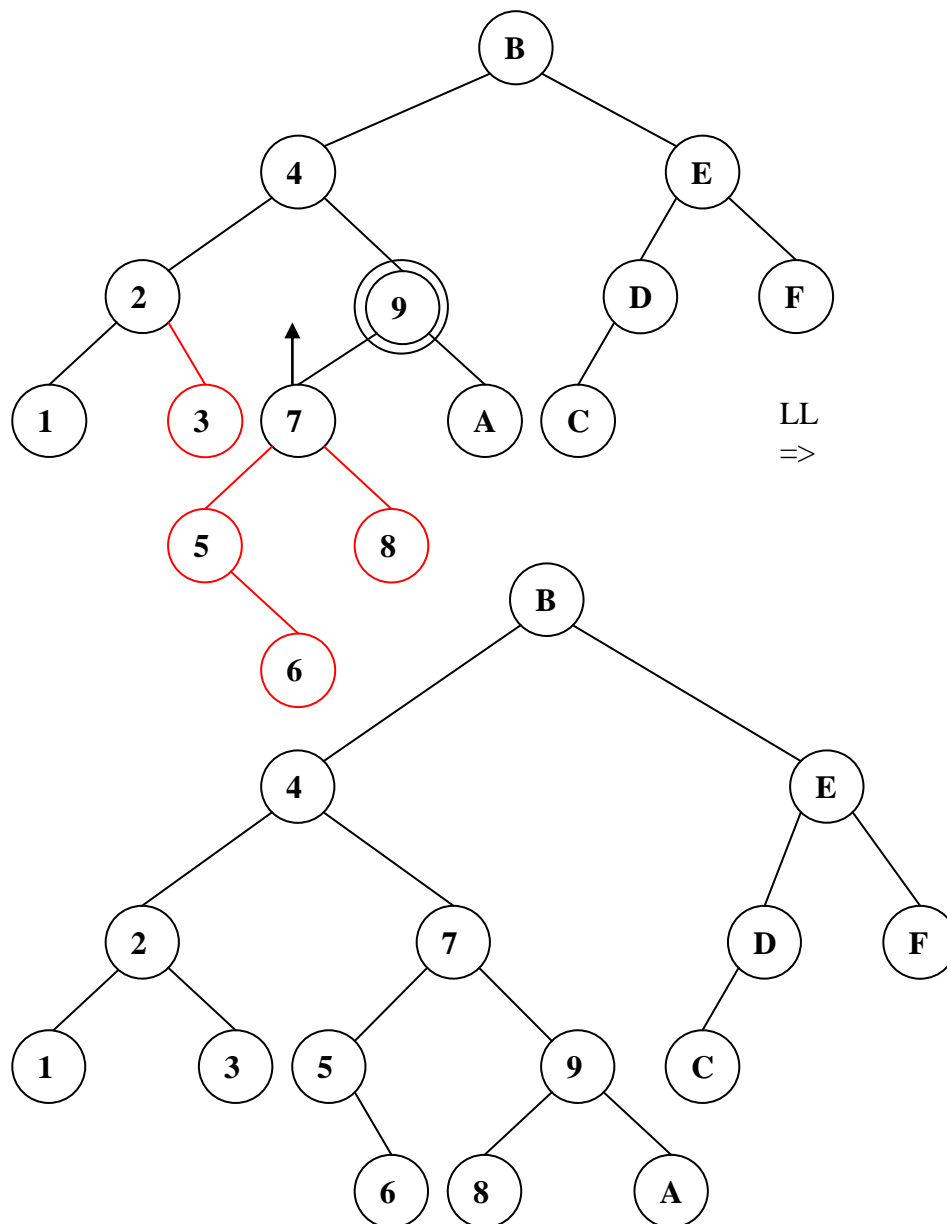


Рисунок 45 Построение АВЛ-дерева

12.4 Удаление вершины из дерева

Очевидно, удаление вершины – процесс намного более сложный, чем добавление. Хотя алгоритм операции балансировки остаётся тем же самым, что и при включении вершины. Балансировка по-прежнему выполняется с помощью одного из четырёх уже рассмотренных поворотов вершин.

Удаление из АВЛ-дерева происходит следующим образом. Удалим вершину так же, как это делалось для СДП. Затем двигаясь назад от удалённой вершины к корню дерева, будем восстанавливать баланс в каждой вершине (с помощью поворотов). При этом нарушение баланса возможно в нескольких вершинах в отличие от операции включения вершины в дерево.

Как и в случае добавления вершин, введём логическую переменную *Уменьшение*, показывающую уменьшилась ли высота поддерева. Балансировка идёт, только если *Уменьшение* = ИСТИНА. Это значение присваивается переменной *Уменьшение*, если обнаружена и удалена вершина или высота поддерева умень-

шилась в процессе балансировки.

Введём две симметричные процедуры балансировки, т. к. они будут использоваться несколько раз в алгоритме удаления:

BL – используется при уменьшении высоты левого поддерева,

BR – используется при уменьшении высоты правого поддерева.

Рисунки 46 и 47 иллюстрируют три случая, возникающие при удалении вершины из левого (для BL) или правого (для BR) поддерева, в зависимости от исходного состояния баланса в вершине по адресу p.

Алгоритм на псевдокоде

BL (p: pVertex, Уменьшение: boolean)

IF (p→Bal = -1) p→Bal := 0

ELSEIF (p→Bal = 0) p→Bal := 1, Уменьшение := ЛОЖЬ

ELSEIF (p→Bal = 1)

IF (p→Left→Bal ≥ 0) <RR1-поворот>

ELSE <RL - поворот> FI

FI

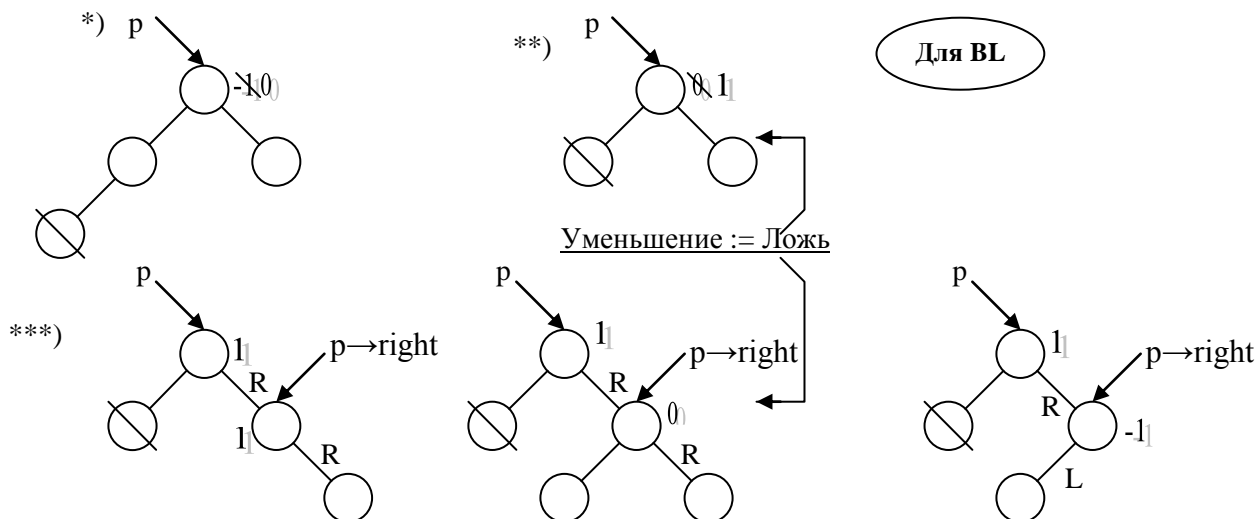


Рисунок 46 Три случая при удалении вершины из левого (для BL) поддерева

Алгоритм на псевдокоде

BR (p: pVertex, Уменьшение: boolean)

IF (p→Bal = 1) p→Bal := 0

ELSEIF (p→Bal = 0) p→Bal := -1, Уменьш := ЛОЖЬ

ELSEIF (p→Bal = -1)

IF (p→Left→Bal ≤ 0) <LL1 - поворот>

ELSE <LR - поворот> FI

FI

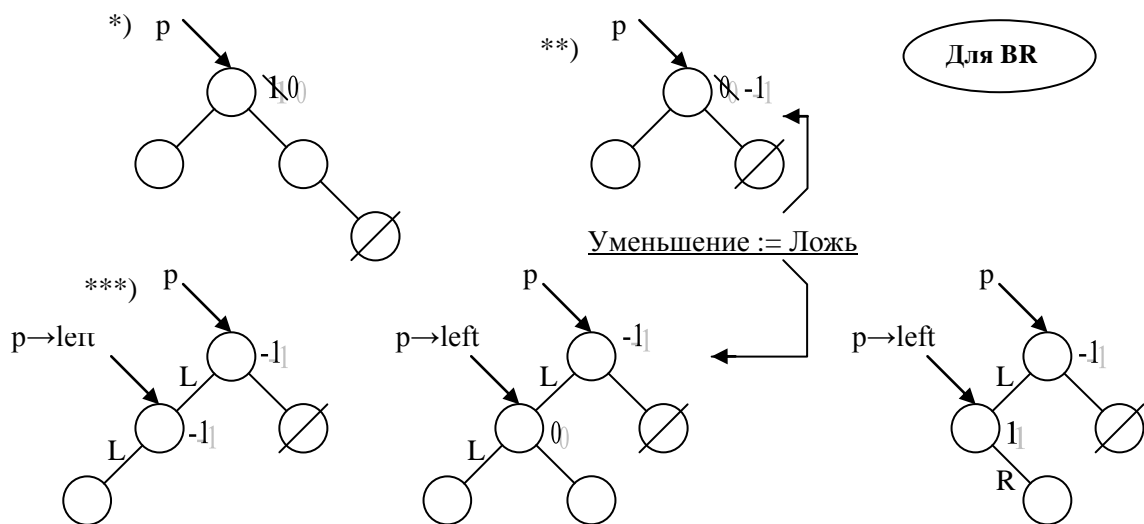


Рисунок 47 Три случая при удалении вершины правого (для BR) поддерева

При добавлении вершины не может быть случая, когда $p \rightarrow \text{left} \rightarrow \text{Bal} = 0$, поэтому LL – поворот необходимо изменить, чтобы учесть эту ситуацию.

Алгоритм на псевдокоде

LL1 – поворот

```

q := p → Left
IF (q → Bal = 0) p → Bal := -1, q → Bal := 1, Уменьш := false
ELSE p → Bal := 0, q → Bal := 0
p → Left := q → Right
q → Right := p
p := q

```

Аналогично изменяется RR – поворот, LR и RL – повороты не изменяются.

Алгоритм на псевдокоде

RR1 – поворот

```

q := p → Right
IF (q → Bal = 0) p → Bal := 1, q → Bal := -1, Уменьшение := ЛОЖЬ
ELSE p → Bal := 0, q → Bal := 0
FI
p → Right := q → Left
q → Left := p
p := q

```

Алгоритм на псевдокоде

Удаление из AVL-дерева (x: Данные, p: pVertex, Уменьшение: boolean)

```

IF (p = NIL) {ключа в дереве нет}
ELSE IF (p → Data > x) Удаление (x, p → Left, Уменьшение)
    IF Уменьшение BL (p, Уменьш) FI
ELSE IF (p → Data < x) Удаление (x, p → Right, Уменьшение)
    IF Уменьшение BR (p, Уменьшение) FI
ELSE IF {удаление вершины по адресу p}

```

```

q := p
IF (q→Right = NIL) p := q→Left, Уменьшение := ИСТИНА
ELSE IF (q→Left = NIL) p := q→Right, Уменьшение := ИСТИНА
ELSE del (q→Left, Уменьшение)
      IF Уменьшение BL (p, Уменьшение) FI
FI
dispose(q)

```

FI

Используемая при удалении процедура del удаляет вершину, имеющую 2 поддерева, т. е. заменяет её на самую правую вершину из левого поддерева.

Алгоритм на псевдокоде

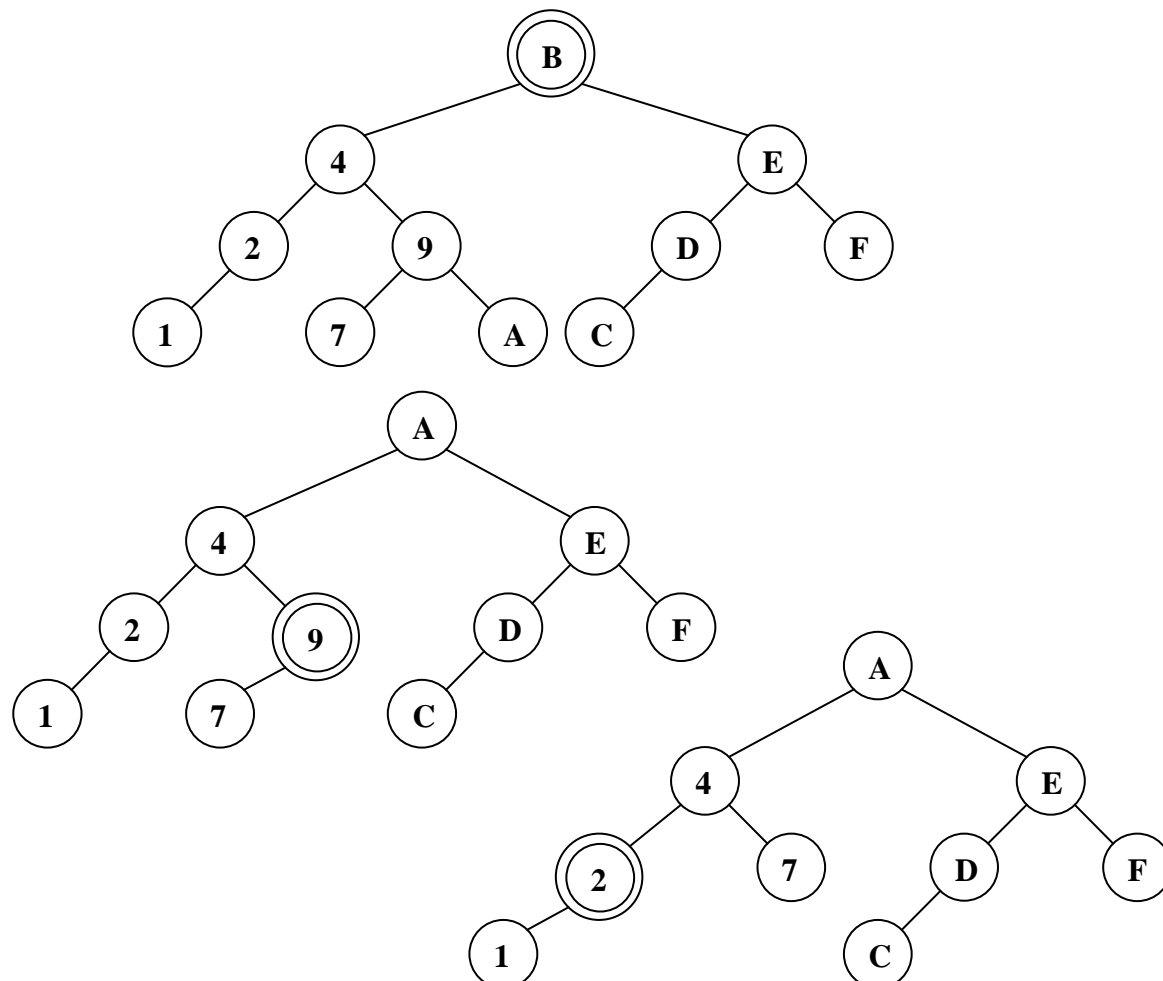
del (r: pVertex, Уменьшение: boolean)

```

IF (r→right ≠ NIL)
  del (r→right, Уменьшение)
  IF Уменьшение BR (r, Уменьшение) FI
ELSE q→Data := r→Data
  q := r
  r := r→Left
  Уменьшение := ИСТИНА
FI

```

Пример: Удаление из АВЛ-дерева вершин В 9 2 4 1 7 Е F



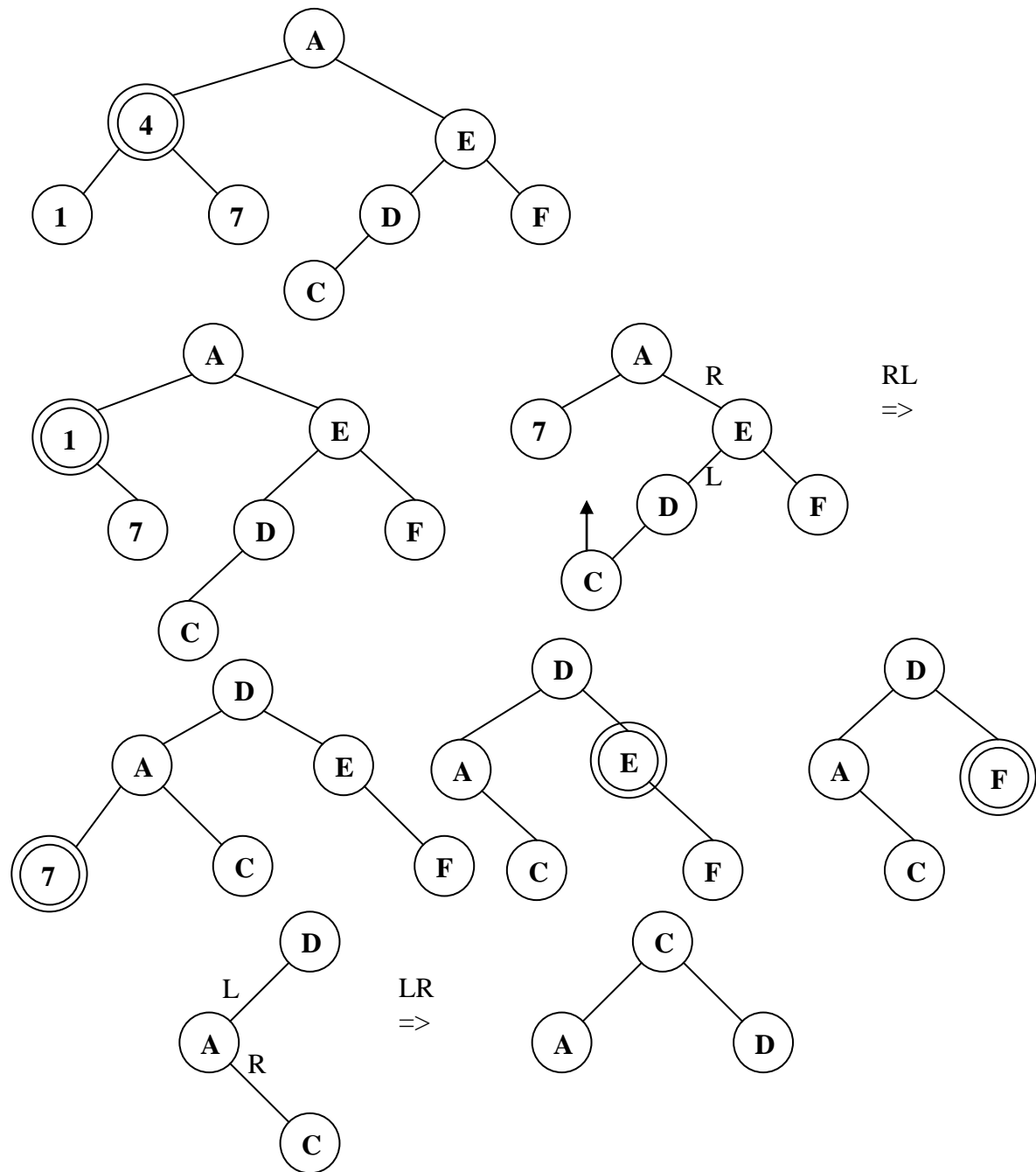


Рисунок 48 Удаление из AVL-дерева

Поиск элемента с заданным ключом, включения нового элемента, удаления элемента – каждое из этих действий в AVL-дереве можно произвести в худшем случае за $O(\log n)$ операций.

Отличие между процедурами включения и удаления заключается в следующем. Включение может привести самое большое к одному повороту, исключение может потребовать поворот во всех вершинах вдоль пути поиска. Наихудшим случаем с точки зрения количества балансировок является удаление самой правой вершины у плохого AVL-дерева (дерева Фибоначчи). По экспериментальным оценкам на каждые два включения встречается один поворот, а при исключении поворот происходит в одном случае из пяти.

12.5 Варианты заданий

1. Написать процедуру, которая определяет, является ли двоичное дерево AVL-деревом. Проверить правильность работы процедуры на различных двоичных деревьях.
2. Написать процедуру для построения деревьев Фибоначчи заданной высоты.
3. Экспериментально определить среднюю длину пути в дереве Фибоначчи.
4. Запрограммировать процедуру добавления новой вершины в AVL-дерево. Определить количество необходимых операций для добавления вершины.
5. Запрограммировать процедуру удаления вершины из AVL-дерева. Определить количество необходимых операций для удаления вершины.
6. Экспериментально сравнить AVL-дерево и ИСДП по высоте.
7. Экспериментально сравнить высоты AVL-дерева и случайного дерева поиска.
8. Экспериментально определить количество поворотов на одно включение вершины в дерево.
9. Экспериментально определить количество поворотов на одно исключение вершины из дерева.
10. Графически изобразить AVL-дерево.

13. Б-ДЕРЕВЬЯ

13.1 Определение Б-дерева порядка m

Деревья, имеющие вершины со многими потомками, будем называть *сильноветвящимися*. Такие деревья могут быть эффективно использованы для решения следующей задачи: формирование и поддержание крупномасштабных деревьев поиска, для которых необходимы включения и удаление элементов, но для которых либо не хватает оперативной памяти, либо она слишком дорога для длительного использования.

В этом случае вершины дерева могут храниться во внешней памяти (например, на диске), тогда ссылки представляют собой адреса на диске, а не в оперативной памяти. Если использовать двоичное дерево для $n = 10^6$ элементов, то потребуется $\log(10^6) = 20$ шагов поиска. Так как каждый шаг включает в себя обращение к диску, то необходимо минимизировать число обращений к диску. Сильноветвящиеся деревья – идеальное решение этой проблемы, т.к. при обращении к диску можно считывать не один элемент, а целую группу, причём размер сектора диска определяет размер минимальной порции (обычно кратен 512 байт). Таким образом, за одно обращение считывается поддерево, которое будем называть *страницей*. Например, пусть для дерева из 10^6 элементов размер страницы равен 100 вершин. Поиск будет требовать в среднем $\log_{100}(10^6) = 3$, а не 20 обращений к диску. Однако если дерево растёт случайным образом, то в худшем случае может потребоваться даже 10^4 обращений. Поэтому Р. Байером и Е. Маккрейтом был сформулирован критерий управления ростом дерева: каждая страница (кроме одной) должна содержать (при заданном постоянном m) от m до $2m$ элементов.

Таким образом, для дерева с n элементами и максимальным размером в $2m$ вершин в худшем случае потребуется $\log_m n$ обращений к страницам (диску). При

этом коэффициент использования памяти не меньше, чем 50%, так как страницы всегда заполнены минимум наполовину. Также эта схема позволяет достаточно просто осуществлять поиск, включения и удаления элементов.

Введем следующее определение. *Б – дерево порядка m* – это дерево со следующими свойствами:

1. В каждой странице хранится k элементов данных $d_1 < d_2 < \dots < d_k$ и $k+1$ указатель p_0, p_1, \dots, p_k . Каждый указатель p_i либо равен NIL, либо указывает на вершину, все элементы которой больше d_i , но меньше d_{i+1} .

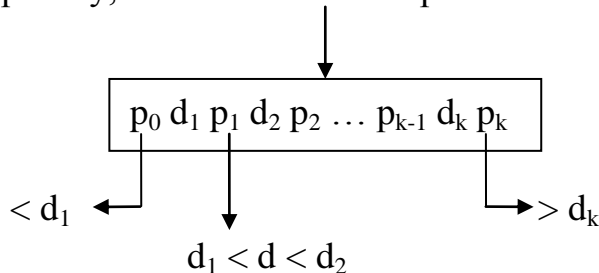


Рисунок 49 Страница Б-дерева

2. Для каждой вершины, кроме корня $m \leq k \leq 2m$, для корня $1 \leq k \leq 2m$.
3. Все листья дерева расположены на одном уровне.

Пример Б-дерева при $m = 2$.

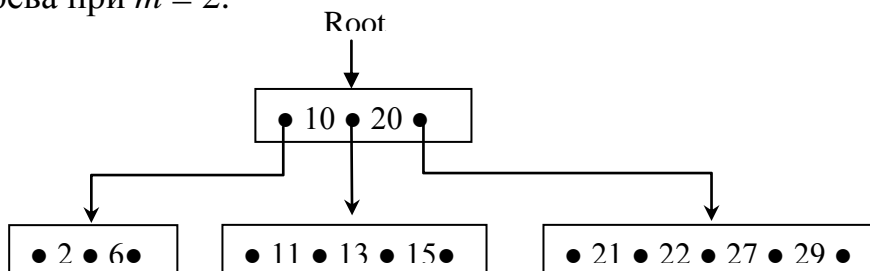


Рисунок 50 Пример Б-дерева

Очевидно, количество обращений к диску равно высоте Б-дерева. Легко видеть, что минимальное количество вершин в Б-дереве при заданной высоте h определяется равенством $n_{\min} = 2(m+1)^{h-1} - 1$. Отсюда высота Б-дерева порядка m с n элементами данных $h = \frac{\log(n+1) - 1}{\log(m+1)} + 1$. Например, при $m=255$ высота h меньше,

чем $\log n/8$, т.е. по сравнению с обычными двоичными деревьями требуется в 8 раз меньше обращений к диску.

13.2 Поиск в Б-дереве

Если спроецировать Б – дерево на один уровень, то элементы расположатся в возрастающем порядке слева направо. Такое размещение определяет способ поиска элемента с заданным ключом. Страница считывается в оперативную память, а затем производится поиск среди элементов d_1, d_2, \dots, d_k (упорядоченных!). Если k мало, то можно использовать простой последовательный поиск, если k достаточно большое, то – двоичный поиск.

Будем считать, что элементы на странице представлены в виде массива. То-

где структура данных может быть следующим образом описана (на Паскале):

```

type pPage = ^Page; {указатель на страницу}
  Item = record {тип элемента}
    Data: integer;
    p: pPage;
  end;
  Page = record {тип страницы}
    k: 0 .. 2*m;
    p0: pPage;
    e: array [1 .. 2*m] of Item;
  end;

```

где m – порядок Б – дерева,

k – количество элементов на странице,

$p0$ – указатель на страницу с элементами $< d_1$,

e – массив элементов на странице.

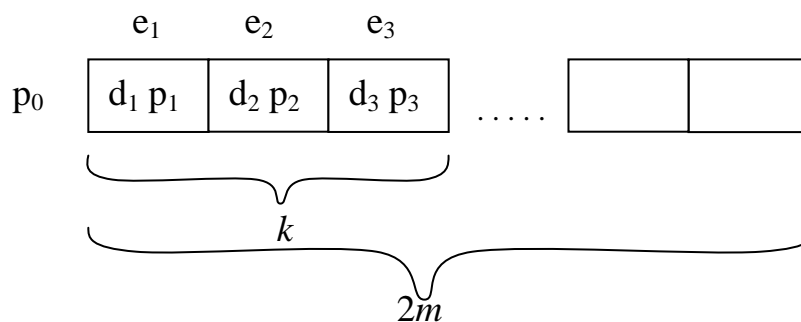


Рисунок 51 Структура страницы Б-дерева

Алгоритм на псевдокоде

Поиск в Б – дереве (D: Данные, a: pPage)

Обозначим L, R левая и правая границы поиска на странице

IF (a = NIL) (Ключа D нет в дереве)

ELSE L := 1, R := k + 1

DO (L < R)

 i := [(L + R)/2]

 IF (a → e_i.data ≤ D) L := i + 1

 ELSE R := i

 FI

OD

R := R – 1

IF (R > 0 и a → e_R.data = D) (ключ D есть в дереве)

ELSE (на этой странице ключа D нет)

 IF (R = 0) Поиск в Б-дереве (D, a → p₀)

 ELSE Поиск в Б-дереве (D, a → e_R.p)

 FI

FI

13.3 Построение Б-дерева

Построение Б-дерева или включение нового элемента данных D в Б-дерево происходит следующим образом:

- Выполним поиск элемента D в дереве.
- Если элемента D нет в дереве, то мы имеем страницу a и позицию R , в которой ожидали найти элемент D .
- Вставим элемент в позицию $R+1$, при этом количество элементов на странице k увеличилось на 1.
- Если $k \leq 2m$, то процесс включения закончен.
- Если $k > 2m$ (переполнение страницы), то создаём новую страницу b , переносим в неё m правых элементов из страницы a , а средний элемент переносим на один уровень вверх на родительскую страницу.
- Включение элемента в родительскую страницу производится по такому же алгоритму.
- Если родительской страницы нет, то она создаётся и в неё включается один элемент.

Эта схема сохраняет все характерные свойства Б-дерева. Получившиеся две новые страницы содержат ровно по m элементов. Включение элемента в родительскую страницу может опять перевести к переполнению, то есть разделение страниц может распространиться до самого корня. В этом случае может увеличиться высота дерева. Таким образом, Б-деревья растут обратно: от листа к корню.

Пример построения Б-дерева при $m = 2$. Предварительно удалим повторяющиеся буквы.



Рисунок 52 Построение Б-дерева

Алгоритм на псевдокоде

Построение Б – дерева (D: Данные, a: pPage, Rost: Boolean, V: Item)

Обозначим D – добавляемые данные

a – адрес страницы

Rost – логическая переменная; (принимает значение ИСТИНА, если дерево выросло).

V – рабочая переменная типа Item для элемента, передаваемого вверх по дереву.

u – рабочая переменная типа Item, фактический параметр при вызове процедуры построения

IF (a = NIL) {D нет в дереве, включение}

V.Data := D

V.p := NIL

Rost := ИСТИНА

ELSE Поиск в Б-дереве (D,a)

IF (R > 0 и $a \rightarrow e_R.data = D$) {элемент D есть в дереве}

ELSE (на этой странице ключа D нет)

IF(R = 0) Построение Б-деревя(D, $a \rightarrow p_0$, Rost, u)

ELSE Построение Б-деревя (D, $a \rightarrow e_R.p$, Rost, u)

FI

IF (Rost = true) {включение u вправо от e_R }

IF (k < 2m) Rost := false {есть место на странице}

k := k+1

DO (i = k, k-1, ..., R+2) $e_i = e_{i-1}$

OD

$e_{R+1} := u$

ELSE {переполнение страницы}

New(b) {создание новой страницы}

IF (R <= m)

IF (R=m) V := u

ELSE V := e_m

DO (i = m, m-1, ..., R+2) $e_i := e_{i-1}$

OD

$e_{R+1} := u$

FI

DO (i:=1, 2, ... m) $b \rightarrow e_i := a \rightarrow e_{i-1}$

OD

ELSE {включение в правую страницу}

R:= R-m

V:= e_{m+1}

DO(i=1, 2, ... R-1) $b \rightarrow e_i := a \rightarrow e_{i+m+1}$

OD

$b \rightarrow e_R := u$

DO (i:=R+1, R+2, ..., m) $b \rightarrow e_i := a \rightarrow e_{i+m}$

OD

```

FI
a → k := m
b → k := m
b → p0 := V.p
V.p := b

```

```

FI

```

```

FI

```

```

FI

```

```

FI

```

При создании Б-дерева необходимо предусмотреть разделение корневой страницы и создания нового корня из одного элемента. Это будет выполняться, если после обращения к процедуре построения Б-дерева с параметром Rost равным ИСТИНА.

Алгоритм на псевдокоде

Разделение корневой страницы

```

Root := NIL
DO <ввести D>
    Построение Б-дерева (D, Root, Rost, u)
    IF (Rost = true) (включение новой корневой страницы)
        q := Root
        new(Root)
        root → k := 1
        root → p0 := q
        root → e1 := u
    FI
OD

```

13.4 Определение двоичного Б-дерева

На первый взгляд кажется, что наименьший интерес представляют Б-деревья первого порядка. Но иногда стоит обращать внимание и на такие исключительные варианты. Очевидно, что Б-дерево первого порядка не имеет смысла использовать для представления больших множеств данных, требующих вторичной памяти, т.к. приблизительно 50% всех страниц будут содержать только один элемент.

Двоичное Б-дерево состоит из вершин (страниц) с одним или двумя элементами. Следовательно, каждая страница содержит две или три ссылки на поддеревья. На рисунке 53 показаны примеры страниц Б – дерева при $m = 1$.

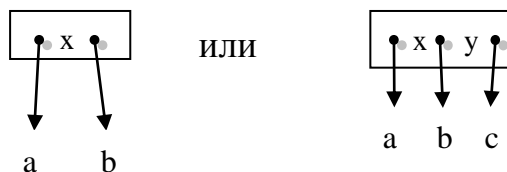


Рисунок 53 Виды вершин ДБД

Поэтому вновь рассмотрим задачу построения деревьев поиска в оперативной памяти компьютера. В этом случае неэффективным с точки зрения экономии памяти будет представление элементов внутри страницы в виде массива. Выход из положения – динамическое размещение на основе списочной структуры, когда внутри страницы существует список из одного или двух элементов.

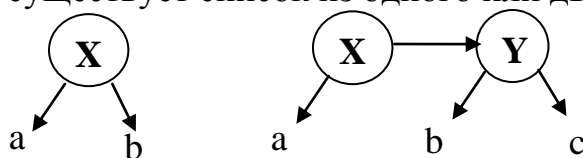


Рисунок 54 Вершины двоичного Б-дерева

Таким образом, страницы Б-дерева теряют свою целостность и элементы списков начинают играть роль вершин в двоичном дереве. Однако остается необходимость делать различия между ссылками на потомков (вертикальными) и ссылками на одном уровне (горизонтальными), а также следить, чтобы все листья были на одном уровне.

Очевидно, двоичные Б-деревья представляют собой альтернативу AVL-деревьям. При этом поиск в двоичном Б-дереве происходит как в обычном двоичном дереве.

Высота двоичного Б-дерева $h = \frac{\log(n+1) - 1}{\log(1+1)} + 1 = \log(n+1)$. Если рассматри-

вать двоичное Б-дерево как обычное двоичное дерево, то его высота может увеличиться вдвое, т.е. $h = 2\log(n+1)$. Для сравнения, в AVL-дереве даже в самом плохом случае $h < 1.44 \log n$. Поэтому сложность поиска в двоичном Б-дереве и в AVL-дереве одинакова по порядку величины.

13.5 Добавление вершины в дерево

Построение двоичного Б-дерева происходит путем добавления новой вершины в уже существующее дерево. Введем логическую переменную VR, показывающую вертикальный рост дерева (в случае, если страница переполнилась и средний элемент передается на вышележащий уровень) и логическую переменную HR, определяющую горизонтальный рост дерева (если новый элемент размещается на этой же условной странице). Также определим показатель баланса BAL для каждой вершины, который принимает значение 0, если у данной вершины есть только вертикальные ссылки (вершина одна на странице), и значение 1, если у данной вершины есть правая горизонтальная ссылка.

При добавлении элементов в двоичное Б-дерево различают 4 ситуации, возникающих при росте левых или правых поддеревьев (см. рис. 55). Самый простой случай (1) — рост правого поддерева вершины A, когда A — единственный элемент на странице. Тогда вертикальная ссылка просто превращается в горизонтальную (HR=1, баланс вершины A равен 1). Если на странице уже два элемента (2), то при добавлении новой вершины C средняя вершина B передается на вышестоящий уровень (VR=1, баланс вершины B равен 0).

В случае роста левого поддерева, если вершина B одна на странице (3), то вершина A добавляется на эту страницу. Однако левая ссылка не может быть го-

ризонтальной, поэтому требуется переопределение ссылок ($HR=1$, баланс вершины A равен 1). Если на странице два элемента, то, как и в случае 2, средняя вершина B поднимается на вышестоящий уровень ($VR=1$, баланс вершины B равен 0).

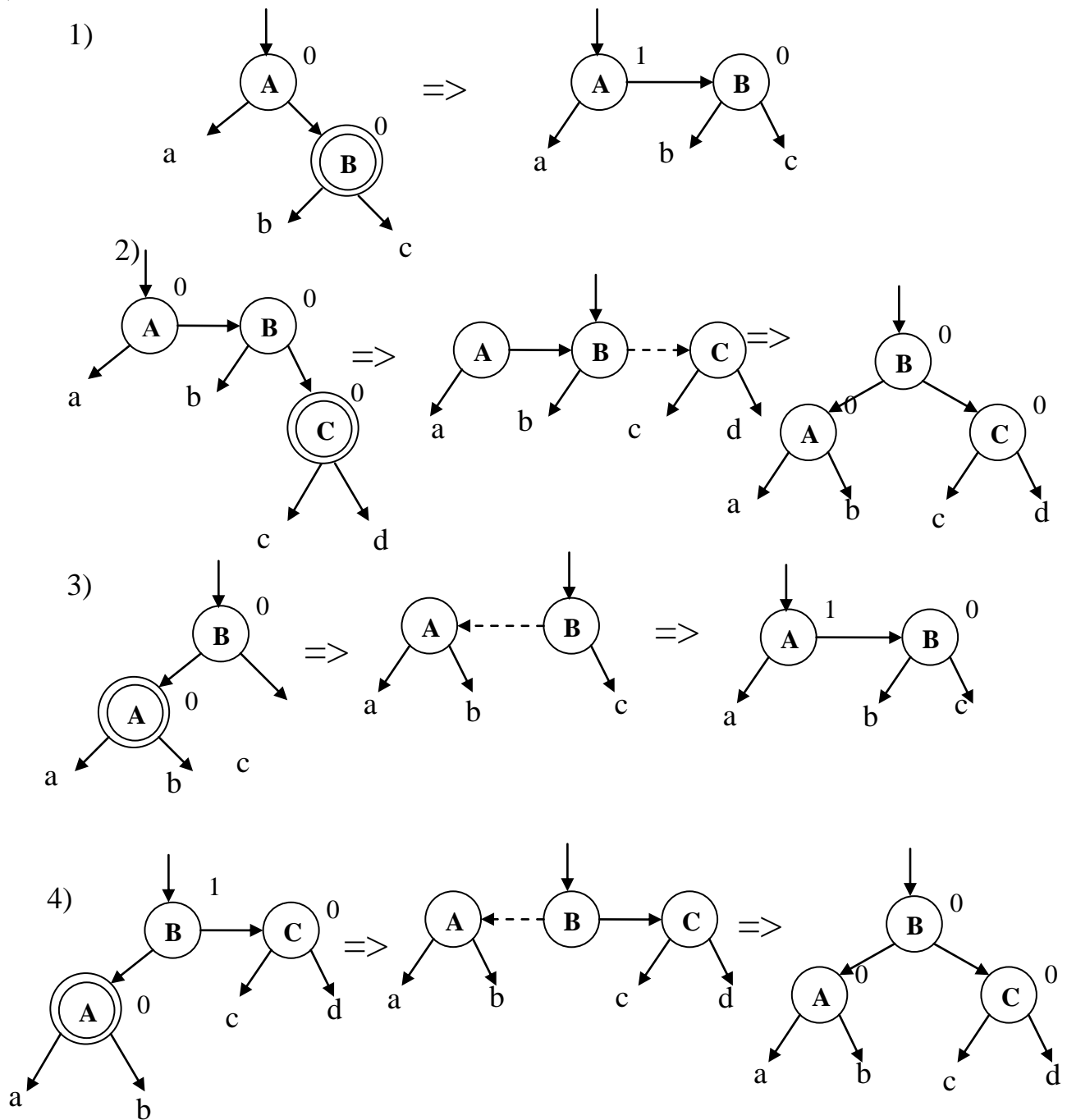


Рисунок 55 Четыре ситуации, возникающих при росте левых или правых поддеревьев

Алгоритм на псевдокоде

Добавление в ДБ-дерево(D : Данные, p : $pVertex$)

Обозначим

VR , HR — логические переменные, определяющие вертикальный или горизонтальный рост дерева (в начале VR , HR равны ИСТИНА)

IF ($p=NIL$)

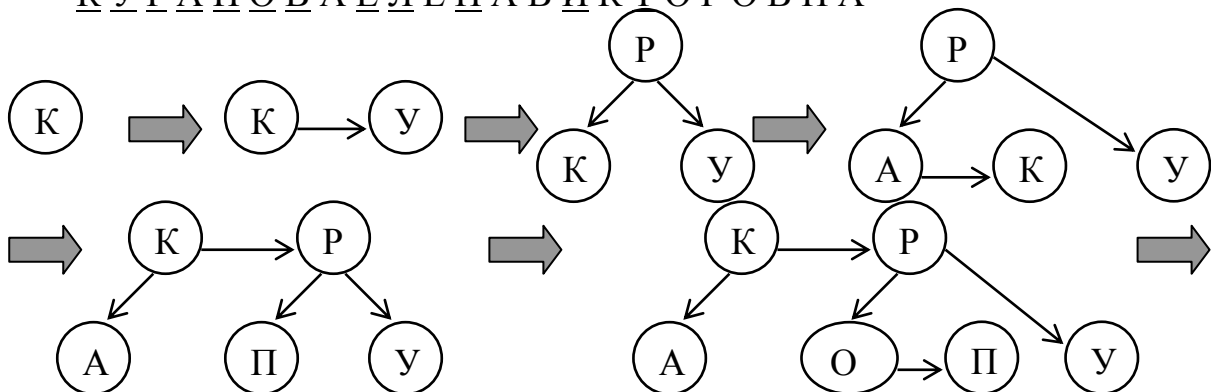
```

new(p); p→Date =D, p→Left= NIL; p→Right = NIL;
p→Balance=0; VR= ИСТИНА;
ELSE
  IF (p→Date> D) Добавление в ДБ-дерево(D, p→Left)
    IF (VR = ИСТИНА)
      IF (p→Balance = 0) q:= p→Left; p→Left:= q→Right;
        q→Right := p; p:=q; q→Balance :=1
        VR := ЛОЖЬ; HR =ИСТИНА;
      ELSE p→Balance:= 0; HR:=ИСТИНА;
    FI
    ELSE HR := ЛОЖЬ;
    FI
  ELSE IF (p→Date< D)
    Добавление в ДБ-дерево (D, p→Right),
    IF (VR = ИСТИНА) p→Balance:= 1; VR:= ЛОЖЬ;
      HR := ИСТИНА;
    ELSE IF (HR = ИСТИНА)
      IF(p→Balance > 0) q := p→Right;
        p→Right := q→Left;
        p→Balance := 0; q→Balance := 0;
        p→Left := p; p :=q
        VR:= ИСТИНА; HR:= ЛОЖЬ;
      ELSE HR:= ЛОЖЬ;
    FI
  FI
FI
FI
FI
FI
FI
FI
FI

```

Пример построения двоичного Б-дерева приведен на следующем рисунке.

К У Р А П О В А Е Л Е Н А В И К Т О Р О В Н А



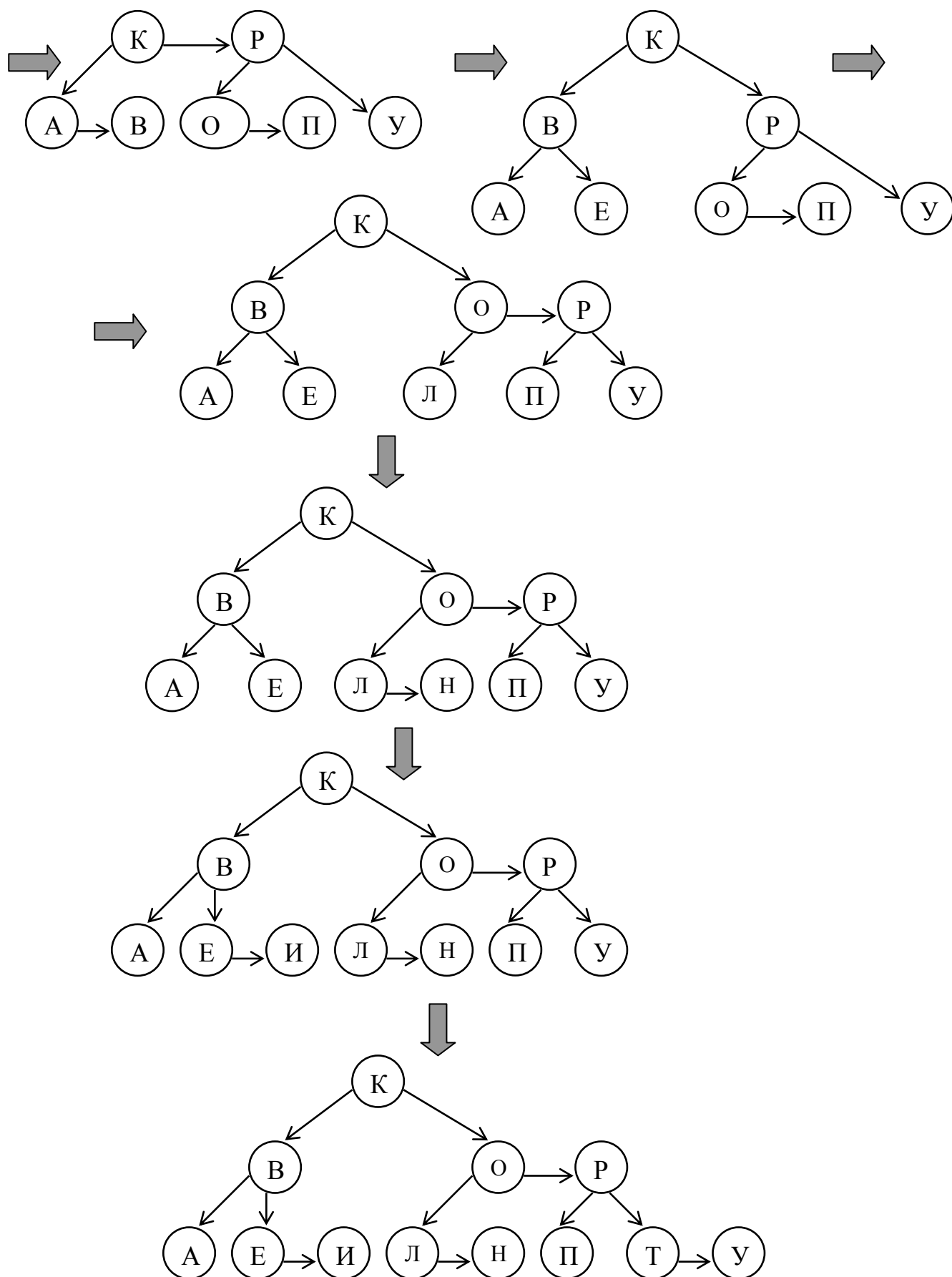


Рисунок 56 Построение двоичного Б-дерева

При построении двоичного Б-дерева реже приходится переставлять вершины, поэтому AVL-деревья предпочтительней в тех случаях, когда поиск ключей

происходит значительно чаще, чем добавление новых элементов. Кроме того, существует зависимость от особенностей реализации, поэтому вопрос о применении того или иного типа деревьев следует решать индивидуально для каждого конкретного случая.

13.6 Варианты заданий

1. Написать процедуру поиска элемента с заданным ключом в Б-дереве порядка m .
2. Определить трудоемкость поиска в Б-дереве порядка m .
3. Написать процедуру определения высоты Б-деревя порядка m .
4. Запрограммировать процедуру добавления нового элемента в Б-деревя порядка m .
5. Графически изобразить Б-деревя порядка 2.
6. Запрограммировать процедуру добавления новой вершины в двоичное Б-деревя. Определить количество необходимых операций для добавления вершины.
7. Написать процедуру определения высоты двоичного Б-деревя.
8. Экспериментально сравнить двоичное Б-деревя и ИСП по высоте как двоичные деревья.
9. Экспериментально сравнить высоты двоичного Б-деревя и случайного дерева поиска как двоичные деревья.
10. Экспериментально сравнить двоичное Б-деревя и АВЛ-деревя по высоте как двоичные деревья.
11. Графически изобразить двоичное Б-деревя.

14. ДЕРЕВЬЯ ОПТИМАЛЬНОГО ПОИСКА (ДОП)

14.1 Определение дерева оптимального поиска

До сих пор предполагалось, что частота обращения ко всем вершинам дерева поиска одинакова. Однако встречаются ситуации, когда известна информация о вероятностях обращения к отдельным ключам. Обычно для таких ситуаций характерно постоянство ключей, т.е. в деревя не включаются новые вершины и не исключаются старые и структура дерева остается неизменной. Эту ситуацию иллюстрирует сканер транслятора, который определяет, является ли каждое слово программы (идентификатор) служебным. Статистические измерения на сотнях транслируемых программ могут в этом случае дать точную информацию об относительных частотах появления в тексте отдельных ключей.

Припишем каждой вершине дерева V_i вес w_i , пропорциональный частоте поиска этой вершины (например, если из каждых 100 операций поиска 15 операций приходятся на вершину V_1 , то $w_1=15$). Сумма весов всех вершин дает вес дерева W . Каждая вершина V_i расположена на высоте h_i , корень расположен на высоте 1. Высота вершины равна количеству операций сравнения, необходимых для поиска этой вершины. Определим средневзвешенную высоту дерева с n вершинами следующим образом: $h_{cp}=(w_1h_1+w_2h_2+\dots+w_nh_n)/W$. Деревя поиска, имеющее

минимальную средневзвешенную высоту, называется *деревом оптимального поиска* (ДОП).

Пример. Рассмотрим множество из трех ключей $V_1=1$, $V_2=2$, $V_3=3$ со следующими весами: $w_1=60$, $w_2=30$, $w_3=10$, $W=100$. Эти три ключа можно расставить в дереве поиска пятью различными способами.

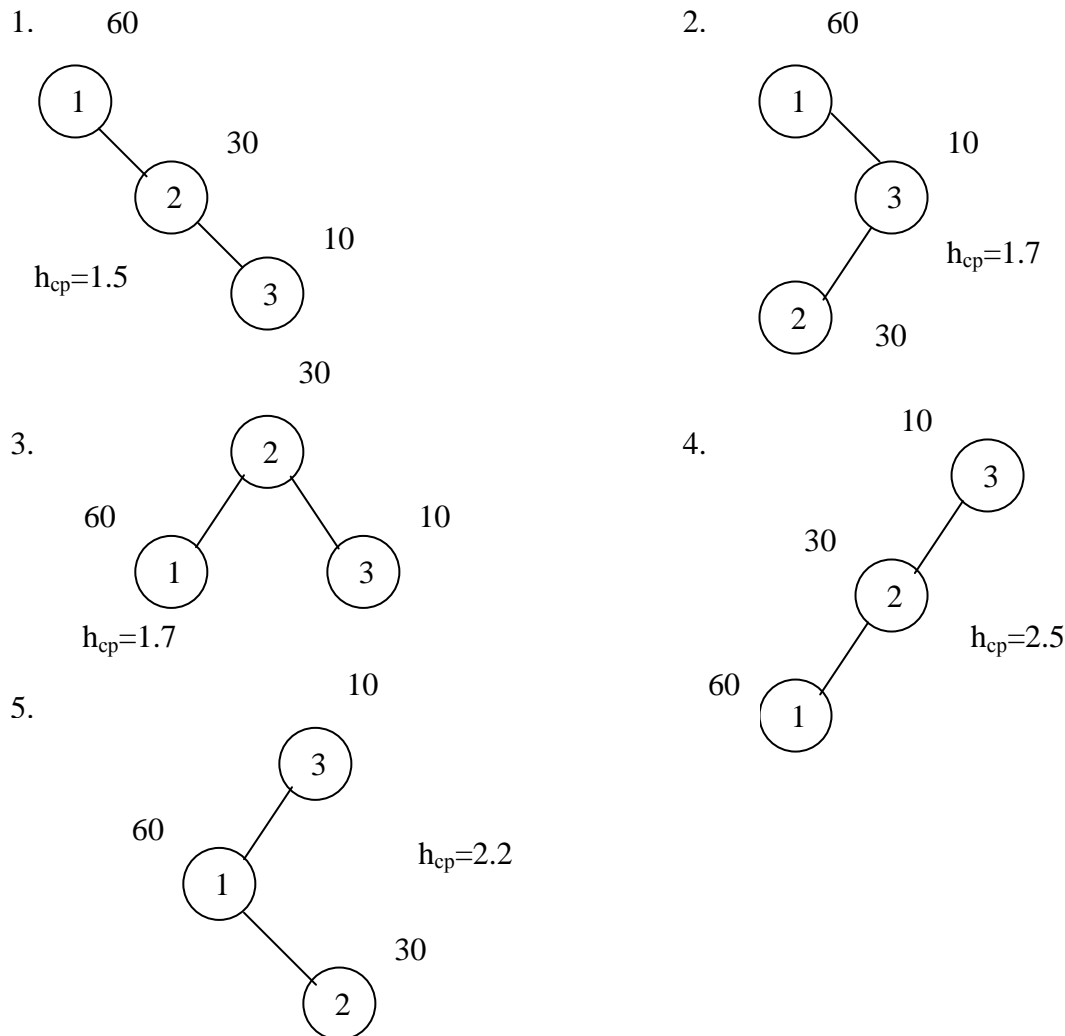


Рисунок 57 Различные деревья поиска с вершинами $V_1=1$, $V_2=2$, $V_3=3$

Легко видеть, что минимальной средневзвешенной высотой обладает дерево 1 на рисунке 57, которое представляет собой список или вырожденное дерево. Дерево 3 не является деревом оптимального поиска, хотя представляет собой идеально сбалансированное дерево. Очевидно, для минимизации средней длины пути поиска нужно стремиться располагать наиболее часто используемые вершины ближе к корню дерева.

Задача построения ДОП может ставится в двух вариантах:

- Известны вершины и их веса.
- Вес вершины определяется в процессе работы. Например, после каждого поиска вершины ее вес увеличивается на 1. В этом случае необходимо перестраивать структуру дерева при изменении весов.

Далее будем рассматривать задачу построения ДОП с фиксированным набором ключей и их весов.

14.2 Точный алгоритм построения ДОП

Поскольку число возможных конфигураций из n вершин растет экспоненциально с ростом n , то решение задачи построения ДОП при больших n методом перебора нерационально. Однако деревья оптимального поиска обладают свойствами, которые позволяют получить алгоритм построения ДОП, начиная с отдельных вершин с последовательным включением новых вершин в дерево. Далее будем считать, что множество вершин, входящих в дерево, упорядочено. Поскольку вес дерева остается неизменным, то вместо средневзвешенной высоты будем рассматривать взвешенную высоту дерева: $P = h_1w_1 + h_2w_2 + \dots + h_nw_n$

Свойство 1. Для дерева поиска с весом W справедливо соотношение $P = P_L + W + P_R$, где P_L, P_R – взвешенные высоты левого и правого поддеревьев корня.

Доказательство. Пусть вершина V_i с весом w_i является корневой для некоторого $i = 1, \dots, n$. Поскольку левое и правое поддерева являются деревьями поиска, то в левое поддерево входят вершины V_1, V_2, \dots, V_{i-1} , а в правое – V_{i+1}, \dots, V_n . Взвешенные высоты этих поддеревьев вычисляются следующим образом.

$$P_L = (h_1 - 1)w_1 + (h_2 - 1)w_2 + \dots + (h_{i-1} - 1)w_{i-1}$$

$$P_R = (h_{i+1} - 1)w_{i+1} + \dots + (h_n - 1)w_n$$

Рассмотрим выражение взвешенной высоты для всего дерева, замечая, что $h_i = 1$

$$P = h_1w_1 + h_2w_2 + \dots + h_nw_n = (h_1 - 1)w_1 + w_1 + (h_2 - 1)w_2 + w_2 + \dots + (h_{i-1} - 1)w_{i-1} + w_{i-1} + w_i + (h_{i+1} - 1)w_{i+1} + w_{i+1} + \dots + (h_n - 1)w_n + w_n = P_L + W + P_R$$

Свойство 2. Все поддерева дерева оптимального поиска также являются деревьями оптимального поиска для соответствующих подмножеств вершин.

Доказательство. Предположим, что одно из поддеревьев, например, правое, не является ДОП, т.е. существует дерево поиска с тем же множеством вершин, но с меньшей взвешенной высотой. Тогда по свойству 1 взвешенная высота всего дерева также не является минимальной. Данное противоречие доказывает свойство 2.

На основе приведенных свойств можно разработать точный алгоритм построения ДОП. Обозначим T_{ij} – оптимальное поддерево, состоящее из вершин V_{i+1}, \dots, V_j . Введем матрицу $AR = \|Ar_{ij}\|$, $0 \leq i, j \leq n$ элементы которой содержат номер корневой вершины поддерева T_{ij} , $0 \leq i < j \leq n$. Взвешенную высоту поддерева T_{ij} обозначим Aw_{ij} , а вес поддерева T_{ij} обозначим Aw_{ij} , $0 \leq i < j \leq n$. Очевидно, что $P = Ar_{0,n}$, $W = Aw_{0,n}$, T_{ii} – пустые деревья (без вершин), $Aw_{ii} = 0$, $Ar_{ii} = 0$, $i = 1, \dots, n$.

Используя свойство 2, величины Aw_{ij} , Ar_{ij} можно вычислить рекуррентно по следующим соотношениям (для всех возможных поддеревьев):

$$Aw_{ij} = Aw_{i,j-1} + Aw_j, \quad 0 \leq i < j \leq n \quad (1)$$

$$Ar_{ij} = Aw_{ij} + \min_{i < k < j} (Ar_{i,k-1} + Ar_{k,j}), \quad 0 \leq i < j \leq n \quad (2)$$

Во время вычислений будем запоминать индекс k^* , при котором достигается минимум во втором соотношении. Значение k^* является индексом корневой вершины поддерева T_{ij} во всем множестве вершин. Занесем в матрицу AR k^* – индекс корня T_{ij} , т.е. $Ar_{ij} = k^*$, $0 \leq i < j \leq n$.

Идея построения дерева состоит в следующем. В матрице AR берем значение $Ar_{0,n}$ (номер корневой вершины всего дерева в упорядоченном массиве вер-

шин), пусть оно равно k . Добавляем вершину V_k в дерево, используя обычный алгоритм добавления вершин в дерево поиска. Затем из матрицы AR берем значения $Ar_{o,k-1}$ и добавляем вершину с этим номером в левое поддерево. Далее берем $Ar_{k,n}$ и добавляем вершину с этим номером в правое поддерево и т.д.

Пример. Построить дерево оптимального поиска с вершинами $V_1=1, V_2=2, V_3=3$ и весами $w_1=60, w_2=30, w_3=10$. Сначала вычислим $Aw_{ij}, Ap_{ij}, Ar_{ij}, 0 \leq i < j \leq n$. Легко видеть, что

$T_{00}, T_{11}, T_{22}, T_{33}$ – пустые поддеревья

T_{01}, T_{12}, T_{23} – поддеревья из одной вершины (1), (2), (3)

T_{02}, T_{13} – поддеревья из двух вершин (1,2) и (2,3)

T_{03} – поддерево из трех вершин (1,2,3)

По формулам (1) и (2) вычислим элементы матрицы весов AW и элементы матрицы взвешенных высот AP , значения матрицы AR запишем в верхних углах ячеек матрицы AP .

Aw_{ij}	0	1	2	3
0	0	60	90	100
1		0	30	40
2			0	10
3				0

Ap_{ij}	0	1	2	3
0	0	60^1	120^1	150^1
1		0	30^2	50^2
2			0	10^3
3				0

$$Ap_{01} = Aw_{01} + \min_{0 < k \leq 1} (Ap_{00} + Ap_{11}) = 60 \quad (k^*=1)$$

$$Ap_{12} = Aw_{12} + \min_{1 < k \leq 2} (Ap_{11} + Ap_{22}) = 30 \quad (k^*=2)$$

$$Ap_{23} = Aw_{23} + \min_{2 < k \leq 3} (Ap_{22} + Ap_{33}) = 10 \quad (k^*=3)$$

$$Ap_{02} = Aw_{02} + \min_{0 < k \leq 2} (Ap_{00} + Ap_{12}, Ap_{01} + Ap_{22}) = 90 + 30 = 120 \quad (k^*=1).$$

$$Ap_{13} = Aw_{13} + \min_{1 < k \leq 3} (Ap_{11} + Ap_{23}, Ap_{12} + Ap_{33}) = 40 + 10 = 50 \quad (k^*=2).$$

$$Ap_{03} = Aw_{03} + \min_{0 < k \leq 3} (Ap_{00} + Ap_{13}, Ap_{01} + Ap_{23}, Ap_{02} + Ap_{33}) = 100 + 50 = 150 \quad (k^*=3).$$

Корневой вершиной будет вершина V_1 , поскольку $Ar_{0,3}=1$. Левое поддерево пустое, корень правого поддерева – вершина V_2 ($r_{1,3}=2$) и т.д. Полученное дерево показано на рисунке.

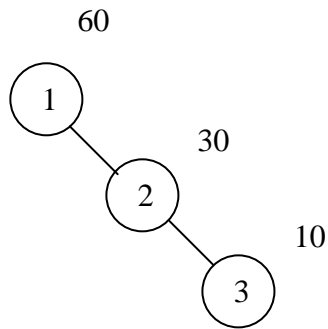


Рисунок 58 ДОП для $w_1=60$, $w_2=30$, $w_3=10$

Поскольку существует около $n^2/2$ значений Ap_{ij} , а вычисление (2) требует выбора одного из $0 < i-j \leq n$ значений, то весь процесс будет занимать $O(n^3)$ операций при $n \rightarrow \infty$. Д. Кнут отмечает, что можно избавиться от одного множителя n и тем самым сохранить практическую ценность алгоритма. Поиск Ar_{ij} можно ограничить, то есть сократить число вычислений до $j-i$ (если найден корень оптимального поддерева T_{ij} , то ни добавление справа новой вершины, ни отбрасывание левой вершины не могут сдвинуть вправо этот оптимальный корень). Это свойство выражается соотношением $Ar_{i,j-1} \leq Ar_{ij} \leq Ar_{i+1,j}$, что и ограничивает поиск Ar_{ij} диапазоном $Ar_{i,j-1} \dots Ar_{i+1,j}$. Это уменьшает трудоемкость алгоритма до $O(n^2)$ операций при $n \rightarrow \infty$.

Алгоритм на псевдокоде

Вычисление матрицы весов AW (формула 1).

```

DO (i = 0,...,n)
  DO (j = i+1,...,n)
    Awij := Awi,j-1 + wj
  OD
OD

```

Алгоритм на псевдокоде

Вычисление матриц AP и AR (формула 2).

```

DO (i = 0,...,n - 1)
  Apij+1 := Awij+1
  Arij+1 := i+1
OD
DO (h = 2,...,n)
  DO (i = 0,...,n - h)
    j := i + h
    m := Ari,j-1
    min := Api,m-1 + Apm,j
    DO (k = m+1,...,ARi+1,j)
      x := Api,k-1 + Apk,j
      IF (x < min) m:=k , min:= x FI
    OD
    Api,j := min + Awi,j
    Ari,j := m
  OD
OD

```

OD

Алгоритм на псевдокоде

*Создание дерева (L, R – границы массива вершин V,
Root – указатель на корень дерева)*

IF (L < R)

k := Ar_{L,R}

Добавить (Root, V_k)

Создание дерева (L, k-1)

Создание дерева (k, R)

FI

Для построения дерева необходимо вызвать процедуру создания дерева с параметрами L=0, R=n, Root=NIL.

14.3 Приближенные алгоритмы построения ДОП

Рассмотренный выше точный алгоритм имеет квадратичную трудоемкость. Однако при больших объемах деревьев такие алгоритмы становятся неэффективными. Известны быстрые алгоритмы, строящие почти оптимальные деревья. Рассмотрим два из них.

Первый алгоритм (A1) предлагает в качестве корня использовать вершину с наибольшим весом. Затем среди оставшихся вершин снова выбирается вершина с наибольшим весом и помещается в левое или правое поддерево в зависимости от ее значения, и т.д.

Алгоритм на псевдокоде

Алгоритм A1 (Root – указатель на корень дерева)

Обозначим

V.use – логическая переменная в структуре вершины дерева, которая показывает, что данная вершина была использована при построении дерева;
V.w – вес вершины.

Root := NIL

DO (i = 1,...,n)

V[i].use = ЛОЖЬ

OD

DO (i = 1,...,n)

max:=0, Index:=0

DO (j = 1,...,n)

IF (V[j].w > max и V[j]. use=ЛОЖЬ)

max:=V[j].w

Index:=j

FI

OD

V [index].use :=ИСТИНА

Добавление в СДП (Root, V[index])

OD

Рассмотрим пример построения дерева почти оптимального поиска для символов строки К У Р А П О В А Е Л Е Н А В И К Т О Р О В Н А. Всего символов в строке 23, т.е. $W=23$. Различные символы определяют различные вершины дерева. Частоты вхождения символов (веса) приведены в таблице.

Таблица 3 Частоты вхождения символов в строку

К	У	Р	А	П	О	В	Е	Л	Н	И	Т
2	1	2	4	1	3	3	2	1	2	1	1

Посчитаем средневзвешенную высоту построенного дерева

$$P=4 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 2 \cdot 3 + 2 \cdot 4 + 1 \cdot 4 + 1 \cdot 4 + 2 \cdot 5 + 2 \cdot 5 + 1 \cdot 5 + 1 \cdot 6 + 1 \cdot 6 = 78$$

$$h_{cp} = P/W = 78/23 = 3,39$$

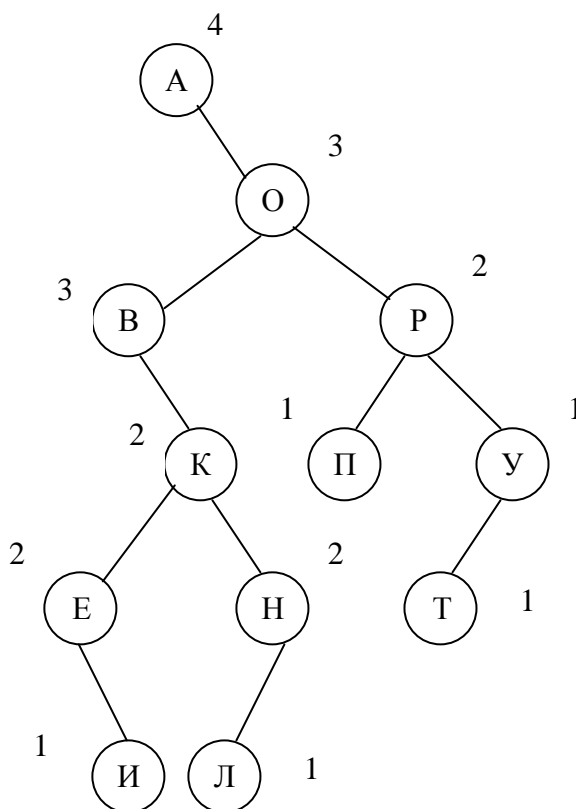


Рисунок 59 Дерево, построенное приближенным алгоритмом А1

Второй алгоритм (А2) использует предварительно упорядоченный набор вершин. В качестве корня выбирается такая вершина, что разность весов левого и правого поддеревьев была минимальна. Для этого путем последовательного суммирования весов определим вершину V_k , для которой справедливы неравенства:

$$\sum_{i=1}^{k-1} w_i < \frac{W}{2} \text{ и } \sum_{i=1}^k w_i \geq \frac{W}{2}.$$

Тогда в качестве "центра тяжести" может быть выбрана вершина V_k , V_{k-1} или V_{k+1} , т. е. вершина, для которой разность весов левого и правого поддерева минимальна. Далее действия повторяются для каждого поддерева

Алгоритм на псевдокоде

*Алгоритм А2(Root – указатель на корень дерева,
L, R – левая и правая границы рабочей части массива)*

wes=0, summa=0

IF (L<=R)

DO (i=L, L+1, ...,R) wes=wes+ V[i].w OD

DO (i=L, L+1,...,R-1)

IF (summa<wes/2 and summa + V[i].w>=wes/2) OD

summa=summa+ V[i].w

OD

Добавление в СДП (Root, V[i])

A2(Root, L, i-1)

A2(Root, i+1, R)

FI

Пример. Рассмотрим процесс построения приближенного ДОП алгоритмом A2 для строки символов из предыдущего примера. Предварительно упорядочим символы по алфавиту.

Таблица 4 Упорядоченный набор вершин

А	В	Е	И	К	Л	Н	О	П	Р	Т	У
4	3	2	1	2	1	2	3	1	2	1	1

В качестве корня дерева выбираем вершину $V_5=K$, поскольку

$$\sum_{i=1}^{5-1} w_i = 4 + 3 + 2 + 1 < \frac{23}{2}, \quad \sum_{i=1}^5 w_i = 4 + 3 + 2 + 1 + 2 \geq \frac{23}{2}.$$

Все вершины левее вершины V_5 образуют левое поддереву, вершины правее V_5 – правое поддереву. Далее алгоритм применяется для каждого из поддеревьев в отдельности. Посчитаем средневзвешенную высоту построенного дерева.

$$P = 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 2 + 4 \cdot 3 + 2 \cdot 3 + 2 \cdot 3 + 2 \cdot 3 + 1 \cdot 4 + 1 \cdot 4 + 1 \cdot 4 + 1 \cdot 4 + 1 \cdot 5 = 65$$

$$h_{cp} = 65/23 = 2.82$$

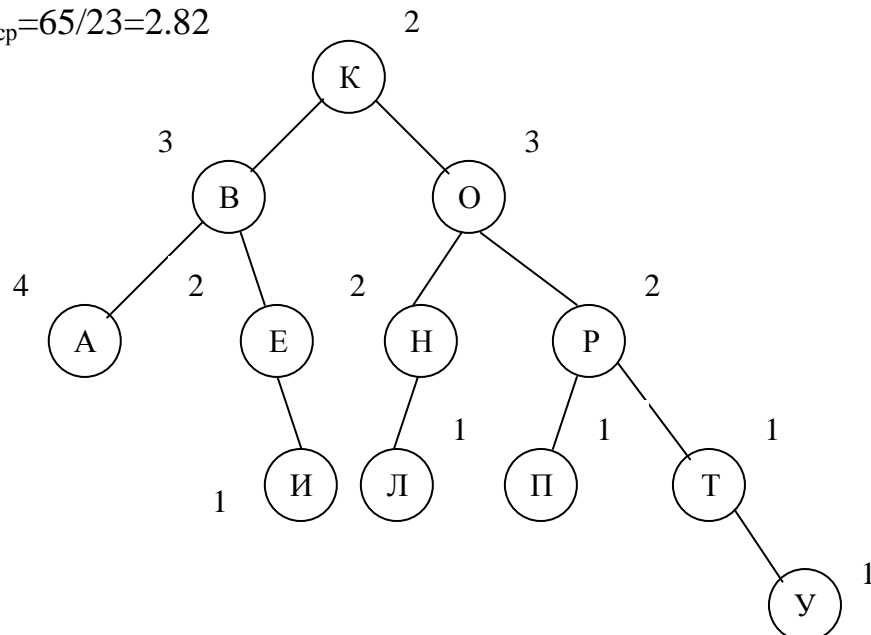


Рисунок 60 Дерево, построенное приближенным алгоритмом A2

Приведем некоторые свойства рассмотренных приближенных алгоритмов:

- 1) Сложность алгоритмов как функция от n (количество элементов) зависит следующим образом: время $T = O(n \log n)$, память $M = O(n)$ при

$n \rightarrow \infty$. (Время определяется трудоемкостью сортировки элементов, а память – размером массива для хранения элементов)

- 2) Дерево, построенное приближенным алгоритмом A_1 , равносильно случайному (с точки зрения средней высоты) при $n \rightarrow \infty$, т.е. алгоритм A_1 – плохой.
- 3) Дерево, построенное приближенным алгоритмом A_2 , асимптотически приближается к оптимальному (с точки зрения средней высоты) при $n \rightarrow \infty$, т.е. алгоритм A_2 является хорошим.
- 4) ИСДП нельзя считать даже приближением к дереву оптимального поиска.

14.4 Варианты заданий

1. Написать процедуру вычисления матрицы весов и матрицы средневзвешенных весов для заданного набора вершин и их весов.
2. Реализовать точный алгоритм построения ДОП.
3. Написать процедуру вычисления средневзвешенной высоты ДОП.
4. Реализовать в виде процедур приближенные алгоритмы построения ДОП.
5. Построить ДОП из 100 чисел с произвольными весами с использованием точного алгоритма. Определить средневзвешенную высоту построенного дерева.
6. Построить почти оптимальные деревья из 100 чисел с использованием приближенных алгоритмов A_1 и A_2 . Сравнить средневзвешенные высоты этих деревьев и ДОП, построенного точным алгоритмом.
7. Построить ДОП ключевых (зарезервированных) слов языка Паскаль (частоты слов определить путем сканирования собственных программ на Паскале).
8. Построить ДОП ключевых (зарезервированных) слов языка Си (частоты слов определить путем сканирования собственных программ на Си).
9. Графически изобразить ДОП из 20 чисел на экране. (Показать значения вершин и их веса).
10. Графически изобразить на экране ДОП ключевых слов языка Паскаль (или Си).

15. ХЭШИРОВАНИЕ И ПОИСК

15.1 Понятие хэш-функции

Все рассмотренные ранее алгоритмы были связаны с задачей поиска, которую можно сформулировать следующим образом: задано множество ключей, необходимо так организовать это множество ключей, чтобы поиск элемента с заданным ключом потребовал как можно меньше затрат времени. Поскольку доступ к элементу осуществляется через его адрес в памяти, то задача сводится к определению подходящего отображения H множества ключей K во множество адресов элементов A .

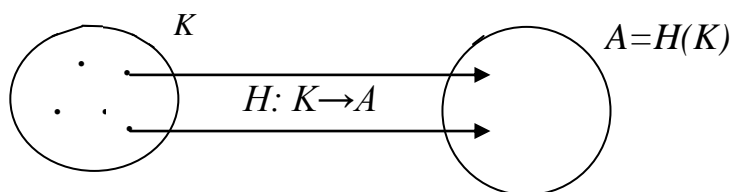


Рисунок 61 Отображение $H: K \rightarrow A$

В предыдущих главах такое отображение получалось путем различного размещения ключей (в отсортированном порядке, в виде деревьев поиска), т.е. каждому ключу соответствовал свой адрес в памяти. Теперь рассмотрим задачу построения отображения $H: K \rightarrow A$ при условии, что количество всевозможных ключей существенно больше количества адресов. Будем обозначать это так: $|K| \gg |A|$. Например, в качестве множества ключей можно взять всевозможные фамилии студентов до 15 букв ($|K| = 32^{15}$), а в качестве множества адресов – 100 мест в аудитории ($|A| = 100$). Функция $H: K \rightarrow A$, определенная на конечном множестве K , называется *хеш-функцией*, если $|K| \gg |A|$. Таким образом, хеш-функция допускает, что нескольким ключам может соответствовать один адрес. *Хеширование* – один из способов поиска элементов по ключу, при этом над ключом k производят некоторые арифметические действия и получают значение функции $h = H(k)$, которое указывает адрес, где хранится ключ k и связанная с ним информация. Если найдутся ключи $k_i \neq k_j$, для которых $H(k_i) = H(k_j)$, т.е. несколько ключей отображаются в один адрес, то такая ситуация называется *коллизией* (*конфликтом*).

Если данные организованы как обычный массив, то H – отображение ключей в индексы массива. Процесс поиска происходит следующим образом:

- 1) для ключа k вычисляем индекс $h = H(k)$
- 2) проверяем, действительно ли h определяет в массиве T элемент с ключом k , т.е. верно ли соотношение $T[H(k)].data = k$. Если равенство верно, то элемент найден. Если неверно, то возникла коллизия.

Для эффективной реализации поиска с помощью хеш-функций необходимо определить какого вида функцию H нужно использовать и что делать в случае коллизии (конфликта). Хорошая хеш-функция должна удовлетворять двум условиям:

- 1) её вычисление должно быть очень быстрым
- 2) она должна минимизировать число коллизий, т.е. как можно равномернее распределять ключи по всему диапазону индекса.

Для разрешения коллизий нужно использовать какой-нибудь способ, указывающий альтернативное местоположение искомого элемента. Выбор хеш-функции и выбор метода разрешения коллизий – два независимых решения.

Функции, дающие неповторяющиеся значения, достаточно редки даже в случае довольно большой таблицы. Например, знаменитый парадокс дней рождений утверждает, что если в комнате присутствует не менее 23 человек, имеется хороший шанс, что у двух из них совпадет день рождения. Т.е., если мы выбираем функцию, отображающую 23 ключа в таблицу из 365 элементов, то с вероятностью 0,4927 все ключи попадут в разные места.

Теоретически невозможно так определить хеш-функцию, чтобы она создавала случайные данные из неслучайных реальных ключей. Но на практике нетрудно сделать достаточно хорошую имитацию случайности, используя простые арифметические действия.

Будем предполагать, что хеш-функция имеет не более m различных значений: $0 \leq H(k) < m$ для любого значения ключа. Например, если ключи десятичные, то возможен следующий способ. Пусть $m=1000$, в качестве $H(k)$ можно взять три цифры из середины двадцатизначного произведения $k \cdot k$. Этот метод «середины квадрата», казалось бы, должен давать довольно равномерное распределение между 000 и 999. Но на практике такой метод не плох, если ключи не содержат много левых или правых нулей подряд.

Исследования выявили хорошую работу двух типов хеш-функций: один основан на умножении, другой на делении.

- 1) *метод деления* особенно прост: используется остаток от деления на m $H(K) = K \bmod m$. При этом желательно m брать простым числом.
- 2) *метод умножения* $H(K) = 2^m (A \cdot K \bmod w)$, где A и w взаимно простые числа.

Далее будем использовать функцию $H(k) = ORD(k) \bmod m$, где $ORD(k)$ – порядковый номер ключа, m – размер массива (таблицы), причем m рекомендуется брать простым числом.

Если ключ поиска является строкой, то для вычисления ее хеш-номера будем рассматривать её как большое целое число, записанное в 256-ичной системе счисления (каждый символ строки является цифрой), т.е.

$$H(S_1 S_2 S_3 \dots S_t) = (S_1 \cdot 256^{t-1} + S_2 \cdot 256^{t-2} + \dots + S_{t-1} \cdot 256 + S_t) \bmod m.$$

Используя свойства остатка от деления можно легко вычислить подобные выражения: $(a+b) \bmod m = (a \bmod m + b \bmod m) \bmod m$. Например, $(47+56) \bmod 10 = (7+6) \bmod 10 = 3$

Алгоритм на псевдокоде

Вычисление хеш-функции для строки S

Обозначим t – длина строки S

$h := 0$

DO ($i=1, 2, \dots, t$)

$h := (h \cdot 256 + S_i) \bmod m$

OD

15.2 Метод прямого связывания

Рассмотрим метод устранения коллизий путем связывания в список всех элементов с одинаковыми значениями хеш-функции, при этом необходимо m списков. Включение элемента в хэш-таблицу осуществляется в два действия:

- 1) вычисление $i = H(k)$
- 2) добавление элемента k в конец i -того списка

Поиск элемента также требует два действия:

- 1) вычисление $i = H(k)$

2) последовательный просмотр i -того списка.

Пример. Составить хеш-таблицу для строки КУРАПОВА ЕЛЕНА. Будем использовать номера символов в алфавитном порядке. Пусть $m=5$, $H(k)=ORD(k \bmod 5)$

Вычислим значения хэш-функции для символов строки

$$H(K)=11 \bmod 5=1$$

$$H(Y)=20 \bmod 5=0$$

$$H(P)=17 \bmod 5=2$$

$$H(A)=1 \bmod 5=1$$

$$H(\Pi)=16 \bmod 5=1$$

$$H(O)=15 \bmod 5=0$$

$$H(B)=3 \bmod 5=3$$

$$H(E)=6 \bmod 5=1$$

$$H(L)=12 \bmod 5=2$$

$$H(H)=14 \bmod 5=4$$

Объединим символы с одинаковыми хеш-номерами в один список

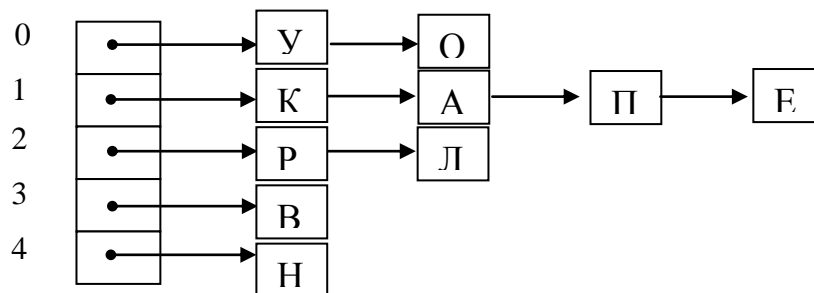


Рисунок 62 Хеш-таблица, построенная методом прямого связывания

Оценим трудоемкость поиска в хеш-таблице, построенной методом прямого связывания. Пусть n – количество элементов данных, m – размер хеш-таблицы. Если все ключи равновероятны и равномерно распределены по хеш-таблице, то средняя длина списка будет $\frac{n}{m}$. При поиске в среднем нужно просмотреть половину списка.

Поэтому $C_{cp} = \frac{n}{2m}$. Если $n < m$, то $C_{cp} < 2$, т. е. в большинстве случаев достаточно одного сравнения. Объем дополнительной памяти определяется объемом памяти, необходимой для хранения $(m+n)$ указателей. Известно, что трудоемкость поиска с помощью двоичного дерева: $C_{cp} = \log n$, объем дополнительной памяти – $2n$ указателей. Метод прямого связывания становится более эффективным, чем дерево поиска, когда

$$\frac{n}{2m} < \log n, \quad m > \frac{n}{2 \log n}$$

Если $n=1000$, то при $m>50$ ($m=53$) метод прямого связывания более эффективен, чем дерево поиска, причем экономия памяти составит около 4 Кбайт. Можно сэкономить еще больше памяти, если отказаться от списков и размещать данные в самой хеш-таблице.

15.3 Метод открытой адресации

Рассмотрим метод открытой адресации, который применяется для разрешения коллизий при поиске с использованием хеш-функций. Суть метода заключается в последовательном просмотре различных элементов таблицы, пока не будет найден искомый ключ k или свободная позиция. Очевидно, необходимо иметь правило, по которому каждый ключ k определяет последовательность проб, т.е. последовательность позиций в таблице, которые нужно просматривать при вставке или поиске ключа k . Если мы произвели пробы и обнаружили свободную позицию, то ключа k нет в таблице. Таким образом, коллизия устраняется путем вычисления последовательности *вторичных хеш-функций*:

$$\begin{aligned}h_0 &= h(x) \\ h_1 &= h(x) + g(1) \pmod{m} \\ h_2 &= h(x) + g(2) \pmod{m} \\ h_i &= h(x) + g(i) \pmod{m}\end{aligned}$$

Самое простое правило для просмотра – просматривать подряд все следующие элементы таблицы. Этот прием называется *методом линейных проб*, при этом $g(i)=i$, $i=1,2,\dots,m-1$. Недостаток данного метода – плохое рассеивание ключей (ключи группируются вокруг первичных ключей, которые были вычислены без конфликта), хотя и используется вся хеш-таблица.

Если в качестве вспомогательных функций использовать квадратичные, т.е. $g(i)=i^2$, $i=1,2,\dots,m-1$, то такой способ просмотра элементов называется *методом квадратичных проб*. Достоинство этого метода – хорошее рассеивание ключей, хотя хеш-таблица используется не полностью.

Утверждение. Если m – простое число, то при квадратичных пробах просматривается по крайней мере половина хеш-таблицы.

Доказательство. Пусть i -ая и j -ая пробы, $i < j$, приводят к одному значению h , т.е. $h_i = h_j$. Тогда $i^2 \pmod{m} = j^2 \pmod{m}$

$$\begin{aligned}(j^2 - i^2) \pmod{m} &= 0 \\ (j+i)(j-i) \pmod{m} &= 0 \\ (j+i)(j-i) &= km \\ i+j &= km/(j-i)\end{aligned}$$

Если m – простое число, то $k/(j-i)$ – целое число больше нуля. В худшем случае $k/(j-i)=1$, тогда $i+j=m$ и $j > m/2$. (Если m – не простое число, то $k/(j-i)$ не обязательно должно быть целым).

На практике этот недостаток не столь существен, т.к. $m/2$ вторичных попыток при разрешении конфликтов встречаются очень редко, главным образом в тех случаях, когда таблица почти заполнена.

Итак, нам нужно вычислять

$$\begin{aligned}h_0 &= h(x) \\ h_i &= (h_0 + i^2) \pmod{m}, i > 0\end{aligned}$$

Вычисление h_i требует одного умножения и деления. Покажем, как можно избавиться от этих операций. Произведем несколько первых шагов при вычислении h_i .

$$h_1 = h_0 + 1$$

$$h_2 = h_0 + 4 = h_0 + 1 + 3 = h_1 + 3 \pmod{m}$$

$$h_3 = h_0 + 9 = h_0 + 4 + 5 = h_2 + 5 \pmod{m}$$

...

Нетрудно видеть, что возникает рекуррентное соотношение:

$$d_0 = 1, h_0 = h(x)$$

$$h_{i+1} = h_i + d_i \pmod{m}$$

$$d_{i+1} = d_i + 2$$

Поскольку $h_i < m$, $d_i < m$, то можно избавиться от деления, заменив его вычитанием $h = h - m$ (см. алгоритм).

Алгоритм на псевдокоде

Поиск и вставка элемента с ключом x

Пусть хеш-таблица является массивом $A = (a_0, a_1, \dots, a_{m-1})$, сначала заполненный нулями. Пусть $x \neq 0$.

$h := x \bmod m$

$d := 1$

DO

IF ($a_h = x$) элемент найден OD

IF ($a_h = 0$) ячейка пуста, $a_h := x$ OD

IF ($d \geq m$) переполнение таблицы OD

$h := h + d$

IF ($h \geq m$) $h := h - m$ FI

$d := d + 2$

OD

Заметим, что если нужен только поиск, то необходимо исключить операцию $a_h := x$.

Пример построения хеш-таблицы методом квадратичных проб ($m=11$) для строки ВЛАДИМИР ПУТИН. Номера символов данной строки приведены в таблице.

Таблица 5 Номера символов строки

В	Л	А	Д	И	М	И	Р	П	У	Т	И	Н
3	12	1	5	9	13	9	17	16	20	19	9	14

Для каждого символа вычисляем его хеш-номер (или последовательность хеш-номеров, если потребуется) и в соответствии с вычисленным номером заносим символ в хеш-таблицу.

В: $h_0 = 3 \bmod 11 = 3$

Л: $h_0 = 12 \bmod 11 = 1$

А: $h_0 = 1 \bmod 11 = 1$

$h_1 = 1 + 1 = 2$

Д: $h_0 = 5$

И: $h_0 = 9$

М: $h_0 = 13 \bmod 11 = 2$

$h_1 = 2 + 1 = 3$

$h_2 = 3 + 3 = 6$

Р: $h_0 = 17 \bmod 11 = 6$

$$h_1=6+1=7$$

$$\text{П: } h_0=16 \bmod 11=5$$

$$h_1=5+1=6$$

$$h_2=6+3=9$$

$$h_3=9+5=3$$

$$h_4=3+7=10$$

$$\text{У: } h_0=20 \bmod 11=9$$

$$h_1=9+1=10$$

$$h_2=10+3=13 \bmod 11=2$$

$$h_3=2+5=7$$

$$h_4=7+7=14 \bmod 11=3$$

$$h_5=3+9=12 \bmod 11=1$$

Просмотр элементов хеш-таблицы на этом заканчивается несмотря на то, что в таблице еще имеются незаполненные ячейки, поскольку следующее значение d уже не будет строго меньше $m=11$. Таким образом, для данной строки не удастся построить хеш-таблицу с $m=11$. Заполненная часть хеш-таблицы выглядит следующим образом

0	1	2	3	4	5	6	7	8	9	10
	Л	А	В		Д	М	Р		И	П

Рисунок 63 Использование квадратичных проб

15.4 Варианты заданий

- 1) Реализовать построение хэш-таблицы методом прямого связывания для слов заданного текста. Экспериментально определить минимально необходимый объем хэш-таблицы.
- 2) Реализовать процедуру поиска с использованием хэш-таблицы (метод прямого связывания). Экспериментально определить среднее число сравнений при поиске.
- 3) Построить хэш-таблицу методом линейных проб для слов заданного текста. Экспериментально определить минимально необходимый объем хэш-таблицы число коллизий при построении.
- 4) Построить хэш-таблицу методом квадратичных проб для слов заданного текста. Экспериментально определить минимально необходимый объем хэш-таблицы число коллизий при построении.
- 5) Экспериментально сравнить объем хэш-таблицы и число коллизий для методов линейных и квадратичных проб.
- 6) Реализовать процедуру поиска с использованием хэш-таблицы (метод открытой адресации). Экспериментально определить среднее число коллизий при поиске.

16. ЭЛЕМЕНТЫ ТЕОРИИ КОДИРОВАНИЯ ИНФОРМАЦИИ

16.1 Необходимые понятия

Теория кодирования и теория информации возникли в начале XX века. Начало развитию этих теорий как научных дисциплин положило появление в 1948 г. статей К. Шеннона, которые заложили фундамент для дальнейших исследований в этой области.

Основной моделью, которую изучает теория информации, является *модель системы передачи сигналов*:

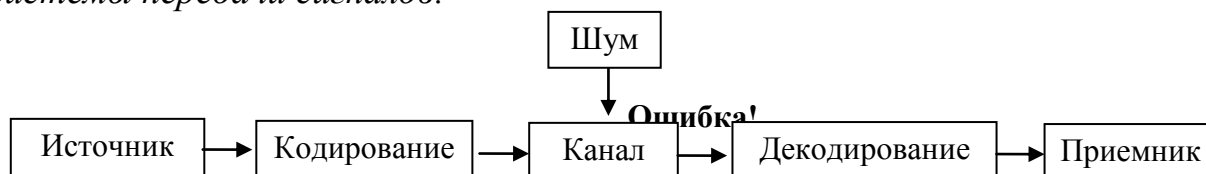


Рисунок 64 Модель системы передачи сигналов

Кодирование – способ представления информации в виде, удобном для хранения и передачи. В связи с развитием информационных технологий кодирование является центральным вопросом при решении самых разных задач программирования:

1. Представление данных произвольной структуры (числа, текст, графика) в памяти компьютера.
2. Обеспечение помехоустойчивости при передаче данных по каналам связи.
3. Сжатие информации в базах данных.

Под *сжатием* понимается *компактное представление данных*, осуществляемое за счет избыточности, содержащейся в сообщениях. Большое значение для практического использования имеет *неискажающее сжатие*, позволяющее полностью восстановить исходное сообщение. При *неискажающем сжатии* происходит кодирование сообщения перед началом передачи или хранения, а после окончания процесса сообщение однозначно декодируется (это соответствует модели канала без шума (помех)).

Начальным звеном в приведенной выше модели является *источник информации*. Мы будем рассматривать *дискретные источники*, в которых выходом является последовательность символов некоторого фиксированного алфавита. Множество всех различных символов, порождаемых некоторым источником, называется *алфавитом источника*, а количество символов в этом множестве – *размером алфавита источника*. Например, можно считать, что текст на русском языке порождается источником с алфавитом из 32 русских букв, пробела и знаков препинания.

Кодирование дискретного источника заключается в сопоставлении символов алфавита А источника символам некоторого другого алфавита В. Причем обычно символу исходного алфавита А ставится в соответствие не один, а группа символов алфавита В, называемых *кодовым словом*. *Кодовый алфавит* – множество различных символов, используемых для записи кодовых слов. *Кодом* назы-

вается совокупность всех кодовых слов, применяемых для представления порождаемых источником символов.

Пример. Азбука Морзе является общеизвестным кодом из символов телеграфного алфавита, в котором буквам русского языка соответствуют кодовые слова (последовательности) из «точек» и «тире».

Далее будем рассматривать *двоичное кодирование*, т.е. размер кодового алфавита равен 2. Конечную последовательность битов (0 и 1) назовем *кодовым словом*, а количество битов в этой последовательности – *длиной кодового слова*.

Пример. Код ASCII (американский стандартный код для обмена информацией) каждому символу ставит в однозначное соответствие кодовое слово длиной 8 бит.

Пусть даны алфавит источника $A = \{a_1, \dots, a_n\}$, кодовый алфавит $B = \{b_1, \dots, b_m\}$, $S \subseteq A^*$ – множество всевозможных сообщений в алфавите A. Тогда функция $F: S \rightarrow B^*$ – *кодирование* (преобразует множество сообщений в алфавит B). Если $\alpha \in S$, то $F(\alpha) = \beta \in B^*$ – кодовое слово. Обратная функция F^{-1} (если она существует) называется *декодированием*.

Задача кодирования сообщения ставится следующим образом. Требуется при заданных алфавитах A и B и множестве сообщений S найти такое кодирование F, которое обладает определенными свойствами (т.е. удовлетворяет заданным ограничениям) и оптимально в некотором смысле. Свойства, которые требуются от кодирования, могут быть различными:

1. Существование декодирования

2. Помехоустойчивость или исправление ошибок: функция декодирования обладает свойством $F^{-1}(\beta) = F^{-1}(\beta')$, $\beta \sim \beta'$ (эквивалентно β' с ошибкой)

3. Обладает заданной трудоемкостью (время, объем памяти).

Известны два класса методов кодирования дискретного источника информации: равномерное и неравномерное кодирование. Под *равномерным кодированием* понимается использование кодов со словами постоянной длины. Для того, чтобы декодирование равномерного кода было возможным, разным символам алфавита источника должны соответствовать разные кодовые слова. При этом длина кодового слова должна быть не меньше $\lceil \log_n m \rceil$ символов, где m – размер исходного алфавита, n – размер кодового алфавита.

Пример. Для кодирования источника, порождающего 26 букв латинского алфавита, равномерным двоичным кодом, требуется построить кодовые слова длиной не меньше $\lceil \log_2 26 \rceil = 5$ бит.

При *неравномерном кодировании источника* используются кодовые слова разной длины. Причем кодовые слова обычно строятся так, что часто встречающиеся символы кодируются более короткими кодовыми словами, а редкие символы – более длинными (за счет этого и достигается «сжатие» данных).

16.2 Кодирование целых чисел

Рассмотрим семейство методов, не учитывающих вероятности появления элементов. В общем случае никаких предположений о свойствах значений элементов не делается, просто каждому целому числу из определенного диапазона

ставится в соответствие свое кодовое слово. Поэтому эту группу методов также называют представлением целых чисел (Representation of Integers).

Основная идея кодирования состоит в том, чтобы отдельно описывать порядок значения элемента x_i («экспоненту» E_i) и отдельно – значащие цифры значения x_i («мантиссу» M_i). Значащие цифры начинаются со старшей ненулевой цифры; порядок числа определяется позицией старшей ненулевой цифры в двоичной записи числа. Как и при десятичной записи, порядок равен числу цифр в записи числа без предшествующих незначащих нулей. Например, порядок двоичного числа 000001101 равен 4, а мантисса – 1101.

Рассмотрим две группы методов кодирования целых чисел. Условно их можно обозначить так:

- Fixed + Variable (фиксированная длина экспоненты – переменная длина мантиссы)
- Variable + Variable (переменная длина экспоненты – переменная длина мантиссы)

Коды класса Fixed + Variable

В кодах класса Fixed + Variable под запись значения порядка числа отводится фиксированное количество бит, а значение порядка числа определяет, сколько бит потребуется под запись мантиссы. Для кодирования целого числа необходимо произвести с числом две операции: определение порядка числа и выделение бит мантиссы (можно хранить в памяти готовую таблицу кодовых слов). Рассмотрим процесс построения кода данного класса на примере.

Пример. Пусть $R=15$ – количество бит исходного числа. Отведем $E=4$ бита под экспоненту (порядок), т.к. $R \leq 2^4$. При записи мантиссы можно сэкономить 1 бит: не писать первую единицу т.к. это всегда будет только единица. Таким образом, количество бит мантиссы равно количеству бит для порядка минус 1.

Таблица 6 Код класса Fixed + Variable

число	кодовое слово	длина кодового слова
0	0000	4
1	0001	4
2	0010 0	5
3	0010 1	5
4	0011 00	6
5	0011 01	6
6	0011 10	6
7	0011 11	6
8	0100 000	7
9	0100 001	7
10	0100 010	7
...
15	0100 111	7
16	0101 0000	8
17	0101 0001	8
...

Коды класса Variable + Variable

В качестве кода числа записываем сначала подряд несколько нулей (их количество равно значению порядка числа), затем единицу как признак окончания экспоненты переменной длины, затем мантиссу переменной длины (как в кодах Fixed + Variable). Рассмотрим пример построения кода этого класса.

Таблица 7 Код класса *Variable + Variable*

число	кодированное слово	длина кодированного слова
0	1	1
1	0 1	2
2	00 1 0	4
3	00 1 1	4
4	000 1 00	6
5	000 1 01	6
6	000 1 10	6
7	000 1 11	6
8	0000 1 000	8
9	0000 1 001	8
10	0000 1 010	8
...	...	

Если в рассмотренном выше коде исключить кодированное слово для нуля, то можно уменьшить длины кодированных слов на 1 бит, убрав первый ноль. В результате получаем гамма-код Элиаса.

Таблица 8 γ -код Элиаса

число	кодированное слово	длина кодированного слова
1	1	1
2	01 0	3
3	01 1	3
4	00 1 00	5
5	00 1 01	5
6	00 1 10	5
7	00 1 11	5
8	000 1 000	7
9	000 1 001	7
10	000 1 010	7
...	...	

Другим примером кода класса Variable + Variable является *омега-код Элиаса* (ω -код Элиаса). В нем первое значение (кодированное слово для единицы) задается отдельно. Другие кодированные слова состоят из последовательности групп длиной L_1, L_2, \dots, L_m , начинающихся с единицы. Конец всей последовательности задается нулем. Длина первой группы – 2 бита и далее длина каждой следующей группы равна значению битов предыдущей группы плюс 1. Значение битов последней

группы является итоговым значением всей последовательности групп, т.е. первые $m-1$ групп служат для указания длины последней группы.

Таблица 9 ω -код Элиаса

число	кодированное слово	длина кодированного слова
1	0	1
2	10 0	3
3	11 0	3
4	10 100 0	6
5	10 101 0	6
6	10 110 0	6
7	10 111 0	6
8	11 1000 0	7
9	11 1001 0	7
..
15	11 1111 0	7
16	10 100 10000 0	11
17	10 100 10001 0	11
..
31	10 100 11111 0	11
32	10 101 100000 0	12

При кодировании формируется сначала последняя группа, затем предпоследняя и т.д., пока процесс не будет завершен. При декодировании, наоборот, сначала считывается первая группа, по значению ее битов определяется длина следующей группы, или итоговое значение кода, если следующая группа – 0.

Рассмотренные типы кодов могут быть эффективны в следующих случаях

1. Вероятности чисел убывают с ростом значений элементов и их распределение близко к такому: $P(x) \geq P(x+1)$, при любом x , т.е. маленькие числа встречаются чаще, чем большие.
2. Диапазон значений входных элементов не ограничен или неизвестен. Например, при кодировании 32-битовых чисел реально большинство чисел маленькие, но могут быть и большие.
3. При использовании в составе других схем кодирования.

Кодирование длин серий (Элиас)

Входной поток для кодирования рассматривается как последовательность из нулей и единиц. Идея кодирования заключается в том, чтобы кодировать последовательности одинаковых элементов (например, нулей) как целые числа, указывающие количество элементов в этой последовательности. Последовательность одинаковых элементов называется *серией*, количество элементов в ней – *длиной серии*. Например, входную последовательность (общая длина 31бит) можно разбить на серии, а затем закодировать их длины.

000000 1 00000 1 0000000 1 1 00000000 1

Используем, например, γ -код Элиаса. Т.к. в коде нет кодированного слова для нуля, то будем кодировать длину серии +1, т.е. последовательность 7 6 8 1 9

7 6 8 1 9 \Rightarrow 00111 00110 0001000 1 0001001

Длина полученной кодовой последовательности равна 25 бит. Метод актуален для кодирования данных, в которых есть длинные последовательности одинаковых бит. В нашем примере, если $P(0) \gg P(1)$.

16.3 Алфавитное кодирование

Кодирование F может сопоставлять код всему сообщению из множества S как единому целому или строить код сообщения из кодов его частей. Элементарной частью сообщения является одна буква алфавита $A = \{a_1, a_2, \dots, a_n\}$.

Пример 1 $A = \{a_1, a_2, a_3\}$, $V = \{0, 1\}$ $a_1 \rightarrow 1001$, $a_2 \rightarrow 0$, $a_3 \rightarrow 010$

сообщение $a_2 a_1 a_2 a_3 \rightarrow 010010010$

Пример 2 Азбука Морзе. Входной алфавит – английский. Наиболее часто встречающиеся буквы кодируются более короткими словами:

$A \rightarrow 01$, $V \rightarrow 1000$, $C \rightarrow 1010$, $D \rightarrow 100$, $E \rightarrow 0$, ...

Побуквенное кодирование задается таблицей кодовых слов: $\sigma = \langle a_1 \rightarrow \beta_1, \dots, a_n \rightarrow \beta_n \rangle$, $a_i \in A$, $\beta_i \in V^*$. Множество кодовых слов $V = \{\beta_i\}$ называется множеством элементарных кодов. Побуквенное кодирование пригодно для любого множества сообщений S : $F: A^* \rightarrow V^*$, $a_{i1} \dots a_{ik} = \alpha \in A^*$, $F(\alpha) = \beta_{i1} \dots \beta_{ik}$.

Количество букв в слове $\alpha = a_1 \dots a_k$ называется длиной слова $|\alpha| = k$. Пустое слово обозначим Λ . Если $\alpha = a_1 a_2$, то a_1 – начало (префикс) слова α , a_2 – окончание (постфикс) слова α .

Побуквенный код называется *разделимым* (или *однозначно декодируемым*), если любое сообщение из символов алфавита источника, закодированное этим кодом, может быть однозначно декодировано, т.е. если $\beta_{i1} \dots \beta_{ik} = \beta_{j1} \dots \beta_{jt}$, то $k=t$ и при любых $s=1, \dots, k$ $i_s = j_s$, т.е. любое кодовое слово единственным образом разлагается на элементарные коды. Например, код из первого примера не является разделимым, поскольку кодовое слово 010010 может быть декодируемо двумя способами $a_3 a_3$ или $a_2 a_1 a_2$.

Побуквенный код называется *префиксным*, если в его множестве кодовых слов ни одно слово не является началом другого, т.е. элементарный код одной буквы не является префиксом элементарного кода другой буквы. Например, код из первого примера не является префиксным, поскольку элементарный код буквы a_2 является префиксом элементарного кода буквы a_3 .

Утверждение. Префиксный код является разделимым.

Доказательство (от противного). Пусть префиксный код не является разделимым. Тогда существует такая кодовая последовательность β , что она представлена различными способами из элементарных кодов: $\beta = \beta_{i1} \dots \beta_{ik} = \beta_{j1} \dots \beta_{jt}$ (побитовое представление одинаковое) и существует L такое, что при любом $S < L$ следует $(\beta_{is} = \beta_{js})$ и $(\beta_{it} \neq \beta_{jt})$, т.е. начало каждого из этих представлений имеет одинаковую последовательность элементарных кодов. Уберем эту часть. Тогда $\beta_{iL} \dots \beta_{ik} = \beta_{jL} \dots \beta_{jt}$, т.е. последовательности элементарных кодов разные и существует β' , что $\beta_{iL} = \beta_{jL} \beta'$ или $\beta_{jL} = \beta_{iL} \beta'$, т.е. β_{iL} – начало β_{jL} , или наоборот. Получили противоречие с префиксностью кода.

Заметим, что разделимый код может быть не префиксным.

Пример. Разделимый, но не префиксный код: $A=\{a,b\}$, $V=\{0,1\}$, $\varphi = \{a \rightarrow 0, b \rightarrow 01\}$

Приведем основные теоремы побуквенного кодирования.

Теорема (Крафт). Для того, чтобы существовал побуквенный двоичный префиксный код с длинами кодовых слов L_1, \dots, L_n необходимо и достаточно, чтобы

$$\sum_{i=1}^n 2^{-L_i} \leq 1.$$

Доказательство. Докажем необходимость. Пусть существует префиксный код с длинами L_1, \dots, L_n . Рассмотрим полное двоичное дерево. Каждая вершина закодирована последовательностью нулей и единиц (как показано на рисунке).

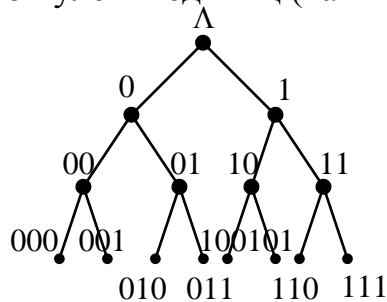


Рисунок 65 Полное двоичное дерево с помеченными вершинами

В этом дереве выделим вершины, соответствующие кодовым словам. Тогда любые два поддерева, соответствующие кодовым вершинам дерева, не пересекаются, т.к. код префиксный. У i -того поддерева на r -том уровне – 2^{r-L_i} вершин. Всего вершин в поддереве 2^r . Тогда $\sum_{i=1}^n 2^{r-L_i} \leq 2^r$, $\sum_{i=1}^n 2^r 2^{-L_i} \leq 2^r$, $\sum_{i=1}^n 2^{-L_i} \leq 1$.

Докажем достаточность утверждения. Пусть существует набор длин кодовых слов такой, что $\sum_{i=1}^n 2^{-L_i} \leq 1$. Рассмотрим полное двоичное дерево с помеченными

вершинами. Пусть длины кодовых слов упорядочены по возрастанию $L_1 \leq L_2 \leq \dots \leq L_n$. Выберем в двоичном дереве вершину V_1 на L_1 уровне. Уберем поддерево с корнем в вершине V_1 . В оставшемся дереве возьмем вершину V_2 на уровне L_2 и удалим поддерево с корнем в этой вершине и т.д. Последовательности, соответствующие вершинам V_1, V_2, \dots, V_n образуют префиксный код.

Пример. Построить префиксный код с длинами $L_1=1, L_2=2, L_3=2$ для алфавита $A=\{a_1, a_2, a_3\}$. Проверим неравенство Крафта для набора длин $\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^2} = 1$.

Неравенство выполняется и, следовательно, префиксный код с таким набором длин кодовых слов существует. Рассмотрим полное двоичное дерево с 2^3 помеченными вершинами и выберем вершины дерева, как описано выше. Тогда элементарные коды могут быть такими $a_1 \rightarrow 0, a_2 \rightarrow 10, a_3 \rightarrow 11$.

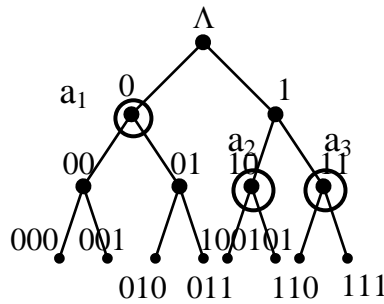


Рисунок 66 Построение префиксного кода с заданными длинами

Процесс декодирования выглядит следующим образом. Просматриваем полученное сообщение, двигаясь по дереву. Если попадем в кодовую вершину, то выдаем соответствующую букву и возвращаемся в корень дерева и т.д.

Теорема (МакМиллан). Для того, чтобы существовал побуквенный двоичный разделимый код с длинами кодовых слов L_1, \dots, L_n , необходимо и достаточно, чтобы $\sum_{i=1}^n 2^{-L_i} \leq 1$.

Доказательство. Покажем достаточность. По теореме Крафта существует префиксный код с длинами L_1, \dots, L_n , и он является разделимым.

Докажем необходимость утверждения. Рассмотрим тождество

$$(x_1 + x_2 + \dots + x_n)^m = \sum_{1 \leq i_1 < \dots < i_m \leq n} x_{i_1} \cdot \dots \cdot x_{i_m}$$

Положим $x_i = 2^{-l_i}$. Тогда тождество можно переписать следующим образом

$$\left(\sum_{i=1}^n 2^{-l_i} \right)^m = \sum_{1 \leq i_1 < \dots < i_m \leq n} 2^{-(l_{i_1} + \dots + l_{i_m})} = \sum_{j=1}^{m l_{\max}} \sum_{j=l_{i_1} + \dots + l_{i_m}} 2^{-j} = \sum_{j=1}^{m l_{\max}} M_j \cdot 2^{-j},$$

где $l_{\max} = \max_i l_i$, M_j – число всевозможных представлений числа j в виде суммы $j = l_{i_1} + \dots + l_{i_m}$. Сопоставим каждому представлению числа j в виде суммы последовательность нулей и единиц длины j по следующему правилу

$$j = l_{i_1} + \dots + l_{i_m} \rightarrow b_{i_1} \dots b_{i_m},$$

где b_s элементарный код длины s . Тогда различным представлениям числа j будут соответствовать различные кодовые слова, поскольку код является разделимым.

Таким образом, $M_j \leq 2^j$ и $\left(\sum_{i=1}^n 2^{-l_i} \right)^m \leq \sum_{j=1}^{m l_{\max}} 2^j \cdot 2^{-j} = m \cdot l_{\max}$. Используя предельный пе-

реход получим $\sum_{i=1}^n 2^{-l_i} \leq \sqrt[m]{m l_{\max}} \rightarrow 1$ при $m \rightarrow \infty$.

Пример. Азбука Морзе – это схема алфавитного кодирования

A→01, B→1000, C→1010, D→100, E→0, F→0010, G→110, H→0000, I→00, J→0111, K→101, L→0100, M→11, N→10, O→111, P→0110, Q→1101, R→010, S→000, T→1, U→001, V→0001, W→011, X→1001, Y→1011, Z→1100.

Неравенство МакМиллана для азбуки Морзе не выполнено, поскольку

$$2 \cdot \frac{1}{2^1} + 4 \cdot \frac{1}{2^2} + 8 \cdot \frac{1}{2^3} + 12 \cdot \frac{1}{2^4} = 3 \frac{3}{4} > 1$$

Следовательно, этот код не является разделимым. На самом деле в азбуке Морзе имеются дополнительные элементы – паузы между буквами (и словами), которые позволяют декодировать сообщение. Эти дополнительные элементы определены неформально, поэтому прием и передача сообщений (особенно с высокой скоростью) является некоторым искусством, а не простой технической процедурой.

16.4 Оптимальное алфавитное кодирование

Побуквенное кодирование пригодно для любых сообщений. Однако на практике часто доступна дополнительная информация о вероятностях символов исходного алфавита. С использованием этой информации решается задача оптимального побуквенного кодирования.

Пусть имеется дискретный вероятностный источник, порождающий символы алфавита $A = \{a_1, \dots, a_n\}$ с вероятностями $p_i = p(a_i)$, $\sum_{i=1}^n p_i = 1$. Основной характеристикой источника является его *энтропия*, которая представляет собой среднее значение количества информации в сообщении источника и определяется выражением (для двоичного случая) $H(p_1, \dots, p_n) = -\sum_{i=1}^n p_i \log_2 p_i$. Энтропия характе-

ризует меру неопределенности выбора для данного источника. Например, если $A = \{a_1, a_2\}$, $p_1 = 0$, $p_2 = 1$, т.е. источник может породить только символ a_2 , то неопределенности нет, энтропия $H(p_1, p_2) = 0$. Максимальная энтропия будет, если все символы равновероятны, например, $A = \{a_1, a_2\}$, $p_1 = 1/2$, $p_2 = 1/2$, тогда неопределенность максимальная, т.е. $H(p_1, p_2) = 1$.

Для практических применений важно, чтобы коды сообщений имели по возможности наименьшую длину. *Основной характеристикой неравномерного кода* является количество символов, затрачиваемых на кодирование одного сообщения. Пусть имеется разделимый побуквенный код для источника, порождающего символы алфавита $A = \{a_1, \dots, a_n\}$ с вероятностями $p_i = p(a_i)$, $\sum_{i=1}^n p_i = 1$, состоящий из n кодовых слов с длинами L_1, \dots, L_n в алфавите $\{0, 1\}$. *Средней длиной кодового слова* называется величина $L_{cp} = \sum_{i=1}^n p_i L_i$ или среднее число кодовых букв на одну букву источника.

Пример. Пусть для имеются два источника с одним и тем же алфавитом $A = \{a_1, a_2, a_3\}$ и разными распределениями $P_1 = \{1/3, 1/3, 1/3\}$, $P_2 = \{1/4, 1/4, 1/2\}$, которые кодируются одним и тем же кодом $\varphi = \{a_1 \rightarrow 10, a_2 \rightarrow 000, a_3 \rightarrow 01\}$. Средняя длина кодового слова для разных распределений будет различной

$$L_{\varphi}(P_1) = 1/3 \cdot 2 + 1/3 \cdot 3 + 1/3 \cdot 2 = 7/3 \approx 2.33$$

$$L_{\varphi}(P_2) = 1/4 \cdot 2 + 1/4 \cdot 3 + 1/2 \cdot 2 = 9/4 = 2.25$$

Побуквенный разделимый код называется *оптимальным*, если средняя длина кодового слова *минимальна* для данного разделения вероятностей символов. *Избыточность* кода является показателем качества кода. *Избыточностью* кода называется разность между средней длиной кодового слова и энтропией ис-

точника сообщений $r=L_{cp}-H$. Задача эффективного неискажающего сжатия заключается в построении кодов с наименьшей избыточностью, у которых средняя длина кодового слова близка к энтропии источника. К таким кодам относятся классические коды Хаффмена, Шеннона, Фано, Гильберта-Мура.

Приведем некоторые свойства, которыми обладает любой оптимальный побуквенный код.

Лемма 1. Для оптимального кода с длинами кодовых слов L_1, \dots, L_n : верно соотношение $L_1 \leq L_2 \leq \dots \leq L_n$ ($p_1 \geq p_2 \geq \dots \geq p_n$).

Доказательство (от противного): Пусть есть i и j , что $L_i > L_j$ при $p_i > p_j$. Тогда

$$\begin{aligned} L_i p_i + L_j p_j &= L_i p_i + L_j p_j + L_i p_j + L_j p_i - L_i p_j - L_j p_i = \\ &= p_i(L_i - L_j) - p_j(L_i - L_j) + L_j p_i + L_i p_j = (p_i - p_j)(L_i - L_j) + L_j p_j + L_i p_i, \end{aligned}$$

т.е. если поменяем местами L_i и L_j , то получим код, имеющий меньшую среднюю длину кодового слова. Противоречие с оптимальностью.

Лемма 2 Пусть $\sigma = \langle a_1 \rightarrow b_1, \dots, a_n \rightarrow b_n \rangle$ схема оптимального префиксного кодирования для распределения вероятностей P , $p_1 \geq p_2 \geq \dots \geq p_n > 0$. Тогда среди элементарных кодов, имеющих максимальную длину, существуют два, которые различаются только в последнем разряде.

Доказательство. Покажем, что в оптимальной схеме кодирования всегда найдется два кодовых слова максимальной длины. Предположим обратное. Пусть кодовое слово максимальной длины одно и имеет вид $b_n = bx$, $x \in \{0,1\}$. Тогда длина любого элементарного кода не больше длины b , т.е. $L_i \leq |b|$, $i = 1, \dots, n$. Поскольку схема кодирования префиксная, то кодовые слова b_1, \dots, b_{n-1} не являются префиксом b . С другой стороны, b не является префиксом кодовых слов b_1, \dots, b_{n-1} . Таким образом, новая схема кодирования $\sigma' = \langle a_1 \rightarrow b_1, \dots, a_n \rightarrow b \rangle$ также является префиксной, причем с меньшей средней длиной кодового слова $L_{\sigma'}(P) = L_{\sigma}(P) - p_n$, что противоречит оптимальности исходной схемы кодирования. Пусть теперь два кодовых слова b_{n-1} и b_n максимальной длины отличаются не в последнем разряде, т.е. $b_{n-1} = b'x'$, $b_n = b''x''$, $b' \neq b''$, $x', x'' \in \{0,1\}$. Причем b' , b'' не являются префиксами для других кодовых слов b_1, \dots, b_{n-2} и наоборот. Тогда новая схема $\sigma'' = \langle a_1 \rightarrow b_1, \dots, a_{n-2} \rightarrow b_{n-2}, a_{n-1} \rightarrow b'x', a_n \rightarrow b'' \rangle$ также является префиксной, причем $L_{\sigma''}(P) = L_{\sigma}(P) - p_n$, что противоречит оптимальности исходной схемы кодирования.

Рассмотрим алгоритм построения оптимального кода Хаффмена.

1. Упорядочим символы исходного алфавита $A = \{a_1, \dots, a_n\}$ по убыванию их вероятностей $p_1 \geq p_2 \geq \dots \geq p_n$.
2. Если $A = \{a_1, a_2\}$, то $a_1 \rightarrow 0$, $a_2 \rightarrow 1$.
3. Если $A = \{a_1, \dots, a_j, \dots, a_n\}$ и известны коды $\langle a_j \rightarrow b_j \rangle$, $j = 1, \dots, n$, то для $\{a_1, \dots, a_j', a_j'', \dots, a_n\}$, $p(a_j) = p(a_j') + p(a_j'')$, $a_j' \rightarrow b_j 0$, $a_j'' \rightarrow b_j 1$.

Пример. Пусть дан алфавит $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$ с вероятностями $p_1=0.36$, $p_2=0.18$, $p_3=0.18$, $p_4=0.12$, $p_5=0.09$, $p_6=0.07$. Будем складывать две наименьшие вероятности и включать суммарную вероятность на соответствующее место в упорядоченном списке вероятностей до тех пор, пока в списке не останется два сим-

вола. Тогда закодируем эти два символа 0 и 1. Далее кодовые слова достраиваются, как показано на рисунке 67.

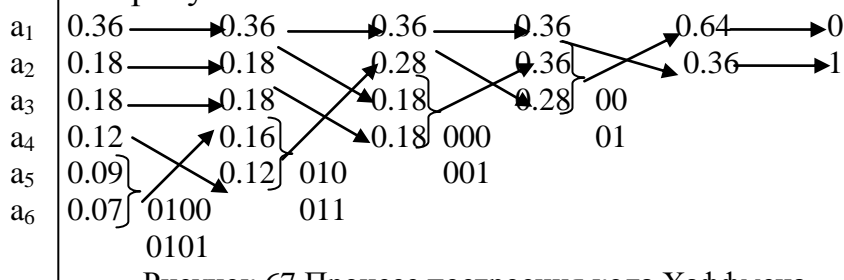


Рисунок 67 Процесс построения кода Хаффмена

Таблица 10 Код Хаффмена

a_i	P_i	L_i	КОДОВОЕ СЛОВО
a_1	0.36	2	1
a_2	0.18	3	000
a_3	0.18	3	001
a_4	0.12	4	011
a_5	0.09	4	0100
a_6	0.07	4	0101

Посчитаем среднюю длину, построенного кода Хаффмена

$$L_{cp}(P) = 1 \cdot 0.36 + 3 \cdot 0.18 + 3 \cdot 0.18 + 3 \cdot 0.12 + 4 \cdot 0.09 + 4 \cdot 0.07 = 2.44,$$

при этом энтропия данного источника равна

$$H = -(0.36 \cdot \log 0.36 + 2 \cdot 0.18 \cdot \log 0.18 + 0.12 \cdot \log 0.12 + 0.09 \cdot \log 0.09 + 0.07 \cdot \log 0.07) = 2.37$$

Код Хаффмена обычно строится и хранится в виде двоичного дерева, в листьях которого находятся символы алфавита, а на «ветвях» – 0 или 1. Тогда уникальным кодом символа является путь от корня дерева к этому символу, по которому все 0 и 1 собираются в одну уникальную последовательность.

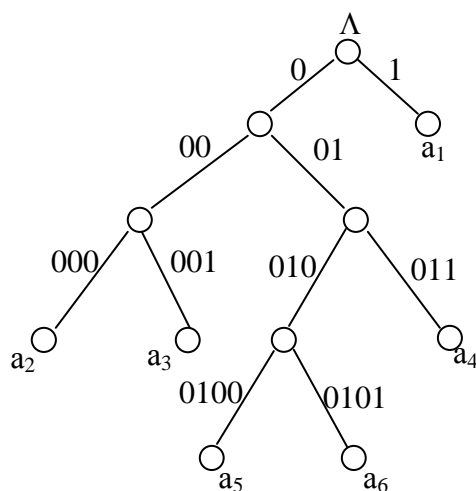


Рисунок 68 Кодовое дерево для кода Хаффмена

Алгоритм на псевдокоде

Построение оптимального кода Хаффмена (n, P)

Обозначим

n – количество символов исходного алфавита

P – массив вероятностей, упорядоченных по убыванию

С – матрица элементарных кодов
 L – массив длин кодовых слов
 Huffman(n,P)
 IF (n=2) C[1,1]:=0, L[1]:=1
 C[2,1]:=1, L[2]:=1
 ELSE q:=P[n-1]+P[n]
 j:=Up(n,q) <поиск и вставка суммы>
 Huffman(n-1,P)
 Down(n,j) <достраивание кодов>
 FI

Функция Up (n,q) находит в массиве P место, куда вставить число q и вставляет его, сдвигая вниз остальные элементы.

DO (i=n-1, n-2,...,2)
 IF (P[i-1]≤q) P[i]:=P[i-1]
 ELSE j:=i
 OD
 FI
 OD
 P[j]:=q

Процедура Down (n,j) формирует кодовые слова.

S:=C[j,*] <запоминание j-той строки матрицы элементарных кодов в массив S>
 L:=L[j]
 DO (i=j,...,n-2)
 C[i,:]:=C[i+1,*] <сдвиг вверх строк матрицы C>
 L[i]:=L[i+1]
 OD
 C[n-1,:]:= S, C[n,:]:= S <восстановление префикса кодовых слов из массива S >
 C[n-1,L+1]:=0, C[n,L+1]:=1
 L[n-1]:=L+1, L[n]:=L+1

16.5 Почти оптимальное алфавитное кодирование

Рассмотрим несколько классических побуквенных кодов, у которых средняя длина кодового слова близка к оптимальной. Пусть имеется дискретный вероятностный источник, порождающий символы алфавита $A=\{a_1, \dots, a_n\}$ с вероятностями $p_i = p(a_i)$, $\sum_{i=1}^n p_i = 1$.

ностями $p_i = p(a_i)$, $\sum_{i=1}^n p_i = 1$.

Код Шеннона

Код Шеннона позволяет построить почти оптимальный код с длинами кодовых слов $L_i < -\log p_i + 1$. Тогда $L_{cp} < H(p_1, \dots, p_n) + 1$. Код Шеннона строится следующим образом.

1. Упорядочим символы исходного алфавита $A=\{a_1, a_2, \dots, a_n\}$ по убыванию их вероятностей: $p_1 \geq p_2 \geq p_3 \geq \dots \geq p_n$.
2. Составим нарастающие суммы вероятностей Q_i :

$$Q_0=0, Q_1=p_1, Q_2=p_1+p_2, Q_3=p_1+p_2+p_3, \dots, Q_n=1.$$
3. Представим Q_i в двоичной системе счисления и возьмем в качестве кодового слова первые $\lceil -\log_2 p_i \rceil$ знаков после запятой.

Для вероятностей, представленных в виде десятичных дробей, удобно определить длину кодового слова L_i из соотношения

$$\frac{1}{2^{L_i}} \leq p_i < \frac{1}{2^{L_i-1}}, \quad i=1, \dots, n.$$

Пример. Пусть дан алфавит $A=\{a_1, a_2, a_3, a_4, a_5, a_6\}$ с вероятностями $p_1=0.36$, $p_2=0.18$, $p_3=0.18$, $p_4=0.12$, $p_5=0.09$, $p_6=0.07$. Построенный код приведен в таблице.

Таблица 11 Код Шеннона

a_i	P_i	Q_i	L_i	кодировое слово
a_1	$1/2^2 \leq 0.36 < 1/2$	0	2	00
a_2	$1/2^3 \leq 0.18 < 1/2^2$	0.36	3	010
a_3	$1/2^3 \leq 0.18 < 1/2^2$	0.54	3	100
a_4	$1/2^4 \leq 0.12 < 1/2^3$	0.72	4	1011
a_5	$1/2^4 \leq 0.09 < 1/2^3$	0.84	4	1101
a_6	$1/2^4 \leq 0.07 < 1/2^3$	0.93	4	1110

Построенный код является префиксным. Вычислим среднюю длину кодового слова и сравним ее с энтропией. Значение энтропии вычислено при построении кода Хаффмена ($H = 2.37$).

$$L_{cp} = 0.36 \cdot 2 + (0.18 + 0.18) \cdot 3 + (0.12 + 0.09 + 0.07) \cdot 4 = 2.92 < 2.37 + 1$$

Алгоритм на псевдокоде

Алгоритм построения кода Шеннона

```

p0:=0, Q0:=0
DO (i=1,...,n)
    Qi := Qi-1+pi
    Li:= -⌈log2pi⌉
OD
DO (i=1,...,n)
    DO (j=1,...,Li)
        Qi-1:=Qi-1*2
        C[i,j]:=⌈Qi-1⌉
        IF (Qi-1>1) Qi-1:=Qi-1-1 FI
    OD
OD

```

Код Фано

Метод Фано построения префиксного почти оптимального кода заключается в следующем. Упорядоченный по убыванию вероятностей список букв алфавита источника делится на две части так, чтобы суммы вероятностей букв, вхо-

дящих в эти части, как можно меньше отличались друг от друга. Буквам первой части приписывается 0, а буквам из второй части – 1. Далее также поступают с каждой из полученных частей. Процесс продолжается до тех пор, пока весь список не разобьется на части, содержащие по одной букве.

Пример. Пусть дан алфавит $A=\{a_1, a_2, a_3, a_4, a_5, a_6\}$ с вероятностями $p_1=0.36$, $p_2=0.18$, $p_3=0.18$, $p_4=0.12$, $p_5=0.09$, $p_6=0.07$. Построенный код приведен в таблице и на рисунке.

Таблица 12 Код Фано

a_i	P_i	кодовое слово				L_i
a_1	0.36	0	0			2
a_2	0.18	0	1			2
a_3	0.18	1	0			2
a_4	0.12	1	1	0		3
a_5	0.09	1	1	1	0	3
a_6	0.07	1	1	1	1	4

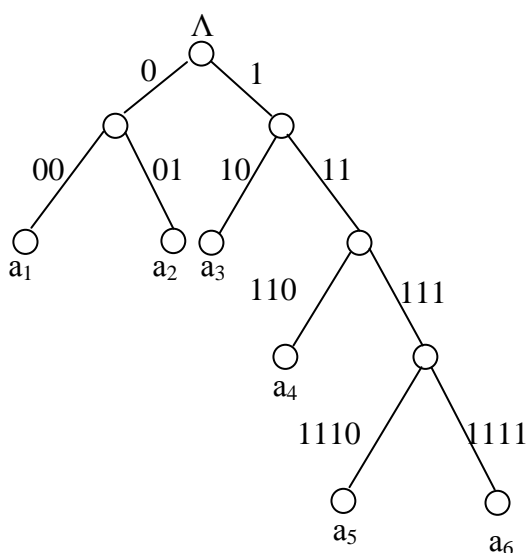


Рисунок 69 Кодовое дерево для кода Фано

Полученный код является префиксным и почти оптимальным со средней длиной кодового слова

$$L_{cp}=0.36 \cdot 2+0.18 \cdot 2+0.18 \cdot 2+0.12 \cdot 3+0.09 \cdot 4+0.07 \cdot 4=2.44$$

Алгоритм на псевдокоде

Алгоритм Фано

Р–массив вероятностей, упорядоченный по убыванию

L – левая граница обрабатываемой части массива Р

Р– правая граница обрабатываемой части массива Р

k – длина уже построенной части элементарных кодов

С – массив элементарных кодов

Length – массив длин элементарных кодов.

Fano(L,R,k)

IF (L<R)

k:=k+1

m:=Med(L,R)

DO (i=L,...,R)

IF (i≤m) C[i,k]:=0, Length[i]:= Length[i]+1

ELSE C[i,k]:=1, Length[i]:= Length[i]+1

FI

OD

Fano (L,m,k)

Fano (m+1,R,k)

FI

Функция Med находит медиану массива P, т.е. такой индекс $L \leq m \leq R$, что

величина $\left| \sum_{i=L}^m p_i - \sum_{i=m+1}^R p_i \right|$ минимальна.

Med (L,R)

S_L:=0

DO (i=L,...,R-1)

S_L:=S_L+p[i] <сумма элементов первой части>

OD

S_R:=p[R] <сумма элементов второй части>

m:=R

DO (S_L ≥ S_R)

m:=m-1

S_L:=S_L+p[m]

S_R:=S_R+p[m]

OD

Med:=m

Алфавитный код Гилберта – Мура

Рассмотрим источник с алфавитом $A = \{a_1, a_2, \dots, a_n\}$ и вероятностями p_1, \dots, p_n . Пусть символы алфавита некоторым образом упорядочены, например, $a_1 \leq a_2 \leq \dots \leq a_n$. Алфавитным называется код, в котором кодовые слова лексикографически упорядочены, т.е. $\varphi(a_1) \leq \varphi(a_2) \leq \dots \leq \varphi(a_n)$.

Е.Н. Гилбертом и Э.Ф. Муром предложили метод построения алфавитного кода, для которого $L_{cp} < H + 2$. Процесс построения происходит следующим образом.

1. Составим суммы Q_i , $i = 1, n$, вычисленные следующим образом:
 $Q_1 = p_1/2$, $Q_2 = p_1 + p_2/2$, $Q_3 = p_1 + p_2 + p_3/2$, ..., $Q_n = p_1 + p_2 + \dots + p_{n-1} + p_n/2$.
2. Представим суммы Q_i в двоичном виде.
3. В качестве кодовых слов возьмем $\lceil -\log_2 p_i \rceil + 1$ младших бит в двоичном представлении Q_i .

Пример. Пусть дан алфавит $A=\{a_1, a_2, a_3, a_4, a_5, a_6\}$ с вероятностями $p_1=0.36$, $p_2=0.18$, $p_3=0.18$, $p_4=0.12$, $p_5=0.09$, $p_6=0.07$. Построенный код приведен в таблице.

Таблица 13 Код Гилберта-Мура

a_i	P_i	Q_i	L_i	кодированное слово
a_1	$1/2^3 \leq 0.18$	0.09	4	0001
a_2	$1/2^3 \leq 0.18 < 1/2^2$	0.27	4	0100
a_3	$1/2^2 \leq 0.36 < 1/2^1$	0.54	3	100
a_4	$1/2^4 \leq 0.07$	0.755	5	11000
a_5	$1/2^4 \leq 0.09$	0.835	5	11010
a_6	$1/2^4 \leq 0.12$	0.94	5	11110

Средняя длина кодового слова не превышает значения энтропии плюс 2

$$L_{cp} = 4 \cdot 0.18 + 4 \cdot 0.18 + 3 \cdot 0.36 + 5 \cdot 0.07 + 5 \cdot 0.09 + 5 \cdot 0.12 = 3.92 < 2.37 + 2$$

16.6 Варианты заданий

- 1) Написать процедуры построения γ -, ω -кодов Элиаса для заданного натурального числа.
- 2) Запрограммировать процедуру кодирования и декодирования последовательности нулей и единиц методом кодирования длин серий.
- 3) Написать процедуру кодирования и декодирования последовательности символов заданным побуквенным префиксным кодом.
- 4) Запрограммировать процедуру, которая определяет является ли заданная схема побуквенного кодирования префиксной.
- 5) Для заданного набора длин кодовых слов написать процедуру построения побуквенного префиксного кода.
- 6) Написать процедуру кодирования текста на русском языке кодом Хаффмена. Определить степень сжатия этого кода.
- 7) Написать процедуру кодирования текста на русском языке кодом Фано. Определить степень сжатия этого кода. Сравнить средние длины кодовых слов кода Хаффмена и кода Фано.
- 8) Написать процедуру кодирования текста на русском языке кодом Шеннона. Определить степень сжатия этого кода. Сравнить средние длины кодовых слов кода Хаффмена и кода Шеннона.
- 9) Написать процедуру кодирования текста на русском языке кодом Гилберта-Мура. Определить степень сжатия этого кода. Сравнить средние длины кодовых слов кода Хаффмена и кода Гилберта-Мура.
- 10) Графически изобразить кодовое дерево заданного префиксного кода.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Вирт Н. Алгоритмы и структуры данных. – М.: , 1989.
2. Кнут Д. Искусство программирования для ЭВМ, сортировка и поиск. – М.: Мир, 1978.
3. Ахо А., Ульман Дж., Хопкрофт Дж. Построение и анализ вычислительных алгоритмов. – М. Мир, 1979.
4. Марченко А.И., Марченко Л.А. Программирование в среде Turbo PASCAL 7.0. Базовый курс. Киев: «ВЕК+», 2003.
5. Березин Б.И., Березин С.Б. Начальный курс С и С++. М: «Диалог-МИФИ», 2001

ПРИЛОЖЕНИЕ А

Псевдокод для записи алгоритмов

Для записи алгоритма будем использовать специальный язык – псевдокод. Алгоритм на псевдокоде записывается на естественном языке с использованием двух конструкций: ветвления и повтора. В круглых скобках будем писать комментарии. В треугольных скобках будем описывать действия, алгоритм выполнения которых не требует детализации, например, <обнулить массив>.

: = Операция присваивания значений.

↔ Операция обмена значениями.

Конструкции ветвления.

1. IF (условие)
 <действие>
 FI
 Если выполняется условие,
 то выполнить действие
 FI указывает на конец этих действий.
2. IF (условие)
 <действия 1>
 ELSE <действия 2>
 FI
 Действия 2 выполняются,
 если неверно условие.
3. IF (условие1)
 <действия1>
 ELSEIF (условие2)
 <действия2>
 ...FI
 Действия 2 выполняются,
 если неверно условие1 и верно условие 2

Конструкции повтора.

1. Цикл с предусловием.
 DO (условие)
 <действия>
 OD
 Действия повторяются
 пока условие истинно.
 OD указывает на конец цикла.
2. Цикл с постусловием.
 DO <действия>
 OD (условие выполнения)
3. Цикл с параметром.
 DO (i=1, 2, ... n)
 <действия>
 OD
 Действия выполняются для значений
 параметра из списка
4. Бесконечный цикл.
 DO
 <действия>
 OD
5. Принудительный выход из цикла.
 DO
 ...IF (условие) OD
 OD
 Если условие истинно, то выйти из цикла.

Елена Викторовна Курапова
Елена Павловна Мачикина

Структуры и алгоритмы обработки данных

Методическое пособие

Редактор Фионов А. Н.
Корректор: Шкитина Д.С.....

Подписано в печать.....
Формат бумаги 62 х 84/16, отпечатано на ризографе, шрифт № 10,
изд. л....., заказ №....., тираж – экз., СибГУТИ.
630102, г. Новосибирск, ул. Кирова, 86.